Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Contents

TCPIP Network Stack Performance in Linux Kernel 2.4 and 2.5

Vaijayanthimala Anand, Bill Hartner
IBM Linux Technology Center
manand@us.ibm.com, bhartner@us.ibm.com

Abstract

We discuss our findings on how well the Linux 2.4 and 2.5 TCPIP stack scales with multiple network interfaces and with the SMP network workloads on 100/1000 Mb Ethernet networks. We identify three hotspots in the Linux TCPIP stack: 1) inter-processor cache disruption on SMP environments, 2) inefficient copy routines, and 3) poor TCPIP stack scaling as network bandwidth increases.

Our analysis shows that the L2 cache_lines_out rate (thereby memory cycles per instruction-mCPI) is high in the TCPIP code path leading to poor SMP Network Scalability. We examine a solution that enhances data cache effectiveness and therefore improves the SMP scalability. Next the paper concentrates on is improving the “Copy_To_User” and “Copy_From_User” routines used by the TCPIP stack. We propose using the “hand unrolled loop” instead of the “movsd” instruction on the IA32 architecture and also discuss the effects of aligning the data buffers. The gigabit network interface scalability workload clearly shows that the Linux TCPIP stack is not efficient in handling high bandwidth network traffic. The Linux TCPIP stack needs to mimic the “Interrupt Mitigation” that network interfaces adopt. We explore the techniques that would accomplish this effect in the TCPIP stack. This paper also touches on the system hardware limitation that affects the gigabit NIC’s scalability. We show that three or more gigabit NICs do not scale in the hardware environment used for the workloads.

1 Introduction

Linux is widely deployed in the web server arena and it has been claimed that Linux networking is optimized to a great extent to perform well in the server network loads such as file serving and web serving, and in packet forwarding services such as firewalls and routers. Linux scales well horizontally in cluster environments which are used for web servers, file servers etc.; however, our studies on IA32 architecture show that the TCPIP network stack in the Linux Kernel 2.4 and 2.5 lack SMP network scalability as more CPUs are added and lack NIC scalability on high bandwidth network interfaces when more NICs are added.

1.1 SMP Network Scalability

Cache memory behavior is a central issue in contemporary computer system performance[6]. Much work has been done to examine the memory reference behavior of application code, operating system and network protocols. Most of this kind of work on network protocols concentrates on uniprocessor systems. This paper discusses the data and instruction memory reference effects of Linux TCPIP stack in a multi-processor system. We examine the L2_cache_line_out rate and instructions retired rate in the receive path of TCPIP and Ethernet driver to understand memory latency.

In IA32 Linux, interrupts from different network interfaces (NICs) are routed to CPUs in a dynamic fashion. The received data, its associated structures that deal with a particular connection, and its activities get processed in different CPUs due to the dynamic routing, which results in non-locality of TCPIP code and data structures, which increases the memory access latency leading to poor performance. This effect is eliminated when the application process, and the interrupt for the particular network interface are aligned to run on the same CPU. By binding the process and interrupt to a CPU, a given connection and its associated activities including the data processing during the duration of that connection are guarenteed to process on the same CPU. This binding results in better locality of data and instructions, and improving the cache effectiveness. Affinitizing process and interrupt to a CPU may be feasible in a single service dedicated server environment, but may not be desirable in all situations. Therefore, reducing the number of L2 lines that bounce between the caches in the TCPIP stack code path is a critical factor in improving the SMP scalability of the TCPIP stack. We use affinity as a tool to understand how TCPIP SMP scalability can be improved.

The Inter-processor cache line bouncing problem can be generally addressed by improving the data memory references and instruction memory references. Instruction cache behavior in a network protocol such as TCPIP has a larger impact on performance in most scenarios than the data cache behavior [6, 2]. Instruction memory refereces may be solely important in scenarios where zero copy [1] is used or scenarios where less data is used. When zero copy is not used, reducing the time spent on the data memory reference considerably improves performance.

In the TCPIP stack, under numerous conditions, the received data is check summed in the interrupt (softirq) handler and is copied to user buffer in the process context. These two contexts, interrupt and process, are frequently executed on different processors due to the dynamic interrupt routing and how processes are scheduled. We proto-typed a patch that forces the csum and copy to happen on the same processor for all conditions resulting in performance improvement. Linux TCPIP does have routines that fold csum into copy; however, these routines are not used in all the code paths. We also show profiling data that supports the need to improve both data and instruction memory references in TCPIP stack.

1.2 Efficient Copy Routines

Not only did we consider reducing the memory reference cycles for data in SMP environment but we also considered it for uniprocessor by improving temporal and spatial locality for data copy. Copying data between user and kernel memory takes a big chunk of network protocol processing time. Zero copy again is the mechanism to reduce this processing time, and improving the packet latency. Even with zero copy, the copy is eliminated only on the send side; improving the copy routines in TCPIP stack would help the receive side processing. We found that the copy routines used in IA32 Linux TCPIP stack can be made more efficient by using “unrolled integer copy” instead of the string operation “movsd.” This paper presents measurements to show that “movsd” is more expensive than “unrolled integer mov.” We also look at the effects of alignment on Pentium III systems. Measurements presented in section 3 show that “movsd” performs better if both the source and the destination buffers are aligned.

1.3 TCPIP Scalability on Gigabit Network

Finally this paper looks at the performance of the TCPIP stack using gigabit bandwidth network interfaces (NICs). We use NIC hereafter in this paper to mean Network adapters. In this paper we concentrate on receive side processing to discuss how the TCPIP stack may be improved for handling high bandwidth traffic. The techniques discussed in this paper for receive side processing are also applicable to transmit side processing but we have not evaluated transmit side performance. This paper presents analysis data to show why TCPIP should process packets in bunches at each layer rather than one packet at a time. High bandwidth network interface cards implement a technique called “interrupt mitigation” to interrupt the processor for a bunch of packets instead of interrupting for each received packet. This paper suggests that this “interrupt mitigation” should be mimicked in higher layer protocols as “packet mitigation.” We also look at how the hardware that we used has limitation that leads to poor gigabit NIC scaling.

The main contribution of this paper is to bring to light the Linux TCPIP SMP scalability problem caused by cache line bouncing among multiple processors in the Linux 2.4.x and 2.5.x kernels. Second this paper points out that the Linux TCPIP stack needs to be tailored to more efficient protocol processing for high bandwidth network traffic.

The rest of the Introduction Section discusses the benchmark, hardware and software environments used. Each section includes the benchmark results, analysis data, description of the problem if one exists, and a technique or a patch that could alleviate the bottleneck. Section 2 deals with the SMP Scalability problem that in Linux 2.4 kernel and also shows how the 0(1) scheduler[8] has improved this scalability to some extent in 2.5 kernel. In section 3 we discuss about efficient copy routines in the TCPIP stack followed by section 4 that discusses TCPIP stack scalability on high bandwidth networks and hardware limitation that causes poor gigabit NIC scalability.

BENCHMARK ENVIRONMENT

Netperf [10] is a well-known network benchmark used in the open source community to measure network throughput and packet latency. Netperf is available at www.netperf.org. Netperf3 is an experimental version of Netperf that has multi-thread and multi-process support. We extended Netperf3 to include multi-adapter and synchronized multi-client features to drive 4-way and 8-way SMP servers and clients. Netperf3 supports streaming, request response, and connection request response functions on TCP and UDP. We used TCP_STREAM for our work so far.

In this paper we evaluate SMP scalability and network NIC scalability using the TCP_STREAM feature of Netperf3. We used multi-adpater workloads between a server and a client for SMP scalability study and used multiple clients and a server with multiple NICs for NIC scalability. The TCP_STREAM test establishes a control socket connection and a data connection then sends streams of data using the data connection. At the end of a fixed time period, the test ends and the total throughput is reported in Mbits per second. The principal metric we used for TCP_STREAM is the throughput and “throughput scaled to 100% CPU Utlization.” Vmstat is used to measure the CPU Utlization at the server. “Throughput scaled to 100% CPU Utlization” is derived using the throughput and the cpu Utlization. The “throughput scaled to CPU” is the throughput which would result if all CPUs could be devoted to the throughput. We refer “throughput scaled to 100% CPU Utlization” as “throughput scaled to CPU” in the rest of the paper.

We used isolated Ethernet 100Mbit and 1000Mbit network for our workloads. Open source profiling tools such as SGI kernprof[7], and SGI lockmeter[3] were used for analysis. Where necessary, additional tools were developed to gather profiling data. In addition to time based profiling of SGI Kernprof, we used Kernprof's Pentium performance-counter based profiling. In performance-counter based profiling, a profile observation is set to occur after a certain number of Pentium performance counter events[4]. We used a frequency of 1000 for our profiling, i.e., for every 1000 cache_lines_out or 1000 instructions retired, the profile records the instruction location. Thus the profile shows where the kernel takes most of its cache_lines_out or where the kernel executes most of its instructions.

HARDWARE AND SOFTWARE ENVIRONMENT

The 100Mb Ethernet test was run on an 8-way Profusion chipset, using 700 Mhz Pentium III Xeon with 4 GB memory and 1 MB L2 cache as our client and a 500 Mhz Pentium III 4-way SMP system with 2.5 GB memory, and 2 MB L2 cache as our server. The NIC cards used were Intel Ethernet PRO/100 Server PCI adapters. For 1000Mb Ethernet, we used the same 8-way system as our server and Pentium III 2-way clients with 1 GB memory (4 of them). The NIC cards used were Intel Ethernet PRO 1000 XF Server PCI adapters. We used both 2.4.x and 2.5.x Linux kernels. In both cases the server and client(s) are connected point-to-point using cross over cables.

2 TCPIP SMP Scalability Using Ethernet 100Mb

2.1 Netperf3 TCP_STREAM results

We measured the SMP scalability of the 2.4.17 and 2.5.3 TCPIP stack using the Netperf3 multi-adapter TCP_STREAM test. The server used was a 4-way 500 Mhz PIII with four 100 Mb ethernet NICs operating in full duplex mode with a MTU size of 1500 bytes. The server NICs (receive side) were connected to the client NIC (send side) point-to-point. We used 8-way, 4-way and 2-way clients to drive 4-way, 2-way and 1-way servers respectively.

A single TCP connection was created between the server and the client on each of the networks for a total of four connections. A process was created for each of the NICs for a total of (4) server processes. The test was run using the application message sizes of 1024, 2048, 4096, 8192, 16384, 32768 and 65536 bytes. The tcp socket buffer size was 65536 bytes and TCP NODELAY OPTION was set on the socket. Both the network throughput (Mb/sec) and CPU utilization were measured on the server. The throughput scaled to CPU utilization is derived from these two measurements. Especially for the 100 Mb Ethernet workload, the "throughput scaled to CPU" is the important measure since the throughput of each of the 4 adapters reaches maximum media speed in all tests.

We ran the test with 1, 2, and 4 CPUs enabled on the server. The 2P and 4P SMP scalablity factor was calculated by dividing the respective "throughput scaled to CPU utilization" by the UP "throughput scaled to CPU utilization."

In Figure 1 and Figure 2, we report the 2P and 4P SMP scalability factor of the 2.4.17 and the 2.5.3 kernel for the Netperf3 TCP_STREAM test using the different Netperf3 message sizes.

Note that the 4P scalability factor for the 2.4.17 kernel using the 65536 message size barely achieves 2/4 scaling. The 4P scalability factor of the 2.5.3 kernel is much improved reaching 2/4 in most of the cases. The better 4P scalability of the 2.5.3 kernel may be attributed to the 0(1) scheduler which provides better process affinity to a CPU than the 2.4.17 scheduler.

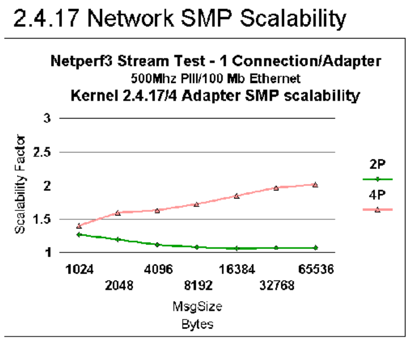


Figure 1: SMP Network Scalability on 2.4.17 Kernel

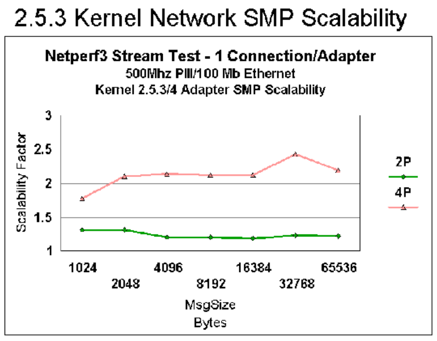


Figure 2: SMP Network Scalability on 2.5.3 Kernel

The SMP TCPIP scalability for 4P and 2P as shown in Figures 1 and 2 is poor. To understand this problem we ran some tests. We choose 4P and 4 adapter cases for our experiment. First we examined the effects of IRQ and process affinity on the Netperf3 TCP_STREAM test for the 2.4.17 kernel using the 4-way server. IRQ affinity involves binding each of the four network adapter IRQs to one of the (4) CPUs. For example, the interrupt assigned to NIC 1 to CPU 0, NIC 2 to CPU 1 etc. PROCESS affinity involves binding each of the four Netperf server processes to one the four CPUs. Both PROCESS and IRQ affinity can be combined by affinitizing the IRQ for NIC and netperf3 server process servicing the tcp connection to the same CPU. In table 1, we report the percent improvement of the Netperf3 TCP_STREAM throughput scaled to CPU when IRQ affinity, PROCESS affinity and BOTH affinities were applied to the server. The percentage improvement is relative to the throughput scaled to CPU utilization when no affinity was applied. The higher the percentage, better the performance.

For the case of IRQ affinity, throughput scaled to CPU utilization improved 4 to 21% for the various message sizes. For the case of PROCESS affinity, throughput scaled to CPU utilization improved 6 to 28%. For the case of both IRQ and PROCESS affinity applied, throughput scaled to CPU utilization improved 42 to 74%. The gain in the third case, where both IRQ and PROCESS affinity are applied, is much greater than the sum of the results of these two affinities applied separately.

Figure 3 shows the comparison of the TCP_STREAM throughput scaled to CPU utilization for the 2.4.17 kernel with 1) no affinity 2) IRQ affinity 3) PROCESS affinity 4) BOTH affinities, and 5) the 2.5.3 kernel with no affinity applied.

The TCP_STREAM results for the 2.5.3 base kernel and the 2.4.17 kernel with PROCESS

Msgsize (Bytes)	IRQ Affinity (%)	PROCESS Affinity (%)	BOTH Affinities (%)
1024	4.34	12.71	74.53
2048	21.45	21.43	66.56
4096	19.01	28.73	74.68
8192	19.18	25.25	68.03
16384	16.36	14.37	55.22
32768	11.38	11.38	47.45
65536	11.31	6.34	42.09

Table 1: Affinity Comparison using 2.4.17 Kernel ON 4P using 4 NICS

affinity are comparable. Again this is probably attributable to the fact that the 2.5's 0(1)scheduler achieves better process affinity to CPU than the 2.4.17 kernel. The best performed test run in figure 3 is the 2.4.17 kernel with both Affinities applied.

The throughput for the results presented in the Figure 3 is mostly constant for most of the tests, and the difference reflected in CPU idle time.

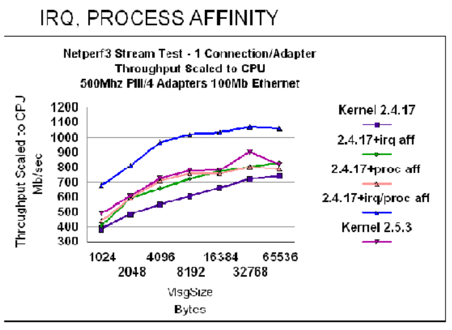


Figure 3: IRQ and PROCESS AFFINITY

2.2 Analysis of IRQ and PROCESS Affinity

```
L2_CACHE_LINES_OUT
```

Next, we analyzed the 4P TCP_STREAM test to understand why IRQ and PROCESS affinity together improves throughput and CPU utilization to a great extent (up to 74%). We used profiling based on Intel Performance counters [4]. We profiled the server during the TCP_STREAM test using the 32K message size on 2.4.17 kernel using the L2_cache_lines_out and total instructions retired events.

By using IRQ and PROCESS affinity, each TCPIPconnection and its activities are bound to happen in the affinitized CPU. This binding leads to lower L2 cache lines out as the data and instructions do not float between CPUs. The gain we achieved through affinity would reflect in this event counter. So we decided to measure L2_cache_lines_out. Because the throughput and throughput scaled to CPU increased in the IRQ and PROCESS affinity case, we decided to measure the total instruction retired count also.

In Table 2 we show the results of the performance counter profiling taken for the event "L2_cache_lines_out" on base Kernel 2.4.17 and Kernel 2.4.17+IRQ and PROCESS affinity. The kernel routines with the highest L2_cache_line_out are listed here. The results in Table 2 show that using IRQ and PROCESS affinity has reduced the number of L2 cache lines out in all of the listed kernel routines. The kernel routines tcp_v4_rcv, mod_timer, and tcp_data_wait are the major benefectors of affinity. Overall the whole of TCPIP receive code path has less L2_cache_lines_out resulting in better performance.

This profile is taken using TCP_STREAM test on 4P server and 8P client. The test ran on 4 adapters using 4 server processes with the configuration of 64K socket buffer, TCP NODELAY ON using 32k message size on 2.4.17 kernel:

Kernel	2.4.17 Kernel	IRQPROCESS
function	Baseline	AFFINITIES
	Frequency	Frequency
poll_idle	121743	72241
csum_partial_copy_generic	27951	13838
schedule	24853	9036
do_softirq	9130	3922
mod_timer	6997	1551
TCP_v4_rcv	6449	629
speedo_interrupt	6262	5066
__wake_up	6143	1779
TCP_recvmsg	5199	2154
USER	5081	2573
speedo_start_xmit	4349	1654
TCP_rcv_established	3724	1336
TCP_data_wait	3610	748

Table 2: L2 Cache Lines Out on 2.4.17

INSTRUCTIONS RETIRED

Next we measured the total instructions retired for the same workload on both baseline 2.4.17 kernel and 2.4.17+IRQ+PROCESS affinity applied kernel. Figure 3 shows the total instructions retired for a short period of time running the TCP_STREAM 4P/4adapter test.

Kernel Function	Instruc. Retired Frequency
2.4.17 Kernel base	32554892
poll_idle on base	19877569
2.4.17 +IRQ+PROCESS	51607249
poll_idle on IRQ+PROCESS	39326958

Table 3: Instructions Retired Count on 2.4.17

The number of instructions executed is high in the affinity case which also supports the fact that lining up process/irq with a CPU brings memory locality and improves instruction memory references. We also presented the number of instructions retired in idle loop. The instructions retired in the idle-loop is doubled in the affinity case. We gained CPU in Affinity case as the memory latency for instructions is decreased. The number of instructions retired excluding the idle-loop case has not improved in the affinity case.

The above analysis clearly indicates that the TCPIP stack SMP scalability can be improved by fixing the inter-processor cache line bouncing by reducing L2_cache_lines_out.

2.3 Combine_csum_copy Patch to reduce the cache_lines_out

Affinitizing both the IRQ and PROCESS to a CPU results in better locality of data and instructions for the TCPIP send and receive path and thus better performance. Because affinity is not feasible in all situations, we analyzed the code to determine if there are code optimizations that could provide better cache effectiveness. It was observed that most of the time, the incoming frames were checksummed in the interrupt context and then copied to the applicatoin buffer in the process context. Often, the interrupt and process context were on two different CPUs. A proto-type patch, csum_copy_patch, was developed to force the checksum and copy operations to execute more often in the process context.

Figure 4 shows the results of the TCP_STREAM test for 1) 2.4.0 kernel baseline, 2) 2.4.0 +IRQ and PROCESS affinity and 3) 2.4.0+csum_copy_patch. The csum_copy_patch improved throughput scaled to CPU utilization by up to 14%. There is additional work to be done in order to bridge the gap between the baseline and the IRQ and PROCESS affinity case. We will continue our work to see how we could close the gap between the non-affinity and affinity case through code improvement.

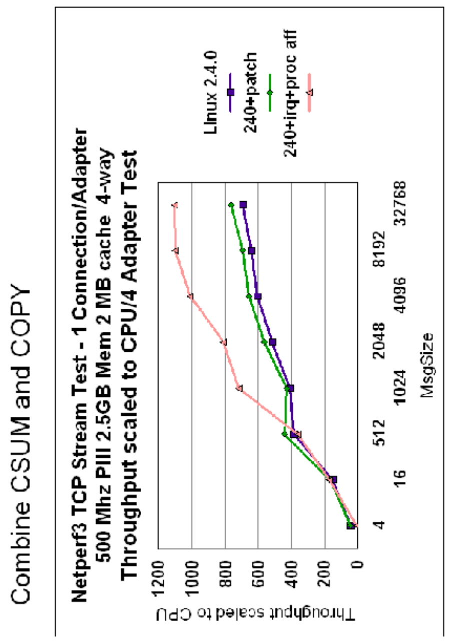


Figure 4: Combine CSUM and COPY on 2.4.0

Kernel function	Frequency
__generic_copy_to_user:	3127
e1000_intr:	540
alloc_skb :	313
TOTAL_SAMPLES	6001

Table 4: Gigabit PC Sampling of 2.4.17 UNI Kernel

3 Copy Routines in TCPIP

3.1 One gigabit NIC's TCP_STREAM Results

We measured the Netperf3 TCP_STREAM throughput for a single connection over a gigabit ethernet network using a 2.4.17 UNI kernel. The Netperf3 message size was 4096 bytes, MTU size was 1500 bytes, and the server (receive side) was an 8-way processor 700 Mhz PIII using a UNI kernel. We observed that the resulting throughput did not achieve maximum media throughput. The CPU was 19% utilized. A time based profile of the kernel revealed that 30–50% of the total ticks were spent in __generic_copy_to_user routine in the receive path of the TCPIP code. The __generic_copy_to_user routine is used to copy data from the network buffers to the application buffers. The profiling data for the TCPIP receive path is given in Table 4

3.2 Copy Routine Analysis

We analyzed the __generic_copy_to_user routine looking for ways to improve the code. The copy code was using the move string (MOVSD) instruction and had no special case code for handling mis-aligned data buffers. We looked at some of the fast memory copy work done previously and in particular the work done by University of Berkeley during the P5 time frame. The Berkeley study [9] compares three types of copies

- STRING: MOVSD string copy.

- INTEGER: unrolled/prefetched using integer registers.
- FLOATING POINT: unrolled/prefetched using floating point registers.

According to their results on P5 machines, the floating point copy method yielded 100% more throughput when compared to the string copy method. The integer copy method yielded 50% better throughput than the string copy method. We adopted both integer copy and floating point copy methods from this technical paper for improving the copy routines in the TCPIP stack.

We developed a user level tool to test these copy methods and found that the integer copy performs better than the other two methods if the source and destination buffers are not aligned on 8-byte boundaries. As shown in Table 5, if the source and the destination buffers are aligned on a 8-byte boundary, the string copy performed better than the "unrolled integer copy." We used a Pentium III system for this test and each test copied 1448 million bytes. In Table 5, the MBytes copied is the throughput and higher the number the copy method is more efficient.

3.3 CopyPatch for Efficient Copy

We created a copy patch using "unrolled integer copy" for the Linux Kernel and tested further with and without alignment to further understand the impact of this patch and buffer alignment on our workload. We decided to test the alignment in the receive path of the TCPIP stack. The gigabit driver was modified to align the receive buffer (which is the source buffer for the copy routine). The destination buffer allocated by netperf3 was already aligned. We instrumented __generic_copy_to_user to measure the CPU cycles spent in this routine. We read the Intel's TSC counter before and after execution of the copy routine with HW interrupts disabled. A user level program was written to retrieve the value of the cycle counter during the test run several times and also after the test is completed.

The results showed in Table 6 are the average cycles (rounded) spent to copy a buffer size of 1448 with and without the patch and with alignment. Lower the cycles better the performance of the copy routine.

Method Used	Cycles Spent
movsd copy routine without alignment	7000
movsd copy routine with 8 byte alignment	3000
IntegerCopyPatch without alignment	4000

Table 6: Measurement of cycles spent in copy methods on 2.4.17 Kernel

The data in Table 6 suggests that the MOVSD instruction has the best performance when the source and the destination buffer addresses are aligned on an 8-byte boundary. However, an 8-byte source and destination alignment may not be possible in the receive path of the TCPIP stack for all general purposes and in all hetrogenous networks. The TCPIP frame header size is variable due to the TCP and IP options in the header. For our analysis purpose we were able to align this as we had a controlled and isolated homogenous network environment. So we decided to implement the "unrolled integer copy" replacing the "movsd" string copy in the copy routines used in this workload as aligning the buffers is out of question in the TCPIP receive path.

Figure 5 and Figure 6 show the baseline and CopyPatch throughput and "throughput scaled to CPU" results on 2.4.17 and 2.5.7 kernels respectively. It is obvious that the unrolled

Method	time taken (sec)	MBytes copied	dst address	src address	aligned
MOVSD	0.609	2378	804c000	804f000	YES
Integer	0.898	1613	8051000	8053000	YES
MOVSD	1.172	1235	804c000	804f004	NO
Integer	0.851	1703	8051000	8053004	NO

Table 5: Comparison of movsd, integer copy, alignment

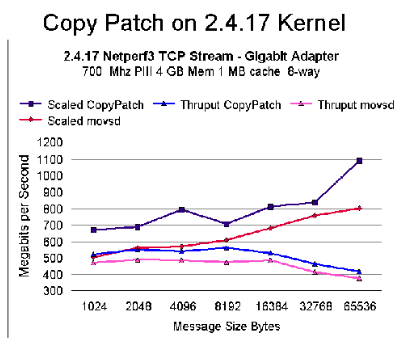


Figure 5: Copy Patch on 2.4.17 Kernel

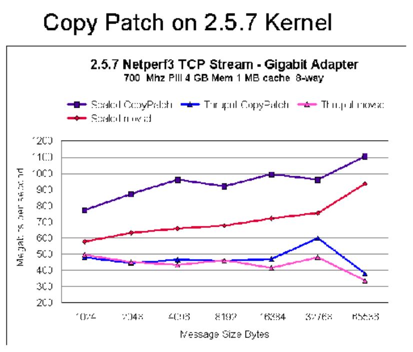


Figure 6: Copy Patch on 2.5.7 Kernel

integer copy routine has improved throughput scaled to CPU for all the message sizes on both kernels. On 2.4.17, the raw throughput improved for all message sizes with the copy-patch, however on 2.5.7 kernel, copypatch improved raw throughput on messages greater than 8k.

There is other copy routines combined with checksum in the TCPIP stack, we have not modified those routines yet. See appendix for integer copy patch. Since there are other methods such as sendfile (only for sendside) and mmx copies that are applicable to cover some situations, the scope of this work may look limited but this kind of string copy is used in other places of the kernel, glibc etc., So we think this work is important to improve the Linux performance in IA32 architecture and for the future "string copy instruction - movsd" implementation in IA32 architecture.

3.4 Future Work

As part of our future work we will look in to improving the following:

- Extend this work to other memcopy routines in the kernel.
- Extend this work to glibc routines.

4 TCPIP GIGABIT NIC SCALABILITY

4.1 Gigabit NIC Scalability Results

The gigabit Ethernet NIC scalability test measures how well multiple gigabit Ethernet NICs perform on an 8-way server. We measured Gigabit Ethernet NIC scalability on the 2.4.7 kernel using the Netperf3 multi-adapter TCP_STREAM test. The server used was an 8-way 700 Mhz Pentium III Xeon CPUs with up to seven Gigabit Ethernet NICs operating in full duplex mode with a MTU size of 1500 bytes. Four 2-way 900 Mhz clients (send side) with two Gigabit Ethernet NICs on each connected to the server (receive side) Ethernet NICs.

The test was first run with only one gigabit NIC, then two NICs, and so on, up to a total of seven gigabit Ethernet NICs. A single TCP connection was created between the server and the client on each of the Ethernet NICs. The test was run with application message sizes of 1024, 2048, 4096, 8196, 16384, 32768 and 65536. The TCP socket buffer size was set to 64K with TCP NODELAY ON. The Ethernet NICs default options are used for the configuration parameters; although, tuning these options did not yield any better results on our hardware. We used SMP kernels enabling 8 processors on the server and 2 processors on each of the client.

The throughput results of the NIC scalability test are found in Figure 7. A single Ethernet NIC achieved only 604 Mb per second. Furthermore, adding a second Ethernet NIC achieved only a total of 699Mb per second for the pair of NICs (yielding a scaling factor of only 58% = 699/604*2). Adding successive Ethernet NIC added minimal throughput. The results of the gigabit Ethernet NIC scalability test inidcates that the gigabit Ethernet NICs tests do not scale on this 8-way system.

4.2 Analysis of TCPIP Scalability on Gigabit Network

We profiled the kernel to understand what could be done in the software to improve the gigabit NICs Scalability. From the TCPIP stack and kernel perspective, our analysis points out that the network stack does not efficiently handle the high rate of incoming frames. One of the main problems we noticed was that the high rate of incoming frames was being processed at the protocol level one at a time even though the NIC mitigates interrupts and causes interrupt once per configurable interrupt delay. Therefore the NIC causes interrupts for a bunch of frames instead of interrupting the processor for each frame. This interrupt mitigation is not mimicked in higher layer protocol processing.

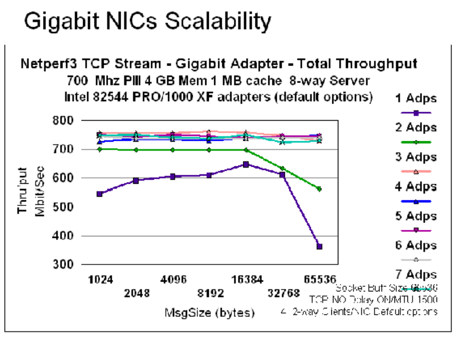


Figure 7: Gigabit NIC Scalability on 2.4.7 Kernel

Kernprof time based annotated call graph profiling taken with MTU=1500 and MTU=9000

By setting the MTU=9000 (jumbo size), on a gigabit adapter/TCP_STREAM, we could reach the max media limit, whereas with MTU set to 1500, we did not reach the maximum media limit on our hardware. We used profiling tools to see the difference. Tables 7 and 8 show the annotated call graph for a three adapter case using 1500 and 9000 MTU sizes. In MTU=1500 case we received less interrupts around 275543 times. But the softirq handler was invoked around 2 million times. There-

Kernel function	Times Invoked
e1000_intr	840790
. ProcessReceiveInterrupts	951848
. . netif_rx	758774
. . . get_fast_time	758774
. . . . do_gettimeofday	758774
. . . get_sample_stats	758774
. . . cpu_raise_softirq	758774
. . eth_type_trans	758774
. . RxChecksum	758774
. . _tasklet_schedule	12339
. . . wake_up_process	3
. . . . reschedule_idle	3367
. ProcessTransmitInterrupts	951848
. . cpu_raise_softirq	368286

Table 8: 2.4.7 Kernel Kernprof's Annotated callgraph for MTU 9000

Kernel Function	Times invoked
e1000_intr	275543
. ProcessReceiveInterrupts	312522
. . netif_rx	2291591
. . . get_fast_time	2291591
. . . . do_gettimeofday	2291591
. . . get_sample_stats	2291591
. . . cpu_raise_softirq	2291591
. . eth_type_trans	2291591
. . RxChecksum	2291591
. . _tasklet_schedule	32369
. ProcessTransmitInterrupts	312522
. . cpu_raise_softirq	189394

Table 7: 2.4.7 Kernel Annotated Callgraph for MTU 1500

fore, we received around 8 frames per interrupt in an average. Whereas in the MTU=9000 case, we received 860970 time interrupts and softirq handler was invoked for 758774 times. We received one frame per interrupt on an average. Therefore, the upper layer protocols are not invoked more than the interrupts and the protocol processing latency was less so we received 3X more interrupts/frames in case of jumbo size frame. Therefore, we suspect that the higher layer protocol processing latency is causing the throughput to go down in the case of MTU=1500 as each layer of the protocol is processing one frame at a time. We propose using a mechanism such as "gather-receive" where the bunch of the frames received are gathered and sent to upper layer protocol for processing. We have not done any proto-type yet to prove that this will help.

Kernel Function	Times invoked
ReceiveBufferFill	32369
. alloc_skb	2291645
. . kmalloc	2481043
. . . kmem_cache_alloc_batch	63866
. . kmem_cache_alloc	1178633
. . . kmem_cache_alloc_batch	7110

Table 9: 2.4.7 Kernel's Kernprof Annotated Callgraph for MTU 1500

We also found a similar problem with the allocation and deallocation of skbs (socket buffers) and data frame buffers in the receive path. The Linux gigabit network device driver has a receive pool of buffers called "receive ring." Gi-

Kernel function	Times Invoked
ReceiveBufferFill	12339
. alloc_skb	758748
. . kmalloc	1127037
. . . kmem_cache_alloc_batch	170
. . kmem_cache_alloc	178716
. . . kmem_cache_alloc_batch	792

Table 10: 2.4.7 Kernel Kernprof's Annotated callgraph for MTU 9000

Table 11: Two NIC using separate PCI buses

Bus		Throughput
NIC 1	Bus A (66 MHz)	385 Mb/sec
NIC 2	Bus B (66 MHz)	387 Mb/sec
Total		782 Mb/sec

Table 12: Two NIC sharing the PCI bus

Bus		Throughput
NIC 1	Bus A (66 MHz)	355 Mb/sec
NIC 2	Bus A (66 MHz)	342 Mb/sec
Total		697 Mb/sec

gabit driver replenishes this pool by allocating skb one at a time. Tables 9 and 10 show that alloc_skb is called 2 million times when ReceiveBufferFill is called only 32 thousand times in the case where MTU was set to 1500. ReceiveBufferFill is the routine that replenishes the receive buffer pool. TCPIP stack should provide a way to allocate these skbs bunch at a time. To take this one step further, we think that the allocated buffers (receive ring in the driver) should be recycled instead of freeing and reallocating them again. We have started looking into creating a proto-type patch and this is our on-going effort.

4.3 Analysis of Hardware for Gigabit NIC Scalability

We performed additional tests in order to understand why multiple NICs do not scale well on this 8-way system. We first investigated the PCI bus. The 8-way server has two 66 Mhz 64-bit PCI buses (A and B) and two 33 Mhz 64-bit PCI buses (C and D). We used gigabit Ethernet NICs on different combinations of PCI busses and reran the tests.

Tables 4.3 and 4.3 indicate that when two gigabit Ethernet NICs are used, placing one on bus A and the other on bus B improved throughput by 10% compared to having both adapters on bus A.

Table 13: Three Adapters, Separate vs. Shared

Three adapters using separate PCI bus		
NIC	Bus	Throughput
1	Bus A (66 MHz)	266 Mb/sec
2	Bus B (66 MHz)	257 Mb/sec
3	Bus C (33 MHz)	259 Mb/sec
Total		782 Mb/sec
Three adapters sharing PCI bus		
1	Bus A (66 MHz)	197 Mb/sec
2	Bus A (66 MHz)	197 Mb/sec
3	Bus B (66 MHz)	356 Mb/sec
Total		750 Mb/sec

Running three adapters on three different buses yields almost the same throughput as the case where 3 NICs share the same 66 Mhz bus. The PCI bus capacity is 532 MByte per second on 66 Mhz and 266 MByte/sec on 33 Mhz [5] But we are not even getting 800 Mbits/sec total in the above cases. We are far from hitting the PCI bus capacity; therefore, we concluded that the PCI bus is not the bottleneck.

Next we looked at the effects of IRQ and Process affinity on the workload. We tested affinity on 2 NICs. Since we have 8 CPUs on our server system we chose different combinations of CPUs to see if selecting one CPU from each side of the Profusion chip set would make a difference. (The 8-way is composed of two 4-ways with independent 800 MBytes/sec front side buses, connected by the Profusion chip to 2 memory cards providing 1600 MBytes/sec and to the PCI buses via a 800 MByte/sec connection.) We affinitize Netperf3 server processes to a combination of CPU sets and the Ethernet NIC's IRQs to a combination of CPU sets. The system is not rebooted between the tests, so the numbers assigned to CPUs by the operating system stayed intact between the tests. The server process was restarted before each test and the IRQs for the NICs were reset after each test. Both the NICs were placed in Bus A (a 66 Mhz/64 bit bus) and interrupts 23 and 24 were assigned to these two NICs. Interrupt 23 was bounded to CPU0 for all the tests and interrupt 24 was affinitized to CPU1, CPU2 etc. Netperf's server process 1 is always affinitized to CPU0 and process 2 is bound to CPU 1, CPU 2 and so on. The results of the IRQ and PROCESS affinity test are in Figure 8. The throughput has not improved that much compared to the baseline. The max throughput that we get with both affinity is around 730 Mbits/sec and our baseline is 699 Mbits/sec.

4.4 Future Work

Neither the IRQ and PROCESS affinity nor selecting CPUs from different sides of the bus improved the throughput much. These 2 and 3 adapter cases are neither CPU bound nor network media limited. So we must contnue our analysis to find the bottleneck. We also did some preliminary investigation using Intel Performance counter profiling; but, found no conclusive leads. We will continue this work further. However, we have shown that the PCI bus is not the limiting factor, and IRQ and PROCESS affinity do not help improve the throughput and scalability. It is not acceptable that even one adapter does not achieve close to media speed using MTU 1500 size and that adding NICs do not scale well. Gigabit NIC scalability will be a focus of our future investigation.

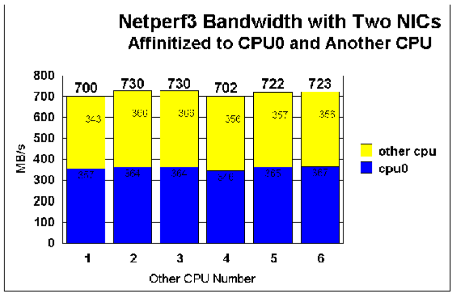


Figure 8: Results of IRQ and Process Affinity with 2 NICs

5 Concluding Remarks

In this paper we have highlighted a few potential areas for improvement in the Linux TCPIP protocol.

- SMP Network Scalability: We presented results showing the SMP network scalability problem and provided analysis to associate the poor scalability to interprocessor cache line bouncing due to high L2_cache_lines_out problem. We believe that this SMP network scalability is one of the areas where TCPIP stack needs to be fixed to improve scalability. We also showed a proto-type to improve the data cache reference in the TCPIP stack. Additionally we examined efficient copy routines for the IA32 Linux TCPIP stack and expressed the belief that this may be a potential area to further investigate to gain performance improvement in the IA32 kernel itself and glibc routines.

- TCPIP Scalability on Gigabit: We examined the effects of using gigabit network on the LINUX TCPIP stack. We presented various test results and analysis data to show that the hardware that we used is not good enough to handle more than 2 NICs. We also emphasized that the implementation of the Linux TCPIP stack itself needs modifications to handle high bandwidth network traffic. As we move to other types of high bandwidth networks such as InfiniBand, we need to keep in mind that the software network stack should also need to scale to utilize fully the high network bandwidth. We conclude that both the system hardware and the software need improvement to take advantage of the high network bandwidth.

We will be working on fixing the above mentioned problems. We look forward to working with the members of the Linux community to discuss, design, and implement solutions to improve the LINUX TCPIP SMP scalability and gigabit network scalability.

6 Acknowledgments

The authors would like to acknowledge the assistance of Fadi Sibai of Intel in interpreting the Pentium Performance counter data. We would like to acknowledge Bill Brantley of IBM and Patricia Goubil-Gambrell of IBM for improving the quality of the paper. We would also like to thank Nivedita Singhvi of IBM and Bruce Alan of IBM for sharing their gigabit NICs scalability test results. This helped us to verify our results.

7 About the authors

Vaijayanthimala (Mala) K Anand works in the IBM Linux Technology center as a member of the Linux Performance team. Mala has worked on the design and development of network protocols, network device drivers and thin clients. Mala is currently working on Linux TCPIP stack performance analysis. She can be reached at manand@us.ibm.com.

Bill Hartner is the technical lead for IBM's Linux Technology Center performance team. Bill has worked in software development for 18 years. For the past 8 years, Bill has worked in kernel development and performance. For the past 3 years Bill has worked on Linux kernel performance. Bill can be reached at bhartner@us.ibm.com.

References

[1] Jeff Chase Andrew Gallatin and Ken Yocum. Trapeze IP:TCPIP at Near Gigabit speeds. http://www.cs.duke.edu/ari/trapeze /freenix/paper.html.

[2] Trevor Blackwell. Speeding up protocols of small messages. ACM SIGCOMM Symposium on Communications Architectures and Protocols,Aug 1996.

[3] R. Bryant and J. Hawkes. Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, Oct 2000.

[4] Intel Corp. Intel Architecture Software Developer's Manual Volume 3: Sysytem Programming. http://www.intel.com.

[5] Adaptec corp's White Paper. PCI, 64-Bit and 66 MHz Benefits. http://www.adaptec.com/worldwide /product/markeditorial.html.

[6] Jim Jurose Erich Nahum, David Yates and Don Towsleyr. Cache Behavior of Network Protocols, June 1997. http://cs-www.bu.edu/faculty/djy.

[7] John Hawkes et. al (Silicon Graphics Inc.). Kernprof. Available at http://oss.sgi.com/projects/kernprof /index.html.

[8] Ingo Molnar. 0(1) scheduler patch. http://www.kernel.org/pub/linux/kernel/ people/mingo.

[9] University of Berkeley. Fast Memory Copy. http://now.cs.berkeley.edu/ Td/bcopy.html.

[10] Hewlett Packard Inc. Rick Jones. Network Benchmarking Netperf. http://www.netperf.org.

8 Appendix

8.1 COPY Patch

This patch changes the copy routines used in tcpip stack.

```
diff -Naur linux-417/arch/i386/lib/usercopy.c \
           linux-417a/arch/i386/lib/usercopy.c
--- linux-417/arch/i386/lib/usercopy.c  Tue Jan 22 21:29:05 2002
+++ linux-417a/arch/i386/lib/usercopy.c Fri Jan 18 12:50:38 2002
@@ -44,7 +44,6 @@
 unsigned long
 __generic_copy_to_user(void *to, const void *from, unsigned long n)
 {
-       prefetch(from);
        if (access_ok(VERIFY_WRITE, to, n))
                __copy_user(to,from,n);
        return n;
diff -Naur linux-417/include/asm-i386/string.h \
           linux-417a/include/asm-i386/string.h
--- linux-417/include/asm-i386/string.h Tue Jan 22 21:29:48 2002
+++ linux-417a/include/asm-i386/string.h        Wed Jan 23 00:11:29 2002
@@ -196,21 +196,65 @@
 return __res;
 }

-static inline void * __memcpy(void * to, const void * from, size_t n)
+static inline void * __memcpy(void * to, const void * from, size_t  size)
 {
-int d0, d1, d2;
+  int __d0, __d1;
 __asm__ __volatile__(
-       "rep ; movsl\n\t"
-       "testb $2,%b4\n\t"
-       "je 1f\n\t"
-       "movsw\n"
-       "1:\ttestb $1,%b4\n\t"
-       "je 2f\n\t"
-       "movsb\n"
-       "2:"
-       : "=&c" (d0), "=&D" (d1), "=&S" (d2)
-       :"0" (n/4), "q" (n),"1" ((long) to),"2" ((long) from)
-       : "memory");
+               "       cmpl $63, %0\n\t"
+               "       jbe  2f\n\t"
+               "       .align 2, 0x90\n\t"
+               "0:     movl 32(%5), %%eax\n\t"
+              "       cmpl $67, %0\n\t"
+               "       jbe 1f\n\t"
+               "       movl 64(%5), %%eax\n\t"
+               "       .align 2, 0x90\n\t"
+              "1:     movl 0(%5), %%eax\n\t"
+                 "       movl 4(%5), %%edx\n\t"
+                 "       movl %%eax, 0(%4)\n\t"
+                "       movl %%edx, 4(%4)\n\t"
+                "       movl 8(%5), %%eax\n\t"
+                "       movl 12(%5),%%edx\n\t"
+                "       movl %%eax, 8(%4)\n\t"
+                "       movl %%edx, 12(%4)\n\t"
+                "       movl 16(%5), %%eax\n\t"
+                "       movl 20(%5), %%edx\n\t"
+                 "       movl %%eax, 16(%4)\n\t"
+                 "       movl %%edx, 20(%4)\n\t"
+                 "       movl 24(%5), %%eax\n\t"
+                 "       movl 28(%5), %%edx\n\t"
+                 "       movl %%eax, 24(%4)\n\t"
+                 "       movl %%edx, 28(%4)\n\t"
+                 "       movl 32(%5), %%eax\n\t"
+                 "       movl 36(%5), %%edx\n\t"
+                 "       movl %%eax, 32(%4)\n\t"
+                 "       movl %%edx, 36(%4)\n\t"
+                 "       movl 40(%5), %%eax\n\t"
+                 "       movl 44(%5), %%edx\n\t"
+                 "       movl %%eax, 40(%4)\n\t"
+                 "       movl %%edx, 44(%4)\n\t"
+                 "       movl 48(%5), %%eax\n\t"
+                 "       movl 52(%5), %%edx\n\t"
+                 "       movl %%eax, 48(%4)\n\t"
+                 "       movl %%edx, 52(%4)\n\t"
+                 "       movl 56(%5), %%eax\n\t"
+                 "       movl 60(%5), %%edx\n\t"
+                 "       movl %%eax, 56(%4)\n\t"
+                 "       movl %%edx, 60(%4)\n\t"
+                 "       addl $-64, %0\n\t"
+                 "       addl $64, %5\n\t"
+                 "       addl $64, %4\n\t"
+                 "       cmpl $63, %0\n\t"
+                 "       ja  0b\n\t"
+                "2:     movl  %0, %%eax\n\t"
+                 "       shrl  $2, %0\n\t"
+                 "       andl  $3, %%eax\n\t"
+                 "       cld\n\t"
+                 "       rep; movsl\n\t"
+                 "       movl %%eax, %0\n\t"
+                "       rep; movsb\n\t"
+                : "=&c"(size), "=&D" (__d0), "=&S" (__d1)
+                :  "0"(size), "1"(to), "2"(from)
+                : "eax", "edx","memory");
 return (to);
 }

diff -Naur linux-417/include/asm-i386/uaccess.h \
           linux-417a/include/asm-i386/uaccess.h
--- linux-417/include/asm-i386/uaccess.h        Tue Jan 22 21:29:43 2002
+++ linux-417a/include/asm-i386/uaccess.h       Tue Jan 22 21:58:11 2002
@@ -256,50 +256,186 @@
```

```
 do {                                                                   \
        int __d0, __d1;                                                 \
        __asm__ __volatile__(                                           \
-               "0:     rep; movsl\n"                                   \
-               "       movl %3,%0\n"                                   \
-               "1:     rep; movsb\n"                                   \
-               "2:\n"                                                  \
-               ".section .fixup,\"ax\"\n"                              \
-               "3:     lea 0(%3,%0,4),%0\n"                            \
-               "       jmp 2b\n"                                       \
-               ".previous\n"                                           \
-               ".section __ex_table,\"a\"\n"                           \
-               "       .align 4\n"                                     \
-               "       .long 0b,3b\n"                                  \
-               "       .long 1b,2b\n"                                  \
-               ".previous"                                             \
-               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
-               : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)      \
-               : "memory");                                            \
+                "       cmpl $63, %0\n"                               \
+                "       jbe  5f\n"                                    \
+                "       .align 2,0x90\n"                              \
+                "0:     movl 32(%4), %%eax\n"                         \
+               "       cmpl $67, %0\n"                               \
+                "       jbe 1f\n"                                     \
+                "       movl 64(%4), %%eax\n"                         \
+                "       .align 2,0x90\n"                              \
+               "1:     movl 0(%4), %%eax\n"                          \
+                "       movl 4(%4), %%edx\n"                          \
+                "2:     movl %%eax, 0(%3)\n"                          \
+               "21:    movl %%edx, 4(%3)\n"                          \
+               "       movl 8(%4), %%eax\n"                          \
+               "       movl 12(%4),%%edx\n"                          \
+               "3:     movl %%eax, 8(%3)\n"                          \
+               "31:    movl %%edx, 12(%3)\n"                         \
+               "       movl 16(%4), %%eax\n"                         \
+               "       movl 20(%4), %%edx\n"                         \
+                "4:     movl %%eax, 16(%3)\n"                         \
+                "41:    movl %%edx, 20(%3)\n"                         \
+                "       movl 24(%4), %%eax\n"                         \
+                "       movl 28(%4), %%edx\n"                         \
+                "10:    movl %%eax, 24(%3)\n"                         \
+                "51:    movl %%edx, 28(%3)\n"                         \
+                "       movl 32(%4), %%eax\n"                         \
+                "       movl 36(%4), %%edx\n"                         \
+                "11:    movl %%eax, 32(%3)\n"                         \
+                "61:    movl %%edx, 36(%3)\n"                         \
+                "       movl 40(%4), %%eax\n"                         \
+                "       movl 44(%4), %%edx\n"                         \
+                "12:    movl %%eax, 40(%3)\n"                         \
+                "71:    movl %%edx, 44(%3)\n"                         \
+                "       movl 48(%4), %%eax\n"                         \
+                "       movl 52(%4), %%edx\n"                         \
+                "13:    movl %%eax, 48(%3)\n"                         \
+                "81:    movl %%edx, 52(%3)\n"                          \
+                "       movl 56(%4), %%eax\n"                          \
+                "       movl 60(%4), %%edx\n"                          \
+                "14:    movl %%eax, 56(%3)\n"                          \
+                "91:    movl %%edx, 60(%3)\n"                          \
+                "       addl $-64, %0\n"                               \
+                "       addl $64, %4\n"                                \
+                "       addl $64, %3\n"                                \
+                "       cmpl $63, %0\n"                                \
+                "       ja  0b\n"                                      \
+               "5:     movl  %0, %%eax\n"                              \
+                "       shrl  $2, %0\n"                                \
+                "       andl  $3, %%eax\n"                             \
+                "       cld\n"                                         \
+                "6:     rep; movsl\n"                                  \
+                "       movl %%eax, %0\n"                              \
+               "7:     rep; movsb\n"                                   \
+               "8:\n"                                                  \
+               ".section .fixup,\"ax\"\n"                              \
+               "9:     lea 0(%%eax,%0,4),%0\n"                         \
+               "       jmp 8b\n"                                       \
+                "15:    movl %6, %0\n"                                 \
+                "       jmp 8b\n"                                      \
+               ".previous\n"                                           \
+               ".section __ex_table,\"a\"\n"                           \
+               "       .align 4\n"                                     \
+               "       .long 2b,15b\n"                                 \
+               "       .long 21b,15b\n"                                \
+               "       .long 3b,15b\n"                                 \
+               "       .long 31b,15b\n"                                \
+               "       .long 4b,15b\n"                                 \
+               "       .long 41b,15b\n"                                \
+               "       .long 10b,15b\n"                                \
+               "       .long 51b,15b\n"                                \
+               "       .long 11b,15b\n"                                \
+               "       .long 61b,15b\n"                                \
+               "       .long 12b,15b\n"                                \
+               "       .long 71b,15b\n"                                \
+               "       .long 13b,15b\n"                                \
+               "       .long 81b,15b\n"                                \
+               "       .long 14b,15b\n"                                \
+               "       .long 91b,15b\n"                                \
+               "       .long 6b,9b\n"                                  \
+                "       .long 7b,8b\n"                                 \
+               ".previous"                                             \
+               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
+               :  "1"(to), "2"(from), "0"(size),"i"(-EFAULT)           \
+               : "eax", "edx", "memory");                              \
 } while (0)

 #define __copy_user_zeroing(to,from,size)                               \
 do {                                                                    \
        int __d0, __d1;                                                  \
        __asm__ __volatile__(                                            \
-               "0:     rep; movsl\n"                                   \
-               "       movl %3,%0\n"                                   \
-               "1:     rep; movsb\n"                                   \
-               "2:\n"                                                  \
-               ".section .fixup,\"ax\"\n"                              \
-               "3:     lea 0(%3,%0,4),%0\n"                            \
-               "4:     pushl %0\n"                                     \
-               "       pushl %%eax\n"                                  \
-               "       xorl %%eax,%%eax\n"                             \
-               "       rep; stosb\n"                                   \
-               "       popl %%eax\n"                                   \
-               "       popl %0\n"                                      \
-               "       jmp 2b\n"                                       \
-               ".previous\n"                                           \
-               ".section __ex_table,\"a\"\n"                           \
-               "       .align 4\n"                                     \
-               "       .long 0b,3b\n"                                  \
-               "       .long 1b,4b\n"                                  \
-               ".previous"                                             \
-               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)               \
-               : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)      \
-               : "memory");                                            \
+                "       cmpl $63, %0\n"                               \
+                "       jbe  5f\n"                                    \
+                "       .align 2,0x90\n"                              \
+                "0:     movl 32(%4), %%eax\n"                         \
+               "       cmpl $67, %0\n"                               \
+                "       jbe 2f\n"                                     \
+                "1:     movl 64(%4), %%eax\n"                         \
+                "       .align 2,0x90\n"                              \
+               "2:     movl 0(%4), %%eax\n"                          \
+                "21:    movl 4(%4), %%edx\n"                          \
+                "       movl %%eax, 0(%3)\n"                          \
+               "       movl %%edx, 4(%3)\n"                          \
+               "3:     movl 8(%4), %%eax\n"                          \
+               "31:    movl 12(%4),%%edx\n"                          \
+               "       movl %%eax, 8(%3)\n"                          \
+               "       movl %%edx, 12(%3)\n"                         \
+               "4:     movl 16(%4), %%eax\n"                         \
+               "41:    movl 20(%4), %%edx\n"                         \
+                "       movl %%eax, 16(%3)\n"                         \
+                "       movl %%edx, 20(%3)\n"                         \
+                "10:    movl 24(%4), %%eax\n"                         \
+                "51:    movl 28(%4), %%edx\n"                         \
+                "       movl %%eax, 24(%3)\n"                         \
+                "       movl %%edx, 28(%3)\n"                         \
+                "11:    movl 32(%4), %%eax\n"                         \
+                "61:    movl 36(%4), %%edx\n"                         \
+                "       movl %%eax, 32(%3)\n"                         \
+                "       movl %%edx, 36(%3)\n"                         \
+                "12:    movl 40(%4), %%eax\n"                         \
+                "71:    movl 44(%4), %%edx\n"                         \
+                "       movl %%eax, 40(%3)\n"                         \
+                "       movl %%edx, 44(%3)\n"                         \
+                "13:    movl 48(%4), %%eax\n"                          \
+                "81:    movl 52(%4), %%edx\n"                          \
+                "       movl %%eax, 48(%3)\n"                          \
+                "       movl %%edx, 52(%3)\n"                          \
+                "14:    movl 56(%4), %%eax\n"                          \
+                "91:    movl 60(%4), %%edx\n"                          \
+                "       movl %%eax, 56(%3)\n"                          \
+                "       movl %%edx, 60(%3)\n"                          \
+                "       addl $-64, %0\n"                               \
+                "       addl $64, %4\n"                                \
+                "       addl $64, %3\n"                                \
+                "       cmpl $63, %0\n"                                \
+                "       ja  0b\n"                                      \
+               "5:     movl  %0, %%eax\n"                             \
+                "       shrl  $2, %0\n"                                \
+                "       andl $3, %%eax\n"                              \
+                "       cld\n"                                         \
+                "6:     rep; movsl\n"                                  \
+                "       movl %%eax,%0\n"                               \
+               "7:     rep; movsb\n"                                  \
+               "8:\n"                                                 \
+               ".section .fixup,\"ax\"\n"                             \
+               "9:     lea 0(%%eax,%0,4),%0\n"                        \
+               "16:    pushl %0\n"                                    \
+               "       pushl %%eax\n"                                 \
+               "       xorl %%eax,%%eax\n"                            \
+               "       rep; stosb\n"                                  \
+               "       popl %%eax\n"                                  \
+               "       popl %0\n"                                     \
+               "       jmp 8b\n"                                      \
+                "15:    movl %6, %0\n"                                 \
+                "       jmp 8b\n"                                      \
+               ".previous\n"                                          \
+               ".section __ex_table,\"a\"\n"                          \
+               "       .align 4\n"                                    \
+               "       .long 0b,16b\n"                                \
+               "       .long 1b,16b\n"                                \
+               "       .long 2b,16b\n"                                \
+               "       .long 21b,16b\n"                               \
+               "       .long 3b,16b\n"                                \
+               "       .long 31b,16b\n"                               \
+               "       .long 4b,16b\n"                                \
+               "       .long 41b,16b\n"                               \
+               "       .long 10b,16b\n"                               \
+               "       .long 51b,16b\n"                               \
+               "       .long 11b,16b\n"                               \
+               "       .long 61b,16b\n"                               \
+               "       .long 12b,16b\n"                               \
+               "       .long 71b,16b\n"                               \
+               "       .long 13b,16b\n"                               \
+               "       .long 81b,16b\n"                               \
+               "       .long 14b,16b\n"                               \
+               "       .long 91b,16b\n"                               \
+               "       .long 6b,9b\n"                                 \
+                " 	.long 7b,16b\n"                              \
+               ".previous"                                      \
+               : "=&c"(size), "=&D" (__d0), "=&S" (__d1)         \
+               :  "1"(to), "2"(from), "0"(size),"i"(-EFAULT)     \
+               : "eax", "edx", "memory");                        \
 } while (0)

 /* We let the __ versions of copy_from/to_user inline, because they're often
@@ -577,24 +713,16 @@
 }

 #define copy_to_user(to,from,n)                                 \
-       (__builtin_constant_p(n) ?                      \
-        __constant_copy_to_user((to),(from),(n)) :     \
-        __generic_copy_to_user((to),(from),(n)))
+        __generic_copy_to_user((to),(from),(n))

 #define copy_from_user(to,from,n)                       \
-       (__builtin_constant_p(n) ?                      \
-        __constant_copy_from_user((to),(from),(n)) :   \
-        __generic_copy_from_user((to),(from),(n)))
+        __generic_copy_from_user((to),(from),(n))

 #define __copy_to_user(to,from,n)                       \
-       (__builtin_constant_p(n) ?                      \
-        __constant_copy_to_user_nocheck((to),(from),(n)) :     \
-        __generic_copy_to_user_nocheck((to),(from),(n)))
+        __generic_copy_to_user_nocheck((to),(from),(n))

 #define __copy_from_user(to,from,n)                     \
-       (__builtin_constant_p(n) ?                      \
-        __constant_copy_from_user_nocheck((to),(from),(n)) :   \
-        __generic_copy_from_user_nocheck((to),(from),(n)))
+        __generic_copy_from_user_nocheck((to),(from),(n))

 long strncpy_from_user(char *dst, const char *src, long count);
 long __strncpy_from_user(char *dst, const char *src, long count);
```

Mobile Cluster Computing Using IPv6

Abdul Basit
Chin-Chih Chang
Wichita State University
Dept. of Computer Science
Wichita, KS 67260-0083, USA
{axbasit, chang}@cs.twsu.edu   http://wireless.cs.twsu.edu

Abstract

Clusters play a major role in scientific computing. They eliminate the need of supercomputers. Communication protocols for cluster computing define efficiency and performance measures for overall system design.

Using IPv6 as a protocol for cluster computing gives benefits such as

- Network load balancing can make use of IPv6 Congestion/Non-Congestion traffic mechanisms (e.g. for handling real-time data requests on clusters).
- Geographically distributed cluster system can make use of embedded IPv6 route optimizations.
- IP Anycast service has the ability to choose the topologically closest server available for handling the request. So it can be used to effectively load balance the network traffic.
- IPv6 Authentication Headers (AH) can be used to define what workstations are allowed to join the cluster.

Moreover, Mobile IPv6 [5] allows transparent geographic mobility without affecting the present connections. This behaviour may be useful in building a clustered environment consisting of mobile agents in which a mobile device (cellphones, PDA's, etc.) submits a computation request to be performed on some local cluster accessible by the Internet. The uses for mobile cluster computing (MCC) can be determining a person's geographical location, mobile business operations (shopping via a cellphone), handling a distributed robot by some mobile device, observing weather information by means of mobile nodes and sending the data to the cluster for future weather prediction (like predicting tornadoes), etc. Issues like timeliness could be better solved by using Mobile IPv6.

This paper covers IPv6, its extension header mechanism, QoS, security, existing network transition, use of IPv6 for cluster computing and mobile cluster computing using IPv6 and its possible *nix [1] implementations.

1 Introduction

High performance distributed computing is always an interesting topic for researchers. During the past decade personal computers have become increasingly powerful, and the communication bandwidth between them is in-

[1]The next generation UNIX

creasing day by day so researchers made a collection of interconnected computers working together as a single system formerly known as 'cluster.' A cluster provides similar (or sometimes better) performance and fault tolerance as the traditional mainframes or supercomputers.

The Internet is growing rapidly since almost 7 years, its continuous growth introduced many problems like IP address space shortage, IP mobility etc. A new protocol named 'IPng – IP next generation' was introduced in 1994 to solve problems in the existing Internet infrastructure. IPng is termed as 'IPv6 – IP version 6' in 1998 [3], IPv6 is expected to replace current IPv4 protocol in year 2005. IPv6 offers many new features like extension header mechanism, IP mobility, IP Anycast, IP route optimization, etc. It provides a very large IP address space up to 340,282,366,920,938,463,374,607,431,770,000,000 IP addresses [4].

Many research-oriented IPv6 backbones like 6BONE, 6TAP, and 6REN are fully operational. IPv6 protocol is designed in such a way that existing networks running IPv4 can natively migrate to IPv6 or they can use IPv6 without fully migrating to it. IPv6 simplifies the header format resulting in less bandwidth usage. IPv6 has embedded support for mobility, priority-based data handling (e.g. video streaming), and security.

According to our definition, mobile cluster computing (MCC) refers to a new paradigm, in which mobile clusters or traditional clusters work together with a set of mobile nodes to carry out a specific task. Mobile cluster computing refers to an environment that offers flexibility in terms of mobility and extendibility, cluster security and a robust mobile network for handling geographically independent high performance computing requests.

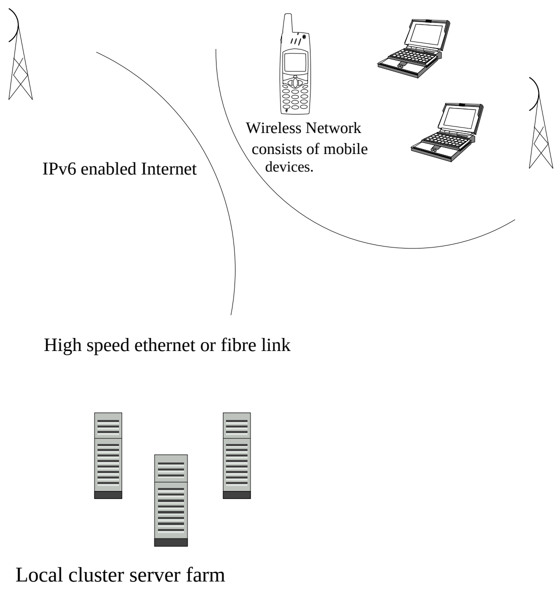

Wireless Network consists of mobile devices.
IPv6 enabled Internet
High speed ethernet or fibre link
Local cluster server farm

Figure 1: Mobile cluster network

IETF standardized mobile IP to handle Internet mobility. However, there exist two standards for mobile IP, they are referred as 'Mobile IPv4' and 'Mobile IPv6.' The Mobile IPv6 protocol offers many benefits over Mobile IPv4 protocol like providing 'Dynamic home agent discovery' mechanism, IP Anycast service, route optimization, etc [7, 6]. In this paper, we use Mobile IPv6 as a working protocol in our definition of 'mobile cluster computing.'

A mobile network typically consists of desktop PC's, wireless devices (such as cellphones, etc.), PDA's, server farms, cluster applications, etc. as shown in Figure 1.

1.1 Mobile and local clusters

A *mobile cluster* refers to a set of interconnected mobile nodes by means of wireless network while a *local cluster* refers to a set of interconnected workstations by means of high speed Ethernet or fibre link. Any cluster node is allowed to migrate from a mobile cluster to a local cluster or from a local cluster to a mobile cluster if this migration is allowed by a *S-node* or *Switching node*. A S-node is any node in a cluster that handles the operation of cluster transition.

Multiple S-nodes can exist either in local or mobile clusters. S-nodes maintain a security map. This security map defines how many nodes are allowed to join a particular cluster at some time. The security map can either be static or dynamic. Static security map is defined initially by the local cluster administrator or by the default policy. The nodes in the security map will grow dynamically if some node can be authenticated by a S-node properly. A S-node performs the network level authentication. A S-node can make use of the AH header of IPv6 packet for authenticating cluster nodes. S-nodes exchange their security map periodically. An IP Authentication Header is shown in Figure 2.

1.2 S-Node cluster authentication mechanism

The *IP Security* (*IPSec*) is designed to provide high quality cryptography-based security for IPv4 and IPv6. The IPv6 protocol has IPSec embedded in it and provides a separate feature different from IPv4. The objectives for using IPSec are to provide integrity, repudiation, confidentiality, encryption, etc. for IP and upper-level layers. Use of IPSec will minimize the need of security for different applications, such as ssh. IPSec provides two traffic security protocols: AH and Encryption Security Payload (ESP). IPSec provides security services at the IP layer. It can select required security protocols, determine the algorithms used for the service, and choose cryptographic keys.

A S-node will act as a 'security gateway' for some particular cluster. A security gateway refers to an intermediate system that implements IPSec. S-node can use AH to provide authentication of the sender of data.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Next Header   |  Payload Len  |          RESERVED             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Security Parameters Index (SPI)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Sequence Number Field                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                Authentication Data (variable)                 |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: IP Authentication Header

For example, a node, A, wants to join a cluster, C_1. Both the node A and the S-node for C_1 speak IPv6. Whenever the node A wants to be authenticated by C_1, it creates an IP extension header along with IPv6 packet and sends to the S-node. Upon receiving the packet that contains an IP AH header, the S-node determines the appropriate Security Association (SA) based on the destination IP address, security protocol in AH and the Security Parameters Index (SPI). A Security Association is a security object shared between two nodes, which contains the data mutually agreed upon for operation on the designated cryptographic algorithm (typically including a key).

The S-node will fetch the source IP address and match it in the specified security map. If this IP matches, the S-node will check authentication data, perform security checking and appropriately grant or deny a ticket to a cluster node according to the designated algorithm. Once the node gets a ticket, it can join that particular cluster. Each ticket has its own pre-defined lifetime.

2 S-node selection algorithm

The question arises that how to select an appropriate S-node when a cluster fires up and stars working. To solve this problem, we propose the following election mechanism.

1. The local cluster administrator specify a password/key in the security map if the default security policy is not employed.

2. Whenever a cluster is made operational, every node in this cluster computes a random number using the password in the security map and multicasts this random number to all nodes in this cluster.

3. Every node will receive the multicast from other nodes within $2n$ time where n is a random time to wait.

4. Every node compares each random number it receive with the local random number. If no random number from other nodes is larger than the local one, then this node will multicast a packet telling other nodes that it is selected as the S-node for a designated cluster.

5. The elected S-node can further delegate access control handling functions to other nodes. The receiving nodes will be synchronized with the security map of the primary S-node. IP AH headers should be used to detect spoofed packets.

3 Dynamic clustering based on type of network traffic

Dynamic clustering refers to the fact that a cluster can distinguish between different types of network traffic. This behaviour is useful in performing network-based load balancing for different types of network data, such as performing network level load balancing based on traffic class.

3.1 Types of traffic

Traffic class is specified in the IPv6 header to identify and distinguish between different classes or priorities of IPv6 packets. In the original specification IPv6 splits traffic into two categories [2]:

1. *Congestion controlled traffic* may be arbitrarily delayed under conditions of congestion.

2. *Non-congestion controlled traffic* would be discarded under conditions of congestion

In our design we keep the notation defined in the original specification until the new specification has been finalized. In that document congestion controlled traffic can be further categorized as shown in the following table where larger priority value indicates higher priority:

Pri.	Description	Example
0	Uncharacterised traffic	Custom use
1	Filler traffic	netnews
2	Unattended data traffic	email
3	Reserved	
4	Attended data traffic	FTP, NFS
5	Reserved	
6	Interactive traffic	telnet, X
7	Internet control traffic	routing, ICMP

Table 1: Congestion-controlled Traffic

Non-congestion controlled traffic is the type of traffic for which constant data rate and constant delay is required such as real-time audio and video transmission. The priority values 8 through 15 are used to specify this type of traffic.

3.2 Using flow labels to perform priority level traffic handling

Priority level traffic handling in IPv6 is based on flow label mechanism where flow refers to a sequence of packets transmitted by some

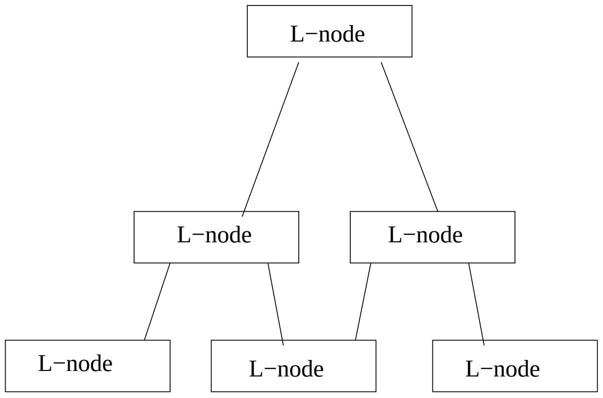


Figure 3: Hierarchal view of load balancing nodes

transmission protocol from source to destination with some QoS requirements. It can enable a cluster to either handle different data on different clusters or be optimized for a specific type of data such as video streaming [8, 1]. Using flow labels, we can route different network traffic through a different route. Previously in IPv4, TOS (Type of service) field offers little control over network traffic.

IPv6 specifications state that every IPv6 header contains a 24-bit flow label field and a 4-bit traffic class field. A flow label is basically a unique identifier between 1 and FFFFFF when it is combined with the source IP address. Flow label 0 indicates that no flow label exists and this packet doesn't need any special treatment. Real-time network flow always has some flow label.

A router associates a flow with some state. If the router doesn't contain any state associated with flow, it may either drop the packet or forward it setting flow label to zero (0). All packets pertaining to the same flow must be originated from same source address, same destination address and same flow label.

A *L-node* refers to a *load balance node* or *load balancer* for a particular cluster. L-node will be responsible for flow control handling. There can be one or more L-nodes for a particular cluster. L-nodes can form a hierarchy to balance load from a set of cluster nodes as shown in Figure 3.

3.3 Communication mechanism between L-nodes

Every L-node upon receipt of some packet will forward the packet (workload) to its corresponding L-node or balance the traffic load if it is connected to some network (that is presence of a single L-node for balancing). L-node will balance the traffic either among different L-nodes or among different cluster-nodes according to following schemes:

- FCFS (First-Come First-Served) – The request that comes first is allocated to L-node first.
- RR (Round Robbin) – Each request gets a quantum time to run.
- Priority based – Service is based on their priority. The one with higher priority gets served first.
- IP Anycast – The L-node uses IP Anycast IPv6 service to distribute network requests to any nearby available node or cluster.

Possible failures can be minimized by suggesting that the network requests don't need to come through the top-level L-node, the network request can hit any L-node in L-node hierarchy and the request will be processed further down in hierarchy. So in the case if top level L-node fails, this will not cause the whole network to be down. In other case, if a child of a L-node fails, the request coming through parent L-node will be timed out. This won't affect a computing too much. The request can be queued when the parent detects a failure node. When the parent detect a timeout, the parent either carry that computation or forward to other L-nodes.

4 Benefits of IPv6 for roaming cluster nodes

A *roaming cluster node* usually refers to a mobile device participating in a cluster. A local node can become a mobile node at any time, but that should not affect local network connections. The transition of a local node to a mobile node should be transparent for cluster operations. They should keep processing without any interruption. One of the facts is that in IPv4 we don't have much global IP addresses left to assign to each mobile node. However, this problem is solved in IPv6 because we can uniquely identify the mobile device with at least one global IP from a very great IP address space provided by IPv6. Moreover, if we transit a local node to a mobile node in IPv4, that will definitely cause a service interruption because possible IP address change occurs when moving from one network subnet to another in IPv4. On the contrary, in IPv6, Mobile IP offers a way so that this node transition won't interrupt the cluster operation. Figure 4 shows basic communication between a mobile node and a cluster.

4.1 Mobile cluster

A mobile cluster is introduced as a set of mobile nodes that participate in a cluster computing. A mobile node can migrate to a local cluster and a local node can become mobile at any time.

We have to consider if we purely use mobile clusters. For example, one PDA sends request to another PDA for performing some task. How can IPv6 be useful in this scenario? Our primitive observation is that due to simplified header mechanism in IPv6 (simplified headers mean that some fields in header are made optional or eliminated) it gives performance increase over IPv4. Since wireless media has a low bandwidth, this saves bandwidth. So IPv6 is more feasible than IPv4 for mobile only clustering.

4.2 Local to mobile cluster node migration mechanism

- A local node detects that it has moved by discovering a new default router.
- A local node becomes mobile node now, it performs a stateless or stateful address auto-configuration mechanism to obtain a new COA (Care of Address) for the new location. All packets destined to this COA will redirect to the mobile node through the current link. If the current link also serves some other cluster, then this node can decide to participate in both clusters (old and newly joined cluster).
- The mobile node then performs a binding update with the home agent of the old cluster.
- The home agent of the old cluster registers this binding and then sends back the binding acknowledgement.
- The home agent of the old cluster will now intercept the packets for migration by using 'proxy neighborhood discovery' protocol. *Proxy neighborhood discovery* means that the home agent multicasts a neighborhood advertisement onto the home link (cluster link) on behalf of the mobile node. The home agent itself also replies to neighbor solicitation on behalf of the mobile node. Each intercepted packet is then tunneled to the new COA address of the migrated node using IPv6 encapsulation.

- This triangular routing can be avoided by means of ‘*route optimization*’ in which a mobile/migrated node can send binding update to any correspondent node. So the correspondent node can later send packet directly to the new COA of migrated node instead to the cluster home agent. But in this way the cluster/newly migrated node can not participate in two clusters at one time. If the mobile node sends packet to any other node, it sends packet directly to its destination (not to home agent), it sets the source address of this packet to the COA, it also includes ‘home address’ destination option.

4.3 Stateless vs. stateful address auto-configuration for mobile nodes

In stateless address auto-configuration, no central server is configured for a host to obtain its interface address (COA in our case) and or configuration information such as DNS servers, gateways, etc. The stateless mechanism allows a host to generate its own address using a combination of locally available information advertised by routers. Routers advertise prefixes (like /64) that identify the subnet(s) associated with this link, whereas the host generates an identifier that uniquely identifies the host on some subnet (usually it uses MAC 48-bit address for first 48-bits). The final address is computed by combining the two. In the absence of router advertisements, the host will only generate link local address. With only link-local address [2], the host can only communicates with other hosts on the same link.

In stateful address auto-configuration, a host obtains interface address and or configuration parameters by contacting a central server (e.g. DHCPv6 server). The server maintains a database that keeps track of assigned COA (IP address).

[2]A link-local address refers to a non-routable address that is unique and valid only on the local network. The link-local address has a prefix of fe80::/64 and it is used for contacting hosts and routers on the same network only. The addresses are not visible or reachable from different subnets.

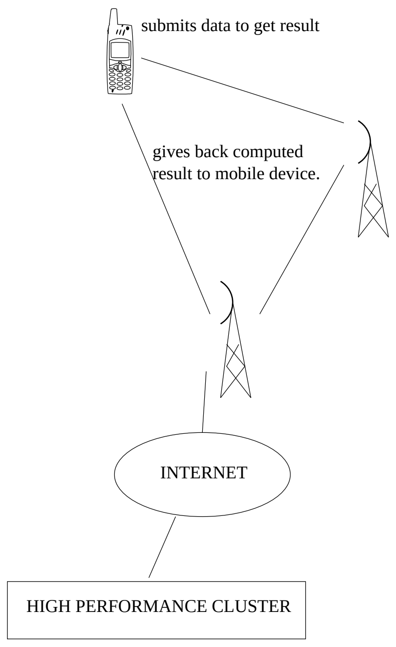

submits data to get result
gives back computed result to mobile device.
INTERNET
HIGH PERFORMANCE CLUSTER

Figure 4: Basic communication between a mobile node and a cluster

4.4 Availability of a computing node

Wireless communication is thought to be an unreliable one because of occasional omission and arbitrary failures caused by the intrinsic property of non-uniform wireless signal. IPv6 could alleviate this situation. For example, if a computing node is beyond the range of the cluster, this node is thought to leave the cluster in IPv4. But in IPv6 the node outside of the range of the cluster can be detected if it is adopted by the remote network via Mobile IP. This could guarantee the computing capability.

We can use the following scheme to seamlessly achieve a cluster computing task in respect to leaving and joining nodes:

1. If a node leaves the current cluster, the task assigned to it will be queued in the requesting agent.

2. The requesting agent continue to carry out its computation in the cluster and wait for some pre-defined time for the leaving node to join the cluster through the Foreign Agent (FA).

3. If the leaving node is detected within the pre-defined time, the queued task will be carried out by the rejoining node. Otherwise, the task will be carried by other available nodes.

This scheme won't be so feasible in IPv4.

4.5 Timeliness Issues

In the previous section we discuss availability of mobile nodes. The situation will become complex when time constraint is imposed. Timeliness issues happen when mobile nodes within a computing cluster migrate from one subnet to another subnet in a wireless network and when time constraint is imposed [9]. To meet the QoS requirement timeliness issue needs to be considered.

In [9] it is proposed that the route optimization is delayed to a certain period and the message from the home network to a new foreign network is tunneled. Several approaches also presented in that paper. In IPv4, this problem is difficult to solve. In IPv6, we can combine their approaches and the scheme we propose for availability of a node to alleviate timeliness issue.

4.6 Mobile process migration

A *mobile process* refers to a process running in either a local or mobile node in some cluster. A mobile process differs from a local process in a way that it can either fully or partially migrate from a node to another or from a node to multiple nodes.

- *Full process migration* refers to the fact that a process running on some node detaches itself from that node and injects itself in some other node.

- *Partial process migration* refers to the fact that a part of process migrates itself to some other node.

- Partial and full process migration can be bi-directional. That is from a mobile node to a local node or vice versa.

- In the case of full process migration, some level of transparency should be addressed so that the local process should not possess dangling references to the mobile process. The full migration of a mobile process may cause possible memory leaks because the process is not running inside the same machine. Hence, possible memory references should be updated. One of the proposed solution is to employ a *MM-node* (*Memory-manager node*). The purpose of this MM-node is to provide a *V-map* (*virtual map*) to map physical memory to distributed memory. Each mobile process takes references to some memory location provided by means of V-map.

- In the case of partial process migration, only a part of process or a thread of a process migrates. The parent process maintains the state for each migrated thread. If a migrated thread further spawns some thread, then this thread can not be migrated.

- Several threads of a mobile process can migrate to different nodes.

- One mobile process can only fully migrate to a node at a time.

- *P-node* (*Processing node*) refers to a node whose sole purpose is to handle incoming and outgoing processes for a particular cluster. A mobile process submits itself to P-node, and P-node will migrate this process to some other local or mobile cluster. Depending on system usage, P-node can either queue, suspend, or stop migration. P-node will migrate the mobile process to P-node of another cluster that will further migrate that process to any node or multiple nodes of its cluster.

- Using IPv6 protocol can protect some node in cluster to pretend as P-node.

- P-node or MM-node in simple words is like a router, that will multicast its address by means of IPv6 packets along with AH extension security headers to prevent spoofed packets.

5 Conclusion

We have discussed some basic issues of mobile cluster computing. We specify some basic concepts which are required in solving these problems. We further propose several solutions based on IPv6. S-nodes are designed to use AH and search the security map to authenticate a node to join a cluster. The flow label mechanism is used to handle priority level traffic. L-node is intended to balance the workload. Then we present how we could benefit from using IPv6. We build a top-down view from the cluster, to the node, and eventually to the process level. Timeliness issues are also discussed.

In this paper we identify problems and outline some design idea. Our future work is to refine our design and put it into implementation.

References

[1] Rahul Banerjee and Sumeshwar Paul Malhotra. A Modified Specification for Use of IPv6 Flow Label for Providing Efficient Quality of Service using Hybrid Approach. *IETF Internet Draft*, February 2002.

[2] S. E. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. *RFC 1883*, December 1995.

[3] S. E. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. *RFC 2460*, December 1998.

[4] R. Hinden and S. E. Deering. IP Version 6 Addressing Architecture. *RFC 2373*, July 1998.

[5] David B. Johnson and Charles Perkins. Mobility Support in IPv6. *IETF Internet Draft*, March 2002.

[6] Charles Perkins and David B. Johnson. Route Optimization in Mobile IP. *Cluster Computing*, 1(2):161–176, 1998.

[7] Charles Perkins and David B. Johnson. Route Optimization in Mobile IP. *IETF Internet Draft*, September 2001.

[8] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering. IPv6 Flow Label Specification. *IETF Internet Draft*, March 2002.

[9] Haihong Zheng, Rajkumar Buyya, and Sourav Bhattacharya. Mobile Cluster Computing and Timeliness Issues. *Informatica*, 23(1):5–17, 1999.

Incrementally Improving the Linux SCSI Subsystem

James E.J. Bottomley
SteelEye Technology, Inc.
http://www.steeleye.com
James.Bottomley@steeleye.com

Abstract

This paper tackles two issues in the current SCSI subsystem: init time probing using the new hotplug infrastructure and improvements to the current error handler. We also include an appendix sketching the operation of the SCSI subsystem and another one listing other outstanding problems not covered in this paper.

1 Introduction

Obviously, the scope of potential incremental improvements to the SCSI subsystem is enormous. In order to narrow the field quite a bit, we will concentrate on just two particular examples.

1.1 Device Scanning and Inquiry

The first addresses the current weaknesses in the device probing and inquiry code. As things stand today, the SCSI subsystem will scan all targets (up to 15) and, depending on compile and run time variables, try to scan all LUNs on those targets. There is also a compiled in exception table stored in `scsi_scan.c` which can cope with the idiosyncrasies of certain devices. The principle disadvantages of this system are

1. It is extremely inflexible and rigid. New devices that need exceptions have to be patched into the kernel table and the kernel recompiled before it does the correct thing, and
2. The exception table is cumbersome and does not cover all cases. For example, how certain devices are probed can depend on the SCSI host adaptor they are attached to (mercifully, this is becoming extremely rare).

It is the thesis of this paper that such complex rules based logic should be abstracted entirely from the kernel and placed in user land, where it can easily be altered and extended without even rebooting; and furthermore, that such a system can be grafted on to the existing code fairly easily.

1.2 Fixing the Error Handler

This topic is also broad, particularly as there are several bio related errors in the 2.5 series kernel that make error handling especially problematic (see appendix B). However, the object here is to concentrate on the SCSI specific region of code in `scsi_error.c` and assume that the kinks in the bio system will be worked out as the kernel evolves.

The essential problem in `scsi_unjam_host()` is that it tends to execute on one command at once, and have limited facilities for understanding that error recovery on one command may affect a large numbers of others. This is particularly acute when drives start misbehaving because they usually have the maximum number of commands queued when error recovery begins. Our presentation will be essentially to clean up the error handler actions and to restart command execution slowly and gently (throttling) instead of slamming an entire set of outstanding commands down again.

1.3 A Historical Context

Years ago, when the first monolithic UNIX kernels were emerging into the light of day it was recognised that you could draw a neat line around most of the code used to boot the system and configure its devices. Further, that this code was never used again in the entirety of the operation of the kernel.

Back in the mists of time, Linux began to separate this initialisation code into a different compiler section and release it after the system had completed booting. This worked well for a while but as modular drivers came along, less of the core kernel code which was used by the boot process could be discarded because it might be used by a module to initialise its devices.

Much later, the concept of hotplugging devices came along, and the Linux hotplug [1] project was started.

2 Hotplug

The essence of the hotplug system and its utility has been described elsewhere [2]. Hotplug is primarily intended for computers whose configurations change on the fly, obvious examples of which are the laptop PCMCIA system, a USB daisy chain, firewire and so on. It was recognised fairly early on in this project that the problem of adding a device to a running system is substantially similar to that of configuring a device at boot up, except that the operating system may not be completely initialised (and thus the hotplug system may not be available) when boot probing is done. For this reason, the boot probe issue is called “coldplugging”.

2.1 Avoiding Coldplug

In general, the coldplug problem is similar to most bootstrap problems. However, there is a fairly neat way to avoid the difficulty for several subsystems (SCSI being among them, fortunately): by using an initial ramdisk. As long as the hotplug system is built into the initial ramdisk, the coldplug bootstrap problem is completely eliminated for any subsystem which can be inserted entirely as modules, since it would be handled as a genuine hotplug event by the initial ramdisk hotplug system.

2.2 Hotplug and Device Scanning

A number of the devices that can be hot plugged are also effectively “bridges”, that is units which make onward connections to other busses which may contain other devices. SCSI Host Bus Adaptor (HBA) cards are a classic example of this, since all they really do is bridge the computer bus (often PCI) to an external or internal SCSI bus.

Whenever any type of bus bridge is added to the system, logic must be invoked to scan the devices beyond the bridge and add them into the system. Usually, all this scanning is performed inside the kernel; however, for this paper we investigate transferring they scanning logic to the user level hotplug system.

2.3 Bridge Insertion Events

In general, hotplug events are designed to allow the system to configure the particular device which has just been inserted and the interaction between the programs executed during the hotplug event processing are designed to perform this configuration. Scanning and configuration of devices beyond the bridge device should obviously not be begun until the bridge device is properly configured and fully functional. It therefore makes sense to fire a separate "bridge scan" hotplug event after bridge configuration and bus sanitisation to trigger probing on the actual bus beyond the bridge.

Following the initiation of scanning beyond the bridge, the job of the scanning routine should be to notify the kernel of the existence of the new devices, but allow the kernel to configure them, or better yet trigger another hotplug event to configure them.

One of the issues in bridge/device configuration that require the bridge to be configured before the scan are the setting or collecting of intrinsic bus properties: things like bus speed, width and configuration, all of which must be known before the devices on the bus may be probed. For instance, the SCSI bus can be configured for various widths (wide or narrow). The width is usually governed by the HBA, but nothing prevents a wide device being connected to a narrow only HBA. Thus, the device configuration is dependent on the parameter rage of the bus, which are controlled by the bridge (the HBA).

2.4 Bridge Configuration

We need a mechanism for making available to both the user and kernel the parameters detected and set during the prior bridge hotplug event. There is an evolving infrastructure in the new driver model[3] that may ultimately be capable of storing this information in a usefully abstracted form. However, for the time being, we opted for a completely opaque bridge programming model so that the bridge hotplug event handler needs exact bridge programming knowledge. Basically, we elected to place the SCSI bus parameters in a SCSI specific field which can be queried by `ioctl`s.

3 Replacing SCSI Scan/Inquiry

This project essentially builds on top of the ideas and code provided by the scsimon [4] project. Although scsimon was designed to provide notifications for device insertions, the code it supplies and the design basics are essentially reusable in the scan/inquiry replacement.

3.1 What `scsi_scan.c` does now

This entire file of code is dedicated to scanning busses and detecting and configuring devices. It is driven entirely from a static table (called `device_list`) which contains inquiry strings matching devices for which special actions are taken. Most of the actions are geared to LUN scanning. Here is an example of some of the flags:

- `BLIST_FORCELUN`: scan for LUNS even if kernel is compiled not to.
- `BLIST_NOLUN`: Never scan LUNS on this device.
- `BLIST_SINGLELUN`: only allow I/O to one LUN at a time.
- `BLIST_NOTQ`: Device claims to support Tag Command Queueing but in reality cannot.
- . . .

Other flags deal with device type misidentification and so forth. All of which can easily be accommodated in user land, with the addition of two extra ioctls: one to set or clear the tag bit (`tagged_supported`) and one to alter the device type.

3.2 Adding the Bridge Insertion Hook

In the SCSI subsystem, the easiest way to add the bus insertion hook is right at the end of `scsi_register_host()`. We eliminate the code in `scan_scsis` except for the hard coded entry used by `add-single-device`.

The Bus insertion hook is now used to begin scanning the targets (at LUN zero) using the `add-single-device` command (the scsi and channel numbers being passed up from the hotplug event).

It is certainly open to debate whether it is worthwhile moving this functionality into user land. However, in principle we could also set up bus transfer parameters or actually opt to use the SCSI-3 report LUNs command instead for the scanning, so it still provides an arguably much more flexible system. This will become particularly important as newer standards are adopted and the process of scanning for devices changes.

3.3 Adding Device Insertion Hooks

Just as the majority of the work is done in `scsis_scan_single()`, so most of the work will be done in the code running after the device insertion hotplug event. The correct place for this is after the initial inquiry command, so that when the hotplug event is called we know the inquiry parameters and can pass them as event parameters.

The internal `device_list` table may now be laced into a flat configuration file (which is thus easily customised) which matches inquiry strings and triggers the appropriate actions. Since we now have more powerful tools at our disposal, the matching can be much more finely controlled using regular expressions. The actions may also be much more dynamic than simply sending parameters down to the kernel: indeed, the system may now be designed to be completely extensible so that we could execute a vendor supplied script, installed in the system, whenever that vendor's device is detected.

There has been recent concern [5] about certain devices not respecting the SCSI standards with regard to inquiry parameter lengths. This could probably be handled either by allowing the maximum inquiry length to be a bus parameter set by the bridge insertion event, or by having the initial inquiry only be the minimum length and allowing the device hotplug event to use the `sg` device to formulate a second inquiry to get all the parameters it needs. The counter argument: that the low level drivers need to snoop the inquiry data to set up their parameters could be avoided by allowing the device insertion hotplug to communicate the relevant parameters to the bridge.

3.4 Device and Bridge Interaction

If we managed to create the correct abstraction of the SCSI bus, there would be no need for special ioctls to be sent to inform the bridge of device bus characteristics, nor would the bridge need to snoop data (like inquiry returns) being passed over the bus to obtain this information.

However, until such an ideal state of affairs is reached, it is possible that the bridge will need to be made aware of extra parameters in the device. For this reason, we permit a "bridge callout" to be done at the end of the device hotplug event script (essentially the hotplug engine checks to see if the bridge wishes to be informed of device insertion events and executes the script provided by the bridge if it does). This should allow for arbitrary setting of bridge/device parameters to suit the bus environment.

In the current implementation, it is the responsibility of the device/bridge script to scan for additional LUNs (if necessary). This leads to the unwanted side effect that `add-single-device` for LUN 0 will now trigger a complete LUN scan, which may not be what was intended. This can be solved by making "have scanned for LUNs" a property of the device and passing it up on the device insertion event.

3.5 Event Flow

The flow of events described above is shown in figure 1 with time moving from left to right in the diagram.

4 Error Handling

The current error handler thread begins when a fatal error is detected (see Appendix A for an operational sketch) it then quiesces the device and proceeds, one command at a time, through its abort and reset sequence. It checks the progress of error recovery at most points by sending down a test unit ready (TUR) command. However, there are common SCSI driver problems that the mid-layer ignores and others that cause it to malfunction.

4.1 Who Owns the Tag Starvation Problem?

When Tag Command Queueing (TCQ) is enabled on a device, we pretty much (although not always) use "unordered" tags. This leaves it entirely within the province of the device firmware to determine command execution order. In theory, the device firmware has a mode page which lays out guarantees about the maximum times it will take for the device to process any given tag. However in practise, most (particularly older) devices govern tag execution by closest head stepping times and thus some I/Os to different parts of the disc surface can find themselves ignored—a phenomenon called tag starvation.

Since the problem can occur on almost every parallel SCSI card, every driver that does TCQ has to be aware of it and evolve a strategy to cope. Therefore, the tag starvation problem is pretty much owned by the low level drivers, which is a pity, since it means code duplication and lots of extra testing.

The two most general ways tag starvation is handled are: sending an ordered tag with the next command, or not sending down any more commands until the starved tag is processed. These are both extremely easy to implement inside the mid-layer and would relieve the low level drivers of a sizable amount of duplicated code.

4.2 What Kinds of Errors Occur?

The SCSI operation model is essentially a giant state machine. There are a wide variety of error conditions which can occur but which are recognised as particular states in the model. The SCSI mid-layer translates these model states into actions in `scsi_decide_disposition()`. However, anything that gets into this code is pretty much part of the state model, since it is invoked using a SCSI return code. Pretty much everything other than an unrecognised return code for a command still in progress will be handled without troubling the error recovery thread.

The point is that the error handler thread is

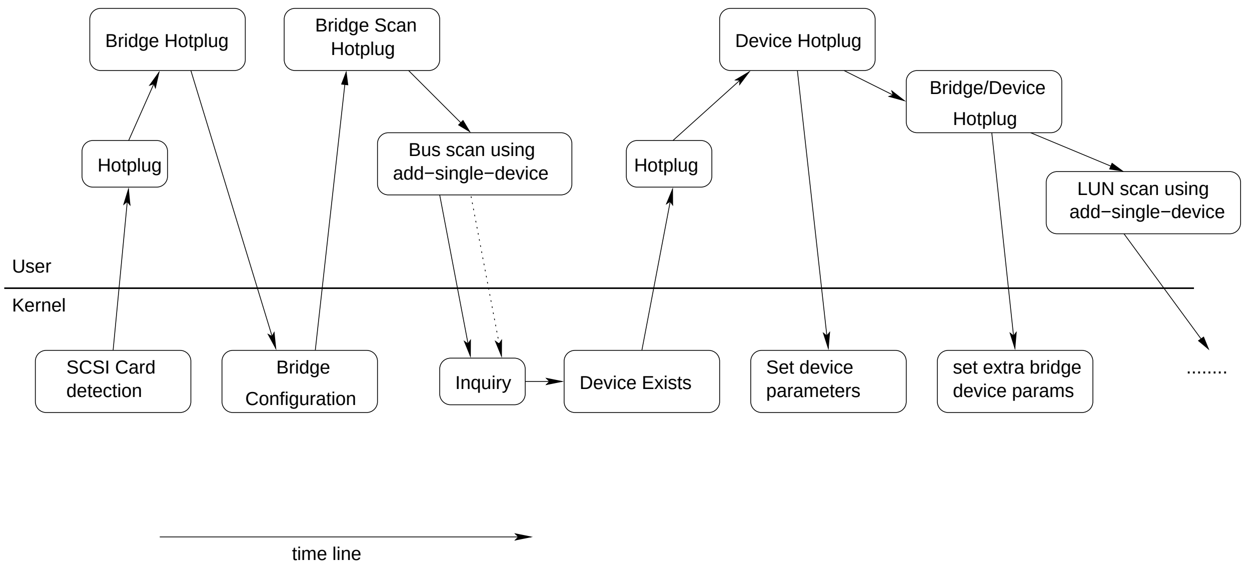


Figure 1: Time line for Hotplug Insertion Events

usually only invoked when the state model has failed[1] or a command has timed out (which is pretty much the same thing, since it means we sent a command to the device and it got lost), so the remedies it applies have usually got to be a drastic kick to get the device back into a state the model recognises and can resume processing from.

4.3 Why Abort?

The first action the error handler thread tries is to issue an abort to the problem command. Abort is a SCSI message that informs the device to discard a particular tag at whatever point it has reached in processing it. It is a command designed to fit inside the SCSI state model and has several uses, particularly for stopping linked commands which have encountered errors. However, its use as the first port of call for the error handler thread trying to recover a device is worse than useless, since it is applying a course of action within the state model to something already outside of it. Following this logic, the abort sequence and its associated driver hook may simply be removed (or at least deprecated) in the SCSI driver model.

4.4 Flavours of Reset

The next courses of actions, if abort failed, will be to begin a series of resets culminating in the complete reset of the HBA. The SCSI protocols actually support three levels of reset: LUN, device and bus. The former is a message addition from SCSI-3 and is only relevant for devices with multiple LUNs. It does everything that a device reset does, but operates only on a single LUN—a device reset operates on all LUNs. A device reset only operates on a single target (but all of its LUNs) and a bus reset operates on every device on a physical bus.

Resets are designedly very intrusive to device operation. A reset basically causes the device to drop everything on the floor and re-initialise itself; it is allowed to spend quite a bit of time (measured in seconds) on this re-initialisation and is entitled to respond “not ready” to any command received during this period.

[1]Here remember that the SCSI state model defined by the standards is much larger than the one *implemented* in the mid-layer.

The error handler, meanwhile, should be aware of the extent of the potential disruption resets cause to the device in question, particularly with regard to losing all outstanding commands. It should pull all commands affected by the reset from both the pending and error queues, cancelling the timers on the pending commands and place them all in abeyance until the error handling completes (it might make sense at this point to push them back into the bio queues so that they can be merged if necessary, but hang on to the command we were initially trying to recover).

After a reset has been sent, it should keep the device quiesced and back off for a while (probably a second) before probing with a TUR—we should also loop in this mode, probing every second or so, until the TUR comes back as not "not ready".

Once we know the device is ready to accept commands again, we should feed the first command from the error thread, wait for it to complete and remove the device quiesce if it returns successfully. We should probably also lower the tag queue depth (if it had one) on the assumption that the error may have been triggered by over feeding.

4.5 Choosing a Reset

Often, the simplest reset for any device driver writer to use is the bus reset. This is because all the other resets are actually phrased as SCSI messages and thus need special processing. The bus reset is activated simply by pulling the reset line on the SCSI bus low for 25ms and is usually triggerable using a simple chip register flip.

In choosing bus reset, one thing to beware of is the SCSI standard soft reset alternative (see section 6.2.2 of the SCSI-2 standard [6]). What this does is allow the device to essentially ignore some of the most useful aspects of the bus reset (i.e. dropping everything and starting over from a clean state). The mid layer picks the flag indicating soft reset alternative out of the inquiry data and sets the `soft_reset` device flag in this case. We have never come across one of these devices, but if we did it would certainly cause huge problems for low level drivers that rely solely on bus resets.

4.6 Device Offlining

The last response of the error handler, if it fails to get the device to accept commands once more is to place it offline. All outstanding commands should be immediately failed with I/O errors. However, the mid layer should continue to accept commands for this device, but should just immediately fail them with I/O errors. This should break out of the unfortunate condition where offlining a device with a huge outstanding bio queue can leave lots of processes stuck in D wait (see appendix B.2).

4.7 Multi-Initiator Scenarios

Previously, multi-initiator (where more than one initiator, or HBA, is connected to the same bus, so multiple machines may be talking to the same devices) was a fairly esoteric configuration primarily limited to clustering environments. With the advent of Fibre Channel, this all changed and shared busses are becoming much more prevalent.

In the classic multi-initator scenario, a reset from another initiator, that you don't see, causes all of your outstanding commands to be lost without trace. This can be particularly nasty in the case where LUN reset is not implemented, because you could be quietly processing exclusively on LUN15 of an array only to be reset because another initator was having issues with LUN3.

About the best way to handle this is to take special action whenever the signature for a reset occurs (which is a check condition followed by unit attention sense on the next command to be sent down to the device). On detecting this condition, we should immediately proceed as though we were the ones resetting the device as part of error recovery: collect all the outstanding commands, cancel the timers probe with a TUR and start feeding them back down again when the device is ready to accept them.

4.8 Testing Error Handler Changes

Once changes are made to error handling, one of the main problems is actually testing them. Most modern SCSI devices really don't actually ever cause the error handler to be invoked. Even transmission line conditions or other problems which cause the SCSI bus to go marginal aren't exactly very useful since there is little chance of correct recovery from them. What is needed is a method for simulating errors in the SCSI subsystem and gauging what happens next.

One particularly useful tool is the debug driver of the Linux SCSI subsystem [7]. Although it currently only comes with the ability to simulate a medium error, persuading it to drop a SCSI command silently (and thus trigger a timeout and error handling) isn't that much of a difficult problem. Once this enhancement is made, it is comparatively easy to trigger a recoverable error an watch how the system behaves.

For those people with access to genuinely malfunctioning devices (my favourite being an old HP C3255 device which seems just to stop working occasionally with high tag queue depths), it is extremely nice to be able to plug them in an watch the system cope (or not, as the case may be).

A A Sketch of How the Current SCSI Subsystem Works

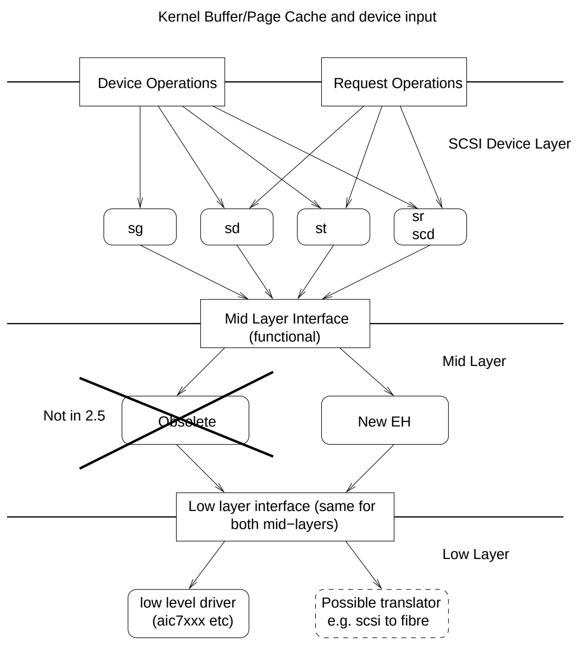

Kernel Buffer/Page Cache and device input
Device Operations
Request Operations
SCSI Device Layer
sg
sd
st
sr
scd
Mid Layer Interface (functional)
Mid Layer
Not in 2.5
Obsolete
New EH
Low layer interface (same for both mid−layers)
Low Layer
low level driver (aic7xxx etc)
Possible translator e.g. scsi to fibre

Figure 2: Block Diagram of the SCSI subsystem

A complete block diagram of the SCSI subsystem is shown in figure 2. The error handler comprises a very small portion of the midlayer (shown as new EH—although for 2.5, this is the only error handler). It's code is entirely in `scsi_error.c` and it is invoked from the bottom half handler routine activated by `scsi_done()`.

A.1 I/O in

All requests come in from the upper layer device drivers through `scsi_request_fn()` which loops over all pending requests. If the device is `in_recovery` or plugged, it returns.

Otherwise, it proceeds as follows:

1. Dequeue the request.

2. Copy the request into a scsi request structure and release the bio request.
3. Call `scsi_init_io()` to set up the scatter/gather list on the request structure.
4. Call the device specific init command to set up the appropriate SCSI command.
5. Initialise the error handler components (mainly zero out sense and set up the timeout).
6. Call `scsi_dispatch_command()` to begin. This sets up the serial number and pid, adds the timer and calls the host `queuecommand()` if it `can_queue` otherwise calls `command()`.

A.2 I/O out

All finished commands come in from the low layer through `scsi_done()` with the queue lock held. They are then added to the bottom half (BH) queue and the BH is notified.

The BH handler (`scsi_bottom_half_handler()`) runs later picking work off the SCSI BH queue until none remains. It calls `scsi_device_disposition()` which returns four possibilities:

- `SUCCESS`: immediately complete the command by calling `scsi_finish_command()`.
- `NEEDS_RETRY`: send the command to `scsi_retry_command()` which will send it immediately down to the lower layer unless the retry count has been exceeded.
- `ADD_TO_MLQUEUE`: call `scsi_mlqueue_insert()` to send the command back to its elevator queue.
- `FAILED`: set `in_recovery`, plug the elevator queue and wake the error handler.

A.3 I/O Error

Once the error handler thread is awoken, it calls the template `eh_strategy_handler()` if it exists, otherwise goes into `scsi_unjam_host`.

`scsi_unjam_host()` loops over all pending commands and looks at their `state` field. Really, it is only interested in the `FAILED` or `TIMEOUT` states.

Essentially, it loops over every failed or timed out command and runs through first abort, then device reset, then bus reset and finally host reset, sending a TUR to test the device if one of these succeeded. If it gets all the way to the end and still has failed commands, it offlines the device.

B Unresolved Issues in the Mid-Layer

This section is really a collection of issues on my todo list, but obviously as I haven't got around to doing them yet, if anyone else wants to step into the crease, they're more than welcome.

B.1 `queuecommand` busy failure

The template hook `queuecommand()` is allowed to return 1 if the command has not been queued. This can be for a variety of reasons, but most commonly because of either tag starvation or static resource exhaustion. What is supposed to happen is that the unqueued command goes back into the bio elevator and is resubmitted at a later time.

It looks like the `scsi_mlqueue_insert()` function or the bio is failing somehow, because on most 2.5.x, as soon as `queuecommand()` returns 1, the buffer hangs forever in D wait.

B.2 Device Offline Failure

After an initial complete failure of the error handler, leading to a device being taken offline, processes trying to use buffers on the device often end hung in D wait. This is indicating that the code which returns I/O errors on all the outstanding I/O requests is missing some. I suspect there may be a problem prizing the rest of the I/O out of the bio, since it seems that the code in the error handler to fail the I/O that has reached the mid-layer is fairly bullet proof.

References

[1] `http://sourceforge.net/projects/linux-hotplug`

[2] Greg Kroah-Hartman *Hotpluggable Devices and the Linux Kernel* Ottawa Linux Symposium 2001

[3] Patrick Mochel *The (New) Linux Kernel Driver Model* `Documentation/driver-model.txt`

[4] Doug Gilbert *Scsimon Driver for Linux* `http://www.torque.net/scsi/scsimon.html`

[5] Martin Wilck *Hack to make Datafab KECF-USB work* `http://marc.theaimsgroup.com/?l=linux-usb-devel&m=101304393027774`

[6] X3T9.2 Project 375D *Information Technology—Small Computer System Interface 2* `ftp://ftp.t10.org/t10/drafts/s2/s2-r10l.pdf`

[7] Originally by Eric Youngdale, but now Maintained and enhanced by Doug Gilbert `http://www.torque.net/sg/sdebug.html`

Lustre: The intergalactic file system

Peter J. Braam
Philip Schwan
Cluster File Systems, Inc.
braam@clusterfs.com, http://www.clusterfs.com

1 Introduction

This is a short overview of Lustre, a new open source cluster file system. The name Lustre embodies "Linux" and "Cluster." Lustre focusses scalability for use on large compute clusters, but can equally well serve smaller commercial environments. Lustre runs over different networks, including at present Ethernet and Quadrics.

Lustre originated from research done in the Coda project at Carnegie Mellon. It has seen interest from several companies in the storage industry that contributed to the design and funded some implementation. Soon after the original ideas came out, the USA NationaL Laboratories and the DOD started to explore Lustre as a potential next generation file system. During this stage of the project we received a lot of help and insight from Los Alamos and Sandia National Laboratories, most significantly from Lee Ward.

Lustre provides many new features and embodies significant complexity. In order to reach a usable intermediate target soon, Mark Seager from Lawrence Livermore pushed forward with Lustre Lite. We are hopeful that Lustre Lite will be the shared file system on the new 800 node MCR Linux cluster during 2002.

This paper provides a high level overview of Lustre.

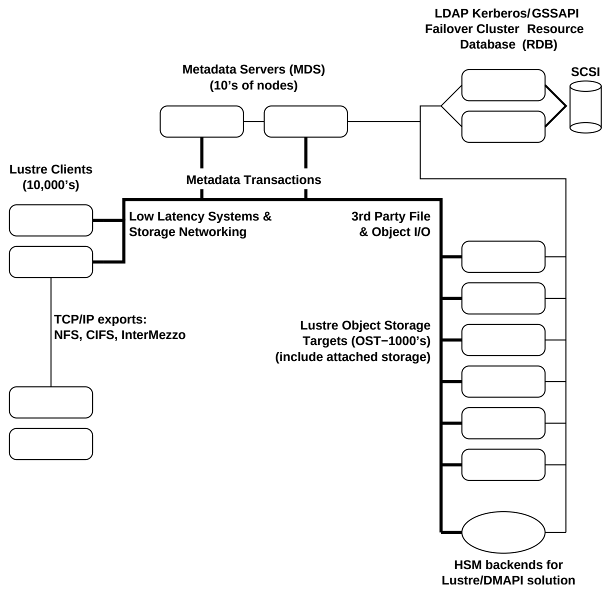


Figure 1: A Lustre Cluster

2 Lustre Components

In Lustre clusters there are three major types of systems, the Clients, the object storage targets (OST's) and metadata server MDS systems. Each of the systems internally has a very modular layout. Many modules, such as locking, the request processing and message passing layers are shared between all systems. Others are unique, such as the Lustre Lite client module on the client systems. Figure 1 gives a first impression of the interactions that are to be expected.

Lustre (see http://www.lustre.org) provides a

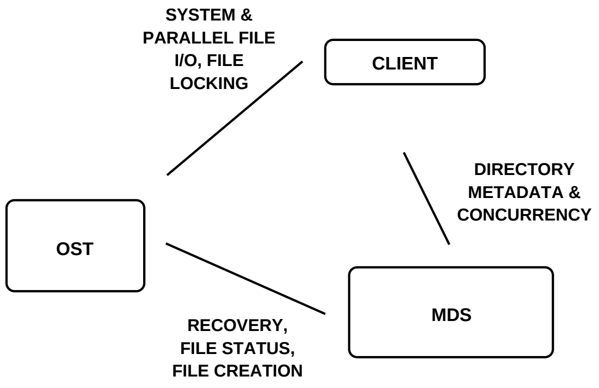

SYSTEM & PARALLEL FILE I/O, FILE LOCKING
CLIENT
DIRECTORY METADATA & CONCURRENCY
OST
MDS
RECOVERY, FILE STATUS, FILE CREATION

Figure 2: outline of interactions between systems

clustered file system which combines features from scalable distributed file systems such as AFS [1], Coda [2] and InterMezzo (see www.inter-mezzo.org), and Locus CFS [3], with ideas derived from traditional shared storage cluster file systems like Zebra [4], Berkeley XFS, which evolved to Frangipani Petal [5], GPFS [6], Calypso [7], InfiniFile [8] and GFS [9]. Lustre clients run the Lustre file system and interact with object storage targets (OST's) for file data I/O and with metadata servers (MDS) for namespace operations. When client, OST and MDS systems are separate, Lustre appears similar to a cluster file system with a file manager, but these subsystems can also all run on the same system, leading to a symmetric layout. The main protocols are described in figure 2.

3 Object Storage Targets

At the root of Lustre is the concept of object storage see [10]. Objects can be thought of as inodes and are used to store file data. Access to these objects is furnished by OST's which provide the file I/O service in a Lustre cluster. The name space is managed by metadata services which manages the Lustre inodes. Such inodes can be directories, symbolic links or special devices in which case the associated data and metadata is stored on the metadata servers. When a Lustre inode represents a file, the metadata merely holds references to the file data objects stored on the OST's.

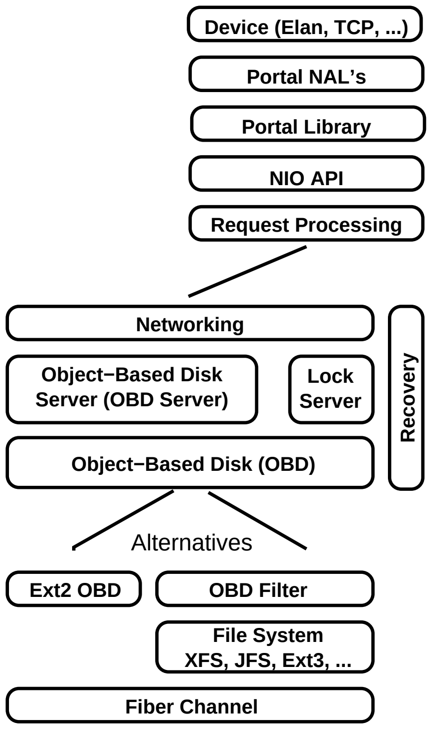


Figure 3: Object Storage Targets (OST)

Fundamental in Lustre's design is that the OST's perform the block allocation for data objects, leading to distributed and scalable allocation metadata. The OST's also enforce security regarding client access to objects. The client - OST protocol bears some similarity to systems like DAFS in that it combines request processing with remote DMA. The software modules in the OST's are indicated in figure 3.

Object storage targets provide a networked interface to other object storage. This second layer of object storage, so-called direct object storage drivers, consists of drivers that manage objects, which can be thought of as files, on persistent storage devices. There are many choices for direct drivers which are often interchangeable. Objects can be stored as raw ext2 inodes by the *obdext2* driver, or as files in many journal file systems by the filtering driver, which is now the standard driver for Lustre Lite. More exotic compositions of subsystems are possible. For example, in some situations an OBD Filter direct driver can run on an NFS file system (a single NFS client is all that is supported).

In the OST figure we have expanded the networking into its subcomponents. Lustre request processing is built on a thin API, called the Portals API which developed at Sandia. Portals interoperates with a variety of network transports through Network Abstraction Layers (NAL). This API provides for the delivery and event generation in connection with network messages and provides advanced capabilities such as using remote DMA (RDMA) if the underlying network transport layer supports this.

4 Metadata Service

The metadata servers are perhaps the most complex subsystem. The provide backend storage for the metadata service and update this transactionally over a network interface. This storage presently uses a journal file system, but other options such as shared object storage will be considered as well.

The MDS contains locking modules and heavily exercise the existing features of journal filesystems, such as ext3 or XFS. In Lustre Lite the complexity is limited as just one single metadata server is present. The system still avoids single points of failure by offering failover metadata services, based on existing solutions such as Kimberlite.

In the full Lustre system metadata processing will be load balanced, which leads to significant complexity related to the concurrent access to persistent metadata.

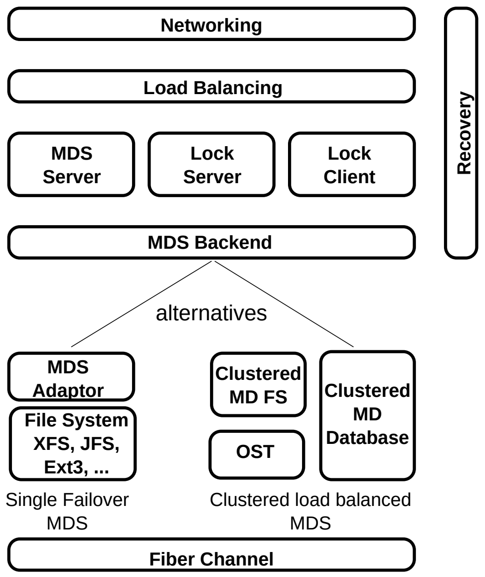


Figure 4: Meta Data Servers (MDS)

5 The client file system

The client metadata protocols are transaction-based and derive from the AFS, Coda and InterMezzo file systems. The protocol features authenticated access, and write-behind caching for all metadata updates. The client again has multiple software modules as shown in figure 5.

Lustre can provide UNIX semantics for file updates. Lock management in Lustre supports coarse granularity locks for entire files and subtrees, when contention is low, as well as finer granularity locks. Finer granularity locks appear for extents in files and as pathname locks to enable scalable access to the root directory of the file system. All subsystems running on Lustre clients can transparently fail over to other services.

The Lustre and Lustre Lite file system provide explicit mechanisms for advanced capabilities

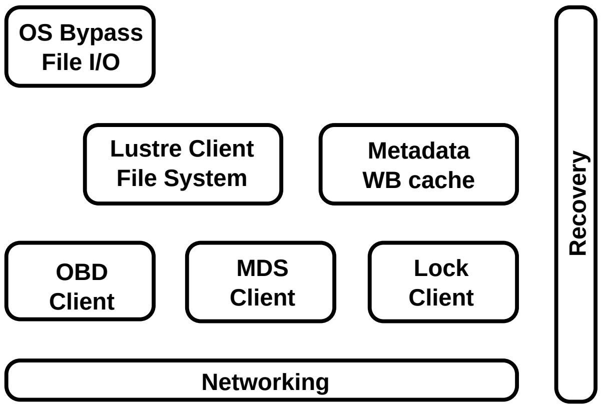


Figure 5: client software modules

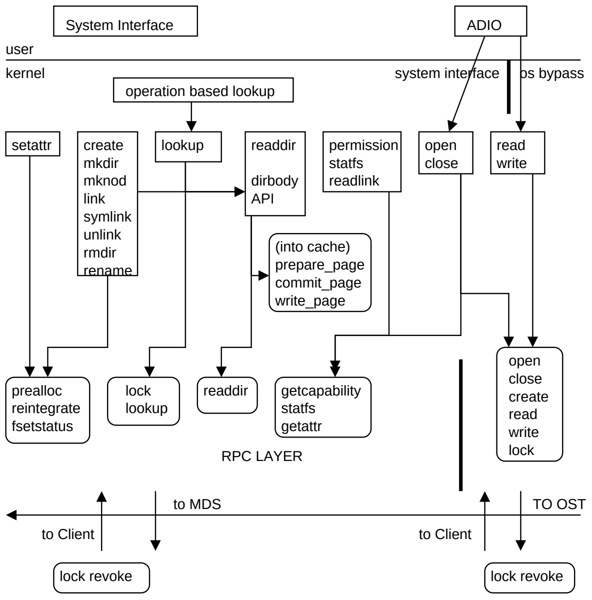

System Interface
ADIO
user
kernel
system interface
os bypass
operation based lookup
setattr
create mkdir mknod link symlink unlink rmdir rename
lookup
readdir dirbody API
permission statfs readlink
open close
read write
(into cache) prepare_page commit_page write_page
prealloc reintegrate fsetstatus
lock lookup
readdir
getcapability statfs getattr
open close create read write lock
RPC LAYER
to MDS
TO OST
to Client
to Client
lock revoke
lock revoke

Figure 6: client file system internals

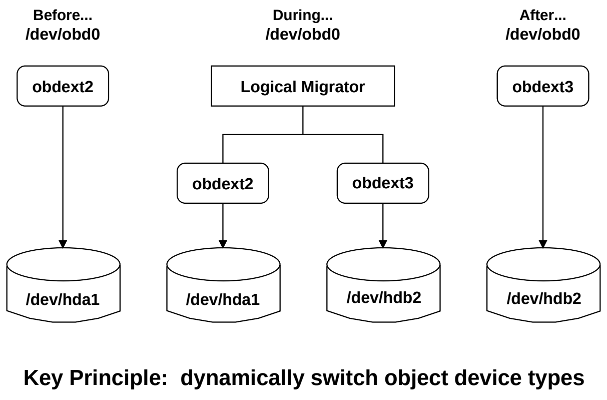


Figure 7: Hot Data Migration using Logical Object Volumes

such as scalable allocation algorithms, security and metadata control. In traditional cluster file systems, such as IBM's GPFS many similar mechanisms are found, but are not independent abstractions, but instead part of a large monolithic file system.

6 Storage Management

Lustre provides numerous ways of handling storage management functions, such as data migration, snapshots, enhanced security and quite advanced functions such as active disk components for data mining. Such storage management is achieved through stacks of object modules, interacting with each other. A general framework is provided for managing and dynamically changing the driver stacks.

An example of stacking object modules is shown in figure 7 for the case of hot data migration from one storage target to another.

7 Conclusions

Lustre provides a novel approach to storage. I heavily leverages existing techniques and software, yet breaks many patterns with new protocols, heavy modularization. The next few years will show if Lustre will establish itself as a mainstream element of the storage industry or will remain an exciting exploration in file system design.

References

[1] J. H. Howard, "An overview of the andrew file system," in *In Proceedings of the USENIX Winter Technical Conference*, 1988.

[2] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," in *Thirteenth ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, U.S., 1991, vol. 25 5, pp. 213–225, ACM Press.

[3] Bruce J. Walker Gerald Popek, *The LOCUS Distributed System Architecture*, MIT Press, 1986.

[4] John H. Hartman and John K. Ousterhout, "The Zebra striped network file system," *ACM Transactions on Computer Systems*, vol. 13, no. 3, pp. 274–310, 1995.

[5] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, "Frangipani: A scalable distributed file system," in *Symposium on Operating Systems Principles*, 1997, pp. 224–237.

[6] IBM, "Gpfs," *FAST proceedings*, 2002.

[7] Murthy Devarakonda, Bill Kish, and Ajay Mohindra, "Recovery in the Calypso file system," *ACM Transactions on Computer Systems*, vol. 14, no. 3, pp. 287–310, 1996.

[8] Yoshitake Shinkai, Yoshihiro Tsuchiya, Takeo Murakami, and Jim Williams, "Alternatives of implementing a cluster file system," pp. 163–178, 2000.

[9] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe, "The Global File System," in *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, College Park, MD, 1996, p. ??, IEEE Computer Society Press.

[10] Garth A. Gibson, David Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka, "File server scaling with network-attached secure disks," in *Measurement and Modeling of Computer Systems*, 1997, pp. 272–284.

Cebolla: Pragmatic IP Anonymity

Zach Brown

zab@zabbo.net

Abstract

Cebolla is an intersection of cryptographic mix networks and the environment of the public Internet. Most of the history of cryptographic mix networks lies in academic attempts to provide anonymity of various sorts to the users of the network. While based on strong cryptographic principles, most attempts have failed to address properties of the public network and the reasonable expectations of most of its users. Cebolla attempts to address this gulf between the interesting research aspects of IP level anonymity and the operational expectations of most uses of the IP network.

1 Introduction

The core concept of providing anonymity of commendations through intermediary relays dates back to the early days of the public network. As initially described by Chaum for email [1], anonymity of the sender can be achieved by sending the message to an agent who encapsulates the email and relays it to a second agent, who relays it to a third, who finally delivers the message. Imagining the communication as a conventional paper letter would conjure an image of each agent opening their letter to discover another letter destined for the next agent. The final agent sees the proper letter destined to the recipient. The response travels in the reverse direction, with each agent putting the incoming letter into a new envelope and addressing it to the previous agent. The sender, upon receiving this large envelope, opens as many layers of envelopes as there were intermediate agents to find the original response.

This anonymizing theory can easily be applied to networks when forwarding instructions are included with each datagram. The included instructions increase the size of the datagrams and verifying the instructions can be very expensive. The common solution to these problems is to negotiate and verify the instructions and require that datagrams reference this existing negotiated state. In Cebolla, this negotiated state is asymmetrical. The initiating sender of all the messages negotiates individual instructions with all the forwarding agents. Each forwarding agent only negotiates state with its immediate neighbours in the path.

Cebolla builds on many previous implementations of anonymizing mix networks:

Wei Dai describes an asymmetric anonymizing network, dubbed PipeNet [2]. While very resilient to attack, it is infeasible to run over the public network. Constant cover traffic makes link usage inefficient and prohibitively expensive. Random path selection punishes users with moderate threat expectations that could tolerate narrowing their traffic to topologically close networks. Finally, rampant frame reordering would confuse popular networking protocols. It has never been publicly implemented.

D. Goldschlag and company at the Navy Research Lab have done much work on Onion Routing [3]. While providing much of the ground work in the field, the implementation is not publicly available, and is covered by US patents.

Zero Knowledge productized a mix network with their Freedom product line. The productized nature of the network motivated Zero Knowledge to remain some amount of centralized control of the network, which turned off some potential users. It was never publicly documented fully, nor were comprehensive sources made available, which prevented any third-party implementations of the protocols to conceivably increase the user-base. It has since been discontinued.

Mike Freedman and company push the envelope by introducing a very scalable peer-to-peer anonymizing network with Tarzan [4]. As it happened, Cebolla and Tarzan were developed at about the same time, with different objectives.

Xor-Trees [5] take the concept to an extreme by describing a network with a fully utilized mesh of dedicated links and synchronized key material generators that can be used to mask both the source and destination of messages between command and control centers. The requirements for fully utilized links and synchronized key material make it infeasible for use on the public network.

Cebolla is an attempt to gel these efforts into an implementation that can be readily used by a group of people on the Internet to efficiently protect their communications. The remainder of the paper will focus on defining the environment that Cebolla considers reasonable, and the methodology behind the implementation.

2 Overview

Cebolla is a unix daemon that maintains UDP connections to a set of peers. Many of these peers connecting together builds an overlay network. Through these UDP connections, called links, peers are able to exchange messages which maintain crypto state, discover the topology of the network, negotiate tunnels on behalf of nodes, and transit encapsulated frames through these tunnels.

Tunnels are the construct that allow messages to be forwarded in a way that masks the identity of the sender. A tunnel is a set of forwarding rules that a client gives to nodes that make up the path of the tunnel. The client also shares keys with each node in the path of the tunnel. Clients and servers run the same software; it is initiating the tunnel that makes us call a node a client.

Like other IP tunnels, Cebolla tunnels have networking devices on the nodes at either end of the tunnel. When a frame is routed into a device, the frame is encapsulated and sent down the tunnel. At the other end, frames exit the tunnel and are received by the device at the end of the tunnel.

The steps performed by a client in a typical Cebolla session might look something like:

- A node in the network is discovered. From this node, the client receives a list of all the nodes in the network.
- The client decides on nodes in the network that a tunnel should include.
- The client establishes a UDP link with the first node in the tunnel.
- With this first node, the client negotiates the first part of a tunnel.
- Through the first part of the tunnel, the client negotiates the second part of the tunnel with the second hop. And so on, until the client only has one hop left.

- Through the tunnel, the client finalizes the tunnel.
- By routing through a local device, the client sends frames down the tunnel.
- Headers on the encapsulated frame tell each hop which tunnel to go down, and the headers are re-written at each hop. Crypto may be performed at each hop, if the client so desired.
- As the frame reaches the final hop, it exits the tunnel and is forwarded out the Internet as a normal IP frame.

3 Threat Model

When describing the threats that Cebolla tries to address, we'll adopt some names for roles that are played out on the network:

- **Alice** – the initiator of the communication, who wishes to remand anonymous.
- **Bob** – the intended recipient of the Alice's communication.
- **Neville** – A corrupt node in the Cebolla path who has honestly participated in the protocol with Alice, but who is trying to leverage that to monitor communications.
- **Patrick** – An attacker who controls the flow of encapsulated frames between Alice and Bob, who can conceivably alter them as they pass.
- **Kiddie** – An attacker with connectivity to the same network as the mesh, but with no control of the path that messages take between Alice and Bob.
- **Smith** – An attacker who is able to monitor communications in the mesh at multiple points and perform deep analysis in real-time.

With these participants in mind, Cebolla attempts to make the following guarantees:

- Alice should be able to determine that she is actually communicating with Bob.
- A Neville working alone should not be able to determine the identities of both Alice and Bob.
- Only Alice and Bob should have access to the actual contents of messages.
- Kiddie should not be allowed to degrade Alice and Bob's communication through trivial a expenditure of resources.

Cebolla also makes the following explicit admissions about its lack of privacy guarantees, as well:

- Two or more Nevilles at the right points in the path may collude, with the help of traffic analysis, to discover many things – the identity of both Alice and Bob, the path that their communication takes, and in unbelievably specific circumstances, even the contents of their communication.
- Patrick may sever communication between Alice and Bob at any time.
- Neville can associate frames that probe the edge with streams he transits, possibly giving rise to the ability to find the node at the edge that terminates a particular communication.
- Smith must to be assumed to be the superset of all possible Nevilles – always knowing which Bobs all Alices are currently in communication with.

4 Secret Negotiation

The Cebolla protocols make heavy use of a four-step secret negotiation that builds shared secrets and negotiates optional features. The message exchange is inspired by Photuris [6]. Following the asymmetric nature of Cebolla's anonymity guarantee, more emphasis is given to protecting the information sent by the client initiating a negotiation than by the server responding to it.

4.1 negotiation phases

The negotiation is split up into four phases:

- **initiation**. The initiator sends an initial negotiation request to the server. The request contains a large random initiator ID and lists of the authentication and shared-secret negotiation schemes that the initiator is willing to use during the negotiation. The entirety of the request is sent in the clear and is readable to those who can monitor the channel it is sent over.
- **response**. The responder parses the request and prepares a response packet. The initiator's random ID is echoed back in the response, and the server provides a responder ID as well. The pairing of these, between the responder and initiator, uniquely defines this negotiation instance. The server parses the lists of offered authentication and shared-secret schemes, and chooses one of each to use for the negotiation. Should it not find any suitable, it can return errors. The response includes authentication data and the responder's half of the secret negotiation, as defined by their respective chosen schemes. The entire response is sent in the clear, but the server appends a signature work-a-like, which the initiator may validate using the chosen authentication scheme.
- **configuration request**. The initiator validates the responder's authentication and prepares a packet containing the IDs that identify the exchange. The initiator appends its half of the shared-secret negotiation to the packet, then combines its half with the responder's half in the incoming packet to calculate the shared-secret. From this secret it derives keys that are used to encrypt the initiator's authentication data and a list of negotiable options that are appended to the packet. The initiator then signs its half of the shared-secret and the encrypted data, appending the clear-text signature material to outgoing packet.
- **configuration acknowledgment**. The responder prepares the final packet in the exchange by parsing the incoming configuration request. After the responder verifies the initiator's signature, it combines the halves of the shared secret and decrypts the initiator's authentication data and option list. The responder choses options from the incoming list and puts them in a list in the outgoing packet. The acknowledgment packet is encrypted with the shared secret.

4.2 negotiation state

An important aspect of the negotiation is that the responder does not maintain state for a negotiation until the configuration request has been successfully parsed. The initiator is responsible for issuing retransmissions until it gives up or the negotiation ends in success or error.

The responder must assume that the initiator will receive the sent configuration acknowledgment because it is the last packet in the exchange. The responder must be careful to deal with the possibility of receiving a retransmitted configuration request from the initiator when the acknowledgment is lost in transit.

4.3 negotiation verification

While the initiator ID is simply a large stream of random bytes, the responder ID is built to provide similar functionality for the responder as syn-cookies [7] do for the TCP handshake.

The responder maintains two private secrets that are alternately replaced at regular intervals. The responder ID is calculated by taking a hash of the most recent private secret and the address of the initiator in the medium of the negotiation. An incoming configuration request must have a responder ID that matches the hash of one of the private secrets and the initiator's address for the responder to be sure that it sent a response to this initiator within the interval that the secrets are updated in.

4.4 negotiation resource consumption

Denial of Service are said to occur when an attack drains resources to the point of excluding others from using those resources. Cebolla doesn't address attacks that exhaust incoming bandwidth because they are best addressed upstream, out of Cebolla's reach.

An attacker wishing to exhaust the CPU resources of the responder is more troubling. The attack comes when an attacker overloads the responder with negotiation packets that look valid based on the responder ID. The responder ID's validity is tied to the source IP of the packets. If the attacker generates the stream of packets through legitimate participation in the protocol the responder can limit the attacker's CPU use based on the IP. Limiting can also be used if the attacker resends an infinite stream of identical packets, all of which must still have a valid IP address to pass the responder ID test. An attacker in the path of regular negotiation traffic can resend packets that it observes, throwing disrupting all negotiation on the path.

It would be generous to describe this protection as incomplete. An attacker is still able to use significant resources on the responder through little effort. Mechanisms like hashcash [8] should be employed to require significant expenditure on the part of the attacker to proceed with the negotiation and convince the responder to spend CPU cycles.

5 Links

Links are the backbone of the Cebolla mesh. All communications between nodes, which include clients, occur over these links. Links use symmetric ciphers to guarantee confidentiality of communications and employ message digests to ensure that communications haven't been tampered with.

Link negotiation occurs between nodes over UDP. Link state is associated with a neighbour's source IP address and UDP port. This association builds the concept of a unique link.

The IP address and UDP port of the responder are assumed to be reachable by all clients. The address of the initiator is never used in the protocol. Initiators may build links from behind routers performing NAT without harm. As is expected, the NAT changing its IP and port mapping will confuse the association of that IP-port pair with its link state.

The primary result of the negotiation is a set of dual transmit and receive keys that the partners of the link use to encrypt and verify frames sent to each other. Separate sets of transmit and receive keys are used to prevent attackers from reflecting frames sent from a node back to the node itself.

5.1 link encapsulation

All messages between link partners are described by a link header. It contains a sequence number, a message type, some flags, and a generic ID field that is used by certain message types. The sequence number, described later, protects attackers from replaying valid frames. No flags are currently defined, and the type is obviously used to decide what to do with the frame.

5.2 dueling link headers

This link header is kept at the size of the block cipher used in the link to enable nodes in the middle of the path to save packet space and CPU time under the right threat assumptions. A client may decide that its traffic is adequately protected by a single-layer full-frame encryption and a MAC check only at the end of the path. The routing process in all transit nodes then simply involves decrypting the header, rewriting the ID, encrypting the header, and forwarding the packet.

A single decryption of the block the header resides in wouldn't be enough to give confidence in the resulting header – it could have been modified in transit. Instead of spending bytes and CPU time on a MAC covering the header, we instead maintain a second key that encrypts and decrypts a second copy of the link header. The recipient decrypts the two copies with the two keys, and if they match it has high confidence that either header has not been modified.

6 Tunnel Negotiation

Cebolla builds up tunnels in an iterative process. The first stage is done between the client and its immediate neighbour who it already has negotiated a link with. A tunnel negotiation builds up similar cryptographic state as is built up in a link negotiation. It also assigns tunnel IDs to each participant. The negotiation can include assigning an IP address to the initiator's endpoint on the final hop negotiation. The negotiation of intermediate hops includes a negotiation parameter that specifies the IP address of the next hop to be negotiated.

Tunnel IDs are used by pairs of nodes to associate frames with a tunnel. Each node has a local ID for a hop that connects to another node. This local ID is uniquely generated by each node and transmitted to the other node during the negotiation. When sending frames down a tunnel the sender uses the remote ID to specify the tunnel to the receiver.

If a multi-hop tunnel is being negotiated, the initiator will include an option in the negotiation that will specify the next hop in the path. The negotiated tunnel is not yet ready to be used with real encapsulated frames. The responder in the negotiation will establish tunnel state and mark it as embryonic and store the next hop. The initiator negotiates an additional hop through the embryonic tunnel by building messages intended for the additional hop. The initiator sends these message down the embryonic tunnel as encapsulated negotiation packets. The embryonic tunnel will unencapsulate the messages and forward them over a link that is established to the additional hop.

This process can be repeated for as many hops as the initiator wishes to build.

As of this writing there are no provisions to stop an initiator to a long time building a tunnel that passes through nodes in the network many, many, times. Such a tunnel allows an initiator to send a single packet down the tunnel, resulting in excessive bandwidth and CPU expenditure by the network.

Preventing this behaviour with a simple time-to-live packet header, as used in IP, would give intermediate nodes information about the length of tunnels. This knowledge can be combined with knowledge of the network graph and measurements of streams to gain a very educated guess as to the actual nodes that make up the tunnel.

6.1 tunnel encapsulation

Tunnel headers communicate details of the encapsulated frame between the initiator and the final hop of the tunnel. The current implementation only contains a type field, which is limited to specifying encapsulated IPv4 frames, an unused flags field, and a sequence number.

The level of protection offered by the tunnel is under full control of the negotiator. Each hop negotiation specifies whether block ciphers or MAC digests are applied to payloads passing through that hop.

7 Keying

Cebolla relies heavily on symmetric ciphers to speed up encryption and decryption. Link and tunnel negotiation both build up a shared secret associated with that link or tunnel. Symmetric keys are derived by hashing the shared secret with a known value for each line of keys that will be used. Further keys in a line are derived by hashing the existing keys with the shared secret. For example, the link payload encryption and decryption keys would differ from the dual link header keys by the known value they were hashed with.

Peers must be sure to derive their encryption and decryption keys so that they match their peers'. The initiator's encryption key must match the responder's decryption key. The current implementation achieves this by requiring the responder to swap its key sets after both peers derive their keys with the same code.

7.1 re-keying

Trust in symmetric keys diminishes the longer they are used in the wild. Key rotation, or re-keying, must be done at regular intervals to lessen the success attackers can have at crypt-analyzing the keys. The rotation must be synchronized between either ends of a resource, allowing for dropped messages, to prevent the keys from becoming out of phase.

Cebolla does this by adopting a protocol for rotating the keys that depends on minimal re-keying messages , knowledge of the role of either end in negotiating the initial resource, and feedback based on which keys succeeded in decrypting incoming frames.

The party who decides to start the re-keying protocol first is dubbed the initiator, the latecomer the responder. Both parties maintain two sets of keys for a given resource, primary and secondary. In the quiescent state the primary keys are in use and the secondary keys are undefined. When re-keying is active, the primary keys are used to send messages and to decrypt messages. Decryption is attempted with the secondary keys only when the primary fail.

- The initiator decides to start re-keying. It derives its secondary keys from the first, and sends a re-keying request to the responder.
- The responder sees the re-keying request and derives its secondary keys from the first, and swaps its primary and secondary keys. Its now primarily encrypting and decrypting with the next generation keys. It sends a dummy message, usually implemented as some form of echo request, over the medium.
- The initiator fails to decrypt the message with its primary keys, but succeeds with

the secondary keys. It takes this to mean that the responder has derived the next generation keys. The initiator swaps its primary and secondary keys, and destroys its secondary keys. It is now only using the next generation keys. It sends another dummy message over the medium.

- The responder succeeds in decrypting the message with its primary keys, and takes this to mean that the initiator has completed the re-keying and destroys its secondary keys.

This mechanism is simple to implement and lets traffic flow during the time it takes the re-keying messages to make the round trips between nodes, which may be particularly important over long, fat pipes. There is always a window during which messages will be doubly decrypted by a mismatch in the primary keys of the sender and receiver.

Initiators can cross the streams. If both parties decide to re-key at the same time, their re-key requests can cross in flight. Both will notice this when they go to process a re-keying request and find that they have initiated a re-keying request themselves. Both initiators know the role they played in negotiating the higher level resource (link or tunnel) and fall back to that role when they discover concurrent re-keying negotiation.

As with negotiation, it is the responsibility of the initiator to re-send re-key requests if it thinks that re-keying is not progressing at a reasonable pace.

8 Sequence Numbers

Cebolla uses sequence numbers in a few places to empower the receiver to discard duplicate frames. A typical advancing window approach is used, implemented almost verbatim from the one specified in RFC 2402. We'll briefly summarize.

The sender always increments the sequence number of frames it transmits. There is only one instance of a sequence number in the lifetime of the sequence, which starts at 1. The receiver maintains a window of sequence numbers that will be accepted. As sequence numbers arrive in that window they are marked off. If that sequence number arrives again its frame will be discarded. If a sequence number arrives that is past the window, the window is shifted so that the largest acceptable sequence number is that of the new arrival. This scheme is simple to implement with bitfields and can withstand reordering and large periods of packet loss on the network.

Care must be taken so that the sequence numbers do not wrap. In the case of tunnel and link payload sequence numbers, the sequence is bound to a key context. When the key contexts are cycled, the sequence is reset to 1. This can be forced when the sequence gets close to wrapping before policy would otherwise dictate that key contexts would be cycled.

9 Network Discovery and Topology Flooding

Cebolla uses a topology flooding scheme which is based on OSPF. Clients must be able to discover nodes in the mesh to communicate through. The clients may wish to make complicated decisions about which nodes to trust, and should be able to trust the information they use in making this decision.

At regular intervals, each node broadcasts a Link State Announcement to each neighbour it has an established link with. These announcements describe the node's static attributes, as well as its current connectivity information. These announcements are signed and contain a sequence number.

As a neighbour receives an announcement, it stores the announcement if it is newer than the previous announcement from that neighbour, and sends back an acknowledgment. If the announcement was new, the neighbour then sends the announcement to all its neighbours. The announcement is not forwarded to the neighbour it was just received from. Announcements are re-sent until their receipt is acknowledged.

The announcement collections of each node are sorted and served via the rsync protocol[9], which allows clients to receive deltas of the largely static information efficiently.

The mechanism can be boot strapped by nearly any out-of-band medium that can communicate IP addresses: email, web pages, DNS, etc.

9.1 topology attacks and accountability

Topology discovery raises many risks. A corrupt node could induce bad announcements into the network. A corrupt node could alter the flow of announcements it transits to the rest of the network. A corrupt node could participate honestly with the rest of the nodes in topology flooding and feed bad information only to clients.

Public key signatures bring a simple first layer of confidence to the system. Announcements can be confirmed to come from the same agent as last time, and trust can be established between that agent and any existing public key trust metric system.

The distributed publication of the connectivity announcements gives multiple views into the announcement circulation at multiple points. This lets clients audit the veracity of the announcements, possibly anonymously. Distributed trust metrics can be adopted by using the sequence numbers to check that all views of the mesh are consistent with others. The rate that sequence numbers advance can be checked to make sure that a node isn't delaying announcements. This lets a mesh permit a node's entry into the mesh without centralized admission. Low initial trust increases over time as behavioral audits confirm that the new node is honest.

The announcements can be extended so nodes can describe themselves. Clients can use parameters like software types, administrative domains, underlying network connectivity, and such, to decide which nodes to build a tunnel with.

10 Acknowledgments

Jerome Etienne should rightfully be considered an author of any document describing these protocols—when he didn't outright design them they're only minimally deviating from his work.

Phil Schwan and Mike Shaver's contributions are treasured, as always.

Special thanks are due to Dr. Adam Back, who never fails to make taking on an interesting challenge entertaining.

Ulf Moeller, Mike Freedman, Anton Siglic, Ian Goldberg, and Adam Shostack were kind enough to humour me by explaining cryptographic principles using small words.

11 Availability

Cebolla should be available under the GPL from

```
http://www.zabbo.net/cebolla/
```

References

[1] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM (USA)*, 24(2), 1981.

[2] Wei Dai. Pipenet. http://www.eskimo.com /˜weidai/pipenet.txt.

[3] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM (USA)*, 42(2):39–41, 1999.

[4] Michael J. Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[5] Shlomi Dolev and Rafail Ostrovsky. Xor-trees for efficient anonymous multicast and reception. Technical Report 98-54, 23 1998.

[6] P. Karn and W. Simpson. [rfc 2522] photuris: Session-key management protocol, March 1999.

[7] Dan Bernstein. Syn cookies. http://cr.yp.to/syncookies.html.

[8] Adam Back. Hashcash. http://www.cypherspace.org/hashcash/, May 1997.

[9] Andrew Tridgell. Efficient algorithms for sorting and synchronization. http://citeseer.nj.nec.com /tridgell99efficient.html.

SE Debian: how to make NSA SE Linux work in a distribution

Russell Coker <`russell@coker.com.au`>,
`http://www.coker.com.au/`

Abstract

I conservatively expect that tens of thousands of Debian users will be using NSA SE Linux [1] next year. I will explain how to make SE Linux work as part of a distribution, and be managable for the administrator.

Although I am writing about my work in developing SE Linux support for Debian, I am using generic terms as much as possible, as the same things need to be done for RPM based distributions.

1 Introduction

SE Linux offers significant benefits for security. It accomplishes this by adding another layer of security in addition to the default Unix permissions model. This is accomplished by firstly assigning a *type* to every file, device, network socket, etc. Then every process has a *domain*, and the level of access permitted to a type is determined by the domain of the process that is attempting the access (in addition to the usual Unix permission checks). Domains may only be changed at process execution time. The domain may automatically be changed when a process is executed based on the type of the executable program file and the domain of the process that is executing it, or a privileged process may specify the new domain for the child process.

In addition to the use of domains and types for access control SE Linux tracks the *identity* of the user (which will be *system_u* for processes that are part of the operating system or the Unix user-name) and the role. Each *identity* will have a list of roles that it is permitted to assume, and each *role* will have a list of domains that it may use. This gives a high level of control over the actions of a user which is tracked through the system. When the user runs SUID or SGID programs the original identity will still be tracked and their privileges in the SE security scheme will not change. This is very different to the standard Unix permissions where after a SUID program runs another SUID program it's impossible to determine who ran the original process. Also of note is the fact that operations that are denied by the security policy [2] have the *identity* of the process in question logged.

For a detailed description of how SE Linux works I recommend reading the paper Peter Loscocco presented at OLS in 2001 [1].

The difficulty is that this increase in functionality also involves an increase in complexity, and requires re-authenticating more often than on a regular Unix system (the SE Linux security policy requires that the user re-authenticate for change of *role*). Due to this most people who could benefit from SE Linux will find themselves unable to use it because of the difficulties of managing it. I plan to address this problem through packaging SE Linux for Debian.

The first issue is getting packages of software that is patched for support of the SE Linux system calls and logic. This includes modified programs for every method of login (/bin/login, sshd, and X login programs), modified *cron* to run cron jobs in the correct security context, modified *ps* to display the security context, modified *logrotate* to keep the correct context on log files, as well as many other modified utilities.

The next issue is to configure the system such that when a package of software is installed the correct security contexts will be automatically applied to all files.

The most difficult problem is ensuring that configuration scripts get run in the correct security context when installing and upgrading packages.

The final problem is managing the configuration files for the security policy.

Once these problems are solved there is still the issue of the SE Linux sample policy being far from the complete policy that is needed in a real network. I estimate that at least 500 new security policy files will need to be written before the sample policy is complete enough that most people can just select the parts that they need for a working system.

2 Patching the Packages

The task of the login program is to authenticate the user, *chown* the tty device to the correct UID, and change to the appropriate UID/GID before executing the user's shell. The SE patched version of the login program performs the same tasks, but in addition changes the security identifier (SID) on the terminal device with the *chsid* system call and then uses the *execve_secure* system call instead of the *execve* system call to change the SID of the child process. The login program also gives the user a choice of which of their authorised roles they will assume at login time.

This is not very different from the regular functionality of the login program and does not require a significant patch.

Typically this adds less than 9K to the object size of the login program, so hopefully soon many of the login programs will have the SE code always compiled in. For the rest we just need a set of packages containing the SE versions of the same programs. So this issue is not a difficult one to solve and most of the work needed to solve it has been done.

A similar patch needs to be applied to many other programs which perform similar operations. One example is *cron* which needs to be modified so cron jobs will be run in the correct security context. Another example is the *suexec* program from *Apache*. An example of a similar program for which no-one has yet written a patch is *procmail*.

Programs which copy files also need to have suitable options for preserving SIDs, *logrotate* and the *fileutils* package (which includes *cp*) have such patches, *cpio* lacks such a patch, and there is a patch for *tar* but it doesn't apply to recent versions and probably needs to be rewritten.

3 Setting the Correct SID When Installing Files

When a package of software is installed the final part of the installation is running a *postinst* script which in the case of a daemon will usually start the daemon in question. However if the files in the package do not have the correct SIDs then the daemon may not be able to run, or will be unable to run correctly!

The Debian packaging system does not currently have any support for running a script after the files of a package are installed but before the *postinst* script. There have been discussions for a few years on how best to do this, as I didn't have time to properly re-write *dpkg* I instead did a quick hack to make it run scripts that it finds in */etc/dpkg/postinst.d/* before running the *postinst* of the package.

When installing an SE Linux system the program *setfiles* is used to apply the correct SIDs to all files in the system. I have written a patch to make it instead take a list of canonical fully-qualified file names on standard input if run with the *-s* switch, which is now included in the NSA source release.

The combination of the *dpkg* patch and the *setfiles* patch allow me to solve the basic problem of getting the correct SIDs applied to files, my script just queries the package management system for a list of files contained in the package and pipes it through to *setfiles* to set the SID on each file.

The next complication is setting the correct SID for the *setfiles* program, by default it gets installed with the security type *sbin_t* because that is the type of the directory it is installed in. However in my default policy setup I have not given the *dpkg_t* domain (which is used by the *dpkg* program when it is run administratively) the privilege of changing the SID of files. So the *setfiles* program needs to have the type *setfiles_exec_t* to trigger an automatic domain transition to the *setfiles_t* domain.

To solve this issue I have the *preinst* script (the script that is run before the package is installed) of the *selinux* package rename the */usr/sbin/setfiles* to */usr/sbin/setfiles.old* on an upgrade. Then the */etc/dpkg/postinst.d/selinux* script will run the old version if it exists.

Here's the relevant section of the *selinux.preinst* file:

```
if [ ! -f /usr/sbin/setfiles.old -a \
    -f /usr/sbin/setfiles ]; then
  mv /usr/sbin/setfiles \
     /usr/sbin/setfiles.old
fi
```

Table 1 shows the contents of */etc/dpkg/postinst.d/selinux*. The first parameter to the script is the name of the package that is being installed. Also I have *"grep ..."* included because setfiles currently has some problems with blank lines and */.* which *dpkg* produces.

4 Running Configuration Scripts in the Correct Context

When a SE Linux system boots the process *init* is started in the domain *init_t*. When it runs the daemon start scripts it uses the scripts */etc/init.d/rc* and */etc/init.d/rcS* on a Debian system (on Red Hat it is */etc/rc.d/rc* and */etc/rc.d/rc.sysinit*). So these scripts are given the type *initrc_exec_t* and there is a rule *domain_auto_trans(init_t, initrc_exec_t, initrc_t)* which causes a transition to the *initrc_t* domain. The security policy for each daemon will have a rule causing a domain transition from the *initrc_t* domain to the daemon domain upon execution of the daemon. This all happens as the *system_u* identity and the *system_r* role.

When the system administrator wants to start a script manually they use the program *run_init* which can only be run from the *sysadm_t* domain, it re-authenticates the administrator (to avoid the possibility of it being called by some malicious code that the administrator accidentally runs) before running the specified script as *system_u:system_r:initrc_t*.

This works fine when the daemon start script is quite simple (most such start scripts just

```
#!/bin/sh

make -s -C /etc/selinux file_contexts/file_contexts

SETFILES=/usr/sbin/setfiles
if [ -x /usr/sbin/setfiles.old ]; then
    SETFILES=/usr/sbin/setfiles.old
fi
dpkg -L $1 | grep ^/.. | $SETFILES -s \
    /etc/selinux/file_contexts/file_contexts
if [ -x /usr/sbin/setfiles.old -a "$1" = "selinux" ]; then
    rm /usr/sbin/setfiles.old
fi
```

Table 1: Contents of */etc/dpkg/postinst.d/selinux*.

check whether the daemon is already running and then run it with appropriate parameters). However this doesn't work for complex scripts, which may copy files, change sysctl entries via */proc*, and do many other things. An example of this is the *devfsd* package where the start script creates device nodes for device drivers that lack kernel support for *devfs*. Getting this to work correctly required that the code for device node creation be split into a separate file with the same SID as the main daemon (*devfsd_exec_t*) which causes it to run in the same domain as the daemon (*devfsd_t*). Such changes will probably have to be made to about 5% of daemon start scripts.

But that is part of the standard proceedure of correctly setting up SE Linux. The package specific part comes when the scripts have to be started from the package installation. To get the correct domain (*initrc_t*) for the scripts I use the rule *domain_auto_trans(dpkg_t, etc_t, initrc_t)* which causes the *dpkg_t* domain to transition to the *initrc_t* domain when a script of type *etc_t* is executed. Now the hard part is getting the identity and the role correct when running *dpkg*. For this purpose I have written a customised version of *run_init* to change to the context to *system_u:system_r:dpkg_t*, *system_u:system_r:apt_t*, or *system_u:system_r:dselect_t*, for the programs *dpkg*, *dselect*, and *apt-get* respectively.

The *apt_t* and *dselect_t* domains are only used for selecting and downloading packages, and then executing *dpkg*, which triggers an automatic transition to the *dpkg_t* domain.

5 Managing the Configuration Files

For normal configuration files in Debian (almost every file under /etc and some files in other locations) the file is registered as a *conffile* in the packaging system, and the package status file contains the MD5 checksum of the file. If a file is changed from its original contents (according to an MD5 check) at the time the package is upgraded and if the new version has a different set of data for the file than that which was provided by the old version of the package (according to MD5) then the user will be asked if they want to replace the old file (with a default of no). However if the new version of the package contains different content and the old content was not changed, then the user will get the new content without even being informed of the fact!

This is OK for many files, but the idea of a file from your audited security configuration being replaced with one you've never seen is not a pleasant one! This is only the first problem with managing policy files, the next problem is the size of the database for the sample policy. If you are using an initial RAM disk (initrd) then you must have the policy database on the initrd. The default initrd size of 4 megabytes is not large enough to accomodate the usual modules and the complete sample policy.

So what we need to solve this is a way of having a set of sample policy files (one per *domain*), of which not all will be used, and when new policy files are added or existing files are changed the user must be prompted as to whether they want to add the new files or apply the changes. Also when adding new policy the matching entries have to be added to the database used by *setfiles* for setting the file context.

In the latest versions of the sample policy the *Makefile* creates a configuration file for *setfiles* to match the program configuration files used. For every application policy file *domains/program/%.te* the matching file *file_contexts/program/%.fc* will be used as part of the configuration. This change will solve the issue of determining the configuration for *setfiles*, but it doesn't entirely solve the problem. One issue with this is that when a file is added to or removed from the configuration the appropriate changes need to be made to the file system. If you make an addition to the policy before installing a new package (the correct proceedure) then you can usually get away without this as long as none of the files or directories previously existed, however this is not always the case, especially when files are diverted or when dealing with standard directories such as */var/spool/mail* which will exist even if you have not installed any software to use them! It should not be that difficult to write a program to relabel the files matching the specifications of the added policy, the question is whether policy additions are common enough to make it worth saving the effort of a relabel. Also there's the risk that a bug in such a program (or its use) could potentially cause a security hole.

The security policy is comprised of one configuration file per application (or class of application, some domains such as the DHCP client domain *dhcpd_t* are used by multiple programs which perform similar functions). Also sometimes an application requires multiple domains which will therefore be defined in the one file, for example my current policy for Postfix has eleven domains (which is excessive, I plan to reduce it to three or four once I've determined exactly what is required). One problem I faced with this is the issue of what to do when one domain needs to interact with another domain, for example the *pppd* process often needs to run *sendmail -q* to flush the mail queue when it establishes a connection. This requires the policy statement *domain_auto_trans(pppd_t, sendmail_exec_t, sysadm_mail_t)*, previously such a statement would be put in either the *sendmail.te* file or the *pppd.te* file, thus making one of them depend on the other. This is a bad idea because there's no reason for either of these programs to depend on the other. The solution I devised is based on the M4 macro language (which was already used for simpler macro functionality in producing the policy file). I created a script to define a macro with the name of each application policy file that is used. So the solution to the PPP and Sendmail problem is to put the following in the *pppd.te* file:

```
ifdef(‘sendmail.te’,
‘domain_auto_trans(pppd_t,
   sendmail_exec_t, sysadm_mail_t)’)
```

The next problem, is how to effectively manage things so that when I ship a new and improved sample policy the administrator can update it without excessive pain.

The current method involves running *diff -ru* and then copying files if you like the changes. This is excessively painful even when managing one or two SE Linux machines! So it obviously won't scale to serious production. I plan to write a Perl script to manage this, the first thing it has to do is track when the administrator doesn't want a policy file. When a file is removed then the fact that the user has chosen not to have that file installed should be recorded, and they should not be prompted to re-install it on the next upgrade. However if the sample policy is upgraded and a new file has been added then they should be asked if they want to install it. Then when a file in the sample policy changes and it is a file that is installed the user should be asked if they want the new file copied over their existing file (and they should be provided with a *diff* to show what the changes would be). Finally if such changes involve the file configuration for *setfiles* then the user should be asked whether they want to relabel the system.

The people who are working on Red Hat packaging are considering other ways of managing the versions of configuration files, one of which involves having symbolic links pointing to the files to be used, if you decide to use your own version instead of one of the supplied policy files then you can change the sym-link.

6 Managing Device Nodes

In Linux there are two methods of managing device nodes. One is the traditional method of having */dev* be a regular directory on the root file system and have device nodes created on it with *mknod*, the other is the *devfs* file system which allows the kernel to automatically create device nodes while the *devfsd* process automatically assigns the correct UID, GID, and permissions to them.

On a traditional (non-devfs) system running SE Linux the device nodes will be labelled in the same way as any other file. On a devfs system things are different, the devfs policy database contains rules for labelling device nodes. However this has some limitations, one being that when the policy database does not have an entry for the device node at the time it is created, then it will never be labelled. Another is that every *type* listed in the devfs configuration rules must be defined, which can cause needless dependencies.

To address these issues I wrote a module for devfsd which adds support for SE Linux. This allows you to change the mapping of SIDs to device nodes and re-apply it at any time, and if a security context listed in the configuration file does not exist in the policy then an error will be logged and the system will continue working.

This is especially useful for the case of an *initrd* as the types for all the possible device nodes won't need to be in the ram disk.

7 Work To Be Done

Initial RAM Disk

When using an initrd to boot a modular kernel the security policy database must be stored on the initrd. The problem is that the default initrd size is 4M, which does not leave much space when libc6 is included, often not enough for the policy you want. Also even if the policy does fit you won't really want to have such a large initrd image. If you are installing SE Linux on a single PC, or even on a network of similar PCs then you are best advised to build a kernel with all modules needed for booting statically linked and not use an initrd. However this is not possible for a distribution vendor who has to support a huge variety of hardware.

Another problem with using an initrd for storing the policy is that when you generate a new policy you then have to regenerate the initrd to avoid having your changes disappear on the next boot, of course a boot script could easily load the updated policy from the root file system before going to multi-user mode. But it is wasteful to have a large policy on the initrd that you then discard before ever using much of it.

The solution is to have a small policy that contains all the settings needed for either the first stage of boot, or alternately for running recovery tools in case a failure prevents the machine from entering multi-user mode. Then after the machine has passed the first stages of the boot process a complete policy can be loaded from the root file system, as long as the two policies don't conflict in any major way this should work well. NB A Major policy conflict is a situation where the initrd defines domains that aren't defined in the new policy and processes are executed in such a domain.

The latest release of SE Linux supports automatically re-loading the policy when the real root file system is mounted. Now all that needs to be done is for someone to write a mini-policy to install on the initrd.

Polishing run_init

Stephen Smalley has suggested that we develop a *run_init* program that incorporates the functionality of my modified program as well as of the original *run_init* program in a more generic fashion. It is apparent that other people will have similar needs for programs to execute programs under a different domain, role, and maybe identity. It is better that one program do this than to have many people writing programs for such things.

Also currently my program is hard-coded for the names of the Debian administration programs. An improved program should handle the needs of Debian, RPM, and the regular run_init functionality.

Writing Sample Policy Files

Currently any serious system will require policy files that are not in the sample policy. This forces everyone who uses SE Linux to start by writing policy files (which is the most difficult and time consuming task involved with the project). Currently we are writing new sample policy files for the variety of daemons and applications, and developing new macros for writing policy files quickly. With the new macros policy files are on average half the size that they used to be (and I aim to reduce the size again by new macros). The macros allow short policy files which are easy to understand, and therefore the user can easily determine how to make any required changes, or how to write a policy file for a new program based on existing programs.

8 Obtaining the Source

Currently most of my packages and source are available at `http://www.coker.com.au/selinux/` however I plan to eventually get them all into Debian at which time I may remove that site.

I have several packages in the unstable distribution of Debian, the first is the *kernel-patch-2.4-lsm* and *kernel-patch-2.5-lsm* packages which supply the Linux Security Modules `http://lsm.immunix.org/` kernel patch. That patch includes SE Linux as well as LIDS and some of the OpenWall functionality. When I have time I back-port patches to older kernels and include new patches that the NSA has not officially released, so often my patches will provide more features than the official patches distributed by the NSA from `http://www.nsa.gov/selinux/index.html` or the patches distributed by Immunix. However if you want the *official* patches then these packages may not be what you desire.

From the *selinux-small* archive I create the packages *selinux* and *libselinux-dev* which are also in the unstable distribution of Debian.

9 Acknowledgments

I would like to thank Stephen Smalley for being so helpful when I was learning about SE Linux, and Dr. Brian May for checking my early packages and giving me some good advice when I first started.

Also thanks to Dr. May, Stephen Smalley, and Peter Loscocco for reviewing this paper.

References

[1] *Meeting Critical Security Objectives with Security-Enhanced Linux*. Peter A. Loscocco, NSA, loscocco@tycho.nsa.gov; Stephen D. Smalley, NAI Labs, ssmalley@nai.com `http://www.nsa.gov/selinux/ottawa01-abs.html/`

[2] *Configuring the SELinux Policy*. Stephen D. Smalley, NAI Labs, ssmalley@nai.com `http://www.nsa.gov/selinux/policy2-abs.html/`

The Long Road to the Advanced Encryption Standard

Jean-Luc Cooke
CertainKey Inc.
jlcooke@certainkey.com, http://www.certainkey.com/˜jlcooke

Abstract

This paper will start with a brief background of the Advanced Encryption Standard (AES) process, lessons learned from the Data Encryption Standard (DES), other U.S. government cryptographic publications and the fifteen first round candidate algorithms. The focus of the presentation will lie in presenting the general design of the five final candidate algorithms, and the specifics of the AES and how it differs from the Rijndael design. A presentation on the AES modes of operation and Secure Hash Algorithm (SHA) family of algorithms will follow and will include discussion about how it is directly implicated by AES developments.

Intended Audience

This paper was written as a supplement to a presentation at the Ottawa International Linux Symposium. The reader should have at least first year university level knowledge of algebra and physics. Someone with no knowledge of mathematics can still benefit from this paper and its associated presentation. This topic of cryptography is covered lightly. Care is taken to present enough useful technical information to be interesting to a technical audience and beneficial to others.

1 Introduction

Two decades ago the state-of-the-art in the private sector cryptography was—we know now—far behind the public sector. Don Coppersmith's knowledge of the Data Encryption Standard's (DES) resilience to the then unknown Differential Cryptanalysis (DC), the design principles used in the Secure Hash Algorithm (SHA) in Digital Signature Standard (DSS) being case and point[NISTDSS][NISTDES][DC][NISTSHA1].

The selection and design of the DES was shrouded in controversy and suspicion. This very controversy has lead to a fantastic acceleration in private sector cryptographic advancement. So intrigued by the NSA's modifications to the Lucifer algorithm, researchers—academic and industry alike—powerful tools in assessing block cipher strength were developed. Some of these tools proved useful in understanding more about the changes made by the NSA.

Taking an objective look at the standardization practices of the USA NSA and NIST organizations, one can make broad assumptions on where the American state is focusing its cryptanalytic resources.

1.1 What the NSA/NIST has Taught Us

By the mid-1970's the private sector began having an interest in digital cryptography. Even if the clearly false IBM statement "global market for computers estimated at 10" had been proven correct, most of the computers in operation at the time were terminal servers. The terminals connected to these servers were carrying progressively more sensitive data. Financial records, payroll information, trade secrets, and intellectual property; all crucial to the success of a business were exposed to the hot new hobby of wire tapping with gator clips.

Before the US government moved to create a single encryption standard the private sector was taking its first steps into design cryptographic algorithms. In what would become crypto folklore, the NSA quietly send out letters of solicitation to a few hand picked cryptographic experts and laboratories. It is important to realize that previous to this the only communication a mathematician would have with the NSA was in the form of a "cease and desist or be thrown in jail for conspiracy" letters.

Of the few responses received by the NSA, only one had actually met the minimum standards set out by the NSA in their solicitation. The Lucifer block cipher designed by Don Coppersmith, Horst Fiestel and company at IBM was the winner practically by default.

There were two distinct differences between the Lucifer algorithm submitted by IBM and the final DES design.

- **T**he effective key space was reduced by several orders of magnitude.
- **T**he core substitution boxes were redesigned.

These changes were made without comment from IBM or the NSA. Reducing the effective key strength of the algorithm and the ominous change to possibly the single most crucial sub-component of the algorithm had everyone second-guessing the DES.

In the subsequent years after the 1976 DES announcement, Shamir and Biham published their paper on Differential Cryptanalysis (DC) (1994, ISBN-0387979301). In this paper, the two cryptographers of RSA fame ('S' to be precise), outlined how to correlate input changes to the output of several variants of the DES. At then end of the day, the Lucifer algorithm fell to the attack of DC while DES remained unbroken. To the shock of sceptics, the NSA appears to have not weakened the DES for their evil purposes but in fact made it impervious to an attack not to be publicized for another eighteen years.

After the announcement of DC, the Lucifer co-designer Don Coppersmith confessed to knowledge of DC at the time of DES standardization. This kept the sky from falling on the heads of the crypto sceptics as you can well imagine.

2 Obsolescence of the DES

A 56bit key space did not provide sufficient protection in lieu of the personal computer explosion of the late 1980's and 1990's. The threat of attack was no longer from a single powerful computer, but from thousands of commodity computers coordinating their efforts. In comes the Triple-DES. Encrypting the data with three distinct keys resulted in a three-fold increase in key material, a three-fold increase in computational effort required for encryption, and a 5.2×10^{33} increase in the key space.

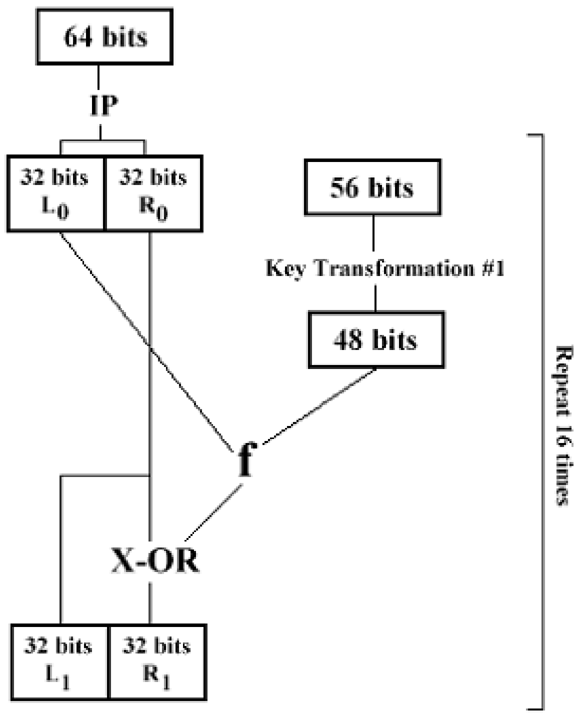


Figure 1: The DES Cipher

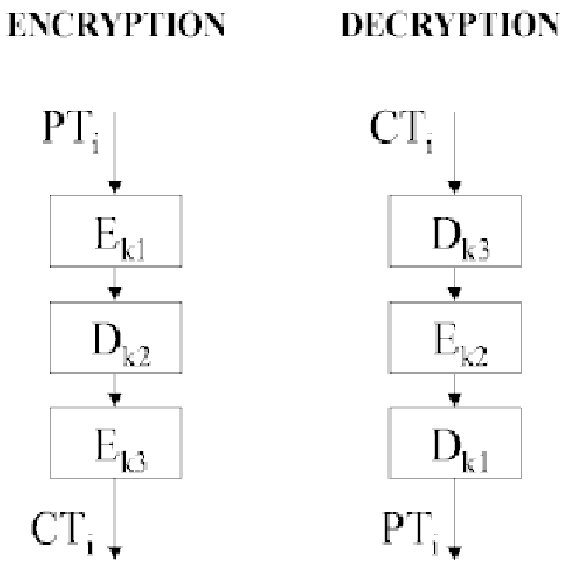


Figure 2: The 3DES Cipher

$$
\begin{aligned}
E &= 3DES_{k1,k2,k3}(D) \\
E &= DES_{k1}(DES^{-1}_{k2}(DES_{k3}(D)))
\end{aligned}
\tag{1}
$$

By the mid 1990's, TripleDES was no longer sufficient. The security of the algorithm was assumed to be good, but there were other shortcomings with TripleDES.

TripleDES was too slow. The private sector's obsession with higher digital communication bandwidths had made the integration of encryption at the data link layer far too costly. Economic conditions then predicted that all data would be encrypted at least once by 2006. Current economic conditions make this prediction conservative.

Private sector dependence on secure data communication was going to continue to grow beyond the capabilities of DES/TripleDES. A new standard with greater security and a long

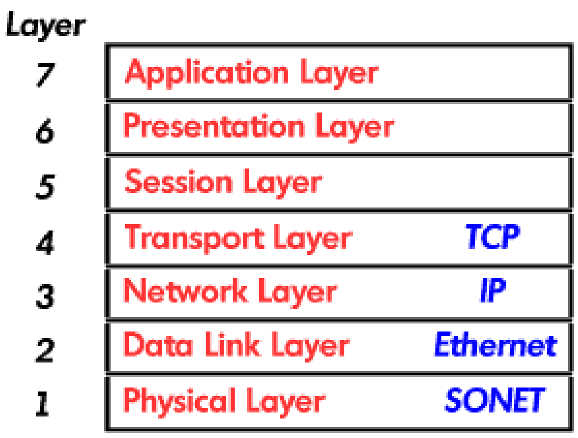


Figure 3: The OSI Network Stack

lifetime needed to be decided to mitigate the impact of migrating to a new standard. "We need to act fast before we're in serious trouble."

Another issue with DES and subsequently TripleDES, was a design limitation. DES was not designed to be efficient in software. The ubiquity of software in modern computer and communication technology dictated an efficient hardware as well as software implementation for this new standard.

3 AES Round One

Contrasting the algorithm solicitation process used in selecting the DES, a very public announcement was made by the NSA/NIST for cipher designs. Appearing at private sector security and cryptography tradeshows, it was made very clear this time the standard was going to be a very public affair.

The NSA/NIST set out minimum requirements for block cipher submission[NISTAESWWW]

$$AESCD1$$

$$AESCD2$$

$$AESCD3$$

.

- **S**ize efficiency of hardware/software implementation
- **S**peed efficiency of hardware/software implementation
- **M**inimum 128bit block sizes
- **K**ey sizes up to 256 bits
- **R**esilience to all known modern attacks.

Fifteen algorithms met these requirements.

- **C**AST-256 - Entrust Technologies Ltd.
  http://www.entrust.com/

  $$AESCD1$$

  $$AESCD2$$

  - **T**he Communications Security Establishment (CSE)—Canadian equivalent to the US NSA—has adopted the CAST5 algorithm as their confidential government data encryption algorithm.
    CSE website: http://www.cse.dnd.ca/
  - **C**AST5 is synonymous with CAST-128
  - **C**AST-256 is an extension to CAST5 for inclusion to the AES
- **C**rypton - Future Systems
  http://www.future.co.kr/

  $$AESCD1$$

  $$AESCD2$$

  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?
- **D**EAL - Outerbridge, Knudsen

http://www.ii.uib.no/~larsr/newblock.html

$AESCD1$

$AESCD2$

  - **T**his is one of two submission co-authored by Knudsen. See also Serpent!
  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?

- **D**FC - Centre National pour la Recherche Scientifique - Ecole Normale Superieure

  http://www.dmi.ens.fr/~vaudenay/dfc.html

  $AESCD1$

  $AESCD2$

  - **V**ive la resistance!
  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?

- **E**2 - Nippon Telegraph and Telephone

  http://info.isl.ntt.co.jp/e2/

  $AESCD1$

  $AESCD2$

  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?

- **F**ROG - TecApro

  http://www.tecapro.com/aesfrog.htm

  $AESCD1$

  $AESCD2$

  - **G**eorgoudis is an amateur cryptographer, the only one in Puerto-Rico in all likelihood! FROG was the first cipher he ever designed, and it was accepted to the AES process!
  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here? Or is Puerto-Rico's special status with the US exempt them for this? Isn't it nice we live in the "free world" and don't have to worry about such ugliness?

- **H**PC - Schroeppel

  http://www.cs.arizona.edu/~rcs/hpc/

  $AESCD1$

  $AESCD2$

  - **A**n academic's block cipher. Schroeppel's paper goes into great detail on the theoretical advantages of his Hasty Pudding Cipher. Not a very practical cipher, underlines the 'openness' of the AES submission process.
  - **B**onus question: Hasty Pudding and Harvard University - what's the connection?

- **L**OKI97 - Brown, Pieprzyk, Beberry

  http://www.unsw.adfa.edu.au/˜lpb
  /research/loki97/

  $AESCD1$

  $AESCD2$

  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?

- **M**AGENTA - Deutsche Telekom

  no url available

  $AESCD1$

  $AESCD2$

  - **A**uthor unknown . . . and for good reason! At the first AES conference, the presenter from DT had nightmare of nightmares happen. The MAGENTA cipher was cracked in real-time! Discussions in the audience between Biham and others led to a mountable attack before the presentation was even over! A paper was written and published within twenty-four hours. And the kicker of it all was there were rumours that MAGENTA had been used in production DT equipment for years but the algorithm was never published. Chalk one up to security though non-obscurity.

  - **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?

- **M**ARS - IBM

  http://www.research.ibm.com/security
  /mars.html

  $AESCD1$

  $AESCD2$

  - **T**he Lucifer/DES design team (most of it) returns. A lot was expected from this team.

- **R**C6 - RSA Laboratories

  http://www.rsasecurity.com/rsalabs/aes/

  $AESCD1$

  $AESCD2$

  - **R**C6, based on RC5, based on RC4. Principle designer Ron Rivest, the R in RSA, the man behind MD1/2/3/4/5, RC1/2/3/4/5/6 and many other publications. If there ever was a crypto rock star, this is he.

- **R**ijndael - Daeman, Rijman

  http://www.esat.kuleuven.ac.be/˜rijmen
  /rijndael/

  $AESCD1$

  $AESCD2$

- **T**wo Flemish Belgians (as apposed to the French Belgians) designed Rijndael. Cinderella story: no big names, modest track record, and European nationalities.
- **A**t the time of AES, this algorithm submission would be allowed to ENTER the US, but not leave. Is something wrong here?

• **S**afer+ - Cylink Corporation

mailto:williams.chuck@cylink.com

$$AESCD1$$

$$AESCD2$$

- **B**ased on the Safer cipher.

• **S**erpent - Anderson, Biham, Knudsen

http://www.cl.cam.ac.uk/~rja14
/serpent.html

$$AESCD1$$

$$AESCD2$$

- **S**trong cipher, big name authors. Biham co-authored the famous paper on Differential Cryptanalysis. This is one of two ciphers Knudsen co-authored in the AES—see DEAL.

• **T**woFish - Counterpane (Schneier, Kelsey, Whiting, Wagner, Hall, Ferguson)

http://www.counterpane.com/twofish.html

$$AESCD1$$

$$AESCD2$$

- **B**ased loosely on BlowFish. B. Schneier we know from his seminal introductory work on cryptography “Applied Cryptography” and his authoring of the most widely analyzed private sector cipher BlowFish.

4 AES Round Two

The finalists are:

• **M**ARS

- **S**tands for: Multiplication Addition Rotation Subtraction. These are the primitive operations used by the cipher.
- **N**o surprise the cipher made it this far. Don Coppersmith and team have the longest track record and the distinction of designing the DES.

• **R**C6

$$AESCD3$$

- **N**o surprise here.
- **S**tands for: Ron’s Cipher number 6. Ron Rivest has written many ciphers the entire world uses daily.

Remember Distributed.net had a distributed effort to crack RC5? Well RC6 makes RC5 look easy to crack, and difficult to implement.

– **T**he frightening simplicity of the RC6 encryption operation can be summed up in 10 lines of ANSI-C code for a 32-bit computer:

```
rc6_encrypt() {
 {A,B,C,D} = plaintext
 B=B + S[0];
 D=D + S[0];
 for (i=0; i<r; i++) {
   t=ROL(B * (2*B + 1), 5);
   u=ROL(D * (2*D + 1), 5);
   A=ROL(A^t, u) + S[2*i  ];
   C=ROL(C^u, t) + S[2*i+1];
   {A,B,C,D}={B,C,D,A};
 }
 A=A + S[2*r + 2];
 C=C + S[2*r + 3];
}
```

And the decryption operation:

```
rc6_decrypt() {
 {A,B,C,D} = ciphertext
 C=C - S[2*r + 3];
 A=A - S[2*r + 2];
 for (i=r; 0<=i; i--) {
   {A,B,C,D} = {D,A,B,C};
   u=ROL(D * (2*D + 1), 5);
   t=ROL(B * (2*B + 1), 5);
   C=ROR(C - S[2*r +1],t)^u;
   A=ROR(A - S[2*r   ],u)^t;
 }
 D=D - S[1];
 B=B - S[0];
}
```

– **Now** don't go off and use this. This cipher is trademarked and patented! The AES process demanded the winning cipher be unencumbered by intellectual property restrictions in all world markets (US export laws don't count?). RSA Labs explained in their submission that if and only if RC6 were to be selected as the AES would they wave royalties, otherwise they only allow use of RC6 for research and educational purposes.

• **R**ijndael

$AESCD3$

– **T**he cipher name is a play on words and the author's names. If you're Flemish I'd like to hear the explanation. Many people can't pronounce the cipher properly and are happy they won so they can just call it "AES." A Canadian wrote to the authors early in the AES process and suggested renaming the cipher to "Bob."

– **U**nlike RC6 Rijndael was developed in academia. There were never any IP restrictions.

• **S**erpent

$AESCD3$

– **N**o surprise here.

– **T**he Linux encrypted file system loop back device had jumped the gun and chose Serpent as the cipher of choice. Newer versions of the encrypted file system support the Rijndael cipher.

• **T**woFish

$AESCD3$

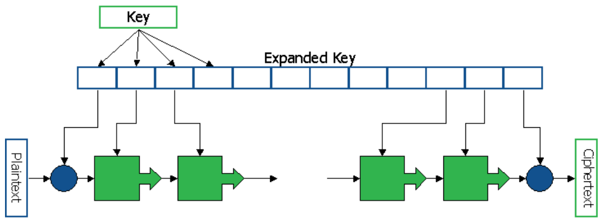


Figure 4: High level design of all AES finalists

- **N**o surprise here.
- **T**hose who know Bruce know he doesn't have a good chance of winning a seat on the United Nations if he ever chose to run. But still, cryptographers respect his abilities more than his tact and Two Fish made it this far on its own merits.

All five finalist algorithms were of excellent design. At a high level, they were all very similar.

- **E**mployed a strong key expansion algorithm
- **P**re- and post-whitening to protect the inner cipher rounds from "unfolding"
- **J**udicial combinations of linear and differential operations to thwart any differential or linier cryptanalysis
- **C**onstructed from sound mathematical principles

5 The Winner - Rijndael

A Flemish cipher chosen to be an American standard, what is the world coming to? After the last AES conference where the five finalists presented their closing comments, the NSA/NIST distributed a questionnaire:

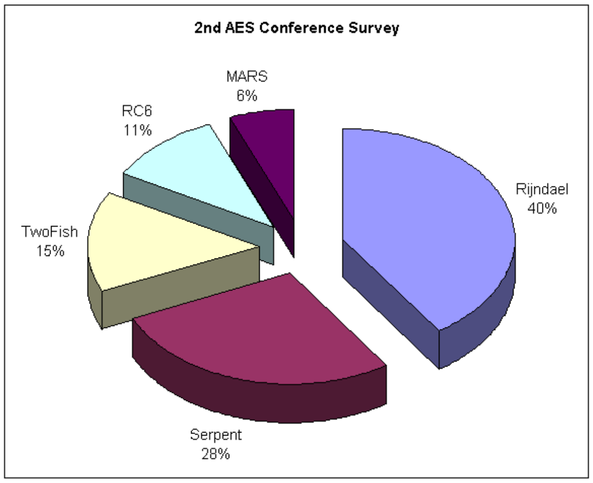


Figure 5: 2^{nd} AES Conference Survey

- ‘‘If only one algorithm were to be selected as the AES, which should it be?"
- ‘‘If a back-up algorithm were to be selected, which should it be?"

Results from question one showed a clear preference for Rijndael, Serpent coming in a distant second, and MARS, RC6 and TwoFish accumulating few votes combined than Serpent.

The four finalists were not as favoured to become the AES for several reasons.

- **R**C6's simplicity in software came at an unacceptable cost to hardware. In smartcards, efficient 32bit multiplication consumes far too much surface area.
- **M**ARS was a strong cipher, with a very complex structure that was not conducive to straightforward analysis. Like RC6, it also used 32bit multiplies.
- **S**erpent's popularity was justified, their design used the DES s-boxes so hotly contested for the past two decades. These S-boxes have been so heavily analyzed (significantly by one of the Serpent authors) that it would simply be unwise to create a

whole new net of S-boxes. And for you Ditributed.Net people, the cracking effort which brute forced a DES key in 22 hours used an optimization technique called "bit slicing." This technique performed 32 parallel 4x4 DES S-box substitutions in only a few instructions. Reducing each S-Box to a Karnough map of bit-wise operators and performing transformations on four 32bit words is how the optimization was accomplished. Effectively transforming a 32bit Single Instruction Single Data (SISD) processor into a Single Instruction Multiple Data (SIMD) processor. Serpent used this "bit slicing" technique on its 128bit (4 x 32bit) blocks. They overcame the speed limitation of DES by using the AES block size requirement and a modern optimization technique. Quite clever!

- **T**woFish's limitations lay in the extra overhead required for full speed optimization. A key and block dependent set of lookup tables are created at the start of the encryption/decryption operation.

Rijndael's design was very tight. Not as simple to implement in software as RC6, but the overall simplicity of its sub-components made it the clear favourite. Hardware implementations could be made so small, that two parallel implementations of the Rijndael algorithm can fit on a single 8-bit smart card!

5.1 One Plus One Equals Zero

The cipher achieves its small footprint and simplicity from its use of Galois Field (GF) theory. Also known as "primitive polynomials" or linier feedback shift registers (LFSR), arithmetic in GF requires a minimum of hardware resources—the principal motivation for their use in cell phone and network telecommunications.

To understand how GF works, start by forgetting everything you learned in grade school—for many of us this is easily done. Next, understand that arithmetic in GF is undefined unless you specify the field. In Rijndael this field is called $GF(2^8)$. Meaning, there are 256 possible values, each value represented by 8 bits, no 7 or 9 bit values exist in $GF(2^8)$.

There are several ways of representing values in GF below we demonstrate a few:

$$\begin{aligned} 01010101 &= '55' \\ &= x^6 + x^4 + x^2 + 1 \\ 10101010 &= 'AA" \\ &= x^7 + x^5 + x^3 + x \\ 11110001 &= 'F1' \\ &= x^7 + x^6 + x^4 + x^5 + 1 \end{aligned} \qquad (2)$$

We define the addition operation. In $GF(2^8)$ this is what we commonly know as the explosive-or (XOR) operation.

$$\begin{array}{rcrcr} C & = & A & + & B \\ 00000000 & = & 00000001 & + & 00000001 \\ 00010001 & = & 00010000 & + & 00000001 \end{array} \qquad (3)$$

Notice addition and subtraction are identical in this "new math."

$$\begin{array}{ccccccc} 1 & + & 1 & - & 1 & = & 1 \\ 1 & XOR & 1 & XOR & 1 & = & 1 \end{array} \qquad (4)$$

Next, we define the "multiply by x" operation. This is where things get a bit strange. To keep closure in $GF(2^8)$ we need to define a primitive polynomial (analogous to prime numbers in an integer field) to "divide" our result by to extract our "remainder."

Multiplying a $GF(2^8)$ polynomial by x is as simple as sifting the bits to the left by one.

$$\begin{array}{rcll} C & = & A & \bullet \; x \\ x^3+x^2+x & = & x^2+x+1 & \bullet \; x \\ 00001110 & = & 00000111 & \bullet \; 00000010 \\ '0D' & = & '07' & \bullet \; '02' \end{array} \qquad (5)$$

In the case where A's most significant bit (x^7) is high, we immediately know the value will no longer be in $GF(2^8)$. We perform a modulo operation with our primitive polynomial on this new C value to return it to $GF(2^8)$.

$$\begin{array}{rcl} P & = & x^8+x^4+x^3+x+1 \\ & = & 100011011 \\ & = & '11B' \end{array} \qquad (6)$$

Notice that '00' will always map itself back to '00' and only after 255 multiplications by '02' will a value return to its starting value. (Equations 6 and 7.)

Using Knuth's binary exponentiation technique where successive squaring of B are used to calculate a^b in $log_2(b)$ loops.

```
Knuth_modExp(a,b) {
  rslt = 1;
  while (b != 0) {
    if (b & 1) rslt = rslt * a;
    a = a * a;
  }
  return rslt;
}
```

Using this same technique of binary $GF(2^8)$ multiplication where successive multiplications by x are used to calculate $a \bullet b$.

```
xtime(x) {
  if (x & 0x80)
    return (x << 1) ^ 0x1b;
  else
    return (x << 1);
```

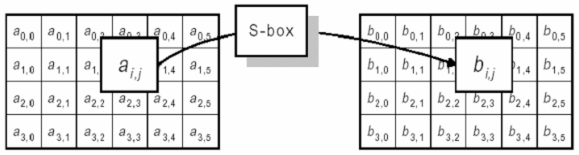


Figure 6: The Rijndael ByteSub Layer

```
}
gf8_mult(a,b) {
  rslt = 1;
  while (b != 0) {
    if (b & 1) rslt = rslt ^ a;
    a = xtime(a);
  }
  return rslt;
}
```

You now know how to do math in the crazy world of Galois Fields.

5.2 Inside Rijndael

The Rijndael algorithm's round function consists of 4 layers:

- **B**yteSub(data)
- **S**hiftRow(data)
- **M**ixColumn(data)
- **B**lendKey(data,exp)

5.2.1 ByteSub

This operation performs a byte level substitution. Unlike DES, this substitution is based on a single bijective transformation defined below. See Figure 6 and Equation 8.

5.2.2 ShiftRow

This layer does not alter the value of the data, it simply moves it about in preparation for later

$$\begin{array}{rcccll} C & = & A & \bullet & x & mod \;\; P \\ & = & 10000001 & \bullet & x & mod \;\; P \\ & = & '81' & \bullet & '02' & mod \;\; '11B' \\ & = & 100000010 & mod & 100011011 & \\ & = & 100000010 & - & 100011011 & \\ & = & 100000010 & XOR & 100011011 & \\ & = & 00011001 & & & \end{array} \tag{7}$$

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{8}$$

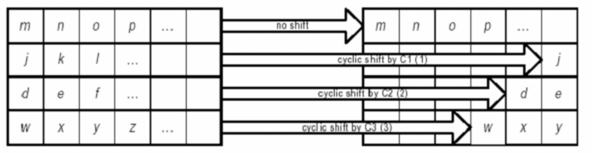


Figure 7: The Rijndael ShiftRow Layer

encryption rounds. It is required to have every bit of input effect every bit of output. See Figure 7.

5.2.3 MixColumn

This operation is a bit more complex. Defining a column as a polynomial of $GF(2^8)$ coefficients, a cross product of this value by a constant polynomial co-prime to x^4+1 is performed. What did I mean by that?

A column of Rijndael data contains 4 bytes. Each byte represents a polynomial in $GF(2^8)$. The column however represents a polynomial of polynomials. When two numbers are co-prime, they do not share any factors other than 1, this applies to all number fields, not just integers. The requirement of co-primality to x^4+1 is required to make this transformation invertible. Invertible operations are nice to have if you ever want to recover the data you're encrypting! See Figure 8 and Equations 9, 10, 11, 12, 13, and 14.

$$c(x) \;=\; '03'x^3 +' 01'x^2 +' 01'x^1 +' 02'x^0 \tag{9}$$

$$b(x) \;=\; c(x) \otimes a(x) \tag{10}$$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{11}$$

$$d(x) \;=\; '0B'x^3 +' 0D'x^2 +' 09'x^1 +' 0E'x^0 \tag{12}$$

$$b(x) \;=\; d(x) \otimes a(x) \tag{13}$$

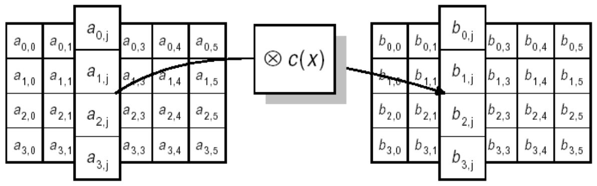


Figure 8: The Rijndael MixColumn Layer

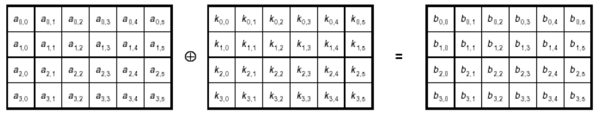


Figure 9: The Rijndael BlendKey Layer

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0E & 0D & 0B & 09 \\ 09 & 0E & 0D & 0B \\ 0B & 09 & 0E & 0D \\ 0D & 0B & 09 & 0E \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{14}$$

5.2.4 BlendKey

An XOR operation with the expanded key is performed on each byte in the cipher's block of data. See Figure 9 and Equation 15.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} \oplus \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{15}$$

5.3 The Sub-Round Function

These operations are combined to create the Rijndael sub-round transformation (see Equation 16). This sub-round operation is performed anywhere from four to eight times each round depending on the block size specified. The Round function is performed any where from 10 to 14 times depending on the key and block sizes specified.

5.4 Implementation

Efficient implementation of the Rijndael algorithm lies in efficient implementation of the Rijndael sub-round transformation. The sub-round transformation has several layers. The ByteSub layer is best implemented as a lookup table, the ShiftRow layer by cyclic array offsets and the BlendKey layer is a trivial matter.

The MixColumn operation requires a bit more thought. The following code segment demonstrates how to mix a single column. This operation would be performed several times though a single sub-round which is itself executed several times in a single round, which in turn is executed several times in a single block encryption operation. This procedure will net you a nice (!) $O(n^3)$ time complexity if implemented in serial!

```
void MixOneColumn(char a[4]) {
  char Tmp, T;
  Tmp = a[0] ^ a[1] ^ a[2] ^ a[3];
  T = a[0] ^ a[1]; T = xtime(T);
    a[0] ^= T ^ Tmp;
  T = a[1] ^ a[2]; T = xtime(T);
    a[1] ^= T ^ Tmp;
  T = a[2] ^ a[3]; T = xtime(T);
    a[2] ^= T ^ Tmp;
  T = a[3] ^ a[0]; T = xtime(T);
    a[3] ^= T ^ Tmp;
}
```

There is room for a significant increase in speed if the implementer is willing to sacrifice size. Collapsing the ByteSub and MixColumn operations into a single 8x32 lookup table and a little coaxing of the equations, the entire sub-round transformation can be reduced to four lookups, four 32bit XORs, and three cyclic bit rotations by 8.

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S\left[a_{0,j}\right] \\ S\left[a_{1,j-C1}\right] \\ S\left[a_{2,j-C2}\right] \\ S\left[a_{3,j-C3}\right] \end{bmatrix} \oplus \begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} \qquad (16)$$

$$T_0\left[v\right] = \begin{bmatrix} S\left[v\right] \bullet' 02' \\ S\left[v\right] \\ S\left[v\right] \\ S\left[v\right] \bullet' 03' \end{bmatrix} \qquad (17)$$

$$\begin{aligned} b_j = \quad & k_j \oplus T_0[a_{0,j}] \oplus \\ & ROR_8(T_0[a_{1,j-C1}] \oplus \\ & ROR_8(T_0[a_{2,j-C2}] \oplus \\ & ROR_8(T_0[a_{3,j-C3}]))) \end{aligned} \qquad (18)$$

At a cost of 4 kilobytes of lookup tables, this can be further reduced to four lookups and four 32bit XORs.

$$\begin{aligned} b_j = \quad & k_j \oplus T_0[a_{0,j}] \oplus T_1[a_{1,j-C1}] \\ & \oplus T_2[a_{2,j-C2}] \oplus T_3[a_{3,j-C3}] \end{aligned} \qquad (19)$$

At first glance, this seems far too simple a transformation to be secure. One is left searching for a backdoor but can find no place to hide it.

5.5 What did/can the NSA do?

Unlike the DES, the AES publication has no changes to the Rijndael algorithm other than limiting the block size from any one of 128, 192 or 256bits to 128bits. I have written another paper suggesting that block sizes and key sizes be identical or the possibility of multiple keys mapping one plaintext input to another ciphertext output would exist. Don Coppersmith took the time to explain there are circumstances where having a larger key to block size would be preferable.

The question remains, did the NSA/NIST do their job? Was the AES a success? The NSA's responsibility to monitor domestic and international communication would naturally lead to developing a backdoor into such a widely used algorithm.

However, the NSA also has the responsibility of protecting American interests for the public as well as private sectors. A strong algorithm would be conducive to one goal but not the other.

The winning block cipher design was not of American origin. It is strange how an American standard came from Belgium. Still, it would appear as though the best cipher won the day. Rijndael is fast (30Mbit/sec ANSI-C, P3 450), simple (see Equation 19), and efficient in hardware as well as software, and widely considered to be secure.

How are these the contradictions in the responsibilities of the NSA reconciled by the AES standard? The answer is simple; an impervious block cipher is insufficient to insure the total security of transmitted data.

There are three broad classes of digital encryption algorithms:

- **M**essage digest or hash algorithms such as MD5 and SHA-1 operate without the use of a key. These algorithms are 'blenders,' they reduce input data to a fixed length binary sequence. These sequences are constructed such that any minute change to the input significantly changes the output.
- **B**lock ciphers such as TripleDES and

AES operate with a single key. With this key, data is encrypted in such a way that it can only be decrypted with the same key.

- **P**ublic-key algorithms operate using two distinct but mathematically related keys. An operation performed by one key can only be undone by its complement. This facilitates the confidential exchange of small pieces of data and the authentication of data origin.

The laws of thermodynamics state the energy in a closed system remains constant. The connection of matter, energy, information and entropy are well understood. Extending these laws to the realm of system-level security as applied by cryptography one can come to three conclusions:

- **M**essage digest algorithms The laws permits the existence of a perfect message digest algorithm. A hash algorithm with no mountable attack other then brute force; in this case the parallel collision attack.
- **B**lock ciphers The laws permits the existence of a perfect block cipher. A cipher with no mountable attach other than brute force.
- **P**ublic-key algorithms The laws do not permit the existence of a perfect public-key algorithm. Looking purely at the flow of information required to conserve confidentiality of key pieces of data, the restrictions the law places on reality forbids such an algorithm to exist.

To demonstrate this we consider two scenarios, confidential exchange with a block cipher/shared secret and with a public-key.

- **B**lock Cipher
  Consider two closed systems A and B communicating over channel C using information k only known to A and B.
  - Assume a symmetric algorithm $c = E_k(d)$ exists with no possible cryptanalytic attack. That is, d cannot be recovered from c unless k is known.
  - System A opens and communicates $c = E_k(d)$ with $n = H(c)$ bits of information though C to B.
  - System C cannot recover d since only c is known.
  - System B can recover d since c and k are known.
  - B now obtains information d possessed by A and not C.
  - This information or entropy was communicated to B though channel C using entropy held only by A and B.

  We have not contradicted ourselves, thus we cannot disprove the existence of a perfect block cipher.

- **P**ublic-Key
  Now consider our two closed systems A and B with channel C communicating with no shared secret.
  - Assume a perfect public-key algorithm $c = P_{pub}(d), d = P_{pri}(c)$. That is, d cannot be recovered from c unless pri is known.
  - System A opens and communicates $c = P_{pub}(d)$ with $n = H(c)$ bits of information though C to B.
  - System C cannot recover d since only c and pub are known.
  - System B can recover d since c and pri are known.
  - B now obtains information d possessed by A and not C.

- This information or entropy was communicated to B though channel C using entropy common to all A, B and C.

Here we have our contradiction, A gave B more entropy than was ever transmitted though C and possessed by A. So this tells us at least one of our assumptions was flawed, either no entropy was transmitted or there can be no perfect public-key algorithm.

What does this mean for RSA, ElGamal, Diffie-Hellman, Elliptic curve systems, and other public-key algorithms? The strength of public-key algorithms stem from our ignorance of their underlying mathematics. Any amateur cryptographer could have told you that, this was just a loose formalization proving it.

Will quantum cryptography come to the rescue? Simply replacing our ignorance of mathematics with our ignorance of physics is not a lasting solution. The Standard Model of subatomic particles explains the communication of force (Gravity, Electro-weak and Strong nuclear forces) as an exchange of virtual particles.

Fixating a quantum-coupled photon will cause its complement to fixate in the opposite spin. The communication between these two photons is suspected to be in the form of some sort of virtual particle. The key in usurping the information between these two parties would lie in detecting the energy state of the virtual particles exchange between the two coupled photons.

It doesn't matter how many ways you skin Schrödinger's Cat. At the end of the day even his feline must obey the laws of thermodynamics.

6 Modes of Operation

Equally important to the good design of a block cipher is how it is used to encrypt data. In 1980 the NSA/NIST published a set of standard modes of operation for the DES.

The publication detailed 4 modes of operation. There are three characteristics that differ from each mode: the primitive data unit, the property of memory and the property of state. S e modes operate at the bit level, others at the block level. Some modes operate with a memory of all data previously encrypted and others are memoryless. Some modes operate with a state variable, which is altered after each encryption operation while others are completely stateless.

- **E**ncrypted Cipher Block (ECB)
  - **D**ata unit: block.
  - **M**emoryless: yes.
  - **S**tateless: yes.
- **C**ipher Block Chaining (CBC)
  - **D**ata unit: block.
  - **M**emoryless: no.
  - **S**tateless: no.
- **O**utput Feed Back (OFB)
  - **D**ata unit: bit.
  - **M**emoryless: yes.
  - **S**tateless: yes.
- **C**ipher Feed Back (CFB)
  - **D**ata unit: bit.
  - **M**emoryless: no.
  - **S**tateless: no.

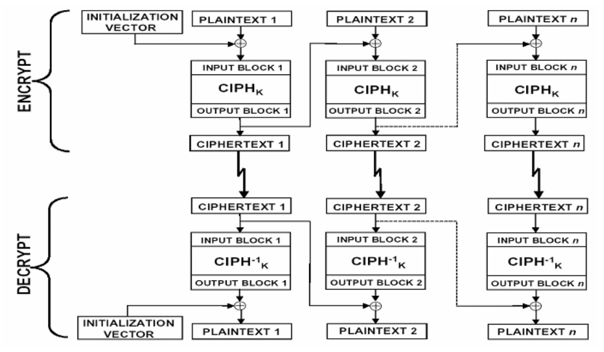


Figure 10: The CBC Mode of Operation

These modes of operation are sound and mathematically provable. However, security is not the only concern in today's cryptosystem deployments. There are serious performance restrictions when using the stronger CBC and CFB modes versus the ECB and OFB modes.

The memoryless modes of operation make parallelism possible. A stateless and memoryless cipher mode leaves an attacker with many opportunities for attacks that do not require breaking any encryption algorithms. The exchange was clear, security for performance.

The CBC mode of operation is the most commonly used, in file and network encryption. The CFB mode is commonly used by network encryption protocols such as SSH where transmitting an entire 128bit block to communicate a single byte of data would be wasteful.

The author of this paper has another publication where he recommends a 'tweak' to the classic CBC mode of operation. The Tweaked-CBC mode proposed reduces the format parsing and API requirements by implicitly encrypting the CBC initialization vector (IV) in the ciphertext payload. While the decryption operation will implicitly assign the IV after the first block is processed and discarded.

The core requirements of the AES were high security and high throughput. A new mode of operation was needed to accommodate the private sector's security and speed requirements.

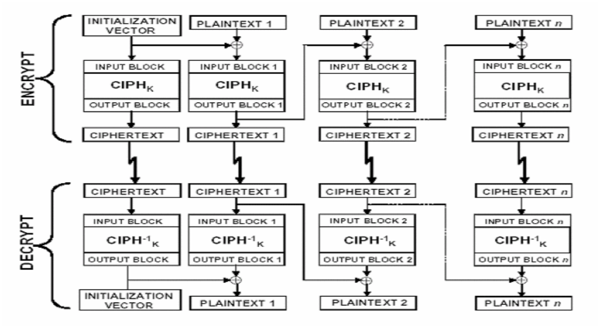


Figure 11: The Tweaked CBC Mode of Operation

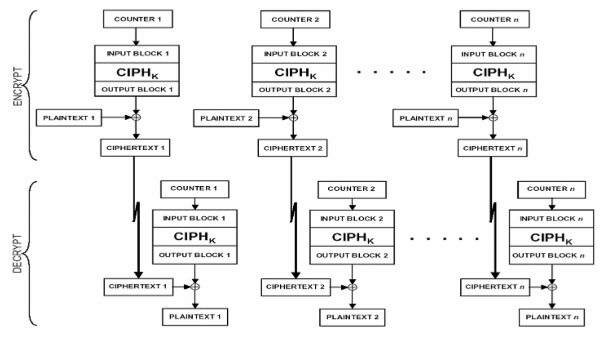


Figure 12: The CTR Mode of Operation

The counter (CTR) mode of operation—designed by Diffie and Hellman in 1979—provides protection from the kinds of attacks mountable against ECB and OFB modes as CBC does, but with high parallelism.

The NSA/NIST has also made known their intention of standardizing on another mode of operation to be used not for confidentiality but for authentication. Message Authentication Codes (MAC) exist today using the memory/state based modes of operation mentioned above. However, confidentiality and authentication are always viewed as orthogonal to each other. One should not assume authenticity when dealing with confidentiality and visa versa.

7 Message Digest Algorithms

Message digest algorithms have followed a history of their own. Hash algorithms are susceptible to a statistical attack known as the Birthday Paradox.

How many people do you need in room before the probability of two people having the same birthday is over 50%? The answer is 20.

Reword the question to “If two random 128bit values are being generated in parallel, how many 128bit number generations are required before the probability of two values matching?” The answer is $2^{\frac{128}{2}}$ or 2^{64}.

In 1994, van Oorschot and Weiner published a design for an MD5 collision machine that could produce a collision in less than 30 days at a cost of $10M. Assuming that Moore’s law was obeyed from 1994 to 2002 (which is in fact a conservative assumption), the cost of such a machine by the time this paper was written was less than $200,000. Suffice it to say, 128bit message digest algorithms should no longer be considered cryptographically secure.

The digest size of MD5 was not the only weakness in it design. The NSA in its Digital Signature Standard (DSS) published what it called the Secure Hash Algorithm (SHA). SHA was highly criticized by the private sector and academia, so an enhanced SHA-1 was published in its place.

SHA-1 digest size was 160bit, $2^{\frac{160-128}{2}}$ or 65,536 times more secure than MD5’s 128bit digest. Also, the SHA-1 algorithm was constructed in such a way that bit input into the algorithm effected every possible output digest bit. This is not a characteristic of MD5. For this reason, research into digest algorithms has stagnated. The private sector feels the NSA has done as good a job as conceivably possible.

With the publication of the AES however, a 160bit hash algorithm with an effective strength of 80bits is mismatched with the 128, 192 and 256 bit key strengths of AES. The NSA has stepped up and published new algorithms to SHA family in a draft processing standard. These algorithms are named SHA-256, SHA-384 and SHA-512 with 256, 384 and 512bit digests and effective strengths of 128, 192, 256bits respectively. These hash algorithms posses effective strengths equal to the AES key sizes.

The SHA-384 algorithm is simply the SHA-512 algorithm with a truncated digest. Many of the SHA-512 core operations are 64bit addition, 64bit rotation and 64bit shifts. The SHA-512 and SHA-384 were not designed with software implementations on 32bit machines in mind.

8 Summary

The Flemish Rijndael block cipher has been chosen as the Advanced Encryption Standard out of an international group of 15 modern algorithms obsoleting the decades old Data Encryption Standard. The AES can be heavily optimized for speed or size in either hardware or software forms. The cipher represents the state-of-the-art in private sector cryptography. This possibility of backdoors for government agencies is negligible due to the simplistic design of the AES.

There are five approved modes of operation for the AES, 4 were adopted from the DES. The new mode of operation is called CTR and is highly parallelizable. Another mode of operation for authentication is still to be announced.

The new Secure Hash Algorithms support 256, 384 and 512bit digests, taking the Birthday Paradox into account their effective strengths are 128, 192 and 256 bits respectively. The new

Secure Hash Algorithms represent the state-of-the-art in message digest algorithms.

References

[CertKeyRes] CertainKey Online Resources, `http://www.certainkey.com/resources/`

[CertKeyOLS2002] CertainKey At OLS 2002, `http://www.certainkey.com/ols2002/`, (2002)

[Cooke2001] Functionally Equivalent Keys in the Advanced Encryption Standard, `http://jlcooke.ca/aes/aes_fek.pdf`, (2001)

[Cooke2001b] Plaintext Dependency of Functionally Equivalent Keys in the Advanced Encryption Standard, `http://jlcooke.ca/aes/aes_fek2.pdf`, (2001)

[Thermo] *Wolfram Research: World of Physics Online Reference*, `http://scienceworld.wolfram.com/physics/ThermodynamicLaws.html`

[NISTDES] *The Data Encryption Standard*, `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`, (1976-1999)

[NISTMODEOP] *DES Modes of Operation*, `http://www.itl.nist.gov/fipspubs/fip81.htm`, (1980)

[NISTAES] *The Advanced Encryption Standard*, `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, (2001)

[NISTAESMODEOPS] *Recommendation for Block Cipher Modes of Operation*, `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`, (2001)

[NISTAESWWW] *The Advanced Encryption Standard Website*, `http://www.nist.gov/aes/`, (a)

[MD5] *The MD5 Message Digest Algorithm*, `http://www.faqs.org/rfcs/rfc1321.html`, (1992)

[NISTSHA1] *The Secure Hash Standard*, `http://www.itl.nist.gov/fipspubs/fip180-1.htm`, (1995)

[NISTSHA2] *The Secure Hash Standard*, `http://csrc.nist.gov/encryption/shs/dfips-180-2.pdf`, (2001)

[NISTDSS] *The Digital Signature Standard*, `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf`, (2000)

[CAST128] *The CAST-128 Encryption Algorithm*, `http://www.faqs.org/rfcs/rfc2144.html`, (1997)

[CAST256] *The CAST-256 Encryption Algorithm*, `http://www.faqs.org/rfcs/rfc2612.html`, (1999)

[AESCD1] *AES CD-1: Documentation*

[AESCD2] *AES CD-2: Source Code*

[AESCD3] *AES CD-3: Finalists*

[DC] *Differential Cryptanalysis of the Data Encryption Standard*, ISBN-0387979301, Shamir Biham, (1994)

[DEAL] *A 128-bit Block Cipher*, `http://www.ii.uib.no/~larsr/newblock.html`, (1998)

[E2] *The 1280bit Block Cipher E2*, `http://info.isl.ntt.co.jp/e2/`, (1999)

[HPC] *The Hasty Pudding Cipher*, `http://www.cs.arizona.edu/˜rcs/hpc/`, (1998)

[LOKI97] *The LOKI97 Block Cipher*, `http://www.unsw.adfa.edu.au/˜lpb/research/loki97/`, (1997)

[MARS] *MARS - a candidate cipher for AES*, `http://www.research.ibm.com/security/mars.html`, (1999)

[RC6] *RC6 Block Cipher*, `http://www.rsasecurity.com/rsalabs/aes/`, (1998)

[Rijndael] *The Block Cipher Rijndael*, `http://www.esat.kuleuven.ac.be/˜rijmen/rijndael/`, (1999)

[Serpent] *Serpent*, `http://www.cl.cam.ac.uk/˜rja14/serpent.html`, (1998)

[TwoFish] *TwoFish: A 128-bit Block Cipher*, `http://www.counterpane.com/twofish.html`, (1998)

System Installation Suite
Massive Installation for Linux

Sean Dague
japh@us.ibm.com

Abstract

The first hurdle that a user or administrator must overcome when migrating to Linux is the installation. In the not so distant past, this was a near Herculean task. Today, with a myriad of Linux distributions available, many focusing on the end user experience, installation of a single machine has become much easier. In some instances it is even Mom proof. This has given rise to a new issue, however, as these methods of installation tend to be distribution specific, and tend to have a single machine view of the world.

System Installation Suite attempts to solve the massive installation problem, i.e. how does an administrator handle installation and maintenance of hundreds or thousands of nodes at once, in Linux. The solution is agnostic of Linux distribution and architecture, and presents a uniform interface on every Linux platform. It does this through the creation of installation images, which are built on a centralized server somewhere. These images are then deployed over the network to client machines. The use of installation images, which are in fact fully instantiated Linux systems stored on an image server, gives rise to some interesting possibilities for system management and maintenance. The design process that went into System Installation Suite, and the possibilities that it provides for will be discussed further in this paper.

1 Background

System Installation Suite is a collaboration between two different open source massive installation tools, SystemImager and LUI (the Linux Utility for Cluster Installation). The design of System Installation Suite came largely from harvesting the strengths of both of these tools, while attempting to leave their short comings behind. It is appropriate that we explore some of the strengths and weaknesses of both LUI and SystemImager before delving into the design of System Installation Suite as a whole.

1.1 LUI - Linux Utility for Cluster Installation

The Linux Utility for Cluster Installation (LUI) was one of the first Open Source projects contributed by the IBM Linux Technology Center. The project was started by Rich Ferri to mature the state of Linux clustering. LUI version 1.0 was released to the world in April of 2000 under the GNU Public License.

LUI is a resource based cluster installation tool, conceptually based on NIM (Network Install Manager), the network installer for AIX. In LUI everything was driven by resources. LUI resources included: the list of packages (RPMs) that will be installed on a client, tarballs that would be expanded on the client, disk partition tables that would be used to setup the disks, custom kernels and ramdisks, post install scripts, or single files that would be propagated. A combination of these resources would fully describe the final makeup of a client.

Resources were first abstractly defined in the LUI database. Then clients were abstractly defined in the LUI database. Finally resources were assigned to clients. LUI also supported arbitrary grouping, so resource and client groups could be defined, and the allocation of resource groups to client groups could be utilized.

LUI installation required nodes with network interface cards that could network boot. For clients which did not have network bootable NICs, an floppy from the etherboot project could be made to simulate this process. The network booted kernel had a remote NFS root on the LUI server, and the installation logic was drive by a **clone** program contained within the NFS root.

LUI had many weak spots where things would often break down. The first issue was the reliance on PXE, TFTP, and NFS v2 which are not entirely reliable, secure, or scalable protocols. [1] When network booting worked properly, it was fantastic, when it failed, it was often extremely difficult to debug the failure. This was especially true due to the fact that there are various versions of the PXE standard which behaved slightly differently.

The second major issue was the timing of client instantiation. All the resources were instantiate into a working client machine on the client when running from the network booted kernel and NFS root. Although some sanity checks were run on the resources before they were allowed to be registered, many checks were either too expensive or too complex to be run. The most common failure was having an inconsistent list of RPMs, i.e. one which did not properly satisfy all package dependencies. By the time such an error was detected (during client installation), it was too late to recover gracefully. In a best case scenario, the machine had remote console access to debug the issue. In the more common case, the machine was hung in the middle of an installation, and a monitor and keyboard had to be wheeled over to the node to examine the failure.

[1] Since the time of LUI's introduction, both NFS v3 and more robust implementations of tftp (such as atftpd) have become available on Linux

The final issue with LUI was overly complicated with its resource model. Once a user understood all the possible resources, how they related, and which ones were really required to bring up a machine, it was great. However this learning curve was often rather steep.

Many of these issues were being looked at for a LUI 2.0 redesign during the spring of 2001. However the interaction with the SystemImager project made the redesign take an entirely different direction.

1.2 SystemImager

SystemImager is a project which was started by Brian Elliot Finley. Its first incarnation was under the name Pterodactyl, where it was a set off programs designed to replicate Solaris installations. It later became a far more robust product firmly rooted in Linux exclusively. SystemImager 1.0 was released in May of 2000 under the GNU Public License.

SystemImager, as the name implies, is an image based installation and maintenance tool. Unlike many other image based tools, SystemImager images are actually full live file systems which exist on the image server. These images are captured from a running machine that has been properly prepared, also known as a **golden client**. The image consists of the entire client file system, and is stored in a directly under the **/var/lib/systemimager/images** tree on the image server. These images can later be deployed to other client machines using the SystemImager autoinstallation process.

The autoinstall process for SystemImager is different from that of LUI. Instead of network booting just a kernel, and using a remote NFS root to drive installation, SystemImager uses an embedded Linux, BOEL (Brian's Own Embedded Linux), to control the installation. The kernel and initial ramdisk for BOEL fit on a floppy, cd, or can be served from the network for machines that support network booting. Once BOEL has brought the client machine onto the network, it reconnects to the image server and fetches an autoinstall shell script. The autoinstall script drives the remainder of the installation. All file transfer between client and server is done using **rsync** [2], which provides a mechanism for remote file synchronization.

The use of live file systems on the server and **rsync** to transfer these file in SystemImager allows for a number of additional features. Because **rsync** can use a number of different underlying file transfer methods, the installation can happen over a secure ssh connection. This allows for unattended installation where client and server are separated by significant physical distance and the only route between them is a public, insecure network. Because images are replicated at a file level and not a package level, the process of file transfer is inherently distribution agnostic. Software on the node needs not have come from a distribution package (RPM, Deb, etc.), but may have been installed from source or binary tarball.

However, SystemImager is not an stand alone installation tool. It doesn't solve the "first node" issue. When using SystemImager for installation, one must first build the golden client using some native installation method. Only after the node is fully installed and configured manually can the image be captured. Before System Installation Suite, SystemImager also did only very minimal customizations to the images after installation. This meant that simple hardware differences like different models of network cards in client machines could require separate images on the server.

[2]http://rsync.samba.org

1.3 System Installation Suite

System Installation Suite takes the best parts of SystemImager and LUI and adds other features that neither of them previously had. The LUI project morphed into a project called SystemInstaller, which uses a much simpler model for instantiating a machine from a set of packages and disk partition files. In System Installation Suite, SystemImager stays largely unchanged from a user perspective, however some of the internals have changed to allow interaction with SystemInstaller, and all of the image customization code was removed from the autoinstallation process and replaced with calls to System Configurator. System Configurator is the third major component of System Installation Suite, a uniform configuration API for installation. It supports features such as network setup, boot loader setup, and ramdisk creation. System Configurator also makes images deployed under System Installation Suite more generic, and applicable to a wider range of hardware.

2 SystemInstaller

SystemInstaller is the image build tool for System Installation Suite. It is very similar to LUI in function, however instead of building clients directly, SystemInstaller builds images on the image server. The images are made to look just as if they were harvested from a live machine that was installed by a native installer.

The design of SystemInstaller came from many lessons learned during the LUI project. The SystemInstaller team attempted to keep all the strengths of LUI without retaining any of the weaknesses.

2.1 SystemInstaller Architecture

The biggest hurdle that any LUI user had to overcome was understanding all the possible resources that existed, and how they interacted with each other. In many instances resources were either co-dependent, redundant, or could have been auto detected and allocated during installation. An example of just such and instance was the ramdisk resource.

Most modern distribution kernels are built extremely modularly. This makes it very easy to use the same kernel on many different types of hardware by only changing the modules.conf file. If the root filesystem of the machine is on a device that needs one or more of these modules to access it, an initial ramdisk needs to be built to support this. LUI did not support the automatic create of an initial ramdisk, and required the user to create, and allocate one manually. If the creation of the initial ramdisk took place on a system running an smp kernel, but was attempted to be used on a up kernel system, the boot would fail. This was a common occurrence for beginner LUI users. Hence one of the early goals of SystemInstaller was to make the resource model much simpler, or even totally transparent to the user.

Even though LUI had weaknesses, it also had many strengths that other install tools did not have. Chief among these was the LUI database. All of the user facing LUI commands did nothing more then add the appropriate meta data into the LUI database. During installation, the **clone** script fetched data from the LUI database and used it to instantiate the resources on the client machines. After installation was done, LUI no longer needed this data at all. However, a decision was made that the data should be persistent, and an API created to access it, in the hopes that some other application might find it useful.

This is exactly what happened. When the OSCAR Clustering project was looking for an installation tool, they choose LUI because it had a cluster database already. OSCAR could just ride on top of LUI's database, and add extra information if required. As OSCAR was one of LUI's biggest users, when looking to replace LUI with System Installation Suite, the concept of a cluster database was a requirement that had to be kept.

With these two key points: simplify the resource model, and provide a cluster database, design on SystemInstaller began. SystemInstaller retained a flat file database, as LUI had, as it was still felt that requiring an SQL database was too significant an overhead for an installation tool. Although this did mean sacrificing some functionality, and the ability for remote access to the data store, it did mean the prerequisites were far lighter. The intent was always to move towards a model where various data stores (flat file, SQL, or even LDAP) could be accessed transparently. However, this goal was a bit beyond the scope of the initial release. In addition to the data store, many convenience functions were defined and exported so that products such as OSCAR would have a clean interface to interacting with System Installation Suite.

SystemInstaller has a far more simple command line interface than LUI had. Only the package list and disk partition table resources were retained. This was possible because the image could be tweaked after initial creation but before deployment manually. The addition of custom kernels, or hand compiled software could be handled at this stage, and did not need to be incorporated into the early SystemInstaller interface. This reduced complexity makes System Installation Suite much easier to use out of the box than LUI ever was.

2.2 SystemInstaller Interface

The SystemInstaller interface consists of a number of non-interactive command line utilities, an interactive **buildimage** program, and a graphical user interface, **tksis** which uses the Perl Tk bindings. Both buildimage and tksis are built on top of the command line interfaces of SystemInstaller and SystemImager. This separation between the functional interface and user interface was extremely important in making it possible for other user interfaces to be built on top of the command line utilities.

There are four extremely important commands in SystemInstaller that do all the real work for creating images and client definitions. These commands are as follows:

- mksiimage - Build an image from a list of packages
- mksidisk - Add disk partition information to an image
- mksimachine - Update, Delete, or Show a machine definition
- mksirange - Create a group of machine entries

The **mksiimage** , **mksidisk** , and **mksimachine** commands are entirely contained within the SystemInstaller package. The **mksirange** command is a wrapper on top of SystemImager's **addclients** command, which also inserts the client definitions into the System Installation Suite database.

Most SystemInstaller commands support the following actions: add (-A), delete (-D), and list (-L). The major exception is the **mksimachine** and **mksirange** combination of commands. This limitation is currently based on limitations of the **addclients** command, specifically its inability to add a single machine definition of arbitrary name.

All of the commands in SystemInstaller serve both manipulate the System Installation Suite database, and perform the actual instantiation of their commands. Because of this, failures in image and client instantiation are immediately returned to the user. The full advantages of this will be discussed later.

3 SystemImager Redux

In order to initially integrate SystemImager into System Installation Suite, only minor enhancements were needed. Many of these enhancements were generally applicable to SystemImager outside of the scope of System Installation Suite, and hence seen as merely additional function to existing SystemImager users.

SystemImager logically breaks up into three components: server, client, and autoinstallation. With the version 2.0 release of SystemImager, which integrated with the other components of System Installation Suite, only elements of the server component were changed. The following is a brief overview of merely the changes in those components. For further information on SystemImager design, please refer to the SystemImager manual.

3.1 SystemImager Server Changes

Before System Installation Suite, SystemImager was a complete product which consisted of a number of command line utilities. Principle among these were two commands that did most of the work on the server.

- getimage - captures images from a prepared client, and creates autoinstall script
- addclients - adds a range of client definitions associated with an image

In order to accommodate SystemInstaller and System Installation Suite the following changes were made:

The **getimage** command was broken into two separate commands. The new **getimage** was responsible only for harvesting the image from a client. This primarily involved rsyncing the entire contents of the golden client node into a directory within /var/lib/systemimager/images. An additional **mkautoinstallscript** command now became the interface for generating the autoinstallation script. This functional separation was needed by SystemInstaller, as its mksiimage and mksidisk commands performed the functional equivalent of getimage. This functional separation also enabled the ability to regenerate autoinstall scripts from within SystemImager. Prior to this change, users had to utilize a documented hack of pointing getimage against the localhost interface.

The second major change was the addition of a non interactive mode to **addclients**. Prior to SIS, addclients could only be run in an interactive mode. Although this provided a simple console user interface that was easy to understand, it hindered the ability for other interfaces to sit above the SystemImager commands. It would have been relatively easy to duplicate the function of addclients within SystemInstaller, however it was it was still felt that it would be extremely beneficial if the exact same math was used to calculate the list of clients irrespective of the interface the user was utilizing. As SystemImager's logic for this existed solely inside **addclients** it was simplest to wrap the **addclients** command from **mksirange**.

In addition to these command line interface changes, the beginnings of a SystemImager library was started. A number of functions were added to this library during the evolution of SystemImager 2.0 which allowed SystemInstaller to add entries to the SystemImager rsyncd.conf file reliably. In SystemImager, this process is handled by the getimage command. When building an image with SystemInstaller, getimage is never called, so this aspect of the SystemImager interface could not be used.

4 Image Deployment

Once an image is either built with SystemInstaller, or harvested with SystemImager, it resides on an image server, generally in /var/lib/systemimager/images. This image is then ready to be deployed to client machines. Although most of the logic for image deployment in System Installation Suite remains unchanged from SystemImager, it is significantly different from most package based install mechanisms that it is worthy of discussion. The differences in image based deployment over package based deployment lead to a number of interesting pros and cons of the two methodologies.

4.1 BOEL

The engine for the autoinstallation process is BOEL (Brian's Own Embedded Linux). BOEL consists of a monolithic Linux Kernel 2.2 and a ramdisk containing Glibc 2.1, BusyBox 0.60.0, rsync 2.4.6, sfdisk, and a number of other utilities. BOEL's entire job is to bring the client machine to a state where it can remotely access the image server, and then it hands off its job to the autoinstallation script. BOEL is based on Tom's Root Boot distribution. More details about BOEL can be found in a recent article in Embedded Linux Journal.

BOEL can be booted from a number of media, including floppy disk, cdrom, network boot, or even the local hard drive in the special case of an autoinstallation update of a client. When booted from a floppy disk or local hard drive, BOEL will attempt to access a local configuration file. This file can contain information about things such as the ip address of the client, the default gateway, and image server ip address. The local configuration file is very useful when doing installations on a network where the installation administrator does not have control over the site dhcp server. If the local configuration file is not available, a dhcp client is used to activate the network devices during boot.

Once the network device is configured, BOEL will attempt to determine which image server it should contact. This information may either be provided via local configuration file or dhcp options. After that, BOEL attempts to determine the host name of the machine it is running on. This is accomplished through either values in the local configuration file, reverse DNS lookup, or a special hosts file served from the image server.

With the network enabled, and the image server and local host name found, BOEL connects to the image server using rsync, and retrieves the autoinstall script for the host. This script is stored in the **scripts** rsync module as the file HOSTNAME.sh. At this point BOEL turns over control to the autoinstall script that it has downloaded.

4.2 The Auto Install Script

BOEL is an extremely constrained environment. It contains only a minimal glibc, a statically linked ash shell, and the BusyBox implementation of standard Linux commands. This means the autoinstall script must be POSIX shell. The tasks the autoinstall script must accomplish are as follows:

1. partition the disk drives with sfdisk
2. format all the partitions appropriately (ext2, ext3, and reiserfs are supported)
3. mount all the partitions to the proper mount points
4. rsync the appropriate rsync module from the server to the local disk
5. run systemconfigurator to setup network, modules.conf, and bootloader
6. execute a specified post install action (one of: beep, reboot, or shutdown)

If at any point the auto installation attempts an operation which fails, it will dump its console out to a shell and await human input. At this point no remote notification is provided in the event of failure.

Because the autoinstall script is merely a POSIX shell script, it can be easily modified to perform other actions beyond the straight scope of the autoinstallation process. The **mkautoinstallscript** command should be considered to generate a template autoinstall script. Although it will work fine for most scenarios without any modification, the possibility exists to easily modify it to add extra function.

4.3 Image Customization

Once the files from the image are transfered to the client machine, the job of installation is nearly complete. The only thing that remains is modifying the abstract image so that it has node specific information in it. For a machine to actually boot and connect to the local area network the network scripts must properly reflect the state of the node, and the bootloader must be installed. Setting up networking is something which tends to be very distribution specific. Making a machine bootable is very architecture specific. Instead of forcing that code into the autoinstall script, where one only has access to the POSIX shell environment, a different approach was taken.

System Installation Suite installs all the software in the image to the client machine. The client machine is a full instantiation of a runnable node. It has all the C, Perl, and Python libraries that a running machine would have. Why not exploit this fact by installing an additional program in the image or on the golden client which is transferred to the client during installation. This program would be called via the **chroot** function, and would present a unified API for configuration of networking, bootloader setup, and other required tasks to the autoinstall script, which would be exactly the same on any architecture or distribution. This program became known as the System Configurator project.

System Configurator implements a unified calling interface to setup both network scripts and boot loaders. In the process of setting up these features, it also can detect local hardware and modify the appropriate underlying files accordingly. This feature is even exploited from SystemImager outside the scope of System Installation Suite. This allows an image to be used on machines which have different network interface cards. Prior to SIS, this was not possible. System Configurator will also auto generate initial ramdisks upon request. This solves the long standing issue with LUI where one had to create an appropriate initial ramdisk if attempting to install with a modular kernel on SCSI hardware.

4.4 Foot Printing

There were many possible ways that System Configurator could have implemented its abstraction. One that was suggested, and firmly rejected, was classifying features in terms of distributions. The problem is to support a dozen or so distributions, over 3 or more releases, means 40+ code paths. Also, significant updates between stable releases of a distribution would be nearly impossible to track or support. The eventual design concept that won was "foot printing".

Let's say you know there are two different programs to create ramdisks, and each of them takes different options. You could either first try to find a comprehensive list of all Linux distributions that use one or the other, and then detect distribution, and use the right one (as stored in your matrix), or you could just say "If program a is there, run it like this, if program b, run it like that". This takes out a whole lot of indirection in detection, and provides for the possibility of supporting distributions that you didn't even know existed.

The idea of foot printing naturally leads to a modular architecture. Each module registers itself for a specific type of job (Network, Bootloader, Hardware, etc.), and when the phase for that job is executed, the modules are asked if their footprint is found. If so, their setup routine is executed. Depending on the type of job being accomplished, it may either be ok to execute all modules which footprint properly, or only the first one to do so. This decision is made per task module.

4.5 System Configurator

System Configurator can do many tasks, and does different tasks when used in a System Installation Suite context or a SystemImager only context. The basic setup directives include support for the following:

- pci hardware detection
- network setup
- initial ramdisk generation
- bootloader configuration file generation
- bootloader setup (running the proper bootloader)
- time zone setup
- network time sync

At every stage the main line code uses footprinting and plugin modules to accomplish tasks. Whenever possible System Configurator calls native setup tools for the distribution it is running on. The attempt is to make it very hard to determine that System Configurator created or modified files, as it did exactly what a native user or tool in the distribution environment would do.

A good example of this is the creation of modules.conf on the Debian distribution. In Debian there is a directory of files in /etc/modutils which are all merged into modules.conf using the **update-modules** command. System Configurator modifies the appropriate files and runs update-modules if the /etc/modutils directory is found.

In the current release of System Configurator 4 different types of networking are supported, which provides support for Red Hat, Mandrake, Conectiva, SuSE, TurboLinux, and Debian. An additional 2 types of networking have been identified that would add support for Caldera and Slackware, but haven't been implemented yet. System Configurator can support as many network adapters as your Linux system can handle, though SystemImager and System Installation Suite only support configuration of the primary network adapter at this time.

System Configurator also supports four different methods for bootloader setup, Lilo and Grub on i386, and two ways to setup Elilo on IA64. There is experimental support for PPC and PA-RISC setup at the moment, though the autoinstallation process does not yet support either of those platforms.

After System Configurator runs, the abstract image has become a real live node, tuned for the distribution and hardware that it is running on. This node will be able to reboot and become network accessible. Once the machine is network accessible, any additional custom setup could be performed by remote shell commands or programs such as cfengine.

5 Maintenance

Images are live file systems stored on an image server. Images get transfered across the network via rsync. Before rsync transfers files it first computes the difference between the source and destination for the files. One image can be applied to many machines.

All these facts taken together paint a picture of how SIS provides an extremely efficient update mechanism for client nodes. Suppose that some core library to your system has a security vulnerability, for instance zlib. Pushing this update to all your running machines is as simple as applying the update to the image, then rsyncing that image back out to the client nodes. In most cases updates of a running node can be performed without a reboot, the notable exceptions being an update to a kernel or commonly used shared library with a security vulnerability.

SystemImager provides an interface to performing this update process via the **update-client** command. Updateclient is intelligent enough to exclude many directories that are used for variable or node specific information, such as /var/log, /var/run/, /var/spool, /tmp, /proc and ext3 journal files. This list of excludes is stored in a file on the client, so it can be tailored to meet site specific needs.

Because the rsync protocol computes the difference between the source and destination file systems, only those files that have changed get propagated. This reduces network traffic significantly. Rsync can even check for changed byte ranges within a large file, so that it can replicate on a sub file level.

6 Image vs. Client Instantiation

As has been shown in this paper, System Installation Suite takes a novel approach to installation. Most other tools used for unattended massive installation pull packages across the network, and instantiate them directly on the client during install. System Installation Suite does this instantiation on the server, then transfers the resultant image across the network. There are many advantages to this methodology. Maintenance mode, and support for non packaged software have previously been discussed. However there are many other advantages, some of which are still not fully exploited, to this approach.

All installation methods must have some code running on the client node to perform the installation. With SIS, all the code run on the client during install is SIS code. Packages are not allowed to run their own scripts during the portion of installation which occurs on the client. This means there are far less moving parts during the SIS install process, and hence less things that can go wrong with the install. In a package based installation, one bad package can prevent the install from working. With SIS, the one package can be manually force fit into the image where the user has far greater latitude (tools like alien might even be used to install non native packages). The SIS install process doesn't care where the content of the image came from. This reduced complexity during actual installation of the clients translates into a smaller number of problems that can occur during the autoinstall phase. This is true in theory, and in has been shown in practice as well.

One of the things that image installation does not do as well as package based install tools, is conserve disk space. In a package based installation environment the main server stores only packages. When the client installs, it will determine which combination of package it needs to complete installation, and fetch only those packages it needs to complete the process. The space required on the server is all the packages off the distribution CDs, plus any additional update packages. For most distribution releases this will amount to about 3 GiB of disk space per distro, per release, per architecture.

With image based installation the images are fully instantiated and stored on the server. An average image with a full load of software ranges from 1 to 2 GiB of data. There have been thoughts about providing an image normalization tool that would reduce the space required to store multiple images on the server, or to support multiple phase images, where many different images would be overlayed to create the final installation. Neither of these options are being seriously explored at this point because of one important fact: a 60 GiB EIDE drive costs less than $100. Disk space is cheap. Adding complexity to the project to save storage space requirements does not seem like a valuable use of developer resource.

The final advantage that System Installation

Suite's image implementation provides is the ability to live test an image on the server before deployment. The image is a live file system. Any user level program run chrooted from an image will run just as if it was a live machine. If you want to know whether an application will run properly in your image, you can chroot $IMAGEDIR $CMD.

This methodology was used when adding SuSE support to SystemInstaller. I harvested a SuSE 7.2 system onto my Mandrake 8.1 development machine. I then chrooted into the SuSE image and built additional SuSE images from inside it. This meant I could develop for multiple distributions on a single machine without having to reboot. This feature of SystemInstaller has begun to be exploited by a number of users that wish to create test and build environments for many distributions, but have a limited number of physical machines. The only limitation to this is testing software which needs access to physical hardware or kernel interfaces, as the host kernel will be used for that. There is a possible way around this limitation, discussed in the next section.

7 Future Work

SIS right now is at the very beginning of its life. There are a number of short comings that it has, and many directions it can go from here. What follows is a few of those thoughts, some of which are very pie in the sky, and some that will probably make it into the code stream by the end of this year.

7.1 Current Weaknesses

SIS has a number of weaknesses currently. The first major one is the fact that images currently contain more than just the software that is applied to the image. Because images also contain files like /etc/raidtab, and /etc/fstab, an image is bound to a partition model. This means that an image build for /dev/hda, cannot be applied to a machine with only SCSI devices. As most software doesn't care about the underlying disk devices, this should be able to be extracted from the main image and put into the autoinstall script.

The lack of multiple adapter support is also a big weakness. SIS currently only will set up the **eth0** interface during installation. Although there are hacks to change which interface is setup and to bring up additional adapters via dhcp, real multiple adapter support needs to be added to the autoinstall script to support this.

Remote logging existed in LUI, but there is no equivalent in SIS. This needs to be put in place before SIS can be considered enterprise ready, as lack of remote logging makes the discovery of failures far more difficult.

The autoinstallation kernel needs a Debian environment to build in. The main reason for this is that Debian provides libc_pic packages, which make it easy to create a smaller version of glibc to go on the autoinstall media. This is a serious limitation to having true source packages that can be rebuilt on any environment. There are a number of possible options here, the use of uClibc, dietlibc, or minilibc are top on the list.

Although building images directly on the server works for most packages, it doesn't for all of them. Occasionally a package will attempt to start a daemon which needs to communicate with other services or directly with hardware. Although it is questionable for a package to do this in a post install script without having a good way to shut it down, it does happen. There has been the possibility of doing some manner of freeze / thaw on post install scripts, so that certain package scripts would be stopped from running, then executed on first boot of the client. This is possible, but the full implications would need to be worked out.

7.2 New Directions

There are many new directions we would like SIS to take, however it is unlikely that more then one or two of these will get accomplished this year due to the size of the development team. So consider some of these pie in the sky ideas that hopefully some eager volunteers will help us do.

7.2.1 SIS to other Platforms

Currently in the pipe is work to bring SIS to PowerPC, HPARISC, and S/390 Linux. Some of these ports should see the light of day this summer. There has always been the thought that SIS could be applied to other operating systems, especially the *BSD family (FreeBSD, OpenBSD, and NetBSD) of operating systems. Any OS which Linux supports creation of, and read / write access to, their filesystem should be able to be supported by SIS in some manner.

7.2.2 Multicast SIS

Once upon a time a multicast library was written for SystemImager called multicaster. The library was never fully finished, but was posted to Source Forge anyway. The funny thing with open source projects, is they pop up in the oddest places. Sometime in October of last year, a new project was announced on Freshmeat called **mrsync** which was a derivative of multicaster with an rsync like command line interface. This is being used in production shops at the moment. The possibility exists of making SIS use mrsync instead of rsync during initial installation to allow installation to scale to hundreds or thousands of simultaneous nodes.

7.2.3 Diskless SIS

SIS creates fully chrootable images on a server which can then be deployed to clients. With very few modifications it should be able to build fully chrootable environments which could be used for diskless environments as well. There is significant interest from certain segments of the high performance computing community for this, so I believe this will happen in the near future.

7.2.4 UML Verification Suite

Because the images on the server are full installations of a running system, it seems possible that a more hearty verification on system integrity could be run on them. The natural choice for this would be User Mode Linux. After image instantiation, a custom verification program could be run in a UML instance which uses the image as its root filesystem. This would allow for a burn in test of the image before it was ever deployed, and could help track down possible conflicting libraries or software revisions. This type of burn it would be essential for large installations that want an assurance test on their images before deployment.

8 SIS in Action

One of the areas that Linux has penetrated extremely well, is the High Performance Computing arena, specifically High Performance Linux Clusters. This arena is very will suited to the strengths of SIS, as all the machines in a cluster tend to be nearly identical. Only a small number of images will be needed to deploy hundreds or thousands of machines. The installation method is distribution independent, so no matter what distribution the user chooses to deploy, the methodology is the same. SIS also has both a command line and graphic user interface, so it can be driven by another application very easily.

OSCAR (Open Source Cluster Application Resource) is a cluster building based on the best known practices in Linux clustering. As of OSCAR 1.2, System Installation Suite is the installer for all the client nodes in an OSCAR cluster. The OSCAR wizard is written in Perl Tk, and hence can use TkSIS panels directly. Every panel in TkSIS allows a callback to be registered. OSCAR uses this feature to do other cluster setup tasks at every stage. This integration was very easy to accomplished, and has given the OSCAR project a very robust distribution independent mechanism for installation. SIS was instrumental in allowing OSCAR support both Red Hat and Mandrake in OSCAR version 1.3.

Other clustering projects such as Clubmask, SCore, and SCE are looking at moving to SIS for their installation so they can support numerous underlying distributions. We expect many tools to leverage the image based framework for systems management that System Installation Suite has created in the future.

9 Conclusion

System Installation Suite is a novel approach to the massive installation problem in Linux which is both distribution and architecture agnostic. It provides an image based framework for extremely scalable installation and maintenance. It has always been the intent of the project to expose as many clean interfaces as possible to other applications, so System Installation Suite can be cleanly integrated into other projects or products requiring an install method that works on many distributions. The expectation is that other components could be easily added to System Installation Suite over time to exploit many of the capabilities of image based systems that have yet to be explored.

Our motto has always been: “Do it once, do it right, do it for every buddy”. We want to support every distribution and every architecture that will run Linux equally well. By doing so, we raise the base line for Linux system’s management, and make Linux easier to deploy for administrators everywhere.

For more information on System Installation Suite: how to use it, how to join the project, and what you can do to help, please visit our web site at **http://sisuite.org**. Links to all the component parts of System Installation Suite are provided there, as well as a network installer which will download and install the latest version of System Installation Suite.

10 Acknowledgements

Credit must always be given where credit is due. System Installation Suite has had many contributors, and hopefully many more (for a full list please see the credits files of all the various projects). Specifically mentioned should be the follow folks that have put long hours into making System Installation Suite happen over the past year, Michael Chase-Salerno, Brian Finley, and Dann Frazier. The following companies have donated a significant amount of time and effort to the project as well, and should be commended for that: Bald Guy Software, Hewlett Packard, and IBM.

Special thanks go out to my editing crew: Trey Belew, Michael Chase-Salerno, Joe Greenseid, and Barb Kane. Without their assistance, this paper would have been far less legible. I also want to thank my employer, IBM, for allowing me to spend the last year working full time on

the System Installation Suite project. I believe this project is a significant contribution to the Linux community, and am thrilled to have been a part of it.

11 References

LUI,
`http://oss.software.ibm.com/lui`

System Configurator,
`http://systemconfig.sf.net`

SystemImager,
`http://systemimager.org`

SystemInstaller,
`http://systeminstaller.sf.net`

System Installation Suite,
`http://sisuite.org`

OSCAR, `http://oscar.sf.net`

12 Trademarks

Making Linux Safe for Virtual Machines

Jeff Dike (jdike@karaya.com)

Abstract

User-mode Linux (UML)[1] is the port of Linux to Linux. It has demonstrated that the Linux system call interface is sufficiently powerful to virtualize itself. However, the virtualization isn't perfect, in terms of individual UML performance, aggregate performance of a number of UML instances sharing a host, or, in one way, functionality.

This paper discusses the current weaknesses in the ability of Linux to host virtual machines and proposes some ways of correcting those shortcomings.

1 Introduction

User-mode Linux (UML) is a port of the Linux kernel to the Linux system call interface. Since this had not been done before, it is impressive that, except for one trivial patch, Linux already provided the functionality needed to virtualize itself.

However, as development of UML has continued, some weaknesses in this support have become evident. Although UML has been successfully implemented using the existing system call interface, in some respects, it is highly non-optimal. This results in poor performance in some areas of UML and poor code in others.

The principal area which hurts UML performance is the Linux `ptrace` interface, which is used to virtualize system calls. Every kernel entry and kernel exit in UML requires two full context switches on the host. This makes system calls, in particular, far more expensive than on the host, but it also hurts the performance of interrupts, which also require four host context switches to complete.

For better context switching performance, each UML process has an associated host process. The host processes are created to gain access to their address spaces. The fact that threads also need to be created wastes host kernel memory and complicates UML context switching. To fix this problem, this paper proposes that Linux address spaces be made independent of threads and that they be created, populated, switched between, and destroyed as separate Linux objects.

Finally, there is the issue of maximizing the capacity of a given server to host virtual machines. Dealing with the issues described above will certainly help, but when considering the hosting throughput of a server, new problems arise. The largest one is memory management. The host and the virtual machines sharing it all have independent virtual memory systems which likely will have completely different pictures of how scarce memory is. Overall, this will lead to inefficient use of the host's memory because some virtual machines will perceive a memory shortage when there isn't one and free memory too aggressively. Similarly, others will not feel any memory pressure when there is some, and will consume memory that could more productively be used elsewhere. So, there will need to be new mechanisms for the host to communicate the true ex-

[1] http://user-mode-linux.sourceforge.net

tent of memory pressure to the UMLs that it's hosting and for the UMLs to respond appropriately to that information.

2 Debugging interface enhancements

2.1 Background

2.1.1 System call virtualization

UML runs the same executables as the host. Since those executables invoke system calls by trapping into the host kernel, UML needs some way to intercept and cancel them, and then execute them in its own context.

This is done through the `ptrace` mechanism. `ptrace` allows one process to intercept the system calls of another, letting it read them out and modify them. UML virtualizes the system calls of its processes by having a master thread, the tracing thread, use `ptrace` to trace the system calls of all UML processes. It annuls them in the host kernel by changing them into `getpid()`.

This works well, but it's slow, since each UML system call involves four context switches, to the tracing thread and back at the start of each system call and again at the end.

2.1.2 Kernel debugging

`ptrace` is exclusive in the sense that a given process can be traced by only one other process at a time. This is inconvenient for UML because the use of `ptrace` by the tracing thread precludes `gdb` from attaching to UML threads, making it harder to use as a UML debugger.

This was solved by having the tracing thread also trace `gdb`, intercepting its system calls, and manipulating the `ptrace` calls in order to fake `gdb` into believing that it's attached to UML.

This works well, but it's inconvenient, and has some unpleasant side-effects. Since `ptrace` reparents the process to whatever has attached to it, the original parent can change its behavior as a result. This generally isn't a problem, since UML usually runs `gdb` itself, or gdb is run under a shell or `emacs`, which don't notice the reparenting. However, a prominent exception is `ddd`, which spawns `gdb`, and calls `wait()` on it periodically. When UML attaches to that `gdb`, `wait` starts behaving differently for `ddd`, resulting in it not working at all.

2.2 A new debugging interface

To solve the performance problems of `ptrace`, UML needs a way for a process to intercept its own system calls. This could be done by delivering a signal whenever it performs a system call and having the signal handler nullify it in the host kernel and execute it inside UML.

Another possibility, which is more elegant, is to introduce a notion of two contexts within a single process. One context would trace the other, gaining control whenever it entered the kernel. This would be implemented by the debugging interface saving the master context state while the traced context runs. When the traced context makes a system call or receives a signal, the master context would be restored. It would have the state of the traced context available, and it could modify it however it saw fit.

This would reduce the cost of a UML system call from four host context switches to about two host system calls.

To allow `gdb` to debug UML, it would also be necessary to simultaneously allow other processes to trace it. This doesn't pose any problems as long as their needs don't interfere with each other or with UML's own system call tracing.

These requirements are sufficiently different from what `ptrace` provides that a new interface is called for. David Howells of Red Hat is working on a `ptrace` replacement. It doesn't allow threads to trace themselves, but that looks like it can be added, and it satisfies all of the other requirements that UML has.

3 Address spaces

3.1 Background

3.1.1 UML address space switching

Currently, each UML process gets a process on the host. This is to provide each UML process with a separate host address space, which makes context switching faster. The alternative, all UML processes sharing a host address space, requires that that address be completely remapped on each context switch. So, if UML switches from a bash to an Apache, that address space would need to be changed from a bash address space to an Apache address space. In fact, this is how UML was first implemented. The change to giving each process a different host address space was done in order to avoid the overhead of walking the address space and remapping it on every context switch.

This optimization nearly converts a UML context switch into a host context switch. The exception is when a process has had its address space changed while it was not running. Most commonly, this is the result of the process being swapped. It can also happen when a threaded process forks. The thread that's out of context will have its address space COWed. When the process is next run, its address space needs to be updated to reflect these changes.

This is done with the help of two architecture-specific flag bits in the pte, `_PROT_NEWPAGE` and `_PROT_NEWPROT`. In a process that's not running, when a page is unmapped or a new page is mapped, `_PROT_NEWPAGE` is set in the page's pte. Similarly, when a page's protection is changed, `_PROT_NEWPROT` is set. When that process is next run, the page tables are scanned for these bits, and the appropriate action (`mmap`, `munmap`, or `mprotect`) is taken on the host in order to bring those pages up to date.

A second complication with bringing an address space up to date during a context switch is the kernel virtual memory area. The kernel, including its VM, is mapped into each process address space. When a process causes a change in the kernel virtual mappings, by adding or removing swap areas or by loading a module, those mappings need to be updated in each process that subsequently runs. The `_PROT_NEWPAGE` and `_PROT_NEWPROT` bits can't be used in this case because the kernel page tables, which store those mappings, are shared by all processes. So, there's no way for a single pte bit to indicate which processes are up to date and which aren't.

This problem is handled by using a counter which increments each time a change is made in the kernel VM area. Each UML process holds in its thread structure the value of that counter when it last went out of context. If the counter hasn't changed, then the process has an up to date kernel VM area. If it has, then it scans the kernel page tables in order to bring its address space up to date.

3.1.2 Execution context switching

The fact that every UML process has an associated host process has implications for switching execution contexts. The obvious way of doing this is for the outgoing process to do the following

1. send the incoming process `SIGCONT`
2. send itself `SIGSTOP`

However, this is unfixably racy. To see this, consider this sequence of events (process A is the outgoing process, and B is the incoming process).

1. A sends B `SIGCONT`
2. B is now runnable on the host, so the host scheduler switches to it, putting A to sleep temporarily
3. B finishes its work before its UML quantum expires, so it switches back to A
4. A now runs on the host and finishes its original UML context switch by sending itself `SIGSTOP`

Now, all UML processes are asleep and will never wake up, effectively hanging the machine.

This was the first implementation of UML's context switching. When this race was discovered, it was eliminated by having the tracing thread mediate context switches. The outgoing process would send a message to the tracing thread asking it to start the incoming process. The tracing thread would do that, leaving the outgoing process stopped. Using the tracing thread as a synchronization point eliminated the race.

In order to eliminate the role of the tracing thread in context switching, two further designs were tried. The first avoided the race by the outgoing process stopping itself with a signal, but blocking that signal and using `sigsuspend` to atomically sleep and enable it:

1. A blocks `SIGTERM`
2. A sends B `SIGTERM`
3. A calls sigsuspend, simultaneously sleeping and enabling `SIGTERM`

In this case, if B runs and switches back to A before it sleeps, then the `SIGTERM` won't be delivered until the `sigsuspend` call, avoiding the race.

However, implementing this involved some non-obvious signal manipulation, so a simpler method was implemented, and it is the current context switching mechanism.

Now, each UML process creates a pipe which is used by other processes to bring it into context. A context switch now works as follows:

1. A writes a character to B's pipe
2. B, which has been blocked in a read on that pipe, returns and continues running
3. A calls read on its own pipe

This avoids races in a similar, but simpler, way to the `SIGTERM` design.

3.2 The solution

So, while assigning a host process to each UML process provides reasonable context switching performance, it has a number of problems of its own:

- Changes to address spaces of other processes can't be effective immediately because one process can't change the address space of another. So, whenever a process is switched in, it must bring its address space up to date if necessary.
- Context switching is complicated by the need to avoid races when one host process continues another and stops itself

The proposed solution is to allow Linux address spaces to be created, manipulated, and destroyed just as processes are. In effect, this would turn address spaces into objects in their own right, separating them from threads. The capabilities that are needed are

- Creation of a new address space
- Changing the mappings in an arbitrary address space
- Switching between address spaces
- Destruction of address spaces

The implementation of this is fairly straightforward. Address spaces are already represented by a separate structure within the kernel, the `mm_struct`. Access to an `mm_struct` can be provided by a new driver which provides userspace access to it through a file descriptor. So, a handle to a process' address space may be obtained by opening `/proc/`*`pid`*`/mm` and a new, empty address space may be created by opening `/proc/mm`. Creating a new handle to an `mm_struct` would increment its reference count, and closing it would decrement it. So, an address space would not disappear as long as there are processes running in it or there are processes which have handles to it.

Allowing a process to change the mappings in another address space would be done with an extension to `mmap`:

```
void *mmap2(void *start,
size_t length, int
prot, int flags, int
src_fd, off_t offset,
int dest_fd);
```

The new argument, `dest_fd`, specifies the address space within which the new mapping is to take place. A value of -1 would specify the current address space, making this interface a superset of the existing `mmap`.

`munmap` and `mprotect` would need to be similarly extended.

3.3 Moving the UML kernel to a separate address space

With the ability to arbitrarily create new address spaces, and the debugging interface described in section 2.2, it is possible to move the UML kernel out of its process address spaces. What's needed is for the debugging interface to switch to the kernel address space when it restores the tracing context.

This would hurt performance somewhat by adding a memory switch to each kernel entry and exit, but would have some large compensating advantages.

The primary gain from doing this would be that it would make UML's data completely inaccessible to its processes. Currently, UML text and data occupy the top .5G of its process' address spaces. By default, this memory is not write-protected when userspace code is running.

This is a problem for applications of UML such as jails and honeypots that need to confine a hostile root user. There is a jail mode in UML which write-protects kernel memory while userspace code is executing by using mprotect to enable write permission on kernel entry and to disable it on kernel exit.

However, there is a severe performance penalty doing this. Implementing jail mode by locating the kernel in a different address space would replace the calls to mprotect with two memory context switches. This is likely to be much faster, and if it's enough faster, it could become the default for UML.

4 AIO

5 Memory management primitives

6 Cooperative memory management between host and guest

6.1 Background

The changes discussed elsewhere in this paper are focussed on improving the performance of an individual UML instance. However, in some applications, such as virtual hosting, the aggregate performance of UML is of equal or greater importance. The aggregate performance is the performance of a set of UML instances running on the same host. Improving this at the expense of the individual UMLs can improve the economics of a virtual hosting installation if the capacity of the host improves enough to increase its overall throughput.

The consumption and use of host memory by UML is a crucial aspect of the server's hosting capacity. There are currently a number of aspects of Linux and UML which cause it to use more host memory than is necessary and to use it less efficiently than it should.

6.1.1 Unused memory is wasted memory

A basic assumption of the Linux VM system is that memory should be used for something, and if memory is plentiful, it doesn't matter what the excess is used for, because it might prove useful in the future. So, data is not thrown out of memory until there is a shortage.

This is fine for a physical machine which contains memory which can't possibly be used by anything else, but this policy hurts the UML hosting capacity of Linux. UML inherits this from the generic, architecture-independent kernel and thus won't free memory until it is feeling a shortage.

The problem is that the host may be short of memory without any of the UMLs it's hosting being short. So, they will hang on to their own data even though they could increase the host's performance by giving up some of it. If the host is swapping enough, they could even improve their own performance by giving up enough of their data to stop the host from needing to swap.

6.1.2 UML memory isn't shared

UML memory is instantiated by creating a temporary file on the host and mapping that into the UML address space. When some of this memory is used for file data, the UML block driver requests, via the `read()` system call, that the data be copied from the host's page cache into its own page cache. There are now two copies of that data in the host's memory. If ten UMLs each boot from separate copies of the same filesystem and copy the data into their own page caches, there will be twenty copies of that data on the host, ten in the host page cache because it loaded ten identical filesystems, and one for each UML. Clearly, this is a waste of memory.

This waste can be alleviated by having them boot from the same filesystem image with separate, private COW files. This will reduce the number of copies of shared data from twenty to eleven. Since they are sharing the same underlying filesystem, the number of copies in the host page cache is now one. However, there is still one copy per UML. This is still a waste of memory.

The problem is that there is currently no mechanism for reducing this any further. Clearly, the copy count could be reduced to one by having UML map data directly from the host page cache into its own page cache.

This would require a different I/O model in the generic kernel. Currently, Linux considers that it has a fixed amount of memory available to is, and when it reads data from disk, it has to allocate memory for that data and copy it from the disk. A UML instance mapping file data directly from the host memory is akin to having that memory, with the data already in it, materialize from nowhere.

6.1.3 VM information isn't shared

A further problem is that the host and the UML instances running on it all have completely independent VM systems which likely will have completely different ideas of how scarce memory is at any given moment. This is a problem because they are all sharing the same memory, and there is only one true picture of how scarce it is, and it is held in the host's VM system.

Somehow, this information needs to be communicated to the UMLs in a way that they can respond to. There are a number of requirements that need to be met if this is to happen:

- UML instances need to be able to free memory to the host. This can be done by unmapping unneeded memory or by calling `madvise(..., MADV_DONTNEED)`. The basic mechanisms are available on the host, but the generic kernel provides no clean way of using them.

- The host needs to be able to demand that a UML free a certain amount of memory of a certain type, i.e. dirty or clean pages, in a given time. This is needed in order for UML instances to feel memory pressure when the host does and to respond appropriately to it.

- UML instances need to be able to tell the host that memory which appears dirty, because it has been changed and not written to its backing store, is really clean, because it is in sync with its backing store from the UML point of view. This would happen when the UML instance had itself swapped the data to its own swap area. In this case, the host could treat the memory as clean and reallocate it without swapping it out.

6.2 Improving memory management cooperation

In contrast to the other problems identified in this paper, this is not a single problem that has a single fix as much as it is a set of problems which will require a set of improvements. This will likely require thought and work for some time to come in order to develop solutions that work reasonably well.

So, I will outline a set of possible partial solutions rather than the single fixes that have been described so far. Some of these may turn out to be of limited value, while other ideas, not described here, may be the ones that solve major parts of the problem.

6.2.1 Passing memory pressure from the host to UML

If there is to be any cooperation at all between the host and its UML instances in memory management, the host needs to be able to communicate memory pressure and its severity to UML. There are currently no mechanisms at all for doing this in Linux.

The host would need to be able to communicate the following information to the UML instances that it is hosting:

- the existence of memory pressure
- the amount of memory that a given instance should release
- the type of memory, i.e. dirty or clean pages, that should be released
- a deadline

The amount of memory that the host asks for may be more a guess than a calculation. It may be more useful for the host to decide what it will do to a UML instance if it doesn't get enough memory released from all sources. This would likely be some number of pages of that UML's memory that will be swapped out. Then, the UML instance can decide what it will do in order to try to avoid that fate. It would likely have a better idea of what memory it can do without than the host, so it could release pages that it thinks it needs less than what the host would choose to swap out.

Whether the host wants clean or dirty pages released depends on whether the host is also I/O-bound and on the timeframe within which it wants the memory. If it is I/O-bound, then freeing dirty pages isn't going to be useful because they would need to be written to swap before they could be freed, adding to the I/O congestion. Similarly, if the need for memory is immediate, then releasing dirty pages won't help because there will be a significant time lag between their release by UML and their availability to the system.

A UML instance will have some pages which it considers to be clean, but which the host considers to be dirty. A mechanism to inform the host that these should be considered clean would increase the proportion of clean pages available for release.

Another possible mechanism for reclaiming memory is for the host to just take clean pages from a UML without it explicitly releasing them. This would require that the host signal the UML instance when it next accesses such a page. It would have to refill the page with its original data before it could continue to be reused. This has the disadvantage that the UML has no choice in what pages are taken, so there is no guarantee that the host will choose pages that the UML instance can do without. The host could make reasonable guesses by looking at the hardware access bit, but it will still not have information about access patterns within the UML that may be relevant.

6.2.2 Freeing memory to the host

Currently, a UML instance is assigned a certain amount of memory which is considered to be its physical memory. It is not allocated on the host immediately. Rather, the host allocates it as it is used. However, once it is used, it is never given back. So an instance with a large amount of memory that has not been used recently will hold onto it even if there are other instances which have a much greater need for it.

As described above, there are mechanisms for a process to free memory back to the host. However, there is no way in the generic Linux kernel to give up memory. With some hooks in the page allocation and page release code, it is possible for the architecture to do some things.

Given a hook in the page release path, the architecture could release a page to the host by unmapping it or calling `madvise` appropriately. It also has a choice between freeing it to the page allocator or not. This decision would be based on whether the UML instance is to reduce the amount of memory it has at its disposal. If the page is made available to the page allocator, it will very likely be reallocated and reused. Thus, the host will need to reallocate the page soon after having it freed. This will limit the benefits to the host of freeing the page in the first place.

So, there would appear to be benefits to not freeing pages to the page allocator if the host is under memory pressure. However, if this is done, there would need to be some way of getting those pages back when the memory pressure on the host has abated.

Given a mechanism of the sort described in section 6.2.1, there should also be an obvious way of indicating that the memory pressure has diminished, and that the UML instance can start reclaiming the memory that it gave up. In both cases, there would need to be some indication from the host of how much memory should be given up and how much can be reclaimed. This would prevent undershooting and overshooting in both cases and make it more likely that the host will have a reasonable amount of free memory.

7 Conclusion

Fixing the problems described above will greatly increase both the performance of individual UML instances and the hosting capacity of a given server.

A more efficient system call interception interface will greatly increase the performance of system-intensive applications running inside UML. Compute-bound processes typically run as fast inside UML as on the host, but other, more system-intensive workloads can run two to three times slower. Allowing a thread to intercept its own system calls would bring the performance of these processes much closer to their performance on the host.

Similarly, using AIO to allow many outstanding I/O requests would help I/O-intensive workloads. Without AIO, UML is limited to one outstanding I/O request at a time. As a result, I/O-intensive processes can spend much of their time idle, waiting for their data to be read from the host. This isn't a waste of the host's processing power like the use of `ptrace` is, so it may not hurt the host's capacity, but it does noticeably hurt the performance of individual UML instances.

Allowing host address spaces to be manipulated as objects separate from threads would help UML's context switching performance. It would also greatly simplify the low-level context-switching code. There would be no need to traverse the address spaces of processes coming into context in order to bring them up to date with whatever changes were made while they were out of context. There would also be no need for the optimizations that have been made in order to make the existing algorithms faster. A further benefit to the code would be that the execution context switch would be completely race-free, since it would no longer be switching between host processes. This code has been significantly simplified over time, but it would become trivial once a single thread could switch between address spaces arbitrarily.

This enhancement, in conjunction with a single-thread system call interception capability, would also allow the UML kernel to be located in a completely different address space than its processes. This would be particularly beneficial to jailing applications, which currently suffer from the poor performance of the current mechanism of protecting UML kernel data from userspace.

Finally, by improving the ability of the host to communicate memory pressure to the UML instances running on it and improving their ability to respond would noticeably improve the hosting capacity of a given server. In some cases, this would also improve the individual performance of the UML instances.

This area is also interesting because this is of much more general use than the others. `ptrace`, AIO, and address spaces are of fairly limited use to most applications. In contrast, memory management is of concern to practically all processes. So, implementing methods of cooperative memory management between the host and UML instances would provide those mechanism to other processes as well. This would open the way to this sort of cooperation being common, presumably with the result that the system performs better than it would otherwise.

This relatively small list summarizes the problems that Linux has as a virtual machine hosting environment. They mostly appear to have fairly straight-forward fixes, and some of these fixes are in progress already. The most complicated area is the cooperative memory management. That is a set of related problems that will require a set of measure to deal with them, rather than a single problem with a single fix. In contrast to the others, it will likely be the subject of study and work for some time to come.

Linux is currently quite viable as a virtual machine platform for a number of applications. Once these problems are fixed, Linux will become even more attractive for hosting virtual machines.

Online ext2 and ext3 Filesystem Resizing

Andreas E. Dilger
adilger@clusterfs.com,
http://www-mddsp.enel.ucalgary.ca/People/adilger/

Abstract

It is difficult to predict the future, yet this is what you have to do each time you partition a disk. There are several HOWTOs giving advice on ways to partition a disk for Linux, yet everyone's usage pattern is different. Invariably, your filesystems fill up, often in the middle of doing something important.

With the advent of Linux LVM in 1999, Linux was finally getting to the stage where one could add space to "partitions" dynamically. The missing link was allowing the widely-used ext2 filesystem to grow to use newly added disk space without having to kill your applications, unmount the filesystem, do an offline resize (which was in its infancy at that time also), and remount the filesystem.

We start with a brief overview of the layout of the ext2 filesystem to give an understanding of the constraints behind the design of the online ext2 resizer. The three ext2 resizing scenarios are discussed, and implementation of each case is presented. The rationale behind offline filesystem preparation is given, and the (incompatible and yet unimplemented) alternative is presented. We continue with the requirements for ext3 online resizing discuss how this leads to a totally different implementation.

1 Introduction

One of the many reasons why a system stops doing the job it is intended to do is because it runs out of space in an important filesystem. While it is possible to increase the size of a partition or disk which is in use (via software or hardware RAID, LVM [LVM], and more recently EVMS [EVMS]) you normally have to stop applications and unmount ext2 and ext3 filesystems in order for the filesystem itself to be resized to take advantage of this increased space. To avoid an interruption to the system (and applications, and users), one has to be able to grow ext2 and ext3 filesystems while they are mounted and in active use (i.e. read and write operations in progress, current working directory of a process, etc).

The `GNU ext2resize` package [resize] is GPL licensed code which contains the `ext2online` tool and a kernel patch, which together allow increasing the size of a mounted filesystem without interruption to processes using that filesystem. In addition, the `ext2resize` tool also allows you to grow and shrink an unmounted filesystem. The `ext2prepare` tool is also part of `GNU ext2resize`, and is discussed later. While `ext2online` only allows one to increase the size of a mounted filesystem, in a vast majority of cases it is increasing the free space in a filesystem which is the critical operation needed to keep an application running. In rare cases you might need to shrink one filesystem in order to grow another, but given the ease of increasing the size of a mounted filesystem on a system using LVM and online filesystem resizing, there is little need to make filesystems too large for their anticipated short-term growth as was needed previously.

The bulk of the ext2 online resizing kernel code was written in the fall of 1999 for the 2.0.36 and 2.2.10 kernels, but has remained relatively unchanged through all of the kernels since then, with only minor changes to locking and patch context. The 2.4 "code freeze" of the fall of 1999 prevented the patch from being added to the kernel at that time, and ongoing stabilization of 2.4 and other tasks have prevented me from doing more than minor code maintenance for the most part. The requirement to have online resizing for ext3 was the first restructuring of the kernel code, and involved a complete code rewrite. It is anticipated that the ext2 and ext3 online resizing code will be submitted for inclusion into the 2.4 and 2.5 kernels some time in the summer of 2002. The user space tools that form the `GNU ext2resize` package have undergone a slow evolution during their lifetime to support newer ext2 features such as large files and offline resizing of ext3 filesystems, as well as having more complete support for unusual filesystem layouts such as RAID stripe aligned metadata and filesystems whose inode tables are not at the same offset in every group.

In this paper we focus primarily on the *online* (mounted) aspect of filesystem resizing. For *offline* (unmounted) filesystem resizing, there are additional aspects of resizing, such as inode renumbering, moving the contents of data blocks and the inode table, and renumbering the data block pointers within an inode. The `ext2resize` tool can do all of these things. In order to keep the amount of kernel code to a minimum (and to make it actually work) these aforementioned operations are never done during online filesystem resizing.

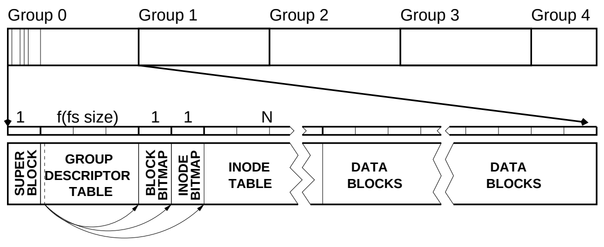


Figure 1: Block Group Layout

2 Anatomy of an ext2 File System

In order to understand the constraints under which the ext2 filesystem resizer operates, we must first have some understanding of the on-disk filesystem layout. For the purposes of filesystem resizing, the ext2 and ext3 on-disk layouts are identical. We do not cover all aspects of the ext2 filesystem layout, such as inodes and directories, because for the purposes of online filesystem resizing those details are mostly irrelevant. They are covered in many other general ext2 papers [ext2].

The on-disk layout of the ext2 filesystem is strongly influenced by the layout of the BSD Fast File System. The disk/partition is divided into one or more sections, called *block groups*. Block groups are of a fixed size, determined at filesystem creation time, and all contain the same number of blocks, except the last block group which may have fewer blocks. By default, ext2 block groups are created at their maximum size (32MB for the default 4kB blocksize), and are numbered from the beginning of the filesystem starting with 0.

Each block group contains several key pieces of filesystem metadata, as shown in

Figure 1. For each block group, there is one block which is the *block bitmap*, one block which is the *inode bitmap*, and one or more blocks which make up the *inode table*. In addition, there may be a copy of both the filesystem *superblock* and the filesystem *group descriptor table* in a block group. Whether a block group will contain either a primary or backup superblock and group descriptor table depends on the group number and/or parameters at filesystem creation time.

The block bitmap describes the allocation status of all data and metadata blocks within that block group. If a bit is set, this indicates that a block is in use as either a data or metadata block, and if it is clear then the block is available for allocation. Since the block bitmap is limited to a single block in size, this imposes the maximum size of a block group - the number of bits which will fit in a single filesystem block is 8 times the blocksize, and this is the maximum number of blocks that can be in a single group. For the last group in a filesystem, the bits representing blocks past the end of the filesystem will be set (marked in use) so that the kernel does not need to special-case the search for free blocks in the last group.

The inode bitmap describes the allocation status of the inodes in its group's inode table. Since the inode bitmap is limited to a single block in size, this imposes the maximum number of inodes that can be allocated in a single group. If there are less than the maximum number of inodes in a group, the bits corresponding to non-existent inodes will be set (in use).

The inode table contains one or more blocks which hold the inode data. Each group has the same number of blocks in the inode table, and this number is determined at filesystem creation time. Multiple inodes are packed into each inode table block, and fill the block completely, so this imposes the minimum number of inodes that can be allocated in each group - the number of inodes that fill a single block. The maximum number of inodes in each group is the same as with the maximum number of blocks in each group - the number of bits that fit within a single filesystem block, so 8 times the blocksize.

The superblock contains critical filesystem configuration parameters (e.g. blocksize, total number of blocks and inodes, group size, number of inodes in each group, etc.) and also dynamic filesystem status (e.g. the number of free blocks and inodes, the number of times the filesystem was mounted, the error status, the last time it was checked, etc). The primary superblock is located at 1024 bytes from the start of the filesystem, and is 1024 bytes in size. There are backup copies of the superblock stored in block groups[1]with numbers which are integer powers of 3, 5, and 7 (i.e. 1, 3, 5, 7, 9, 25, 27, 49, ...). Under normal operation, only the primary copy of the superblock is ever used, and the backups are only needed by `e2fsck` in case the primary copy is corrupted or overwritten.

The group descriptor table contains one or more blocks which hold *group descriptors*. A group descriptor contains the location of the block bitmap, inode bitmap, and the start of the inode table for its block group. It also contains the count of free blocks, free inodes, and allocated directories for its group. There is a group descriptor for each group in the filesystem, so the number of blocks that make up the group descriptor depends on the number of groups in the filesystem, which in turn depends on the size of the filesystem. Because the group descriptor table is critical in locating the filesystem metadata, backup copies of the group descriptor table are placed in the same groups as

[1]When the ext2 on-disk layout was first developed, backup copies of the superblock were placed in *every* block group. For large filesystems, the amount of space consumed by the backup superblocks and group descriptor tables became too large, so modern ext2 filesystems only place backups in *sparse* groups (i.e. as described here).

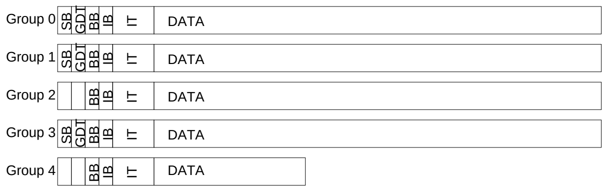


Figure 2: Several block groups make up filesystem

backup superblocks.

Figure 2 shows a filesystem with several block groups. Note that all but the last group have the same number of blocks. The inode table is the same size in each block group, and each block group has both an inode and block bitmap. The group descriptor table is the same size in each group, if it exists. While this example shows the most common case of the bitmaps and inode table in the same location in each group, the ext2 format allows the bitmaps and inode table each to be in any (non-overlapping) location within the group. The actual location within each group is solely determined by the entries in the group descriptor table for that group. The backup superblock and group descriptor table must be located in the first blocks of the group so that they can be located in case of filesystem corruption. By default `mke2fs` will create the bitmaps and inode table in the same position within each group.

3 Three Resizing Scenarios

3.1 Common Resizing Operations

There are several operations that are common to all of the growth scenarios discussed here:

- Increasing the total number of filesystem blocks in the primary and backup superblocks by the number of blocks added to the filesystem. Since it is possible that the primary superblock may be corrupted at some later date, we also need to update the backup superblocks to reflect the new size of the filesystem. Otherwise, `e2fsck` would believe that all block numbers higher than the old filesystem size are invalid and the data therein would be discarded.

- Increase the count of free filesystem blocks in the primary superblock and group descriptor for that group. The count of free blocks, like other dynamic ext2 metadata, does not necessarily need to be updated in the backup superblocks or group descriptors. The total and per-block-group count of free filesystem blocks can be recovered by counting the bits set in each of the block bitmaps, and is only really a convenience for efficient `statfs()` and `ENOSPC` implementation.

- Increase the number of reserved filesystem blocks proportional to the number of new blocks added to the filesystem.

- In order to notify the filesystem that it should begin its resizing operation, the filesystem is remounted with the option `-o remount,resize=<new size>`. While this seems somewhat awkward, it does have the benefit that it can be done from the command-line with only the `mount` command. It is also very practical from the point of view that the resize operation uses all of the same checks and setup code in `ext2_setup_super()` as the initial `mount` call.

Since the goal of online resizing is to allow the filesystem to continue to be used while the resizing operation is being done, we need to make sure that we do the appropriate locking of the filesystem structures that we update. Currently, there is only a single lock for the superblock and all of the group descriptors, so we only need to hold this superblock lock to ensure our operations are safe. Also, the critical filesystem values are each stored in only one place so we don't need to ensure consistency between multiple data fields of different in-memory data structures. Obviously we want to hold the superblock lock for as little time as possible to avoid excluding other processes from doing filesystem operations. It is anticipated that in the future the filesystem locking will become more fine-grained to allow increased parallelism in block and inode allocation. This will likely be done by having a read/write lock for each group's block bitmap, inode bitmap, and group descriptor, separate from the superblock lock, and may be added to the kernel during 2.5 development.

One other aspect of increasing the filesystem size which makes it relatively trouble-free is the fact that all of the blocks which are added to the system are new. This means that there can not be any users of these blocks or pages or buffers mapped to them, so we do not need to be concerned with hashing or locking or other such aspects of block I/O which may cause deadlocks or data corruption. This is one of the major reasons why shrinking a mounted filesystem is completely impractical. In limited cases it might be possible to shrink a mounted filesystem by a single group, the nature of the ext2 block and inode allocation algorithms mean that the last group will almost always have inodes and blocks allocated in them. While it might be possible to relocate data blocks on a mounted filesystem (excluding issues such as `FIBMAP` of those blocks exporting the block numbers to user space), the relocation of inodes is even more problematic because of the use of inode numbers in directories and NFS file handles, let alone the locking issues involved.

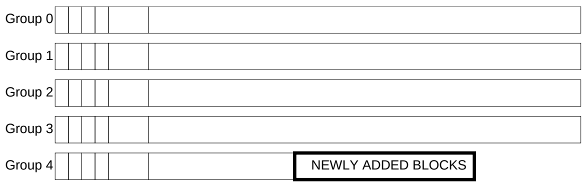


Figure 3: Adding blocks to a single group

3.2 Adding Blocks to a Single Block Group

The first and simplest filesystem growth scenario is adding blocks to the end of a single block group, as shown in Figure 3. In order to efficiently and safely implement the operations of updating the block bitmap and increasing the free blocks count in the superblock and group descriptor, we take a short cut and create a fake inode which spans the newly-added blocks at the end of the last block group. All we have to do now is increase the total number of filesystem blocks in the superblock, and delete the fake inode via `ext2_free_blocks()`. This will take care of updating the bitmaps and the free blocks count in the superblock and group descriptor. The fake node is only created in-memory and only has enough fields filled in to satisfy those accessed by `ext2_free_blocks()` and what it calls. This also has the advantage that any locking changes which take place in the ext2 code will be handled for us.

We can easily do everything we need for this resizing operation from within the kernel, since it has no more impact on performance than deleting a file of the same size (less actually, since we don't need to update the on-disk inode data). Since this resize operation is virtually identical to deleting a file, it is almost impossible for it to fail, excluding bugs in the core ext2 code. This resizing operation needs no additional updates to the on-disk metadata. Depending on the blocksize of the filesystem, this scenario would allow us to grow a filesystem up to the next 8MB, 16MB, or 32MB bound-

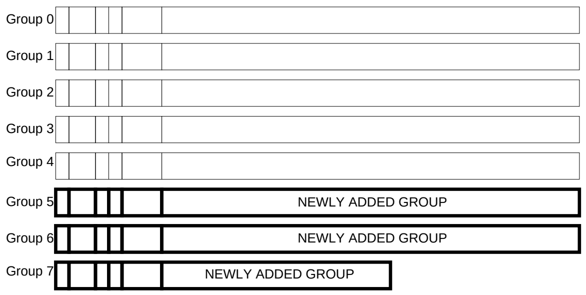


Figure 4: Adding new block groups within the same descriptor block

ary (for 1kB, 2kB, or 4kB blocks, respectively) for all filesystems.

The only operation which is left to user-space is that of copying the new primary superblock to the backup superblock locations, if any. Since the backup superblocks are never accessed by the kernel, there is no problem writing them directly to the block device from user space. If the resize command is done manually via the `mount` command instead of `ext2online`, the superblock is not copied to the backup locations. Under all but the most extreme failure conditions that will be OK, and the ability to do a filesystem resize using an available tool is convenient. The backup superblocks will be updated the next time `e2fsck` does a full check of the filesystem.

3.3 Adding a Block Group Within a Group Descriptor Block

The second filesystem growth scenario is that of adding a new group to the filesystem, as shown in Figure 4, after the end of a full block group. Each block group must have a block bitmap, inode bitmap, and inode table, and possibly a backup superblock and group descriptor table. This means that we need to add at least enough new blocks to the filesystem to hold all of this metadata before we can actually increase the size of the filesystem. Since we are adding an inode bitmap and inode table, we also need to update the total number of inodes in the primary and backup superblocks, and the number of free inodes in both the primary superblock and group descriptor for the new group. All of these operations can be done by simply updating the appropriate fields in the superblock while holding the superblock lock.

There are two things that distinguish this case from the first case:

- We need to create a new block bitmap, inode bitmap, and an inode table for each group added to the filesystem. The bitmaps have to be filled to reflect the availability of blocks and inodes within the new group.
- We need to add a new entry to the group descriptor table for the new group. The group descriptor table is stored on disk and is accessed in the kernel via an array of buffer heads. If we are adding a group that fits within the last group descriptor block then we do not need to add a new buffer head, but we do need to increase the number of groups in the filesystem so that the block and inode allocation routines know to check for free blocks in the new group(s).

In order to minimize kernel code growth and in-kernel processing overhead, the creation of the block bitmap, inode bitmap, and inode table are done from user space before the kernel is told about the new group(s). This is perfectly safe even on a busy filesystem because the blocks being modified from user space are beyond the end of the existing filesystem, so there will not be any other processes reading from or writing to these blocks.

The creation of the new group descriptor entry is slightly more problematic, because it is sharing a block with other group descriptors that are in active use. Reading the block to user space, modifying it, and writing it back to disk cannot be done safely on an active filesystem because it would lose any updates that had been done by the kernel since the group descriptor block had been read. Fortunately, existing kernels have cache coherency between the block device and the buffer heads in the kernel, so it is possible with ext2 to write a new group descriptor entry into the last group descriptor block and have it visible to the kernel without corrupting the existing group descriptors. The new group descriptor entry has the correct values for the free block and inode counts of that group, in addition to the location of the bitmaps and inode table.

The new group descriptors are written to the disk from user space before the resize operation takes place, as are the bitmaps and inode table. The filesystem is notified via the `mount` parameter of the new filesystem size and adds the new blocks and inodes to the free and total block and inode counts in the superblock. As the group's free block and inode counts are added into the totals in the superblock (under the superblock lock, of course), the new groups are immediately available to the filesystem for allocation. Again, once the kernel is finished its resizing operation the new superblock and group descriptor table is copied to the backup locations, if any. Since we need to create the bitmaps and inode table from user space, it is not practical to do this resize operation with anything other than the `ext2online` tool, which also handles the updating of the backup superblock and group descriptor tables once the resize operation has completed successfully.

Resizing the filesystem in this manner allows it to add groups until the last group descriptor block is full. Since each group descriptor is a fixed size (32 bytes), the number of group descriptors that fit into a single filesystem block depends on the blocksize. Likewise the size of each group also depends on the blocksize. The boundaries for group descriptor blocks are 256MB, 1GB, and 16GB for 1kB, 2kB, and 4kB blocks, respectively. This allows one to resize any default 4kB blocksize filesystem a considerable amount without any prior preparation.

The failure scenarios for this resizing case are very minimal. The in-kernel code basically is just adding the new block and inode counts from the group descriptors into the superblock. Although the group descriptors, bitmaps, and inode table were just written from user space, the kernel does some validity checks of everything before activating each group to avoid any problems.

3.4 Adding a Block Group in a New Group Descriptor Block

The final resizing scenario happens when the last block in the group descriptor table is full. This brings up one of the major limitations imposed by keeping on-disk compatibility with the existing ext2 format. Because the number of groups in the filesystem is linearly dependent on the filesystem size, and we keep all group descriptors in each group descriptor table, eventually we need to allocate new blocks for the group descriptor table in each group that has a copy. As Figure 1 shows, the default configuration of metadata within a group is to have the superblock and group descriptor table first (these two *have* to be first in the group), with the block bitmap, inode bitmap, and inode table following immediately afterward. This means that the group descriptor table cannot normally allocate a new block at the end because the block bitmap is using this block.

Fortunately, the design of the ext2 on-disk layout allows us to circumvent this problem by moving the bitmaps and inode table further

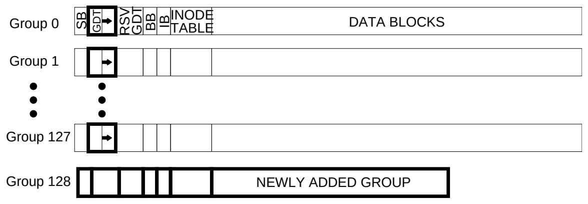


Figure 5: Adding a new block group in a new group descriptor block

from the start of the group to allow the group descriptor table to grow, as shown in Figure 5. Since the location of the bitmaps and inode table for each group can be set in the group descriptor, it is simply a matter of creating a filesystem with the bitmaps and inode table offset from the end of the group descriptor table. For existing filesystems, the `ext2resize` package has an `ext2prepare` tool which will relocate the bitmaps and inode table. Since the operation of relocating the bitmaps and inode table is also necessary for resizing the filesystem (for the same reason - because the group descriptor table will grow to need the blocks they occupy) the `ext2prepare` code can re-use most of the same code as `ext2resize`. The moving of the bitmaps and inode table must be done while the filesystem is unmounted.

The second issue for this case is how to reserve the blocks after the end of the group descriptor table so that they are not allocated to regular files. One way would have been to store an extra field in the superblock which holds the number of reserved blocks. If the blocks were set unused in the block bitmap, kernels and `e2fsck` that do not know about this scheme would happily assign those blocks to files and potentially prevent the filesystem from being resized later. If the blocks were set as used in the block bitmap, then an `e2fsck` which didn't know about this scheme would think they were unused and clear them the next time it checked the filesystem. Instead, one of the reserved ext2 inodes (#7) was used to hold these reserved group descriptor blocks. To both the kernel and `e2fsck` inode #7 is using the reserved group descriptor blocks and everyone is happy.

In order to do an online resize when we need to add a new group descriptor block, we perform all of the steps as before:

- Write a group descriptor, block bitmap, inode bitmap, and inode table for each group to the disk from user space.
- Tell the filesystem to grow to the new size.
- The kernel adds the new inodes and blocks from each group to their respective total and free counts in the superblock.
- Update the backup superblocks and group descriptors from user space once the resize has completed successfully.

There are two additional steps which must be done in order to handle the new group descriptor block:

- The kernel must read the new block into a buffer and add it into the array of group descriptor block buffer heads before adding these groups to the filesystem. In order to minimize kernel code growth, the existing code to allocate the array and read the buffers was extracted into a function which could start reading the group descriptor table at a non-zero offset. The existing buffers are left untouched in order to avoid having other parts of the filesystem code with pointers to no-longer-existing buffer data. Only the group descriptor array is re-allocated at the new size and filled in with pointers to the existing and new buffers. A failure during this part of the resizing information will simply cause the resize not to happen, as the old array is not freed until after the new array and buffer(s) have been allocated.

- We must transfer the blocks from the reserved inode to the group descriptor table. This is actually relatively simple. Because the blocks are already marked as in-use in the block bitmap, we do not need to change it for blocks assigned to the group descriptor table. The number of blocks in the group descriptor table is calculated from the total number of blocks in the filesystem, so we do not need to update the superblock to reflect this (we have already updated the total number of blocks). The only thing that remains, for consistency, is to deallocate the blocks from the reserved inode. This can be done from user-space by writing directly to the block device after the resize is complete, because this reserved inode is not accessible from the mounted filesystem. A failure at this point will mean that `e2fsck` will complain about blocks shared between the reserved inode and the group descriptor table, and will automatically clean it up.

3.5 Incompatible Online Resizing

Adding the reserved group descriptor blocks to the reserved inode is done by `ext2prepare` while the filesystem is unmounted as it moves the bitmaps and inode table out of the way. The number of blocks to be reserved for each group descriptor table copy is calculated from the desired future maximum filesystem size given by the user on the command line. For the default 4kB block size, we only need to reserve 1 block for each 16GB of future growth, so the overhead of reserving enough blocks for a 2TB filesystem is fairly minimal —only 512kB per group descriptor copy. The requirement for users to prepare a filesystem while it is unmounted, before doing large online resizes is a fairly minimal price to pay in order to keep 100% forward and backward compatibility with the existing ext2 filesystem layout. This requirement could be removed by having `mke2fs` create new filesystems that already have these blocks allocated to the reserved inode.

The alternative to doing offline filesystem preparation is to make an incompatible change to the on-disk layout when we run out of space in the last group descriptor block. This change would involve storing new group descriptor blocks at the beginning of the first group that needs a new block for its group descriptor. The backup of this block would be stored in the second group that needs this group descriptor block. This has the added benefit that the group descriptor of a group is relatively closer to the groups that it describes, which may reduce seeking some small amount. The major drawback of this scheme is that it results in a filesystem that can not be mounted by older kernels, nor can older ext2 filesystem tools work with it. It also adds some small amount of additional code to the kernel to deal with the two different layouts of the group descriptor table, although this is fairly minimal. There is also a proposed filesystem change to allow larger contiguous extents on the disk to be allocated which would also benefit from this format change, so there may be additional justification for making such an incompatible change.

Given that people use online filesystem resizing when they are running out of space, they may choose to pay this penalty in order to keep their system running smoothly. The potential drawbacks are only a problem if you have back-level filesystem tools or need to boot an older kernel. This problem could be mitigated by adding functionality to `ext2resize` or `ext2prepare` to remove the incompatibility from unmounted filesystems after the fact.

This would be done by moving the bitmaps and inode table out of the way and moving the new group descriptor block(s) to their normal position, in a manner very similar to resizing the filesystem.

4 Online Resizing an ext3 Filesystem

The ext3 filesystem is the journaling version of ext2[ext3]. Interestingly, although ext2 and ext3 share a virtually identical on-disk format (the ext3 journal is simply a regular file stored inside the filesystem), the requirement for a 100% consistent filesystem in the face of a crash at any point during the resize made the ext3 online resize implementation totally different from that of ext2. Writing into the filesystem from user space for ext2 online resizing is not safe to do with an ext3 filesystem because the journal layer may make copies of a buffer while it is being written to disk, so there is no guarantee of cache coherency between user space access to the block device and kernel space[2]. The requirement that the resize operation be atomic and leave the filesystem correct also precludes writing directly to the block device, because these writes will not be in the journal.

Instead, all of the operations which were formerly done inside journaled transactions[3] in the kernel. The user space code is still responsible for determining the locations of the metadata structures within each group, but the kernel code is responsible for creating the actual data. This turns out to be less complex than initially thought, as most of the operations are simply to `memset()` the buffers to zero (for the inode table and group descriptor and most of the bitmaps), then use the kernel bitmap handling routines to mark appropriate bits set, and finally `memset()` any large parts of the bitmap to 0xffffffff as necessary. Since we are creating the data in the kernel, we do not need to verify its correctness, excluding a limited number of parameters passed from user space.

The addition of each group is placed into its own journal operation to avoid trying to create too large of a single transaction. Transactions which do not need to be atomic because they do not affect filesystem recovery such as zeroing the blocks in the inode table and copying data to the backup superblock and group descriptor tables are put into separate transactions to allow other filesystem operations to happen concurrently. Those operations which need to be atomic, such as moving blocks from the reserved inode to the group descriptor table and updating the superblock to include new groups, are done within a single transaction. Because the journal imposes ordering between transactions, it is enough that a previous transaction was closed and a new one opened to ensure that the previous operations will exist after a crash if any following operation also exists.

In order to keep in-kernel processing to a minimum, the layout of the reserved group descriptor inode was changed from that used with ext2. Under ext2 the updating of the reserved inode was done in user space and searching the inode for blocks that had been added to the group descriptor table was acceptable. With ext3 the layout was changed to allow group descriptor blocks to be put into use by updating

[2]In fact, the ext3 journaling layer has a large number of assertions embedded in it which will catch buffers which enter the ext3 "food chain" incorrectly, to avoid data corruption. This prevents such access as a rule, rather than letting it succeed most of the time but fail mysteriously at other times.

[3]Actually, the ext3 journal layer has a larger operation called a *transaction*. What is referred to here as a transaction is actually called a journal *handle* in the code, but transaction more clearly represents the operation being described and the actual functionality is the same.

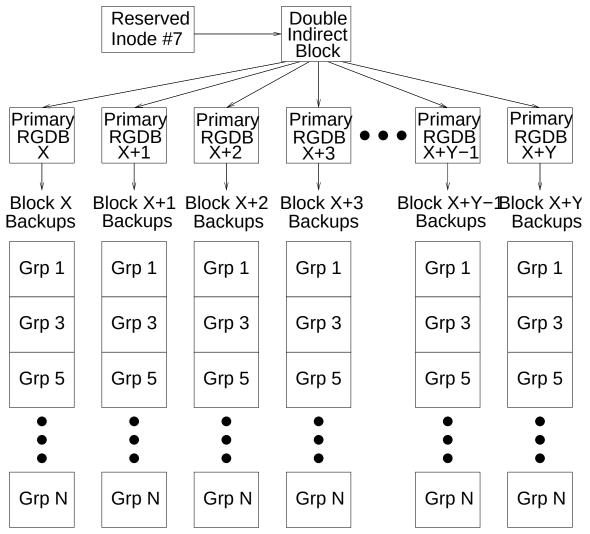


Figure 6: New arrangement of blocks in the reserved inode

only a single inside the transaction, and without searching the inode. This required a change to the user space `ext2prepare` tool to set up the new format. I took advantage of this format change to also add an ext2 compatible feature flag to the ext2 superblock, so that `e2fsck` could properly detect and verify the more rigid layout requirements for the reserved inode.

Figure 6 shows the layout of the reserved GDT blocks in the inode. The reserved blocks are arranged such that the double indirect inode block points to the primary copy of each reserved group descriptor block, which is an indirect inode block. The reserved group descriptor blocks are shown numbered starting with `X`, which is the block number for the block immediately following the last group descriptor block currently in use. The offset of the primary reserved group descriptor block is `X mod (blocksize/4)`. which allows us to locate each reserved group descriptor block and its backups without any searching.

All of the backup copies for this group descriptor block are leaf blocks attached to the indirect (primary) block. There is one backup group descriptor block for each of the "sparse" groups which have a backup superblock and group descriptor table. Using this layout allows us to transfer the reserved descriptor blocks from the reserved inode to the group descriptor table by simply zeroing out the indirect block pointer in the double indirect block. This is done within the same journal transaction as creating the group descriptor data in the primary descriptor block and adding that group to the filesystem, so if there is any error during this process the filesystem remains consistent.

There is one additional point which needs to be handled by the ext3 resizing code which cannot exist in ext2. In the case where a resize operation was completed in the journal, but the system crashes before the resize is flushed from the journal to the filesystem, the filesystem size in the superblock will still reflect the old filesystem size. Journal recovery is normally done by `e2fsck`, but for the root filesystem or other ext3 filesystems mounted without an fsck, journal recovery is done after the superblock has been read from the disk. We detect this case by comparing the filesystem size from before journal recovery to that after journal recovery, and if the filesystem has grown we need to read any additional group descriptor blocks from disk in the same manner we would do in a normal resizing operation.

5 Conclusion

The ext2 filesystem resizing code is a fairly mature piece of code, even thought it has not been included in the stock kernel yet. While the thought of resizing the filesystem while it is mounted and in use is somewhat daunting, the actual simplicity of the resizing operations gives little room for error. In fact, other than minor math errors in updating the group or superblock inode or block counts during development (which are easily detected and/or fixed by both the kernel and `e2fsck`) , I have never had a reported filesystem corruption from the online resizing.

The advent of online resizing for ext3 does add additional complexity to the kernel code, but the requirements for high-availability systems demand both online resizing and journaling support, so the extra overhead can be justified. Since the online resizing code is only used very rarely, it would be possible to put the bulk of this code into a separate module that is only loaded when a resize operation is done. The cleaner layout of the reserved inode and the `e2fsck` support for it mean that it is desirable to change the ext2 online resizing support over to use the new inode format also. In the short term this is accomplished by an updated `ext2online` user tool.

As was previously mentioned, the `GNU ext2resize` package is available from Sourceforge at `http://sf.net/projects/ext2resize/` in `.tgz` and `.rpm` formats. There is also a very low volume mailing list dedicated which can be accessed from this same page. General ext2 and ext3 filesystem design and coding discussions take place on the `ext2-devel@lists.sourceforge.net` mailing list.

Special thanks go to Ted Ts'o and Stephen Tweedie, who helped me understand ext2 and the kernel when I was first trying to learn what kernel programming was all about, and all the interesting ext2/ext3 discussions we've had since then. Lennert Buytenhek was the original author of the `GNU ext2resize` code and `libext2resize`, and this gave me the foundation on which to build the user tools for online resizing and offline filesystem preparation.

Thanks also go to Miguel de Icaza, whose ext2-volume patch[volume] gave me a rough idea of where I should look at in the kernel to find the ext2 filesystem code and how it all fit together. The ext2-volume code was written to allow one to concatenate full ext2 filesystems together to form a single filesystem, prior to the availability of MD RAID and LVM in the kernel, but had the major drawbacks that it only supported offline resizing, and produced a totally incompatible filesystem.

References

[resize] Lennert Buytenhek, Andreas Dilger, *GNU ext2resize*, `http://sf.net/projects/ext2resize/`

[ext2] Rémy Card, Theodore Ts'o, Stephen Tweedie, *Design and Implementation of the Second Extended Filesystem*, `http://e2fsprogs.sf.net/ext2intro.html`, (1994)

[LVM] Michael Hasenstein, *The Logical Volume Manager*, `http://www.sistina.com/lvm/lvm_whitepaper.pdf`, (2000)

[EVMS] Ben Rafanello, John Stiles, Cuong H. Tran, *Emulating Multiple Logical Volume Management Systems within a Single Volume Management Architecture*, `http://oss.software.ibm.com/developerworks/opensource/evms/doc/EVMS_White_Paper1.ps.gz`, (2000)

[ext3] Stephen Tweedie, *Journaling the ext2fs Filesystem*, LinuxExpo '98 Proceedings, `ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz`, (1998)

[volume] Miguel de Icaza, *Ext2fs volume support*, `http://www-`

`mddsp.enel.ucalgary.ca`
`/People/adilger/online-ext2`
`/ext2-volume-0.1.tar.gz`,
(1997)

Running Linux on a DSP? Exploiting the Computational Resources of a programmable DSP Micro-Processor with uClinux

Michael Durrant *Jeff Dionne* *Michael Leslie*

1 Introduction

Many software developers in recent years have turned to Linux as their operating system of choice. Until the advent of uClinux, however, developers of smaller embedded systems, usually incorporating microprocessors with no MMU, (Memory Management Unit) could not take advantage of Linux in their designs. uClinux is a variant of mainstream Linux that runs on "MMU-less" processor architectures. Perhaps a DSP? If a general purpose DSP has enough of a useable instruction set why not!

Component costs are of primary concern in embedded systems, which are typically required to be small and inexpensive. Microprocessors with on-chip MMU hardware tend to be complex and expensive, and as such are not typically selected for small, simple embedded systems, which do not require them.

Benefits

Using Linux in devices, which require some intelligence, is attractive for many reasons:

- It is a mature, robust operating system
- It already supports a large number of devices, filesystems, and networking protocols
- Bug fixes and new features are constantly being added, tested and refined by a large community of programmers and users
- It gives everyone from developers to end users complete visibility of the source code
- A large number of applications (such as GNU software) exist which require little to no porting effort
- Linux's very low cost

2 uClinux System Configurations

Embedded systems running uClinux may be configured in many ways other than that of the familiar UNIX-like Linux distribution. Nevertheless, an example of a system running uClinux in this way will help to illustrate how it may be used.

Kernel / Root Filesystem

The Arcturus uCdimm is a complete computer in an so-DIMM form factor, built around either a Motorola 68VZ328 "DragonBall" microcontroller, the latest processor in a family widely popularized by the "Palm Pilot" or a Motorola CF5272 "ColdFire" microcontroller. It is equipped with 2M of flash memory, 8M of SDRAM, both synchronous and asynchronous serial ports, and an Ethernet controller. There is a custom resident boot monitor on the device, which is capable of downloading a binary image to flash memory and executing it. The image that is downloaded consists of a uClinux kernel and root filesystem. In UNIX terms, the kernel makes a block device out of the memory range where the root filesystem resides, and mounts this device as root. The root filesystem is in a read-only UNIX-like format called "ROMFS". Since the DragonBall runs at 32MHz, the kernel and optionally user programs execute in-place in flash memory. Faster systems like the MCF5272 ColdFire running at 48 MHz or 66 MHz benefit from copying the kernel and root filesystem into RAM and executing there.

The Micro Signal Architecture MSA developed by ADI (and Intel) is very interesting. While a DSP processor, it possesses enough general-purpose processor instruction support to allow operating systems like uClinux to be deployed. The ADI BlackFin is one such processor.

Other embedded systems may be inherently network-based, so a kernel in flash memory might mount a root filesystem being served via nfs (Network File System). An even more network-centric device might request its kernel image and root filesystems via dhcp (Dynamic Host Configuration Protocol) and bootp. Note that drivers for things like IDE and SCSI disk, CD, and floppy support are all still present in the uClinux kernel.

User Space

The contents of the root filesystem vary more dramatically between embedded systems using uClinux than between Linux workstations.

The uClinux distribution contains a root filesystem which implements a small UNIX-like server, with a console on the serial port, a telnet daemon, a web server, nfs (Network File System) client support, and a selection of common UNIX tools.

A system such as an mp3 (MPEG layer 3 compressed audio) CD player might not even have a console. The kernel might contain only support for a CD drive, parallel I/O, and an audio DAC. User space might consist only of an interface program to drive buttons and LEDs, to control the CD, and which could invoke one other program; an MPEG audio player. Such an application specific system would obviously require much less memory than the full-fledged uClinux distribution as it is shipped.

3 Development Under uClinux

Development Tools

Developing software for uClinux systems typically involves a cross-compiler toolchain built from the familiar GNU compiler tools. Software that builds under gcc (GNU C Compiler) for x86 architectures, for example, often builds without modification on any uClinux target.

Debugging a target via gdb (GNU debugger) presents a debugging interface common to all the platforms supported by gdb. The debugging interface to a uClinux kernel on a target depends on debugging support for that target. If the target processor has hardware support for debugging, such as IEEE's JTAG or Motorola's BDM, gdb may connect non-intrusively to the target to debug the kernel. If the processor lacks such support, a gdb "stub" may be incorporated into the kernel. gdb communicates with the stub via a serial port, or via Ethernet.

uClibc

uClibc, the C library used in uClinux, is a smaller implementation than those which ship with most modern Linux distributions. The library has been designed to provide most of the calls that UNIX-like C programs will use. If an application requires a feature that is not implemented in uClibc, the feature may be added to uClibc, it may me linked in as a separate library, or it may be added to the application itself.

Differences Between uClinux And Linux

Considering that the absence of MMU support in uClinux constitutes a fundamental difference from mainstream Linux, surprisingly little kernel and user space software is affected. Developers familiar with Linux will notice little difference working under uClinux. Embedded systems developers will already be familiar with some of the issues peculiar to uClinux.

Two differences between mainstream Linux and uClinux are a consequence of the removal of MMU support from uClinux. The lack of both memory protection and of a virtual memory model are of importance to a developer working in either kernel or user space. Certain system calls to the kernel are also affected.

Memory Protection

One consequence of operating without memory protection is that an invalid pointer reference by even an unprivileged process may trigger an address error, and potentially corrupt or even shut down the system. Obviously code running on such a system must be programmed carefully and tested diligently to ensure robustness and security.

Virtual Memory

There are three primary consequences of running Linux without virtual memory. One is that processes, which are loaded by the kernel, must be able to run independently of their position in memory. One way to achieve this is to “fix up” address references in a program once it is loaded into RAM. The other is to generate code that uses only relative addressing (referred to as PIC, or Position Independent Code). uClinux supports both of these methods.

Another consequence is that memory allocation and deallocation occurs within a flat memory model. Very dynamic memory allocation can result in fragmentation, which can starve the system. One way to improve the robustness of applications that perform dynamic memory allocation is to replace *malloc*() calls with requests from a preallocated buffer pool.

Since virtual memory is not used in uClinux, swapping pages in and out of memory is not implemented, since it cannot be guaranteed that the pages would be loaded to the same location in RAM. In embedded systems it is also unlikely that it would be acceptable to suspend an application in order to use more RAM than is physically available.

System Calls

The lack of memory management hardware on uClinux target processors has meant that some changes needed to be made to the Linux system interface. Perhaps the greatest difference is the absence of the *fork()* and *brk()* system calls.

A call to *fork()* clones a process to create a child. Under Linux, *fork()* is implemented using copy-on-write pages. Without an MMU, uClinux cannot completely and reliably clone a process, nor does it have access to copy-on-write.

uClinux implements *vfork()* in order to compensate for the lack of *fork()*. When a parent process calls *vfork()* to create a child, both processes share all their memory space including the stack. *vfork()* then suspends the parent’s execution until the child process either calls *exit()* or *execve()*. Note that multitasking is not otherwise affected. It does, however, mean that older-style network daemons that make extensive use of *fork()* must be modified. Since child processes run in the same address space as their parents, the behaviour of both processes may require modification in particular situations.

Many modern programs rely on child processes to perform basic tasks, allowing the system to maintain an interactive "feel" even if the processing load is quite heavy. Such programs may require substantial reworking to perform the same task under uClinux. If a key application depends heavily on such structuring, then it may be necessary to either re-create the application, or an MMU-enabled processor may also be needed.

A hypothetical, simple network daemon, *hyped*, will illustrate the use of *fork()*. *hyped* always listens on a well-known network port (or socket) for connections from a network client. When the client connects, *hyped* gives it new connection information (a new socket number) and calls *fork()*. The child process then accepts the client's reconnection to the new socket, freeing the parent to listen for new connections.

uClinux has neither an autogrow stack nor *brk()* and so user space programs must use the *mmap()* command to allocate memory. For convenience, our C library implements *malloc()* as a wrapper to *mmap()*. There is a compile-time option to set the stack size of a program.

4 Brief Anatomy Of The uClinux Kernel

This section describes the changes that were made to the Linux kernel to allow it to run on MMU-less processors.

Architecture-Generic Kernel Changes

The architecture-generic memory management subsystem was modified to remove reliance on MMU hardware by providing basic memory management functions within the kernel software itself. For those who are familiar with uClinux, this is the role of the directory */mmnommu* derived from and replacing the directory */mm*. Several subsystems needed to be modified, added, removed, or rewritten. Kernel and user memory allocation and deallocation routines had to be reimplemented. Support for transparent swapping / paging was removed. Program loaders which support PIC (Position Independent Code) were added. A new binary object code format, named "flat" was created, which supports PIC and which has a very compact header. Other program loaders, such as that for ELF, were modified to support other formats which, instead of using PIC, use absolute references which it is the responsibility of the kernel to "fix up" at run time. Each method has advantages and disadvantages. Traditional PIC is quick and compact but has a size restriction on some architectures. For example, the 16-bit relative jump in Motorola 68k architectures limits PIC programs to 32K. The runtime fix-up technique removes this size restriction, but incurs overhead when the program is loaded by the kernel.

Porting uClinux To New Platforms

The task of adding support for a new CPU architecture in uClinux is similar to doing so in Linux proper. Fortunately, there is a great deal of code in Linux that can be ported with minor adaptations and reused in uClinux. Machine dependent startup code and header files already exist in Linux for MMU versions of processors in the ARM, Motorola 68k, MIPS, SPARC and other families. This code may be adapted to support non-MMU versions of these processors in uClinux.

Driver code, which already exists in Linux, is often easily portable to run under uClinux. Issues in porting such code may involve endian issues or memory handling code, which assumes the presence of MMU support.

5 The Future of uClinux

Numerous enhancements are in the works for uClinux. The diversity of the innovations that mainstream Linux receives from the community pave a good path for the development of uClinux. The uClinux developer community is very active; enhancements and innovations are frequently made.

Real-Time

Linux is now a platform for hard real-time application development (that is, applications with deterministic latency under varying processor loads). The Linux kernel scheduler already provides non-deterministic, or "soft", real-time, and systems such as RT-Linux and RTAI (Real-Time Application Interface) upgrade the Linux kernel to provide hard (deterministic) real-time support. Real-time applications in Linux have access to the extensive resources of the Linux kernel without sacrificing hard real-time performance. Efforts are underway to provide the RTAI subsystem for use on various MMU-less processors.

Adding DSP support or adding general purpose processor support to a DSP

Processors like the ColdFire MCF5272 are primarily general purpose processors with a RSIC instruction set. Yet Motorola included limited DSP functionality with a multiply and accumulate DSP functions. This is an approach of adding functionality into general purpose processors. In this case the ColdFire processor can and does have uClinux support. Other processors like the Analog Devices BlackFin (MSA DSP), the processor is primarily a DSP with added general purpose processor support. uClinux is portable to the BlackFin and expected to be publicly available in the fall of 2002.

The attached paper *"Exploiting the Computational Resources of a Programmable DSP Micro-processor (Micro Signal Architecture MSA) in the field of Multiple Target Tracking" (Hussain et al 2001)*, is a technical representation of the computational requirements for a multiple target acquisition system. Such a system would require a DSP and would greatly benefit from a UNIX like operating system afforded by uClinux. Commercial uses for such a system would include traditional sonar/radar devices allowing for affordable collision detection systems, for robots, and automobiles.

uClinux 2.4

uClinux 2.4, with support for Motorola DragonBall and ColdFire, was released in January of 2001. New ports, including MIPS, Hitachi SH2, ARM, and SPARC, will be made to the uClinux 2.4 tree, which is based on Linux 2.4. but enhancements are also still being made to the uClinux 2.0 tree. uClinux 2.4 will give developers access to many of the new features added to Linux since 2.0, including support for USB, IEEE Firewire, IrDA, and new networking features such as bandwidth allocation, (a.k.a. QoS: Quality of Service) IP Tables, and IPv6.

Since uClinux is Open Source, development effort spent on uClinux will never be lost. Engineering professionals world-wide, are using uClinux to create commercial products and a significant portion of their work is contributed back to the open source community.

****

References:

"Running Linux on low cost, low power, MMU-less processors", Michael Durrant, Arcturus Networks Inc.

"Building Low Cost, Embedded, Network Appliance with Linux", Greg Ungerer, SnapGear Inc.

"Embedded Coldfire-Taking Linux On-Board", Nigel Dick, Motorola Ltd

"When hard real-time goes soft", D. Jeff Dionne, Arcturus Networks Inc.

Real-Time Application Interface (RTAI): **http://www.rtai.org**

The uClinux project: **http://www.uClinux.org**

EXPLOITING THE COMPUTATIONAL RESOURCES OF A PROGRAMMABLE DSP MICRO-PROCESSOR (Micro Signal Architecture MSA) IN THE FIELD OF MULTIPLE TARGET TRACKING

Principal Researcher: Dr. D.M. Akbar Hussain
Contributions: Michael Durrant *michael.durrant@ArcturusNetworks.com*
Jeff Dionne *jeff.dionne@ArcturusNetworks.com*

Arcturus Networks Inc. 195 The West Mall, Suite 608, M9C 5K1, Toronto CANADA
Tel: 416 621 0125 Fax: 416 621 0190

Abstract: During the last few decades the improved technology available for surveillance systems has generated a great deal of interest in algorithms capable of tracking large number of objects.. Typical sensor systems, such as radar or sonar using information from one or more sensors can obtain noisy information data returns from true targets and other possible objects. The tracking problem requires the processing of this data to produce accurate estimates of the position and velocity of the targets. There are two types of uncertainties involved with the measurement data, first the position inaccuracy, as the measurements are corrupted by noise, and second the measurement origin since there may be uncertainty as to which measurement originates from which target. These uncertainties lead to a data association problem and the tracking performance depends not only on the measurement noise but also upon the uncertainty in the measurement origin. Therefore, in a multiple target environment extensive computation may be required to establish the correspondence between measurements and tracks every radar scan.

It is also true that tremendous advancement has also been made in the computational capabilities of a processing unit to deal with such demanding tasks. In this paper we present a simulated study of implementing a recursive multiple target tracking (MTT) algorithm using a track splitting filter, study uses a MSA processor from Analog Devices Inc. (ADI) which is a programmable Digital Signal Processor (DSP) with the added functionality to realize many of the programming advancements more normally associated with Micro-controllers. In addition, the study also explores the porting and support of an embedded real time operating system to such an architecture.

Keywords: DSP, RTAI, Kalman Filter, Target Tracking, State Estimation, uClinux.

1. INTRODUCTION

In the ideal situation of tracking a single target, where one noisy measurement is obtained, standard Kalman filter technique can be used at each radar scan In the multi-target case, an unknown number of measurements are received at each radar scan and, assuming no false measurements, each measurement has to be associated with an existing or new target tracking filter. When the targets are well apart from each other forming a measurement prediction ellipse around a track to associate the measurement with the appropriate track is a standard technique [1]. When targets are near to each other, more than one measurement may fall within the prediction ellipse of a filter and prediction ellipses of different filters may interact. The number of measurements accepted by a filter will therefore be quite sensitive in this situation to the accuracy of the prediction ellipse. Several approaches may be used for this situation [2, 3]. One such approach is called the Track Splitting Filter algorithm. In this algorithm, if *n* measurements occur inside a prediction ellipse, then the filter branches or splits into *n* tracking filters.

This situation, which results in an increased number of filters, requires more processing power and in some cases the system may saturate. A mechanism for restricting excess tracks splitting is required, since eventually this process may result in more than one filter tracking the same target. The first criterion is the support function, which uses the likelihood function of a track as the pruning criterion. The second, similarity criterion, which uses a distance threshold to prune similar filter tracking the same target [4]. The flow chart shown in Fig. 1 depicts the actual processing sequence of a recursive MTT algorithm.

2. TARGET MOTION MODEL AND STATE ESTIMATION

The motion of a target being tracked is assumed to be approximately linear and modeled by the following equations

$$\underline{x}_{n+1} = \Phi \underline{x}_n + \Gamma \underline{\omega}_n \tag{1}$$

$$\underline{z}_{n+1} = H \underline{x}_{n+1} + \underline{\nu}_{n+1} \tag{2}$$

where the state vector

$$\underline{x}^T_{n+1} = (x\, x^\bullet\, y\, y^\bullet)_{n+1} \tag{3}$$

is a four dimensional vector, $\underline{\omega}_n$ is the two dimensional disturbance vector, $\underline{z_{n+1}}$ is the two dimensional measurement vector and $\underline{\nu}_{n+1}$ is the two dimensional measurement error vector. Also Φ is the assumed (4x4) state transition matrix, Γ is the excitation matrix (4x2) and H is the measurement matrix (2x4) and are defined respectively by,

$$\Phi = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$\Gamma = \begin{bmatrix} \Delta t^2/2 & 0 \\ \Delta t & 0 \\ 0 & \Delta t^2/2 \\ 0 & \Delta t \end{bmatrix} \tag{5}$$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6}$$

Here Δt is the sampling interval and corresponds to the time interval (scan interval) assumed constant, at which radar measurement data is received. The system noise sequence $\underline{\omega}_n$ is a two dimensional Gaussian sequence for which

$$\mathrm{E}(\underline{\omega}_n) = 0 \tag{7}$$

where E is the expectation operator. The covariance of $\underline{\omega}_n$ is

$$\mathrm{E}(\underline{\omega}_n \underline{\omega}_n^T) = Q_n \delta_{nm} \tag{8}$$

where Q_n is a positive semi-definite (2x2) diagonal matrix and δ_{nm} is the Kronecker delta defined as

$$\delta_{nm} = \begin{matrix} 0 & n \neq m \\ 1 & n = m \end{matrix}$$

The measurement noise sequence $\underline{\nu}_n$ is a two dimensional zero mean Gaussian white sequence with a covariance of

$$\mathrm{E}(\underline{\nu}_n\ \underline{\nu}^T{}_n) = R_n\delta_{nm} \tag{9}$$

where R_n is a positive semi-definite symmetric (2x2) matrix given by

$$R_n = \begin{bmatrix} \sigma^2{}_x & \sigma_{xy} \\ \sigma_{xy} & \sigma^2{}_y \end{bmatrix} \tag{10}$$

σ_x^2 and σ_y^2 are the variances in the error of the x, y position measurements, and σ_{xy} is the covariance between the x and y measurements errors. It is assumed that the measurement noise sequence and the system noise sequence are independent of each other, that is

$$\mathrm{E}(\underline{\nu}_n\ \underline{\omega}^T{}_n) = 0 \tag{11}$$

The initial state $\underline{x}_0$ is also assumed independent of the $\underline{\nu}_n$ and $\underline{\omega}_n$ sequences that is

$$\mathrm{E}(\underline{x}_0\ \underline{\omega}^T n) = 0 \tag{12}$$

$$\mathrm{E}(\underline{x}_0\ \underline{\nu}^T n) = 0 \tag{13}$$

$\underline{x}_0$ is a four dimensional random vector with mean $\mathrm{E}(\underline{x}_0) = \underline{x}_{0/0}$ and a (4x4) positive semi-definite covariance matrix defined by

$$\mathbf{P}_0 = \mathrm{E}[(\underline{x}_0 - \underline{x}^-{}_0)(\underline{x}_0 - \underline{x}^-{}_0)^{\mathrm{T}}] \tag{14}$$

where $\underline{x}^-{}_0$ is the mean of the initial state $\underline{x}_0$. The Kalman filter is an optimal filter as it minimizes the mean squared error between the estimated state and the true state (actual) provided the target dynamics are correctly modeled.

The standard Kalman filter equations for estimating the position and velocity of the target motion described by eqns. (1) and (2) are

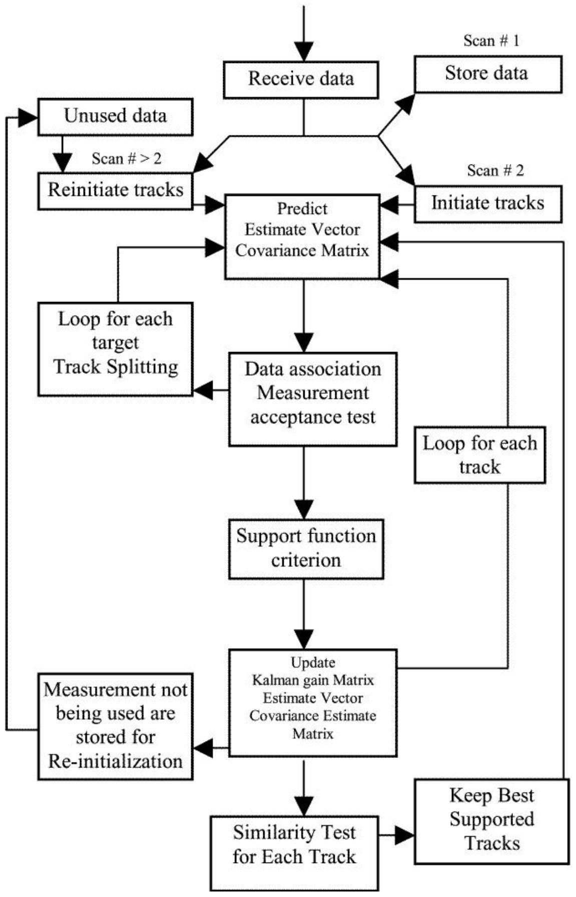

Scan # 1
Receive data
Store data
Unused data
Scan # > 2
Scan # 2
Reinitiate tracks
Initiate tracks
Predict Estimate Vector Covariance Matrix
Loop for each target Track Splitting
Data association Measurement acceptance test
Loop for each track
Support function criterion
Update Kalman gain Matrix Estimate Vector Covariance Estimate Matrix
Measurement not being used are stored for Re-initialization
Keep Best Supported Tracks
Similarity Test for Each Track

Figure 1: Recursive MTT

$$\underline{x}^\wedge_{n+1/n} = \Phi \underline{x}^\wedge{}_n \tag{15}$$

$$\underline{x}^\wedge_{n+1} = \underline{x}^\wedge_{n+1/n} + \mathbf{K}_{n+1}\ \underline{\nu}_{n+1} \tag{16}$$

$$\mathbf{K}_{n+1} = \mathbf{P}_{n+1/n}\mathbf{H}^{\mathbf{T}}\mathbf{B}^{-1}{}_{n+1} \tag{17}$$

$$\mathbf{P}_{n+1/n} = \Phi\mathbf{P}_n\Phi^{\mathbf{T}} + \Gamma\mathbf{Q}^{\mathbf{F}}_n\Gamma^{\mathbf{T}} \tag{18}$$

$$\mathbf{B}_{n+1} = \mathbf{R}_{n+1} + \mathbf{H}\mathbf{P}_{n+1/n}\mathbf{H}^{\mathbf{T}} \tag{19}$$

$$\mathbf{P}_{n+1} = (\mathbf{I} - \mathbf{K}_{n+1}\mathbf{H})\mathbf{P}_{n+1/n} \qquad (20)$$

$$\underline{\nu}_{n+1} = \underline{z}_{n+1} - \mathbf{H}\underline{x}^{\wedge}{}_{n+1/n} \qquad (21)$$

where $\underline{x}^{\wedge}_{n+1/n}$, $\underline{x}^{\wedge}_{n+1}$, $\mathbf{K}_{n+1}$, $\mathbf{P}_{n+1/n}$, $\mathbf{B}_{n+1}$, and $\mathbf{P}_{n+1}$ are the predicted state, estimated state, the Kalman gain matrix, the prediction covariance matrix, the covariance matrix of innovation, and the covariance matrix of estimation respectively. $\mathbf{Q}_n^{\mathbf{F}}$ is the covariance of the measurement noise assumed by the filter which is normally taken equal to $\mathbf{Q}_n$. In a practical situation, however, the value of this covariance is not known so the choice should be such that the filter can adequately track any possible motion of the target. To start the computation an initial value is chosen for $\mathbf{P}_0$. Even if this is a diagonal matrix, then clearly from the above equations the covariance matrices $\mathbf{B}_{n+1}$, $\mathbf{P}_{n+1}$ and $\mathbf{P}_{n+1/n}$ for a given n, do not remain diagonal when $\mathbf{R}_{n+1}$ is not diagonal.

3. MSA

The MSA processor has five independent, full functional computation units: Two Arithmetic/Logic Units (ALU), Two Multiplier/Accumulator Units and a Barrel Shifter, Fig. 2 shows the MSA. The processor units can process 8-bit, 16 bit, 32 bit and 40 bit data, depending upon the type of function being performed.

- ***Data Address Generator (DAG)***

The Micro Signal Architecture processor base is a dual data path, modified Harvard Architecture. The two DAG support the sophisticated operations required in DSP algorithms, such as reversed addressing, circular buffering. In addition, auto increment, auto decrement and base plus immediate offset addressing are possible. These units update dedicated register files, the DAG register file and the pointer register file. In general, DSP mathematical operations involving circular buffers uses the DAG register file. The DAG register file contains four sets of 32 bit Index, Length, Modify and Base registers, for a total of sixteen 32 bit registers. With these two independent DAGs, MSA can generate two 32 bit addresses in a single cycle, fetching or storing two 32 bit or four 16 bit operands. The pointer register file is used for more general operations. It has six general purpose 32 bit addressing registers and two dedicated 32 bit Stack Pointers for stack manipulation. The Micro Signal Architecture processor can access a unified 4 GB linear address space.

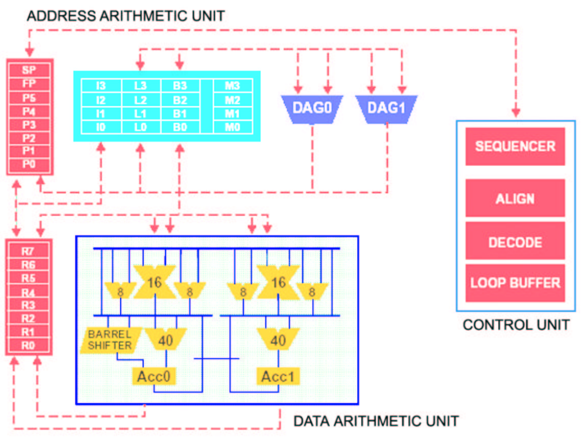


Figure 2: MSA

- ***Memory Management Unit (MMU)***

The memory Management unit (MMU) provides protection to individual tasks that may be operating on the Micro Signal Architecture Processor and may protect system Memory Mapped Registers from unintended access. The architecture includes two Memory Management Units: one for instruction memory and the other one for data memory. These MMUs control accesses to caches, on chip

SRAM and off chip memories. The Micro Signal Architecture Processor supports multiple pages of memory, in four page sizes: 1 K byte, 4 K byte, 1 M byte, 4 M byte. The instruction MMU may designate as many as sixteen distinct memory pages, each with a separate set of criteria governing its cache and protection properties. The Data MMU may designate a further sixteen memory pages.

The Micro Signal Architecture Processor uses a modified Harvard Architecture in combination with a hierarchical memory structure. Memory closest to the Core are referred to as level 1 (L1), and generally have a single cycle, zero latency access by the core. Other memories on chip are referred to as level 2 (L2) and may have multiple-cycle access or latency. Off-chip memories are usually seen at the same hierarchical level as the on chip L2 memory. At the L1 level, the instruction memory holds instructions only and the two data memories hold only data. At L2 level a single unified memory space exists, holding both instructions and data. Also L1 instruction memory and L1 data memories may be configured as either SRAM or caches. The Micro Signal Architecture Processor has a dedicated scratch data memory that is always configured as an additional L1 data SRAM. Scratchpad memory is designed to store stack and local variables.

- ***Program Sequencer***

The program sequencer is used to get instructions from the L1 instruction memory and determine if these instructions are 16 bit ‘Control’ instructions, 32 bit ‘DSP’ instructions or 64 bit ‘DSP multi-function’ instruction. The sequencer manages the control of data through the processor core, insuring that the pipe line is fully interlocked and that zero overhead looping is correctly managed.

- ***Event Controller, Timer and JTAG Interface***

The event controller supports nested and prioritized events. The controller has five basic types of events: Emulation, Reset, Non-maskable Interrupt (NMI), Exception and Interrupts. The programmable interval timer is used to generate periodic interrupts. The 8-bit pre-scale register can be used to set the number of cycles for decrementing a 32 bit counter register. The number of clock cycles per timer decrement may be one to 256. An interrupt is generated when this count register reaches zero. The register may be automatically reloaded from a 16 bit period register and the count resumed.

The JTAG interface provides the method by which the Micro Signal Architecture emulations interact with the processor core. The emulation unit contains an instruction register and data register that is accessed through the JTAG port. These two registers are used to control and interrogate the processor during emulation mode. Additionally the trace unit can be used to store the last 16 non-sequential PC values, which can be used to reconstruct the processor sequence.

- ***Performance Monitor Unit (PMU)***

During the operation of the Micro Signal Architecture Processor, the performance monitor unit can be used to review the efficiency of certain operations, e.g., cache misses, and provide the information for code optimization. The performance unit consists of six instruction address watch points and two data address watch points. These address watch points may be combined in pairs to create address range watch points, which additionally may be associated with various counters for evaluating the performance and profiling the processor code.

- ***Instruction Set***

The Micro Signal Architecture Processor assembly level instruction set employs an algebraic syntax, presenting code to the programmer that is very readable even at the assembly level. The instructions have been specifically tuned to provide a very flexible, yet densely encoded instruction set that will compile to a very small final memory size. The instruction set also provides fully featured multi-function instructions that allow the programmer to use any of the Micro Signal Architecture Processor core resources in a single instruction. Coupled with a variety of enhanced features more often seen on micro-controllers, this instruction set is very efficient when compiling code for C and C++ languages.

- ***Modes of Operation***

The Micro Signal Architecture Processor has five distinct modes of operations: User, Supervisor, Emulation, Idle and Reset. User mode has restricted access to certain system resources, thus providing a protected software environment. Supervisor and Emulation modes have unrestricted access to core resources. Idle and Reset modes prevent software execution, so resource access is not an issue.

4. DSP PROGRAMMING MODEL

The instruction set for the Micro Signal Architecture processor provides two types of instructions: one primarily for micro-controller and general tasks, and those used for DSP oriented computation. Specific Micro Signal Architecture instructions are tuned for their corresponding task, but instructions can easily be combined. Table 1 shows the resources available to the DSP processing. DSP instructions usually read two 32 bit operands from the register file, compute results and either store results back to the register file or accumulate them in the two accumulators as shown in Fig. 3.

Resources	Description
Data Execution Unit 0	• 16 x 16 bit MAC unit (MAC 0) • 40 bit accumulator (a0) • 40 bit shifter
Data Execution Unit 1	• 16 x 16 bit MAC unit (MAC 1) • 40 bit accumulator (a1)
Data Register File	8, 32 bit wide, accessible as register halves

Table 1: Execution Units and Register File

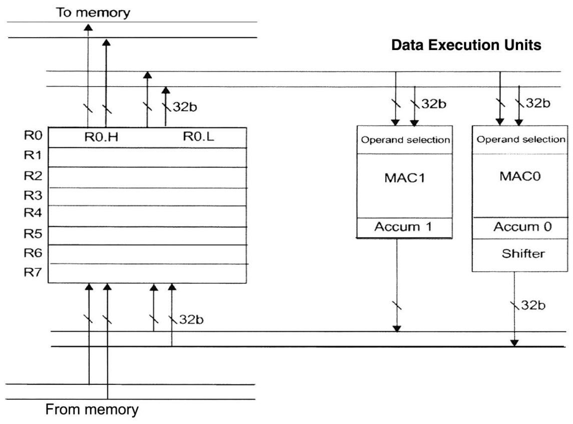


Figure 3: Register Files and Execution Unit

The register file delivers two 32 bit operands to MAC units and accepts two 32 bit results in return. In addition, the register file delivers two 32 bit values to the memory system or it receives two 32 bit values from memory. To perform multiplication, each MAC unit select two 16 bit operands from the two 32 bit words that it receives from the register file as shown in Fig. 4.

It also means that each MAC unit uses four possible combinations of input operands. Therefore, when both MAC units operate in parallel, sixteen possible combinations of 16

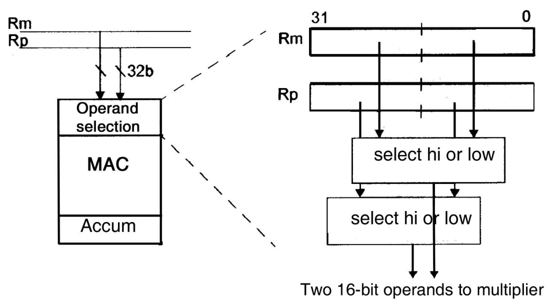


Figure 4: Operand Selection

bit input operand results. Multiply and accumulate operations can be performed on four combinations of input operands, as shown in Fig. 5, for MAC 0. Assembly instruction example of dual MAC operation.

```
A0 += R1.H*R2.L, A0
+=R1.L*R2.L;
```

In addition to performing a multiply accumulate, the multiplier results may optionally be written to the register file, independently of each other. In addition to multiply accumulate, in which the contents of the accumulator are effected, MSA also has multiply instructions which does not effect the contents of the accumulator.

The ALUs have a different mechanism than MACs, ALU 0 as the primary source of arithmetic and logic operation, it can perform the following operations:

- 32 bit operation on two 32 bit inputs producing one output.
- One 16 bit operation on two 16 bit inputs residing in arbitrary register halves producing one 16 bit output.
- Two 16 bit (dual) operation on four 16 bit inputs (in two registers). Dual 16 bit ALU operations can perform four combinations

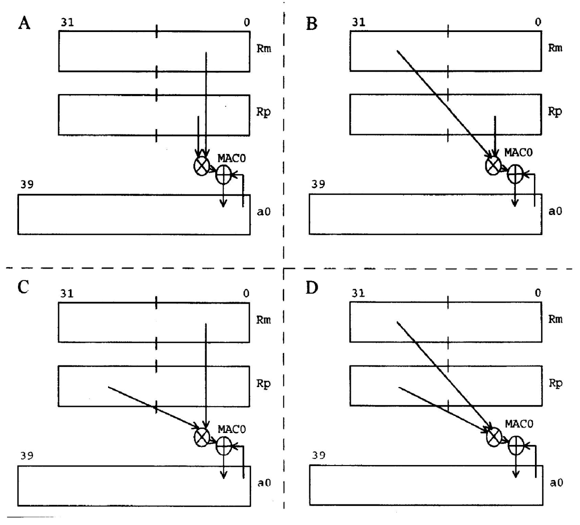


Figure 5: MAC 0 Possible Choices

of addition and subtraction on the input operands as shown in Fig. 6.

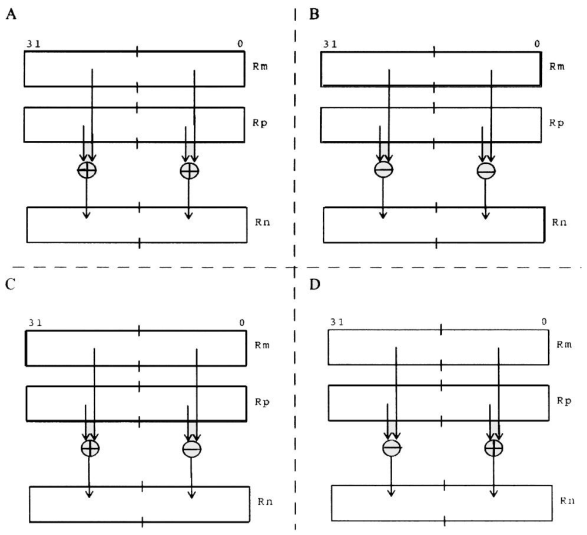


Figure 6: Dual 16-bit ALU Operations

The result from the upper halves are placed into the upper half of the destination register and lower halves to the lower half of the destination register. It also supports the cross option, in which case the order of the two 16 bit result is inverted. In addition to the operations supported on the primary computation unit ALU

0, a small number of instructions support the secondary arithmetic unit ALU 1 in parallel with ALU 0. The Micro Signal Architecture core does not support full dual ALU functionality, because the maximum number of input operands that may be transferred from the register file to the execution units is limited to two 32 bit operands. Therefore, parallel ALU operations on ALU 0 and ALU 1 can be performed only on the same two 32 bit words.

5. IMPLEMENTATION

For the implementation of the target tracking algorithm employing a track splitting filter, a sensor was simulated in two dimensions to generate data for different scenarios. The simulating program is capable of generating up to 20 targets in a predefined scan window, two versions of the program can be used: one, in which initial position, heading, noise and other parameters are taken as default, in the second case all this data can be entered by the programmer. The generated data basically provides the following input information corresponding to each individual target.

x	y	σ_x^2	σ_y^2	σ_{xy}	Flag	CID

Where x, y is the noisy position measurements, σ_x^2, σ_y^2 and σ_{xy} are the measurement noise covariance values, Flag is an index used to see if a measurement has been used by the filter and CID is another index color ID used for identification of each tracking filter/measurement. Basically, first five variables will be available in real time to the tracking algorithm through an interface, in an actual implementation. Here the input data is given to the algorithm through hand coded assembly instructions.

6. RESULT AND CONCLUSION

For the evaluation of our algorithm, a simulator for MSA hardware architecture was written as the actual silicon for MSA is expected in September, 2001. For our evaluation a two crossing target scenario was used, the sensor was moving parallel to the target, towards the targets at a very high speed compared with the targets. The two targets at start up were approximately 30 Km from the sensor and cross each other at 30th scan. After successful initialization of two targets, tracking proceeded with each tracking filter accepting only one measurement as they are well apart. Track splitting (branching) starts at the 24th scan, maximum number of tracks occur at 31st scan. At 38th scan all the redundant tracks are pruned and normal tracking of two targets resume. It should be noted here that lots of detail about tracking is not revealed [5][6] as the actual emphasis is about utilizing the architecture advantages of the processor.

The main concern was to implement the computationally expensive algorithm in such a way to take the advantages of the DSP instruction set for MSA also, one of the reason of selecting Blackfin was its DSP assembly language algebraic syntax which is probably ideal for such an application. As pointed out earlier, the algorithm is hand coded in assembly in order to exploit the DSP. Where ever necessary C is used for the actual implementation. As the architecture supports richly encoded instruction set for such a demanding task, in some cases multiple operations are performed with less number of cycles, this obviously improves real time processing. Here, we just present one particular sequence where the implementation helps in achieving our mission of exploiting its architecture. At each radar scan incoming measurements (returns) are to be tested with each established track (path) for a possible match. The following computation sequence that includes multiplication of three matrices of dimensions [(1x2)*(2x2)*(2x1)] is to be calculated and then compared with a known value to find out if a possible match exits. Therefore, each track should accept the true measurement from the same target, although this situation changes at the time of multiple measurements existing in the same vicinity. In our case when the target cross each other. Now to evaluate this expression

$$\mathbf{d}^2{}_n = \underline{\nu}_n{}^{\mathbf{T}}\mathbf{B}^{-1}{}_n\underline{\nu}_n$$

suppose $\nu_n^{\mathbf{T}}$ of dimension (1x2) is stored in data register R0 such that:

$$\mathrm{R0.L} = \underline{\nu}_n{}^{\mathbf{T}}(0,0) \quad \mathrm{R0.H} = \underline{\nu}_n{}^{\mathbf{T}}(0,1)$$

(0, 0) represents first row, first column element of the matrix and so on. Similarly $\mathbf{B}_n^{-1}$ is stored in two registers as

$$\mathrm{R2.L} = \mathbf{B}_n^{-1}(0,0) \quad \mathrm{R2.H} = \mathbf{B}_n^{-1}(0,1)$$

$$\mathrm{R3.L} = \mathbf{B}_n^{-1}(1,0) \quad \mathrm{R3.H} = \mathbf{B}_n^{-1}(1,1)$$

By using MSA instruction set we can perform this task in just four steps:

$$\left.\begin{array}{r}\mathrm{A0} = \mathrm{R0.L} * \mathrm{R2.L}\\ \mathrm{A1} = \mathrm{R0.L} * \mathrm{R2.L}\end{array}\right] (\mathrm{Parallel\ \ Op})$$

$$\left.\begin{array}{r}\mathrm{R4.L} = \mathrm{A0}+ = \mathrm{R0.H} * \mathrm{R3.L}\\ \mathrm{R4.H} = \mathrm{A1}+ = \mathrm{R0.H} * \mathrm{R3.H}\end{array}\right] (\mathrm{Parallel\ \ Op})$$

$$\left.\begin{array}{r}\mathrm{A0} = \mathrm{R4.L} * \mathrm{R0.L}\\ \mathrm{A1} = \mathrm{R4.H} * \mathrm{R0.H}\end{array}\right] (\mathrm{Parallel\ \ Op})$$

$$\mathbf{d}^2{}_n = \mathrm{A0} + \mathrm{A1}$$

The above calculation is just a replica of calculations performed repeatedly in a recursive filter, which basically suite such DSP architecture. Because DSP processors are characterized by tight code loop and in-fact data flow driven. Therefore, this type of loop/calculations can be implemented very efficiently. Actually, the number of times $\mathbf{d}^2{}_n$ is evaluated to find a probable track-measurement pair increases exponentially with increasing number of track splitting. If it is possible to do such calculations quickly the efficiency will increase in terms of time. The simulated study carried out here has logically determined the advantage of using MSA core for such an application. The simulated study after running a number of scenarios and careful evaluation reveals that MSA may provide 20 to 30% improved performance in processing for such a computationally intensive application. Although, no bench mark evaluation criterion is used for such evaluation.

The selection of an operating system suitable for implementation in a target tracking system presents many challenges. The main challenge is selecting a processor and complimentary operating system able to handle the computationally expensive load. The combination of Blackfin architecture and Linux operating system meets this challenge with Linux Kernel. The second being the ability of the operating system to respond in a deterministic manner. This can be achieved by operating system that are designed to operate in Real Time. Linux was selected in this study as its characteristic performance achieves close to a predictable real time response under known loads. However, Linux on its own is not a suitable Real Time Operating System (RTOS) and some characterize Linux's response time as "Soft Real Time", the observed jitter is larger than the jitter associated with running Linux under the control of a real time scheduler such as found in "Real Time Executive in C for DSP

(RTXCDSP)" or the Real Time Application Interface (RTAI) [11].

Recently, efforts are underway to provide the RTAI subsystems for use on various MMU-less processors. If one is to develop an embedded system using such a DSP (MSA) for real time applications, uClinux with uClibc may provide an excellent platform for such real time target tracking implementation. The work is already underway for porting uClinux and uClibc for this processor (MSA).

7. REFERENCES

[1] P. L. Smith and G. Buechler, "A branching algorithm for discriminating and tracking multiple objects," IEEE Transactions Automatic Control, Vol. AC-20, February 1975, pp 101-104.

[2] Y. Bar-Shalom and T. E. Fortmann, "Tracking and Data Association", Academic Press, Inc. 1988.

[3] S. S. Blackman, " Multiple Target Tracking with Radar Applications", Artech House, Inc. 1986.

[4] D. P. Atherton, E. Gul, A. Kountzeris and M. Kharbouch, "Tracking Multiple Targets using parallel processing," Proc. IEE, Part D, No. 4, July 1990, pp 225-234.

[5] D. P. Atherton, D. M. A. Hussain and E. Gul, "Target tracking using transputers as parallel processors," 9th IFAC Symposium on Identification and System Parameter Estimation, Budapest, Hungary July 1991.

[6] M. Kharbouch, "Some investigations on target tracking," D. Phil thesis, Sussex University, 1991.

[7] The Carmel DSP core user's manual, infineon, 1999.

[8] Clifford Liem, Pierre Paulin, Ahmed Jerraya, "Address calculation for retargetable compilation and exploration of instruction set architecture."

[9] The scientist and engineer's guide to Digital Signal Processing.

[10] 1999, DSP architecture directory.

[11] Michael Durrant, Michael Leslie, Using Linux for MMU-less micro-processors. Electronics Engineering UK, Feb. 2001.

8. ACKNOWLEDGEMENTS

The author would like to thank Arcturus Networks Inc. and Lineo Canada Corp for their support in every aspect for doing such a study, and special thanks to Analog Devices Inc., for providing all the technical details, documents etc., about MSA which really encouraged the author to carry out such investigation. The author also likes to extend his appreciation and gratitude to his colleagues, having been very kind to extend their comments at difficult times. Also there is lot of literature available on the Internet about DSP, few of the references are given here for the acknowledgement of this wonderful technology feast, thanks Internet.

An Approach to Injecting faults into Hardened Software

Dave Edwards, Lori Matassa
Intel Corporation

Abstract

There are many efforts within the Linux* community to produce a distribution of Linux* that meets industry standards for quality and reliability. There has been acknowledgment for the need to introduce faults into various software layers of the Linux* OS to achieve this. This paper focuses on the results of our development of a prototype fault injection harness. The prototype focused on a black box approach for injecting faults into device drivers. The technology proved in this prototype can be applied to any software layer in the operating system. This presentation proposes and proves the feasibility of a method for injecting faults called, “state analysis.” This method is the key to our black box approach for driver hardening. It does not require a test writer to have intimate knowledge of the implementation for the driver. It also provides a solid foundation for driver developers to augment the fault injection harness to meet whatever the Linux community presents to the world in the way of driver hardening criteria. The target audience includes developers focusing on Linux* hardening (Drivers and Kernel), test engineers looking for a starting point to fault injection, and anyone looking for input into the kinds of capabilities that can be provided by the use of fault injection.

1 Introduction

A hardware device has a finite set of functions to perform and a rigid programmatic method for utilizing its functions. A device driver contains many code paths that exercise the functionality of a hardware device. This paper discusses the learning obtained from a prototype fault injection test harness in which hardware faults are emulated and injected into Linux* device drivers.

1.1 Purpose and Scope

The purpose of this paper is to provide insight into fault injection through the discussion of a prototype fault injection harness implementation. This paper and will provide its audience with one proposed method for ensuring the hardening level of a device driver.

Actual design details of the entire prototyped implementation are not included in this document.

1.2 Intended Audience

This paper is intended for development and test engineers and anyone interested in designing, implementing or utilizing fault injection capabilities that are reproducible, portable across software revisions and flexible.

1.3 Recommended Reading

The appendix of this paper contains background and reference information. This information is provided to assist in clarifying concepts that are touched upon in this document. It is recommended that these be looked at closely once the basic concepts are understood. It is expected that these sections will provide sufficient detail to explain anything that has not been directly addressed in the main portion of this paper.

The overview sections contain information about the purpose behind fault injection and how each of the software components are related with regards to their interfaces.

The section titled, "Fault Injection (FI) Protoype" describes each major component of the prototype in a little more detail. The purpose of which is to describe the intent and major function of each sub-component.

Definition of Terms

State: A deterministic path from one starting point to another. A state refers to the various states of the hardware as it is programmed (by a driver) for its particular function.

State Analysis: The process of tracking the state of hardware and making decisions about what to do as various hardware states are encountered.

State Machine: The mechanism that can track and respond to changes in hardware state. The machine itself consists of a collection of code segments.

Code Segment: A simple code fragment that provides the functionality of the state machine.

State Machine Test: This is the input file used by the State Machine Compiler to create a binary state table that can be dynamically loaded.

FI Engine: Fault Injection Engine. The concept of an engine refers to the central control component for monitoring state and injecting faults into a device driver.

2 Driver Hardening Overview

Device drivers can be a source of operating system instability and are often contributors to system degradation and/or unscheduled outages. Therefore, device drivers must be robust. A hardened device driver is a robust device driver. Hardened device drivers are designed and developed with the focus of minimizing the instability and downtime of the system.

Measuring the hardness of a driver is difficult and unclear. A concept known as driver hardening levels is documented in a white paper titled, *"Device Driver Hardening and Manageability."* The white paper can be found at the **http://developer.intel.com** web-site. These levels are used to define fundamental hardened driver guidelines, measure the hardness of the driver and create a better understanding as to how robust a driver is. These guidelines are used by device driver writers who wish to support higher levels of availability through the use of some or all of the hardening techniques described in each level. The levels include:

Level 1 - Stability and Reliability Includes good coding practices and requires fault injection testing.

Level 2 - Manageability Provides information that can be used by driver management applications to understand the status of the system and to identify potential problems that might be growing. This information includes driver statistics, event logging and driver diagnostics. All of this information is essential in proactively recognizing potential problems. Together, this information can identify a problem and report it immediately so that downtime can be prevented or at least minimized. Therefore, handling the fault gracefully.

Level 3 - High Availability This is the highest level of a hardened driver. High availability systems minimize system downtime. Guidelines in this level support high availability features which enable a driver to repair or reconfigure devices without needing to power down or reboot the system. These guidelines also include fault recovery to the extent that when a fault is identified, the driver repairs the fault if it can keep the device in service and, at a minimum, isolates the operating system from being affected.

Fault injection can be applied to any level of driver hardening. for illustration purposes, this paper focuses on level 1 where a developer is ensuring the integrity of the source code through fault injection, before introducing the work effort to validation.

For purposes of this paper, this paper concentrates on Level 1 Hardening. Level 1 Hardening guidelines specify that hardened drivers must be fault injected tested.

Good coding practices alone cannot ensure the stability and reliability of a system. Device drivers typically are written and tested with emphasis on the normal operation of the hardware. Details as to how a driver identifies and recovers from faulty hardware or system conditions are often minimal.

Hardened device drivers are designed to be more robust because they are coded to expect anomalies and process them in a way that minimizes the impact to the overall system, thus preventing unplanned downtime of the system. The implementation of the code should test for such things as: values that are illegal, states that should never occur and expect that the device should complete a command within a defined amount of time.

The only way to test a driver's robustness is to include tests that purposefully inject conditions that simulate hardware and system faults. This is known as Fault Injection Testing.

There are several ways to inject faults into a system. The most common method involves altering the branch paths to purposefully modify good data into bad data. This can be accomplished with many tools. For instance, the use of in-circuit emulators (ICE's) or in-target probes (ITP's) can change the execution path and data values at run-time. Other methods include the use of debuggers or special code additions with the specific purpose of causing error paths to be exercised. This is known as "white box" testing where the goal is to maximize code coverage.

White box tests pinpoint exact areas and values within the software that are changed. This means that the test is implemented knowing exactly where and what will occur. There are a few issues with white box testing. First, the setup for the test is very labor intensive and in some cases, requires complete manual intervention to control and execute tests, thus making it nearly impossible to repeat test results consistently. Second, there is the possibility that the object code under test is not the same object code that is shipped as the final product. This is not acceptable to most suppliers of high availability and hardened systems.

There is a method of implementing fault injection tests that is fully automated, can provide reproducibility in test results and uses the same version of object code for fault injection testing as the shipped product. This method, known as "black box" testing. Black box testing uses carefully designed tests that emulate faults at software layers below the component under test. In the case of the prototype fault injection effort, faults are emulated in the hardware layer and the component under test is a hardened device driver.

This method for injecting faults that can be automated, provides reproducible test results, is portable across driver revisions, and is simple to augment as software capabilities and test requirements change. This concludes the overview of driver hardening, why fault injection testing is required and different approaches testing. The remainder of this document will describe the key concepts and critical design details for a prototype implementation.

3 Software Overview

There are three main components to the prototype, the System Driver, the Fault Injection Engine and the Common Driver Interface (hooks). The system driver is the component under test that utilizes a Common Driver Interface (CDI) that contains software hooks to interface with the FI Engine component transparently. The FI Engine is the core software component, providing all of the services necessary to inject faults into a device driver.

Device drivers execute in kernel space. As such, the FI Engine is a driver with interfaces designed to allow connections to occur between a system driver and itself. The CDI (used by the system driver) contains special software hooks that allow it to connect to the FI Engine during the driver initialization sequence. Once the connection is made there is a method to make a direct call to the FI Engine from the connected system driver. For the prototype, the CDI consisted of macros that were created to represent an abstraction on the existing Linux* macro set for programmed IO, DMA and PCI configuration.

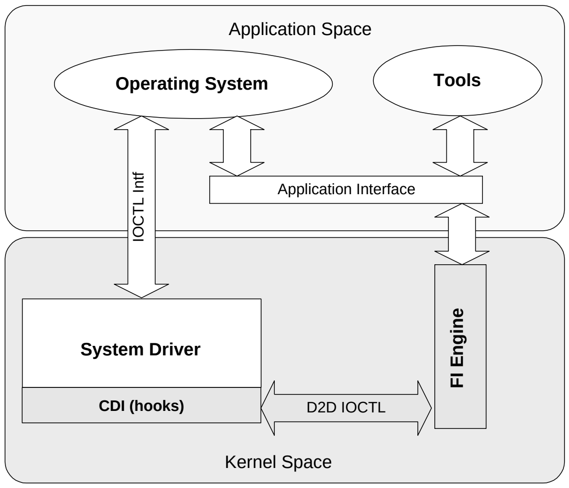


Figure 1: Major Software Components

State analysis is the process of tracking I/O transactions for the purpose of 1) detecting hardening violations 2) injecting faults at specific points in the usage model for that hardware and 3) to emulate the appropriate hardware behavior from the time a fault is injected to the time the hardware is expected to be executing normally.

Hardening violations are categorized by warnings, and rule violations. A warning is some indication that a driver inappropriately accessed registers given the current state of the hardware. This can be useful for detecting known hardware problems that have potential for causing failures under adverse conditions, but aren't guaranteed to do so every time. This is a way for a test engineer to create warning flags for special events. A hardening violation is one in which a driver responds to a failure in such a way that it is known to be inappropriate. For example, a driver may not clear interrupts within a control register after a given fault. In these situations, it is thought that the driver will cause a system failure either immediately, or shortly thereafter.

The process of writing a test begins by using data sheets and any other hardware documentation that specifies where the hardware can fail. Once hardware failures are understood, a test writer can create fault scenarios in which the hardware could fail during its normal operation. These scenarios are then translated to state machine form in which the state of the hardware is tracked, and at various points a decision can be made to inject a fault between state transitions. Once failure scenarios are translated to a state machine form the test writer can, compile the test and load the test into the FI engine before the system driver is loaded.

Figure 2: "State Machine Capabilities" illustrates the services that a state machine engine provides and the types of information that a test writer must have to track the state of hardware, inject faults, and emulate hardware fault behavior.

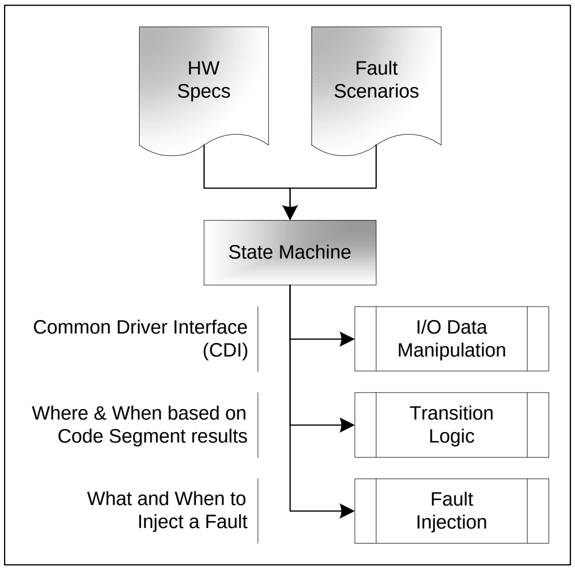

HW Specs
Fault Scenarios
State Machine
Common Driver Interface (CDI)
Where & When based on Code Segment results
What and When to Inject a Fault
I/O Data Manipulation
Transition Logic
Fault Injection

Figure 2: State Machine Capabilities

Figure 2 also illustrates the three main components of the state machine, I/O Data Manipulation, Transition Logic and Fault Injection capabilities. I/O Data Manipulation consists of passing the CDI input parameters to the FI engine and allowing the engine to use the values to determine state and to manipulate the values, i.e. the hooks. Transition Logic consists of a method for executing code associated with a state and being able to specify which state to transition to. This could also be considered execution flow control. The Fault Injection piece defines the method in which the test writer specifies exactly when I/O Data is to be manipulated and how the FI engine will respond for subsequent calls to the CDI.

When all of these things are used properly, a software engineer can track the state of a hardware device. This allows them to specify exactly when to inject a fault and how to behave once the fault is injected.

4 Fault Injection (FI) Prototype

The prototype consists of a system driver, state machine compiler, and an FI engine driver. The core technology described in this paper is the state analysis component of the FI engine. It is responsible for the state tracking and fault injection capabilities. Figure 1: "Major Software Components" outlines the relationship between all of these components which are described in the following sections.

4.1 System Driver

The System Driver (illustrated in Figure 3) is built using the CDI to access hardware resources. The main advantage is that this method gives the engine access to all the possible input and output parameters of the transaction. As such, a test engineer can make decisions and modify any of the data that gets transmitted between the CDI and hardware. In general, execution control is passed to the engine after a read transaction and before a write transaction such that the FIE can decide what to do with the data that will be returned to the driver or written to hardware.

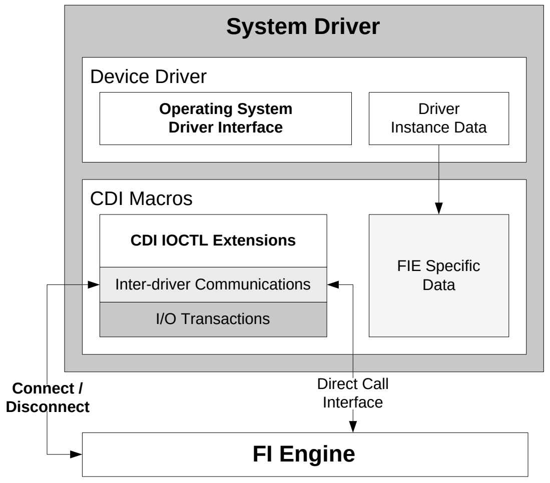


Figure 3: System Driver Block Diagram

The process for initiating fault injection testing involves a dynamic connection sequence. When a system driver is loaded the driver initiates a connection sequence by calling kernel routines that return a handle to the FI Engine driver. Once the handle is known, the system driver initiates driver to driver IOCTL calls through the kernel to the FI Engine. The first call to the FI Engine sends a request to connect. The FI Engine grants this request and returns a handle to a data structure representing instance data for that connection. The instance handle contains the address of the FI Engine entry point function. The CDI uses the entry point address to call directly into the FI Engine.

When a system driver is unloaded, a disconnect sequence is initiated through the same driver to driver IOCTL interface. At which time the FI Engine will perform cleanup on any state machine configuration data associated with the connection.

4.2 Fault Injection Engine

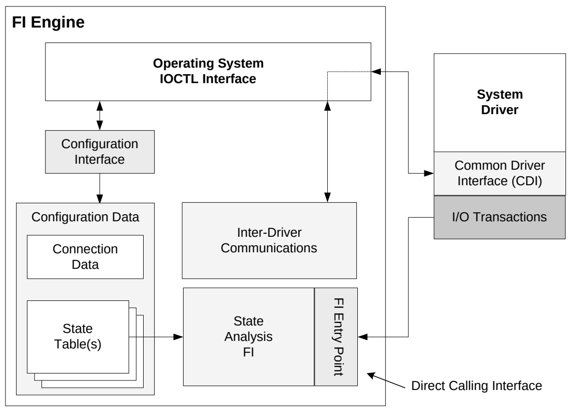


Figure 4: Fault Injection Engine Block Diagram

The FI Engine is illustrated in Figure 4: "Fault Injection Engine Block Diagram." The various blocks within the diagram represent key internal components. Lines are drawn to show the component's inter-relationships. Arrows have also been added to hint at data flow. The system driver connects through special macros of the CDI and is intercepted by the Inter-Driver Communications (IDC) component. The IDC creates data structures and initializes internal subsystems. The Configuration Interface provides an application with the ability to load a state machine table dynamically. The state analysis component is called directly by the CDI once the connection sequence completes.

NOTE: the actual I/O access will occur in the system driver, not the engine. The engine only modifies the data before or after the actual I/O.

Figure 5: "CDI Hooks to FI Engine Flow Diagram" illustrates the flow of execution with respect to the CDI hooks and the FI Engine. It starts with a driver making use of a CDI Macro. Typically an I/O transaction involves either a read or a write. Thus the concept is fairly simple. You inject a fault into a driver by modifying values returned by the read transaction to trick the driver into thinking that a status register contains an error value. You can also inject a fault into a driver by modifying data from a write transaction before the data is transmitted through the I/O interface.

Once the FI Engine receives execution control, it begins traversing the state table, executing code segments that are related to the current state of the hardware. On entry to the engine, the state machine determines where it left off the last time the engine was called. When a driver first connects, the starting state will be the very first state of the state table. The state machine parses the state table entry for the code segment and then executes code associated with that code segment. The return value of the code segment is then used to determine which state should be traversed to. This will continue until the ExitStateMachine code segment is executed. This is a special piece of code that allows a test writer to specify where

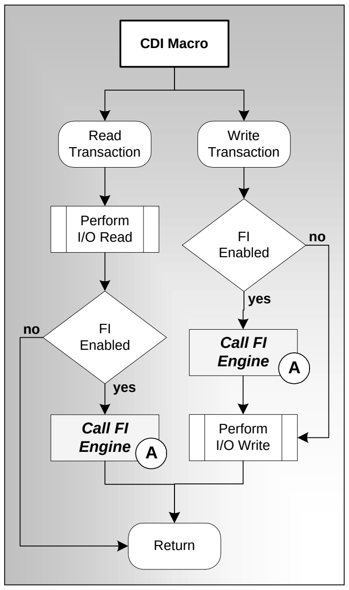


Figure 5: CDI Hooks to FI Engine Flow Diagram

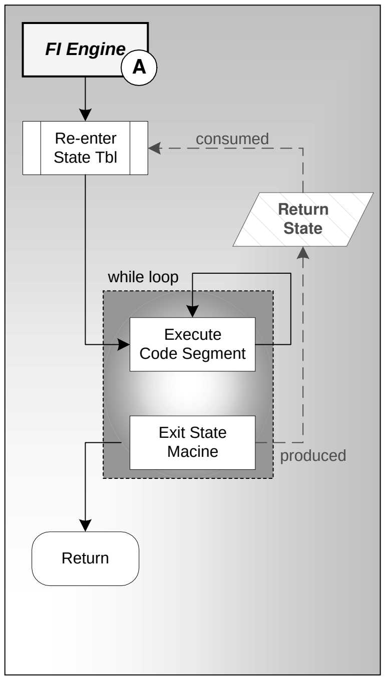


Figure 6: FI Engine Flow Diagram

the state machine will continue the next time it is called and exits the FI Engine to allow the driver to continue running.

4.3 FI Compiler

The compiler (Figure 7: FI Compiler Component Diagram) produces a binary output file that is loaded into the engine with a command line tool. The compiled file is translated to a state table and stored for retrieval when the device driver makes a connection to the FI Engine.

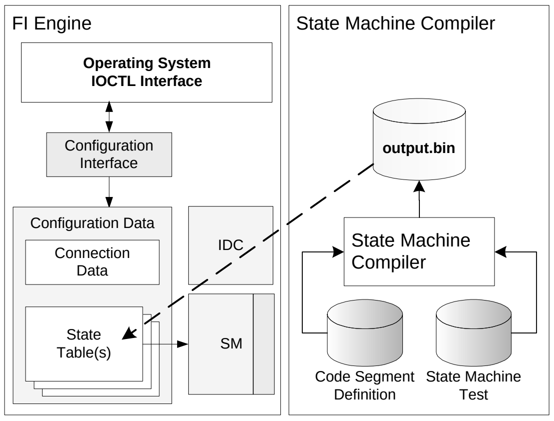


Figure 7: FI Compiler Component Diagram

Input for the compiler consists of a state machine test file and a code segment definition file. The state machine test file contains "source" text that is translated to binary form by the state machine compiler. The code segment definition file is created from the code segment definition data structure mentioned in the next section. The code segment definition file is created as part of the build process. When a test is compiled the code segment definition file is used to validate the input parameters in the test being compiled.

4.4 State Machine

The overview section of this paper made claims that this prototype proves the feasibility of creating reproducible test results, portability across driver revisions, and simple augmentation of the state machine. The first portion of this section is dedicated to explaining why it can do these things and the remainder gives insight into how it can do them.

The state machine is central to the reproducibility of test results consistently across software revisions. It does this by allowing a test developer to track the state of the hardware and specify the exact moments in which a fault will be injected. By tracking hardware state, the test developer does not have to tie the test to the implementation of the driver, only the implementation of the hardware. Thus, as long as the hardware fault scenarios don't change, neither does the test, for any revision of the driver.

The state machine is designed to be augmented as test requirements and driver design dictate the need for change. There are three supporting components to the state machine, the state machine switch statement, the state table, and the code segment definition structure. These will be described a bit more, shortly. A change to the machine is a change only to each of these three items.

The state analysis component of the FI Engine (Figure 4) contains the state machine. The state machine is responsible for traversing a list of states within a state table. At each state transition, a code segment is executed, which can be considered analogous to a CPU executing an instruction. Code segments are simple, small code fragments that do not depend on one another to complete execution. However, they do have a mechanism for passing data to between states. For the prototype, this was done through a special Reverse Polish Notation (RPN) based stack.

The state machine is a while(1) loop with a big switch statement inside. Code segments are the case statements with the associated code to be executed on a transition into the state. Each state table entry contains a list of states to transition to, based on the return value from the code segment. As illustrated in the Figure 8: "Sample State Table Linkage," the next state list is an array where the next state is determined by using the return value as the index to the array. The next state is pointed to by the contents of the value indexed in the array. The code sample at the end of this section illustrates this process. The end result is very low overhead between executing code segments.

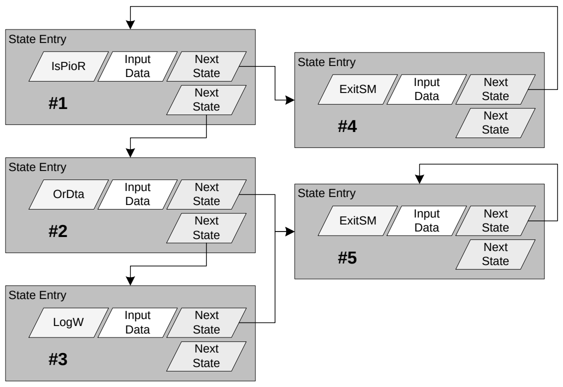


Figure 8: Sample State Table Linkage

Figure 8 also shows a sample diagram representing the contents of a state table and how the flow is controlled. The first field is the code segment tag; in this case there are four code segments, IsPioRead, OrData, LogWarning and ExitStateMachine. These would be used by the test engineer to create a state machine test.

In order to extend the state machine, a data structure that defines the code segments is created. Each table entry contains the following information:

String: The reserved word for the code segment.

Number of Inputs: Number of data parameters required for input.

Number of Returns: Number of possible return values.

This structure is shared between the state machine and the compiler. The compiler uses it in the form of a file (saved to disk by the build process) called the code segment definition. The compiler can load the structure to validate the syntax of the test source code, created by the test developer. The state machine engine uses the code segment definition structure to traverse the state table and access data input parameters. A typical code segment definition would look like the following:

Reserved Word	Inputs	Returns
"PioRead",	0	2
"OrData",	1	1
"LogWarning",	0	1
"ExitMachine",	1	1

The following is an example of the state machine implementation referenced in Figure 8, and its associated code segments. See Appendix B for a complete, functional state machine example.

```
FIE_EntryPoint(InstanceHandle *handle) {
   StateTableEntry *this_state = handle->CurrentStTblEntry;
   CdiParameters   *cdi_param  = handle->CdiParam;
   BOOL             exit_flag = FALSE;

   while (exit_flag == FALSE) {
      switch(this_state->CodeSegment) {
         case CODE_SEG_IsPioRead:
            cs_return = 0; /* FALSE */
            if (cdi_param->Transaction == CDI_PIO_READ) {
               cs_return = 1; /* TRUE */
            }
         break;
         case CODE_SEG_AndData:
            cs_return = 0; /* only 1 return */
            cdi_param &= this_state->StateData;
         break;
         case CODE_SEG_LogWarning:
            cs_return = 0; /* only 1 return */
            WriteLog("Warning Violation...");
         break;

         case CODE_SEG_LogViolation:
            cs_return = 0; /* only 1 return */
            WriteLog("Hardening Violation...");
         break;

         case CODE_SEG_ExitStateMachine:
            cs_return = 0; /* only 1 return */
            /* This code segment’s data item contains a pointer
             * to the state to execute on the next entry to the
             * state machine.
             */
            handle->CurrentStTblEntry = this_state->StateData;
            exit_flag = TRUE;
         break;
         default: { /* invalid code segment */ };
      }
      /* Set the next code segment to be executed based on the
       * return value of the current code segment.  This is just
       * an array of pointers to state table entries.
      this_state->CodeSegment = this_state->NextState[cs_return];
   } /* end while */
}
```

5 Summary

The prototype fault injection harness proved that it is possible to create a state machine language and that it is possible to track the state of hardware from initialization all the way through the normal execution cycles of a driver.

Tracking the state of hardware is critical to making decisions about when to inject a fault. It is also what allows test results to be repeatable and allows a test to be portable across driver revisions.

Keep in mind that even though this paper focuses on drivers, any software component can make use of the principles.

6 Acknowledgments

Special thanks to my fellow co-workers for their contribution to this effort: Donald Long, Daniel Vanhoozier, Kimberly Davis, Ryan Kummet, Susan Foster, Victoria Genovker, Grace Hawley, and Lori Matassa.

References

[Intel] Lori Matassa *Device Driver Hardening and Manageability* Intel Corporation. `http://developer.intel.com/` (2001)

7 Appendix A - Sample Test

This is a sample state machine test file. The text below describes three major aspects of the system driver under test, normal execution flow, a special IOCTL interface and controlled hardening violations.

A sample driver was designed to illustrate concepts and prove the feasibility of a state machine language. The LCD device is a simple serial port device that plugs into COM1. The sample driver was written to interface to the LCD device. Any programming of the hardware occurs with the serial controller.

The compiled state machine test is loaded into the FI engine prior to loading the LCD driver. When the LCD driver is loaded it makes a connection to the FI Engine. From the moment the driver successfully connects, the state machine is monitoring programmed I/O.

Once initialization completes (successfully) the normal operation of the driver can begin. An application running in the background sends constant messages to the driver through the read/write operating system interface as a character mode driver. The state machine test is designed to inject a fault into the write status register every 1000 character writes to the display. When the driver detects a failure it flashes an error message to the display, re-initializes the device and continues to accept character messages from the background application.

There are some special flags that can be set to demonstrate different aspects of fault injection. The LCD driver has a custom IOCTL interface to allow an application to do three things, re-initialize the driver, set the Warning Violation code path and set the Hardening Violation code path. The background application has the ability to utilize these features on demand.

Figure 9: "Flow Diagram for the Sample (LCD) Driver" illustrates the various code paths that can be exercised depending on the run-time switches controlled by the driver's custom IOCTL interface. There are three demonstrations from this, fault injection during driver initialization, hardening warnings, and hardening violations.

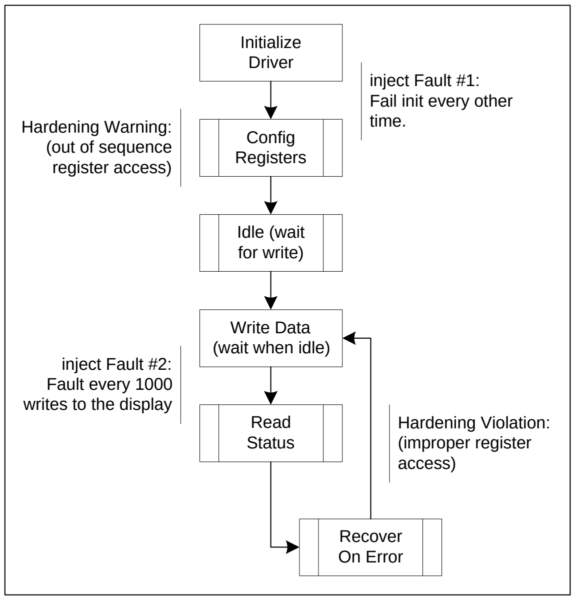

Initialize Driver
inject Fault #1: Fail init every other time.
Hardening Warning: (out of sequence register access)
Config Registers
Idle (wait for write)
Write Data (wait when idle)
inject Fault #2: Fault every 1000 writes to the display
Read Status
Hardening Violation: (improper register access)
Recover On Error

Figure 9: Flow Diagram for the Sample (LCD) Driver

There is a run-time switch within the initialization sequence that forces the driver to execute a warning violation when initialization sequence is repeated. The state machine will detect an out of sequence register violation and write a string to the log file. To demonstrate the ability to inject faults during initialization, the state machine is also designed to inject a fault every other time the background application re-initializes the LCD driver.

The final demonstration is to inject a fault during the character write process and force the driver to inappropriately recover from the failure; this is documented as a hardening violation. The driver simply fails to clear the display after detecting a failure and the state machine can detect the absence of that action.

The state machine definition (below) file is fairly complicated to read, but the fact that it can do what it's supposed to, is a major milestone to prove the capabilities of this method of injecting faults and detecting violations. The code segment definitions for the sample machine below, are not defined in this document. The intent is to provide a reference to a working state table and the driver model specified in previous pages.

```
## Comments are permitted in a state definition file if they are
#  preceded by a ’\#’.
#
#  The syntax of this state machine test file is as follows
#  (in BNF notation):
#
#    <state> := STATE <state_name> <cs_name> [<data>...] [<result>...];
#       <cs_name> := <alpha_num>
#       <state_name> := <alpha_num>
#       <result> := <state_name>
#       <data> := <integer>

########################################################################
#-----------------------------------------------------------------------
# Initialization
#-----------------------------------------------------------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetTraceLevel   SetTrace            0
stPushFmFiCnt; STATE stPushFmFiCnt     StkPush             0
stInitFmFiCnt; STATE stInitFmFiCnt     StkPopStore         1
stPushLcFiCnt;

STATE stPushLcFiCnt     StkPush             0         stInitLcFiCnt;
STATE stInitLcFiCnt     StkPopStore         2         stDlabSPend;

#-----------------------------------------------------------------------
# Wait for the DLAB bit to be set
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stDlabSPend       CheckTransaction    1         stDlabSNot
                                                      stChkDlabSAddr;
STATE stDlabSNot        ExitStateMachine    0         stDlabSPend;
STATE stChkDlabSAddr    CheckAddress        0x3FB     stDlabSNot
                                                      stChkDlabSBit;
STATE stChkDlabSBit     PushTransData       0         stPushDlabSMask;
STATE stPushDlabSMask   StkPush             0x80      stDlabSAnd;
STATE stDlabSAnd        DataAnd             0         stDlabSPushCmp;
STATE stDlabSPushCmp    StkPush             0x80      stDlabSCompare;
STATE stDlabSCompare    CompareEq           0         stDlabSNot
                                                      stGotoDLsbPend1;
STATE stGotoDLsbPend1   ExitStateMachine    0         stSetDLsbPend1;

#-----------------------------------------------------------------------
# Wait for the Divisor LSB to be set first
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetDLsbPend1    CheckTransaction    1         stGotoDLsbPend1
                                                      stChkDivLsbAddr1;
STATE stChkDivLsbAddr1  CheckAddress        0x3F8     stChkDivMsbAddr2
                                                      stGotoDMsbPend1;
STATE stGotoDMsbPend1   ExitStateMachine    0         stSetDMsbPend1;
```

```
#------------------------------------------------------------------------
# Wait for the Divisor MSB to be set second
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetDMsbPend1    CheckTransaction    1         stGotoDMsbPend1
                                                      stChkDivMsbAddr1;
STATE stChkDivMsbAddr1  CheckAddress        0x3F9     stGotoDMsbPend1
                                                      stGotoDlabUPend;
STATE stGotoDlabUPend   ExitStateMachine    0         stDlabUPend;

#------------------------------------------------------------------------
# Wait for the Divisor MSB to be set first
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stChkDivMsbAddr2  CheckAddress        0x3F9     stGotoDLsbPend1
                                                      stPrtDivSetWarn;
STATE stPrtDivSetWarn   LogWarning          0         stGotoDLsbPend2;
STATE stGotoDLsbPend2   ExitStateMachine    0         stSetDivLsbPend2;

#------------------------------------------------------------------------
# Wait for the Divisor LSB to be set second
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetDivLsbPend2  CheckTransaction    1         stGotoDLsbPend2
                                                      stChkDivLsbAddr2;
STATE stChkDivLsbAddr2  CheckAddress        0x3F8     stGotoDLsbPend2
                                                      stGotoDlabUPend;

#------------------------------------------------------------------------
# Wait for the DLAB bit to be unset
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stDlabUPend       CheckTransaction    1         stDlabUNot
                                                      stChkDlabUAddr;
STATE stDlabUNot        ExitStateMachine    0         stDlabUPend;
STATE stChkDlabUAddr    CheckAddress        0x3FB     stDlabUNot
                                                      stChkDlabUBit;
STATE stChkDlabUBit     PushTransData       0         stPushDlabUMask;
STATE stPushDlabUMask   StkPush             0x80      stDlabUAnd;
STATE stDlabUAnd        DataAnd             0         stPushDlabUCmp;
STATE stPushDlabUCmp    StkPush             0         stDlabUCompare;
STATE stDlabUCompare    CompareEq           0         stDlabUNot
                                                      stGotoSetLcPend;
STATE stGotoSetLcPend   ExitStateMachine    0         stSetLcPend;

#------------------------------------------------------------------------
# Wait for set of the line control data
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
```

```
#---------------------#-------------------#---------#-------------------
STATE stSetLcPend       CheckTransaction    1         stNotSetLc
                                                      stChkLcAddr;
STATE stNotSetLc        ExitStateMachine    0         stSetLcPend;
STATE stChkLcAddr       CheckAddress        0x3FB     stNotSetLc
                                                      stChkLcFip;

#-----------------------------------------------------------------------
# Fault injection point. Inject a line control data fault at every 'X'
# interval.
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stChkLcFip        IncStore            2         stPushLcFiReg;
STATE stPushLcFiReg     StkPushStore        2         stPushLcFiCmp;
STATE stPushLcFiCmp     StkPush             2         stCompLcFi;
STATE stCompLcFi        CompareEq           2         stGotoWritePend
                                                      stSetLcError;
STATE stSetLcError      PushTransData       0         stPushLcErrData;
STATE stPushLcErrData   StkPush             0xFC      stGetLcTransData;
STATE stGetLcTransData  DataAnd             0         stSetLcTransData;
STATE stSetLcTransData  PopTransData        0         stPushLcFi0Cnt;
STATE stPushLcFi0Cnt    StkPush             0         stResetLcFiCnt;
STATE stResetLcFiCnt    StkPopStore         2         stGotoWritePend;

#----------------------------------------------------------------------
# Wait for the write. If the write is text goto the "wait for write
# completion section. If the write is the DLAB then go back to the
# initalization section.
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stWritePend       CheckTransaction    1         stNotWrite
                                                      stChkWriteAddr;
STATE stNotWrite        ExitStateMachine    0         stWritePend;
STATE stChkWriteAddr    CheckAddress        0x3F8     stChkWrtLcAddr
                                                      stGotoWrtVerify;
STATE stChkWrtLcAddr    CheckAddress        0x3FB     stNotWrite
                                                      stWrtChkDlabSBit;
STATE stWrtChkDlabSBit  PushTransData       0         stWrtPshDlabSMask;
STATE stWrtPshDlabSMask StkPush             0x80      stWrtDlabSAnd;
STATE stWrtDlabSAnd     DataAnd             0         stWrtDlabSPushCmp;
STATE stWrtDlabSPushCmp StkPush             0x80      stWrtDlabSCompare;
STATE stWrtDlabSCompare CompareEq           0         stNotWrite
                                                      stGotoDLsbPend1;
STATE stGotoWrtVerify   ExitStateMachine    0         stWrtVerify;

#-----------------------------------------------------------------------
# Wait for the write to complete
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stWrtVerify       CheckTransaction    0         stNotWrtVerify
                                                      stChkWrtVfyAddr;
```

```
STATE stNotWrtVerify    ExitStateMachine    0         stWrtVerify;
STATE stChkWrtVfyAddr   CheckAddress        0x3FD     stNotWrtVerify
                                                      stPushLsr;

STATE stPushLsr         PushTransData       0         stPushXmitMask;
STATE stPushXmitMask    StkPush             0x20      stAndXmitStat;
STATE stAndXmitStat     DataAnd             0         stPushXmitCmp;
STATE stPushXmitCmp     StkPush             0x20      stChkXmitStat;
STATE stChkXmitStat     CompareEq           0         stNotWrtVerify
                                                      stChkFmFip;

#-----------------------------------------------------------------------
# Fault injection point. Inject framing errors at every 'X' interval.
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stChkFmFip        IncStore            1         stPushFmFiReg;
STATE stPushFmFiReg     StkPushStore        1         stPushFmFiCmp;
STATE stPushFmFiCmp     StkPush             1000      stCompFmFi;
STATE stCompFmFi        CompareEq           1000      stChkWrtError
                                                      stSetFmError;
STATE stSetFmError      PushTransData       0         stPushFmErrData;
STATE stPushFmErrData   StkPush             0x08      stGetFmTransData;
STATE stGetFmTransData  DataOr              0         stSetFmTransData;
STATE stSetFmTransData  PopTransData        0         stPushFmFi0Cnt;
STATE stPushFmFi0Cnt    StkPush             0         stResetFmFiCnt;
STATE stResetFmFiCnt    StkPopStore         1         stGotoResetPend;

#-----------------------------------------------------------------------
# Check for errors on the write
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stChkWrtError     PushTransData       0         stPushWrtFmMask;
STATE stPushWrtFmMask   StkPush             0x08      stWrtStatFmAnd;
STATE stWrtStatFmAnd    DataAnd             0         stPushCmpFm;
STATE stPushCmpFm       StkPush             0x08      stCmpFmError;
STATE stCmpFmError      CompareEq           0         stGotoWritePend
                                                      stGotoResetPend;
STATE stGotoWritePend   ExitStateMachine    0         stWritePend;
STATE stGotoResetPend   ExitStateMachine    0         stResetPend;

#-----------------------------------------------------------------------
# Wait for the reset
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stResetPend       CheckTransaction    1         stNotResetWrite
                                                      stChkResetAddr;
STATE stNotResetWrite   ExitStateMachine    0         stResetPend;
STATE stChkResetAddr    CheckAddress        0x3FD     stLogResetErr
                                                      stGotoWritePend;
# Check what is set?
STATE stLogResetErr     LogViolation        0         stChkWriteAddr;
```

```
##########################################################################
```

8 Appendix B - Sample State Machine Implementation

```
 /*-----------------------------------------------------------------------
 * FILE NAME: state_machine.cpp
 *
 * IMPORTANT:  READ BEFORE DOWNLOADING, COPYING, INSTALLING OR USING.
 * By downloading, copying, installing or using the software you agree
 * to this license.  If you do not agree to this license, do not
 * download, install, copy or use the software.
 *
 *                        Intel Open Source License
 *
 * Copyright (c) 2002 Intel Corporation
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * #  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * #  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * #  Neither the name of the Intel Corporation nor the names of its
 *    contributors may be used to endorse or promote products derived from
 *    this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE AND NONINFRINGEMENT ARE DISCLAIMED. IN NO EVENT SHALL INTEL OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *-----------------------------------------------------------------------*\

/*-------------------------------------------------------------------------
 *    DESCRIPTION:  This program is and example program of how one could
 *                implement a finite state machine.  This has been written
 *                as and example to be used in a class.  The doucmentation
 *                in the source is limited.  It is also assumed that the
 *                person being taught has a general understanding of
 *               what a state machine is.
 *
 *    Agruments:    [{TRACE [=] {ON|OFF}|?}]
 *
 *                   TRACE is used to turn trace of the statemachine on
 *                   or off.  The default is off.  This will stay set
 *                   for the complete run of the program.  Currenlty no
 *                   method is coded to allow trace to be controlled during
 *                   runtime.
 *
 *                   This command is not case sensitive.
 *
 *    AUTHOR:   Donald W. Long
 *
 *    HISTORY:  1.0     -   First Release
 *-----------------------------------------------------------------------*/

#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

    // General defines
#define ProgramVersion          "1.0"
#define ProgramVersionDate      "10-3-2001"
#define TRUE                    -1
#define FALSE                   0

// These are the code seqment names with the values that they can return
// All sets of code segment return values must start with 0 and go
// up by 1, no holes allowed.
typedef enum {
    CS_PrintBanner = 0,
    CS_AskForNum1,
    CS_AskForNum2,
    CS_AskForFunc,
    CS_Add,
    CS_Sub,
    CS_Times,
    CS_Divide,
    CS_OutPutResults,
    CS_Exit,
    CS_BadInput
} CodeSeqmentTF;

   // Misc Defines
#define CS_PrintBanner_OK       0

#define CS_AskForNum1_OK        0
#define CS_AskForNum1_BAD       1
#define CS_AskForNum1_Exit      2

#define CS_AskForNum2_OK        0
#define CS_AskForNum2_BAD       1
```

```
#define CS_AskForNum2_Exit      2

#define CS_AskForFunc_Add       0
#define CS_AskForFunc_Sub       1
#define CS_AskForFunc_Times     2
#define CS_AskForFunc_Divide    3
#define CS_AskForFunc_Exit      4
#define CS_AskForFunc_Unknown   5

#define CS_Add_OK               0

#define CS_Sub_OK               0

#define CS_Times_OK             0

#define CS_Divide_OK            0

#define CS_OutPutResults_OK     0

#define CS_Exit_OK              0

#define CS_BadInput_OK          0

// This table is used to output the text name of the code segement
// and its values if trace is turned on.
// Format:  <Num Values>  <code segment name>  <Names of values,
//                                              occurs for <Num Values>>
// The Value string names are in the order they are defined
//    (i.e., 0, 1, 2, ...)
//
// At the end of each code segment value list a -1 must occur.
// At the end of the table another -1 must occur.
//
static char CodeSeqmentNames[] =    {
    1, 'P', 'r', 'i', 'n', 't', 'B', 'a', 'n', 'n', 'e', 'r', 0,
        'P', 'r', 'i', 'n', 't', 'B', 'a', 'n', 'n', 'e', 'r',
        '_', 'O', 'K', 0,
    -1,
    3, 'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1',
             '_', 'O', 'K', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1',
             '_', 'B', 'A', 'D', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1',
             '_', 'E', 'x', 'i', 't', 0,
    -1,
    3, 'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2',
             '_', 'O', 'K', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2',
             '_', 'B', 'A', 'D', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2',
             '_', 'E', 'x', 'i', 't', 0,
    -1,
    6, ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’, 0,
        ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’,
             ’_’, ’A’, ’d’, ’d’, 0,
        ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’,
             ’_’, ’S’, ’u’, ’b’, 0,
        ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’,
             ’_’, ’T’, ’i’, ’m’, ’e’, ’s’, 0,
        ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’,
             ’_’, ’D’, ’i’, ’v’, ’i’, ’d’, ’e’, 0,
        ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’,
             ’_’, ’E’, ’x’, ’i’, ’t’, 0,
        ’A’, ’s’, ’k’, ’F’, ’o’, ’r’, ’F’, ’u’, ’n’, ’c’,
             ’_’, ’U’, ’n’, ’k’, ’n’, ’o’, ’w’, ’n’, 0,
    -1,
    1, ’A’, ’d’, ’d’, 0,
        ’A’, ’d’, ’d’, ’_’, ’O’, ’K’, 0,
     -1,
    1, ’S’, ’u’, ’b’, 0,
        ’S’, ’u’, ’b’, ’_’, ’O’, ’K’, 0,
    -1,
    1, ’T’, ’i’, ’m’, ’e’, ’s’, 0,
        ’T’, ’i’, ’m’, ’e’, ’s’, ’_’, ’O’, ’K’, 0,
    -1,
    1, ’D’, ’i’, ’v’, ’i’, ’d’, ’e’, 0,
        ’D’, ’i’, ’v’, ’i’, ’d’, ’e’, ’_’, ’O’, ’K’, 0,
    -1,
    1, ’O’, ’u’, ’t’, ’P’, ’u’, ’t’, ’R’, ’e’, ’s’, ’u’,
             ’l’, ’t’, ’s’, 0,
        ’O’, ’u’, ’t’, ’P’, ’u’, ’t’, ’R’, ’e’, ’s’, ’u’,
             ’l’, ’t’, ’s’, ’_’, ’O’, ’K’, 0,
    -1,
    1, ’E’, ’x’, ’i’, ’t’, 0,
        ’E’, ’x’, ’i’, ’t’, ’_’, ’O’, ’K’, 0,
    -1,
    1, ’B’, ’a’, ’d’, ’I’, ’n’, ’p’, ’u’, ’t’, 0,
        ’B’, ’a’, ’d’, ’I’, ’n’, ’p’, ’u’, ’t’, ’_’, ’O’, ’K’, 0,
    -1,
    -1
};

// This gives us the names of the states we will be using.
// It should be noted that this order must also be followed
// in the StateTable.
typedef enum {
    ST_Start        = 0,
    ST_GetNum1      = 1,
    ST_GetNum2      = 2,
    ST_GetFunc      = 3,
    ST_Add          = 4,
    ST_Sub          = 5,
    ST_Times        = 6,
    ST_Divide       = 7,
    ST_OutPutR      = 8,
    ST_Exit         = 9,
    ST_Num1Bad      = 10,
    ST_Num2Bad      = 11,
    ST_FuncBad      = 12
} States_TF;


int *StateTablePtr = 0; // This is the location in the state table
                        // we are at and the state table.
#define StateTableNumElements   8   // If you add the ability to have more
                                    // values you must increase this
                                    // by that number.

// Format:  <State> <Code Segment>  <state for val1> <state for val2>
//                                  <state for val3> <state for val4>
//                                  <state for val5> <state for val6>
//
//   If the value is not used then set to -1.
//
//   The order of the states must match the order of the states
//   defined in StateTF.
static int  StateTable[] =  {
    ST_Start,   CS_PrintBanner,  ST_GetNum1,     -1,         -1,         -1,
             -1,         -1,
    ST_GetNum1, CS_AskForNum1,   ST_GetNum2,     ST_Num1Bad, ST_Exit,    -1,
             -1,         -1,
    ST_GetNum2, CS_AskForNum2,   ST_GetFunc,     ST_Num2Bad, ST_Exit,    -1,
             -1,         -1,
    ST_GetFunc, CS_AskForFunc,   ST_Add,         ST_Sub,     ST_Times,   ST_Divide,
             ST_Exit,    ST_FuncBad,
    ST_Add,     CS_Add,          ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_Sub,     CS_Sub,          ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_Times,   CS_Times,        ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_Divide,  CS_Divide,       ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_OutPutR, CS_OutPutResults, ST_GetNum1,    -1,         -1,         -1,
             -1,         -1,
    ST_Exit,    CS_Exit,         ST_Exit,        -1,         -1,         -1,
             -1,         -1,
    ST_Num1Bad, CS_BadInput,     ST_GetNum1,     -1,         -1,         -1,
             -1,         -1,
    ST_Num2Bad, CS_BadInput,     ST_GetNum2,     -1,         -1,         -1,
             -1,         -1,
    ST_FuncBad, CS_BadInput,     ST_GetFunc,     -1,         -1,         -1,
             -1,         -1
};

    // State names, format is <name><null> .... Order must match the order
    // of the state values (see StateTF).
    // Last byte is -1 to show end of table.
static char StateNames[] =  {
    'S', 't', 'a', 'r', 't', 0,
    'G', 'e', 't', 'N', 'u', 'm', '1', 0,
    'G', 'e', 't', 'N', 'u', 'm', '2', 0,
    'G', 'e', 't', 'F', 'u', 'n', 'c', 0,
    'A', 'd', 'd', 0,
    'S', 'u', 'b', 0,
    'T', 'i', 'm', 'e', 's', 0,
    'D', 'i', 'v', 'i', 'd', 'e', 0,
    'O', 'u', 't', 'P', 'u', 't', 'R', 0,
    'E', 'x', 'i', 't', 0,
    'N', 'u', 'm', '1', 'B', 'a', 'd', 0,
    'N', 'u', 'm', '2', 'B', 'a', 'd', 0,
    'F', 'u', 'n', 'c', 'B', 'a', 'd', 0,
    -1
};

    // General flags for the program
int Trace = FALSE;      // if not 0 then we will trace the statemachine

    // Function prototypes
int Init(int argc, char *argv[]);
int PrintTrace( int  *StateTableEntry, char *CodeSeqmentNames,
                char *StateNames,      int CodeSegmentValue);
int ConvertToNum(char *InputData, double *Result);
char *StripSpaces(char *Line);
void ToUpper(char *Data);

/*
 ** main
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     argc    -   number of argments passed to the program
 *                  argv    -   address list of arguments.
 *
 *  DESCRIPTION:    main entry point for program
 *  RETURNS:        0   -   Program terminates OK
 *                  1   -   Program had an internal error
 *                  2   -   Invalid argument passed to program
 */
int main(int argc, char* argv[])
{
  int     CodeSegmentValue=-1;  // Code segment value returned from
                                // switch statement
  double  Num1=0;               // Contains the last value inputed
                                // for CS_AskForNum1
  double  Num2=0;               // Contains the last value inputed
                                // for CS_AskForNum2
  double  FuncResult=0;         // Results of the operation requested.
  char    InputData[100];       // String to store the input data into
                                // for all cin operations.
  char    *ptr;                 // General pointer.

      // Init the program
  if (!Init(argc, argv)) {
      cerr << endl <<  "**** Invalid arguments passed ****"  << endl <<
              "SampleStateMachine [{TRACE [=] {ON|OFF}|?}]"   << endl << endl;
      return(2);
  }
      // Main loop for program
  StateTablePtr=StateTable;
  while (TRUE) {
      switch (StateTablePtr[1]) {
          case CS_PrintBanner:
              cout << "Test StateMachine (" << ProgramVersion << "-" <<
                                  ProgramVersionDate << ") ** TRACE = ";
              if (Trace==TRUE) {
                  cout << "ON";
              } else {
                  cout << "OFF";
              }
              cout << endl << endl;
              CodeSegmentValue=CS_PrintBanner_OK;
              break;
          case CS_AskForNum1:
              cout << endl << "Please enter the first number or ’exit’? ";
              cin >> InputData;

              ptr = StripSpaces(InputData);
              ToUpper(ptr);

                  // Process the line
              if (!strcmp(ptr, "EXIT")) {
                  CodeSegmentValue=CS_AskForNum1_Exit;
              } else {
                  if (ConvertToNum(ptr, &Num1)) {
                      CodeSegmentValue=CS_AskForNum1_OK;
                  } else {
                      CodeSegmentValue=CS_AskForNum1_BAD;
                  }
              }
              break;
          case CS_AskForNum2:
              cout << endl << "Please enter the second number or ’exit’? ";
              cin >> InputData;

              ptr = StripSpaces(InputData);
              ToUpper(ptr);

                  // Process the line
              if (!strcmp(ptr, "EXIT")) {
                  CodeSegmentValue=CS_AskForNum2_Exit;
              } else {
                  if (ConvertToNum(ptr, &Num2)) {
                      CodeSegmentValue=CS_AskForNum2_OK;
                  } else {
                      CodeSegmentValue=CS_AskForNum2_BAD;
                  }
              }
              break;
          case CS_AskForFunc:
              cout << endl << "Please enter the function to perform"  << endl <<
                              "  Add"                             << endl <<
                              "  Sub[tract]"                      << endl <<
                              "  Times"                           << endl <<
                              "  Div[ide]"                        << endl <<
                              "  Exit"                            << endl <<
                              "? ";
              cin >> InputData;
              ptr = StripSpaces(InputData);
              ToUpper(ptr);
              if (!strcmp(ptr, "ADD")) {
                  CodeSegmentValue=CS_AskForFunc_Add;
              } else if (!strcmp(ptr, "SUB") || !strcmp(ptr, "SUBTRACT")) {
                  CodeSegmentValue=CS_AskForFunc_Sub;
              } else if (!strcmp(ptr, "TIMES")) {
                  CodeSegmentValue=CS_AskForFunc_Times;
              } else if (!strcmp(ptr, "DIV") || !strcmp(ptr, "DIVIDE")) {
                  CodeSegmentValue=CS_AskForFunc_Divide;
              } else if (!strcmp(ptr, "EXIT")) {
                  CodeSegmentValue=CS_AskForFunc_Exit;
              } else {
                  CodeSegmentValue=CS_AskForFunc_Unknown;
              }
              break;
          case CS_Add:
              FuncResult=Num1+Num2;
              CodeSegmentValue=CS_Add_OK;
              break;
          case CS_Sub:
              FuncResult=Num1-Num2;
              CodeSegmentValue=CS_Sub_OK;
              break;
          case CS_Times:
              FuncResult=Num1*Num2;
              CodeSegmentValue=CS_Times_OK;
              break;
          case CS_Divide:
              FuncResult=Num1/Num2;
              CodeSegmentValue=CS_Divide_OK;
              break;
          case CS_OutPutResults:
              cout << endl << "Your results are " << FuncResult << endl << endl;
              CodeSegmentValue=CS_OutPutResults_OK;
              break;
          case CS_Exit:
              CodeSegmentValue=CS_Exit_OK;
              return(0);
          case CS_BadInput:
              CodeSegmentValue=CS_BadInput_OK;
              cout << "Input is not valid - " << InputData << endl;
              break;
          default:
          cerr << endl <<
          "********************************************************" << endl <<
          "********************************************************" << endl <<
          "**                  INTERNAL ERROR                    **" << endl <<
          "**          The Code Segment Does not exist           **" << endl <<
          "** Either state table is bad, code segment not added,  **" << endl <<
          "**     or logic error in moving thru state table       **" << endl <<
          "********************************************************" << endl <<
          "********************************************************" << endl
          << endl;
            return(1);
      }
      if (Trace==TRUE) {
          if (!PrintTrace(StateTablePtr, CodeSeqmentNames,
              StateNames, CodeSegmentValue)) {
              return(1);
          }
      }
      StateTablePtr=&(StateTable[(StateTablePtr[CodeSegmentValue+2])
                      * StateTableNumElements]);
  }
}

/*
 ** Init
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     argc    -   number argument passed
 *                  argv    -   Address array to arguments
 *
 *  DESCRIPTION:    This will parse out the program arguments.  If
 *                  any errors will exit with a value of 2.
 *
 *  RETURNS:        0   -   Agruments invalid
 *                  1   -   Agruments processed
 *
 */
int Init(int argc, char *argv[])
{
    char *mode;
    char *ptr;


    if (argc>1) {
        if (argc<5 && argc >2) {
            if (argc==4) {
                mode=argv[3];
                ptr=StripSpaces(argv[2]);
                if (strcmp(ptr, "=")) {
                    return(0);
                }
            } else {
                mode=argv[2];
            }
            ptr=StripSpaces(argv[1]);
            ToUpper(ptr);
            if (!strcmp(ptr, "TRACE")) {
                mode=StripSpaces(mode);
                ToUpper(mode);
                if (!strcmp(mode, "ON")) {
                    Trace=TRUE;
                } else if (!strcmp(mode, "OFF")) {
                    Trace=FALSE;
                } else {
                    return(0);
                }
            } else {
                return(0);
            }
        } else {
            if (argc==2) {
                if (!strcmp(argv[1], "?")) {
                    cout << endl << "SampleStateMachine [{TRACE [=] {ON|OFF}|?}]"
                                                         << endl << endl;
                    exit(0);
                } else {
                    return(0);
                }
            } else {
                return(0);
            }
        }
    }
    return(-1);
}

 /*
 ** PrintTrace
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:
 *    StateTableEntry  - Pointer to the current state table location
 *                       that is being processed
 *    CodeSegmentNames - Pointer to the table that contains all the
 *                       code segment names and they code segment values
 *    StateNames       - Pointer to the table that contains the State Names.
 *    CodeSegmentValue - The value that was returned from the last code
 *                       segment that was executed for the current state.
 *
 *  DESCRIPTION:
 *    This will take the current state information (after execution) and
 *    output in text what has occured and the new state that will occur.
 *
 *    The output goes to cerr and is in the following format
 *       <CurState>(val) - <CodeSegment>(val) - <Codesegment value>(val) ->
 *                                                    <NewState>(val)<eol>
 *  RETURNS:
 */
int PrintTrace( int  *StateTableEntry, char *CodeSeqmentNames,
                char *StateNames,      int CodeSegmentValue)
{
    int     CurState;
    int     NewState;
    int     CodeSegment;
    char    *CurStateName;
    char    *NewStateName;
    char    *CodeSegmentName;
    char    *CodeSegmentValName;
    int     NumCodeSegments;
    int     i;
    int     ii;


        // Get the items from the state table entry.
    CurState    = StateTableEntry[0];
    CodeSegment = StateTableEntry[1];
    NewState    = StateTableEntry[CodeSegmentValue+2];

        // Get the current state name.
    for (CurStateName=StateNames, i = 0;
      i<CurState && CurStateName[0]!=-1; i++) {
        for (; *CurStateName!=0; CurStateName++);
        CurStateName++;
    }
    if (*CurStateName==-1) {
        cerr << endl <<
         "**********************************************" << endl <<
         "**********************************************" << endl <<
         "**               INTERNAL ERROR             **" << endl <<
         "**  Current State Not In State Name Table  **" << endl <<
         "** State table is  bad or State Name Table **" << endl <<
         "**********************************************" << endl <<
         "**********************************************" << endl << endl;
        return(0);
    }

        // Get the new state name.
    for (NewStateName=StateNames, i = 0;
      i<NewState && NewStateName[0]!=-1; i++) {
        for (; *NewStateName!=0; NewStateName++);
        NewStateName++;
    }
    if (*NewStateName==-1) {
        cerr << endl <<
         "**********************************************" << endl <<
         "**********************************************" << endl <<
         "**               INTERNAL ERROR             **" << endl <<
         "**      New State Not In State Name Table  **" << endl <<
         "** State table is  bad or State Name Table **" << endl <<
         "**********************************************" << endl <<
         "*********************************************" << endl << endl;
        return(0);
    }

        // Get the codesegment name.
    for (CodeSegmentName=CodeSeqmentNames, i=0;
      i<CodeSegment && CodeSegmentName[0]!=-1; i++) {
        NumCodeSegments=CodeSegmentName[0];
        for (; *CodeSegmentName!=0; CodeSegmentName++);
        CodeSegmentName++;

            // Skip the codesegment return value names.
        for (ii=0; ii<NumCodeSegments && CodeSegmentName[0]!=-1; ii++) {
            for (; *CodeSegmentName!=0; CodeSegmentName++);
            CodeSegmentName++;
        }
        if (ii!=NumCodeSegments && CodeSegmentName[0]!=-1) {
            cerr << endl <<
          "***********************************************" << endl <<
          "***********************************************" << endl <<
          "**            INTERNAL ERROR                 **" << endl <<
          "** Code Segment Vak Not in Code Segment Table **" << endl <<
          "** Code Segment is bad  or Code Segment Table **" << endl <<
          "***********************************************" << endl <<
          "***********************************************" << endl << endl;
            return(0);
        }
        CodeSegmentName++;  // Skip the -1 at the end of the
                            // codesegment values.
    }
    if (*CodeSegmentName==-1) {
        cerr << endl <<
         "***********************************************" << endl <<
          "**********************************************" << endl <<
          "**              INTERNAL ERROR             **" << endl <<
          "**   Code Segment Not in Code Segment Table  **" << endl <<
          "** Code Segment is bad or Code Segment Table **" << endl <<
          "**********************************************" << endl <<
          "**********************************************" << endl << endl;
        return(0);
    }

        // Setup for getting the codesegment value name.
    NumCodeSegments=CodeSegmentName[0];
    CodeSegmentName++;
    for (CodeSegmentValName=CodeSegmentName; *CodeSegmentValName!=0;
      CodeSegmentValName++);
    CodeSegmentValName++;

        // Get the code segment value name
    for (i=0; i<CodeSegmentValue && CodeSegmentValName[0]!=-1; i++) {
        for (; *CodeSegmentValName!=0; CodeSegmentValName++);
        CodeSegmentValName++;
    }
    if (CodeSegmentName[0]==-1) {
        cerr << endl <<
          "*********************************************" << endl <<
          "*********************************************" << endl <<
          "**            INTERNAL ERROR                **" << endl <<
          "** Code Segment Vak Not in Code Segment Table **" << endl <<
          "** Code Segment is bad  or Code Segment Table **" << endl <<
          "*********************************************" << endl <<
          "*********************************************" << endl << endl;
        return(0);
    }

        // Output format is <CurState>(val) - <CodeSegment>(val) -
                            <Codesegment value>(val) -> <NewState>(val)
    cerr << CurStateName    << "(" << CurState          << ") - "   <<
         CodeSegmentName    << "(" << CodeSegment       << ") - "   <<
         CodeSegmentValName << "(" << CodeSegmentValue  << ") -> "  <<
         NewStateName       << "(" << NewState          << ")"      << endl;
    return(-1);
}

/*
 ** ConvertToNum
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:  InputData   -   String that is to be converted to a number
 *               Result      -   Address to a double to put the results in
 *
 *  DESCRIPTION: This function will take an ascii string and convert it to
 *               a double.  This function assumes all spaces have been
 *               removed from the start of the string and the end.
 *
 *  RETURNS:     -1  -   Converted ok
 *                0  -   Invalid data in InputData (not a number)
 *
 */
int ConvertToNum(char *InputData, double *Result)
{
    char *ptr;


        // Make sure all charactors are valid.
    for (ptr=InputData; *ptr!=0; ptr++) {
        if (!isdigit(*ptr)) {
            if (*ptr=='.') {    // Floating Point Number.
                for (ptr++; *ptr!=0; ptr++) {
                    if (!isdigit(*ptr)) {
                        return(0);
                    }
                }
                break;  // Leave the for loop so we can do the atof function.
            } else {
                return(0);
            }
        }
    }
    *Result=atof(InputData);
    return(-1);
}

/*
 ** StripSpaces
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     Line    -   The line to remove spaces
 *
 *  DESCRIPTION:    This will remove all the spaces at the start and end
 *                  of Line.
 *
 *  RETURNS:        Address of first charactor in Line that is not a space
 *
 */
char *StripSpaces(char *Line)
{
    char *stptr;
    char *ptr;


        // Strip off all spaces
    for (stptr=Line; *stptr == ' ' && *stptr != 0; stptr++);
    for (ptr=stptr; *ptr != ' ' && *ptr != 0; ptr++);
    *ptr=0;
    return(stptr);
}

/*
 ** ToUpper
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     Data    -   Data to convert to upper case
 *
 *  DESCRIPTION:    This function will convert a staring to upper case
 *
 *  RETURNS:
 *
 */
void ToUpper(char *Data)
{
    char *ptr;

    for (ptr=Data; *ptr!=0; ptr++) {
        *ptr=toupper(*ptr);
    }
}
```

Advanced Boot Scripts

Richard Gooch
EMC Corporation
rgooch@atnf.csiro.au
http://www.atnf.csiro.au/∼rgooch/linux/boot-scripts/

Abstract

This paper describes the design and implementation of a dependency-based scheme for system boot scripts. This scheme preserves the modularity of SysV-style boot scripts but does not suffer from it's limitations (such as a complicated directory tree populated with symlinks, and the need for global dependency knowledge).

The dependency-based scheme simplifies the creation and integration of boot scripts by requiring only knowledge of direct dependencies (i.e. local rather than global knowledge). Dependency management is performed by **simpleinit(8)**, which may execute boot scripts in parallel, when those scripts have no cross dependencies.

This paper seeks to expose this new scheme to a wide audience, including disribution maintainers, with the hope that more widespread adoption follows.

1 Introduction

I propose a new mechanism for booting user-space on Unix-like systems. This scheme is a significant departure from existing boot mechanisms, and is a response to their respective limitations. The two main existing schemes are the so-called “BSD” and “SysV” styles. Each have their disadvantages, discussed below.

1.1 BSD-style

1.1.1 Mechanism

In this scheme, booting is controlled by one of a very few number of boot scripts. Often, there is a master boot script (typically `/etc/rc`) which orchestrates the whole boot procedure. This scheme is fairly easy to understand, as it has only a small number of scripts to read and the order in which things are started up is quite clear from the master boot script. It is fast, simple, and efficient.

1.1.2 Limitations

Where this scheme fails is in its scalability. If a 3rd-party package needs to have an initialisation script run during the boot procedure, it needs to *edit* one of the existing boot scripts. Such editing is dangerous, as boot scripts are fragile at the best of times. A simple mistake by the installer can lead to an unbootable system.

1.2 SysV-style

1.2.1 Mechanism

This scheme places a number of mini-scripts in a master directory (typically `/etc/rc.d/init.d/`) which collectively can boot most of the system. Each of these mini-scripts starts and stops one "service." This is quite neat and modular. A master boot script is used to orchestrate the boot process, which does some "special" setup (i.e. anything which was considered too difficult to put into a mini-script), and then proceeds to run each of the mini-scripts *in another directory*. The order is based on shell wild-card expansion rules.

The "other directory" is populated with symbolic links back into `/etc/rc.d/init.d/` (where the scripts are kept). Each script usually has two links to it. One starts with "S" and the other with "K". The "S*" scripts are called when booting up the system, the "K*" scripts are called when shutting down the system. The desired ordering is achieved by using numbers after the "S" and "K" in the link names.

So a link with name "S10" will run before "S15", which in turn runs before "S20". It is the responsibility of the system integrator to name these links such that services are started and stopped in the correct order. A 3rd-party software installer can "simply" place their startup script in `/etc/rc.d/init.d/` and then create a symbolic link to the script in the "other directory." The installer has to pick a name that is not already taken, and has to determine the number to use (which depends on how far into the boot procedure the script must be run).

The author of the system boot scripts must therefore allocate numbers with sufficient gaps between them to allow for later insertions. Typically, the numbers 10, 20, 30, 40, 50, 60, 70 and so on are chosen. This reminds me of when I was a youngster programming BASIC on my Apple][. Every line had to be given a number, and you had to be careful to leave "space" for later insertions. The SysV numbering isn't quite so restrictive, as it is possible to append an arbitrary string to the number, which effectively increases the number space. Typically, the name of the script is appended, such as `10inetd` and `10named`. Thus, it is possible to "group" scripts so that the order between groups is well-defined, while ordering within a group is unknown (or knowable but not important).

The SysV booting scheme also supports the concept of "runlevels." What this means is that the system may be booted "all the way" (by convention, this is runlevel 5) by default, but may also be booted only part of the way. The most common purpose is to allow the system to be booted "single-user" (i.e. maintenance/repair mode), where only a handful of services are started. The runlevel scheme is supported by splitting the symlinks in the "other directory" into a number of directories, each directory corresponding to a runlevel. These directories are typically named:

```
/etc/rc.d/rc0.d/
/etc/rc.d/rc1.d/
/etc/rc.d/rc2.d/
/etc/rc.d/rc3.d/
/etc/rc.d/rc4.d/
/etc/rc.d/rc5.d/
/etc/rc.d/rc6.d/.
```

The master boot script will start all scripts in the runlevel directory corresponding to the desired runlevel. Thus, the system can be booted to runlevel 1 by running the scripts in `/etc/rc.d/rc1.d/` (this is often "single-user" mode). Then, perhaps after some maintenance work the system can be booted all the way by switching to runlevel 5 by stopping services for runlevel 1 and starting the scripts in `/etc/rc.d/rc5.d/`. Similarly, the system can be taken from a higher runlevel to a lower one by stopping services.

These boot scripts, in the tradition of SysV, can do anything. They are flexible and scalable and are designed to run large systems.

1.2.2 Limitations

A significant disadvantage of this scheme is its complexity. A simple measure of its complexity is the quantity of text describing it compared to that required for describing the BSD-style scripts. Due to this complexity, it is often difficult to see how the various scripts fit together and determine the execution sequence.

This intricate web of scripts, directories and symbolic links is difficult to construct and difficult to administer. Even an experienced system administrator can be confused by this scheme when first exposed to it. Novice administrators may be expected to be quite perplexed. With the growing popularity of Linux, the vast majority of Linux users are not experienced system administrators, but must still administer their systems. The SysV scheme does not cater to their needs.

While the SysV scheme is more scalable than the BSD scheme, there remains a problem for 3rd-party boot scripts: which symlink name should be chosen? Usually the script is started in runlevel 6, because by that time "most" services are available. The simplest solution is to pick a random high number, which "should" work.

Finally, the use of numerical runlevels is far from intuitive. While old-guard SysV administrators may feel the runlevel definitions are simple to learn, the reality is the numbers convey no meaning. Certainly novice system administrators (the bulk of the Linux population now) will just scratch their collective heads and say, "Ah well, I guess that's just Unix."

2 An Alternative

As indicated, each existing scheme has advantages and disadvantages. The use of mini-scripts provides scalability, and thus this aspect of the SysV scheme should be preserved. What is required is a mechanism that starts mini-scripts in an ordered fashion yet is easy to understand and does not suffer from name-space or number-space limitations.

The proposed solution is simple yet powerful. There is *no* master script which orchestrates everything. Instead, all scripts are executed in parallel. Ordered sequencing is obtained by allowing each script to declare which services it needs available (i.e. what it depends on) in order to successfully complete. Even the master script found in SysV-style booting schemes is eliminated.

Whenever a script declares that it needs another, it is suspended (blocked) until the required service is available or is determined to be unavailable. This simple mechanism enforces strict sequencing with precisely the level of granularity desired.

Placing dependency information inside each script has the following advantages:

- it is immediately clear what other services a script depends on
- the information is localised, requiring no global orchestration by the system integrator
- 3rd-party scripts can fine-tune their dependencies
- 3rd-party script installers need not be aware of the global sequencing details.

A beneficial side-effect of executing scripts in parallel is that some services will be started in parallel, once the common services they depend on are available. This can reduce the time taken to boot the system.[1]

3 Implementation

The **init(8)** programme is responsible for executing the boot scripts and orchestrating the correct sequencing. To accomplish this, I modified the **simpleinit(8)** programme from the **util-linux** package to support dependency-based boot sequencing. I wrote the **initctl(8)** programme which is used by scripts to declare their dependencies, and a set of boot scripts using this new mechanism. These boot scripts may be used as a guide for writing another set, or may be used directly in a production system.

The mini scripts are kept in a directory and **init(8)** runs *all* of them, in random order. Ordering of the mini scripts is controlled by the scripts themselves. Each script runs any other scripts it depends on, using the **need(8)** programme (an alias of **initctl(8)**) which ensures that a script is only run once. It doesn't matter which order **init(8)** starts running the scripts, as **need(8)** ensures scripts wait as required.

3rd-party scripts need only use **need(8)** to ensure services they require are running. This eliminates the problem of deciding where to place the script in the sequence.

The changes to **simpleinit(8)** and the new **initctl(8)** have been incorporated into the **util-linux** package.

3.1 Implementation details

By default, **simpleinit(8)** will run `/etc/rc` as its startup script. The modified version allows the system administrator (either at the boot prompt or in `/etc/inittab`) to specify an alternative script to run. If the script specified is in fact a directory, all the scripts in that directory are run, in random order.

[1]consideration must be given to the effect this may have on disc head seek times, which could eliminate gains due to parallelism

In the new scheme, **init(8)** is configured to run all mini startup scripts in `/sbin/init.d/`. Each script starts/stops one service (i.e. printing, file-system checks, NFS mounting and so on). Take the example of the NFS export script, which starts the daemons **rpc.mountd(8)** and **rpc.nfsd(8)**, but must wait until the RPC portmapper is running before starting the NFS daemons. The sample script below demonstrates this:

```
#! /bin/sh
# /sbin/init.d/nfs-export

case "$1" in
  start)
    need portmap || exit 1
    rpc.mountd
    rpd.nfsd
    ;;
  stop)
    killall rpc.nfsd && \
      killall rpc.mountd
    ;;
esac
# End
```

The **need(8)** programme is used to run a script, and wait for its completion. If the programme has not been run before, **need(8)** will run it. If it has already run, **need(8)** does nothing. The exit code indicates whether the service (the portmapper in this case) started successfully or not. Since the portmapper is required, the script tests the exit code from **need(8)** and fails if it is unavailable for any reason.

3.1.1 Single-user and runlevels

For single-user mode, **init(8)** can be configured to run a specific script (or directory). This script can provide a simple or arbitrarily complex single-user mode, at the discretion of the designer of a set of boot scripts.

Different runlevels are supported in a similar fashion. Whatever argument is passed to **init(8)** at the command line (boot prompt), it is appended to a configurable prefix and together they specify the script (or directory) to run. Thus, you can pass in "single", "3", "6" or "multi" and all that is required is the appropiately named script or directory.

There are two ways in which traditional runlevels can be supported. One is that an appropriate directory is created with symlinks back into `/sbin/init.d/`. This approach may be used when a rapid implementation of runlevels is desired. A more elegant solution is to have a script for each runlevel. An example script is shown below:

```
#! /bin/sh
# /sbin/init.d/runlevel.3

case "$1" in
  start)
    need runlevel.2 || exit 1
    need portmap || exit 1
    mount -vat nfs
    ;;
  stop)
    umount -vat nfs
    ;;
esac
# End
```

In this example, the distinction between runlevels 2 and 3 is that runlevel 3 will additionally mount remote file-systems. Thus, runlevel 2 is required as is the portmapper.

3.1.2 The initctl(8) implementation

Originally, I had intended to put most of the intelligence into **initctl(8)** and have **init(8)** only maintain the database of scripts. This approach was quickly discarded, since it would require reliable, full-duplex inter-process communication (IPC) services. Under Linux, these may be available as loadable modules, and thus may not be available at the time **init(8)** starts. The only IPC facilities that may be relied on are named pipes (FIFOs) and Unix signals. These are not suited to parallel, full-duplex communications.

The approach I adopted was to place the responsibility for script starting and stopping, as well as database management, with the **init(8)** programme, and have a simple control interface. By limiting the amount of information that is sent from **init(8)** to **initctl(8)** to a simple available/not-available/failed status, the need for a second FIFO (for each instance of **initctl(8)**) is avoided, and Unix signals may be used instead.

The **initctl(8)** control interface is a trivial programme which simply writes service requests to the control FIFO and waits for a success/failure signal.

Because the dependency table for **init(8)**-started processes is kept in **init(8)**, it makes partial and complete rollbacks (switching between runlevels and orderly shutdowns) easier to implement. Since **init(8)** never dies, and doesn't crash (if it does, the system will hang), it is quite safe to maintain the database inside the virtual memory space of **init(8)**. Also, since the database is quite small, there is no significant resource consumption.

3.2 Optimisations

A simple optimisation which can reduce booting time is the pre-fetching of all the script files, which can reduce the number of disc head seeks. Without this optimisation, the disc head may have to travel back and forth between the script files, the daemons they start and their configuration files. Assuming the script files are close to each other on the disc media, a small number of seeks will suffice for pre-fetching the script files. This optimisation has been implemented, by reading the scripts in file-system order into a dummy buffer.

3.3 Runlevels and rollback

Because **init(8)** maintains a table of which boot scripts have been run and which have failed (if any), and since **init(8)** runs for the lifetime of the booted system, it is ideally suited to managing orderly shutdown of the system. Further, since at any time **need(8)** may be used to run another boot script, with full dependency checking, then **init(8)** may also be used to switch between runlevels.

An orderly shutdown is as simple as rolling back the entire table. The algorithm is trivial: obtain the last entry in the table and run the appropriate stop script (which is then removed from the table). The process is repeated until the table is empty. All services will then have been stopped in the reverse order in which they were started.

Increasing runlevel is also quite simple: just run the desired runlevel script. Thus going from runlevel 2 to 3 involves running `runlevel.3` under the dependency management scheme.

Going from runlevel 3 to 2 is slightly more complicated, but not much. Again, the system needs to be rolled back, stopping each script/service in reverse order. As each is stopped, its entry is removed from the dependency table. The process is stopped at `runlevel.2` (without stopping `runlevel.2` itself).

This scheme works because `runlevel.3` is added to the dependency table *after* it registers new dependencies (because it's added to the list once it completes). So once the system has rolled back to `runlevel.2`, we can be sure that all the services `runlevel.3` has started have been stopped, plus all the services it depended on, *but not the services runlevel 2 depended on, or runlevel 2 itself.*

For this switching between runlevels to work, the burden is placed on the runlevel scripts, not **init(8)**, which is an important point, because it provides maximum flexibility in the construction of boot scripts and keeps **init(8)** simple. The same rollback mechanism required for orderly shutdown may be used to switch runlevels. No extra logic is required.

3.4 Multiple providers and provide(8)

Sometimes, there may be multiple service providers for the same generic service. For example, you might have **sendmail(8)** and **qmail(8)** installed on your system, and each has a boot script associated with it. Each one provides the `mta` (Mail Transport Agent) service.

In this case, only one of these scripts should be started. It might not matter which one is started, or perhaps each script may check some system-specific configuration to determine whether or not it should start the service. In either case, all scripts providing the generic service should be run, but only one should start the service.

The solution to this is the **provide(8)** programme. It tells **init(8)** that the calling programme/script is able to provide the generic service. **init(8)** then makes sure that only one provider will actually provide this service. An example script follows:

```
#! /bin/sh
# /sbin/init.d/sendmail

case "$1" in
 start)
  if [ ! -f \
     /etc/mail/sendmail.cf ];
       then exit 2
  fi
  provide mta || exit 2
  need portmap
  /usr/sbin/sendmail -bd -q15m
  ;;
 stop)
  killall sendmail
  ;;
esac
# End
```

Here, the script first checks to see if its configuration file `/etc/mail/sendmail.cf` is available. If not, the script exits with a "not available" status code. Then, the script registers its intention to provide the `mta` service. If given permission, it proceeds to start the service, otherwise it exits.

4 Future Work

I've considered keeping a full dependency history inside **simpleinit(8)** (right now it only keeps track of the currently depended-on service for each script). This would allow any service to be stopped and all services which depend on it to be stopped (dependent services would be stopped first, of course). This would be more flexible than either runlevels or rollback. In addition, a stopped service could still be recorded in the database and thus restarted with all the services that depended on it also being restarted. It is not clear whether these features would yield sufficient benefit to justify the implementation effort.

5 Acknowledgements

This work is the result of an evening discussion session between Patrick Jordan[2] and myself. The basic concept of a dependency-based booting scheme, and the semantics of the **need(8)** programme, were established during that session. I thank Patrick for his enthusiasm for this project and his willingness to try the new, experimental boot scripts.

The implementation of the multiple providers feature (discussed in section 3.4) was added as a result of discussions with Wichert Akkerman (then Debian Project leader, email: **wichert@cistron.nl**), where the needs ot Debian were raised.

A similar (although less complete) dependency-based booting scheme has been independently developed by David Parsons for his Mastodon Linux[3]. Thanks to Larry McVoy for pointing this out.

Except where otherwise noted, all work is my own.

[2]http://www.ariel.com.au/∼patrick/

[3]http://www.pell.portland.or.us/∼orc/Mastodon/

Porting Drivers to HP ZX1

Grant Grundler
Linux Development Lab
Hewlett Packard
Cupertino, CA, USA, 95014
grundler@cup.hp.com

Abstract

"Porting" doesn't accurately describe how one gets a Linux driver to run on different architectures. If a driver doesn't "just work," generally it's a matter of figuring out which wrong assumptions about the HW (or OS) are embedded in the driver. The goal of this talk is to describe the *HP ZX1* IO subsystem and some of the wrong assumptions I've found in 2.4.17 kernel drivers.

A Block Diagram of the ZX1 IO subsystem is quite similar to current PA-RISC systems. In contrast to Intel Itanium boxes, neither supports legacy x86 IO space. For booting, *EFI drivers* (ugh, DOS is back) are required and IA32 Expansion ROMs are ignored. *Platform Services* must be used for DMA mapping, interrupts, PCI device discovery. I'll discuss how those services are different between HP's ZX1 platform and my (weak) understanding of IA32. Fortunately, use of these services is the same between both architectures.

I was surprised by which drivers did not *Just Work* (e.g. tulip, acenic) and will talk about why they didn't. Primarily, the timing of CPU interaction with IO devices is different. ZX1 IO subsystem is also less tolerant of driver "quirks"—things that are wrong but other platforms don't puke on. Lastly, I'll explain what an MCA is and how it's useful for debugging IO driver problems.

1 HP ZX1 IO Subsystem

The HP ZX1 chip set doesn't have many surprises to folks who've worked on RISC systems. Other architectures including PA-RISC, Alpha, and SPARC have similar block diagrams. The main parts of HP's implementation are the *System Bus Adapter* (SBA) and *Lower Bus Adapter* (LBA).

From a very high level, most IO subsystems aren't that different since PCI bus behaviors are defined by the various PCI specifications. IO Interrupts (IRQ Line), IO Port, and MMIO functionality provided have the same semantics as on IA32. This is good since it makes it possible to write (mostly) portable drivers.

The SBA provides an IO MMU, memory controller, and interconnect between the *ropes* bus and *McKinley* bus. The IO MMU design is based on the implementation used in PA-RISC workstations and low end servers. Two significant differences is how 64-bit DMA addressing is supported and cache coherency model. Other less obvious differences are greater bandwidth of both the McKinley bus and ropes bus.

The LBA is the PCI Host bus adapter and also contains the IO SAPIC. Unlike its

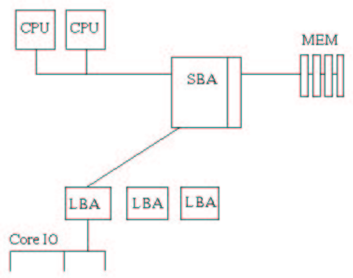


Figure 1: HP ZX1 architecture

PA-RISC predecessor, this LBA supports PCI-X (133MHz, 64-bit). The IO SAPIC was also used in PA-RISC platforms. I'm still amazed that 80% of the code is the same between the IA64 and PA-RISC implementations. Because of NDAs with Intel, both implementations were developed completely independently inside HP and published on the same day (Feb 3, 2000) when the IA64 source tree was published. (See `http://lists.parisc-linux.org/hypermail/parisc-linux-cvs/2860.html`).

This type of architecture has some clear performance advantages over legacy North/South bridge topology in IA32 systems and also introduces some new issues. The performance advantage is more raw IO bandwidth between IO, memory, and CPU which exceeds the single PCI bus model by orders of magnitude. Some obvious problems are ordering of transactions (e.g. IRQ vs. DMA), DMA latency, and PIO latency.

2 DMA Mapping

Use of PCI DMA mapping services is required for several reasons:

- **A**ddress Translation: The primary purpose is to provide a device view of memory for DMA.
- **3**2-bit DMA: IO MMU provides 32-bit devices that ability to DMA into memory which lives above 4GB address boundary. This provides at least 3x better performance than SW for block IO.
- **P**ortability: The old interface, `virt_to_bus()`, could only support systems w/o IOMMU or the IOMMU could map all of host memory statically. HP ZX1 IOMMU can only map 1GB at a time. That's not as bad as it sounds since only 32-bit PCI devices are required to use the IO MMU. 64-bit PCI devices (capable of DAC) can bypass the IO MMU.

3 Interrupts

`request_irq()` works the same as before. What's really different from IA32 is the number and type of IRQs available. IA64 defines 256 vectors vs the woefully inadequate 15 in legacy IA32. The following sections describe some of the high level behaviors of IO SAPIC and implementations.

3.1 Message Signalled Interrupts

As far as I can tell, no one is using this. At least not directly. The IO SAPIC translates the line based IRQ into a transaction on the "upstream" bus. The Local SAPIC in the CPU is the target of this transaction and is identified by its *EID*. The data portion of the transaction identifies which interrupt vector is being delivered.

System Firmware assigns EIDs and initializes the Local SAPICs. The IO SAPIC driver reads the Interrupt Routing Table from ACPI. This table describes how each of the 4 IRQ *Pins* from each PCI device or slot is routed to a particular IO SAPIC IRQ Line. When a device driver registers its interrupt handler via `request_irq()`, the IO SAPIC driver programs the IRTE (an internal IO SAPIC register) for the IRQ input line.

Note that Foster CPU is the first IA32 CPU to use IO xAPIC (˜= SAPIC) and Local xAPIC. Support for IO xAPIC was only recently added to i386 arch in order to properly distribute IRQs across CPUs. All architectures with IO xAPICs (PA-RISC, IA64, IA32) direct interrupts at specific CPUs. None use XTPR transactions to enable the HW to redirect interrupts to a "lower priority" CPU. For IA64 and IA32, this is by design to preserve driver state in the CPU cache associated with a given PCI interface card. PA-RISC has no Local SAPIC or XTPR support and consumes the interrupt transaction directly.

So why talk about MSI? There's several good reasons for devices to use MSI:

- **I**nterrupt Code Path: It allows the driver interrupt to be directly called from the trap handler—no traversing lists or lookup tables. Typically though, a layer of indirection is only needed if the HW can't generate an EOI to the IO SAPIC or the IRQ Line is shared.
- **E**xclusive Vector: The driver can avoid shared PCI IRQ line and the the resulting shared vector. IO SAPIC implementations to date typically only have 7 IRQ lines—not really enough if the PCI bus hosts multiple devices/slots.
- **D**MA ordering: Normally, when the IRQ is a line, it bypasses the normal DMA data path. Thus race conditions exist where a DMA might not reach memory before the IRQ is delivered and acted upon. For PCs and the like this isn't a problem since all the IO paths are short.

  For larger systems, this can be a problem. When the interrupt is a transaction on the bus, PCI ordering rules prevent it from bypassing any inbound DMA transaction. Thus, when the interrupt finally reaches the CPU, one can be certain all DMA has reached memory as well and not stuck in any coalescing buffers between the IO device and the memory it was writing to. Thus one doesn't need any additional magic to guarantee the in-flight DMA is coherent with CPU caches.
- **T**arget multiple CPUs: This is wish list. Given the right services, a smart device can target transaction completions at different CPUs by generating interrupt transactions for specific Local SAPICs. The goal is to service the interrupt on the same CPU that initiated the transaction. Tradeoffs between driver D-cache footprint and interrupt latency would help determine applications for this. Clustering folks I've talked were looking at this but didn't prototype anything to test it out.

3.2 More than 256 Interrupts?

Large systems (32 CPU and up) can end up using more than 256 vectors and exhaust the Interrupt Vector Table. One example is cc-Numa machines where one really doesn't want to (or can't) deliver interrupts across the fabric. Linux can gracefully work around this issue by defining *IRQ Regions*. For IA64, each region could represent a different Interrupt Vector Table. PA-RISC uses IRQ regions for every level of the interrupt handling that has to decode a bit mask or handle an array of IRQs.

The ACPI Interrupt Routing Tables may not need to collude if arch specific code can correctly direct interrupt transactions generated by IO SAPIC to the targeted CPU. I'm not familiar with details of ccNUMA support but know it has been done.

4 Posted vs Non-Posted Writes

Nearly all linux drivers started out using *IO Port* address space since ISA/EISA was the standard when linux was born. On IA32, special instructions (yes, most of you know this already) exist to access this alternative address space.

What's key here is IO Port space also has different semantics. One similarity is reads and writes to either IO Port or MMIO space do not interact with CPU Cache. A subtle difference is writes to IO Port space are *Non-Post-able*. This means **t**he CPU stalls waiting for the transaction to complete.

4.1 IO Port space sucks

IO Port space has several serious issues:

- **I**SA Aliasing: Most of IO port address space isn't available because of ISA compatibility where to many ISA devices only support 10 (or more) address lines and alias everything above that.
- **L**egacy IO: serial, timers, fd, parallel and a host of other devices occupy de-facto standard addresses in IO port space.
- **M**ore Registers: Many new devices require more register space. Either more mail boxes or on-board RAM. Just isn't room for a 4k or bigger shared RAM in IO port space. Maybe for single devices, but I've been told that's not useful when 4 or more cards need to be installed in the system.
- **D**evice Discovery: For devices which don't have legacy addresses assigned, they had to poke around in IO port space to discovery where their devices where. Fortunately with PCI, that's no longer necessary though some drivers still do that for ISA compatibility.

Combined, all of these issues have encouraged nearly all PCI devices to move to MMIO space regardless of the Non-Post-able semantics.

4.2 Memory Mapped IO is better

Since PCI has become a standard, many PCI devices support both IO Port space and MMIO address space to provide compatibility and a transition for drivers to use MMIO space. And that transition has been taking place. In implementing this transition, many driver writers assume MMIO is the same as IO port space and there's just more of it. **T**hat's wrong.

MMIO space is *Post-able*. The CPU writes the data and just continues doing other work. The CPU may not even wait for the transaction to hit the *Central Bus* (aka Front Side Bus) before continuing. This is good. It means a burst of writes are exactly that.

4.3 MMIO is harder to get Right

Even **g**ood driver writers get MMIO space usage wrong.

This is from acenic, a "mature" driver. But here is an example of this wrong assumption:

```
  writel(local, &regs->LocalCtrl);
  udelay(ACE_LONG_DELAY);
  mb();
  local |= EEPROM_CLK_OUT;
  writel(local, &regs->LocalCtrl);
```

The problem is the CPU starts executing the udelay() before the data reaches the device. The `writel()`s are timing sensitive. And the `mb()` is orthogonal to the udelay(). Switching the order around shouldn't change things. The fix is **a**dd a `readl()` after the first `writel()`. PCI transaction ordering rules require the write get pushed to the device before the read. Since the CPU has to wait for the read return, the write is effectively flushed. We don't care what the read returns in this case.

In all fairness, Jes Sorensen caught what was going on right away and accepted my patch. I added 35 readl() calls. He did gripe about my formatting. That's OK. That's Jes and it's his driver.

Dave Miller was a slightly harder sell for a patch to tg3. Jeff Garzik caught on right away and provided Dave with the explanation that I somehow didn't.

```
http://linux.bkbits.net:8080
/linux-2.4/cset@1.383.17.6
?nav=index.html|ChangeSet@-4w|
```

tg3 driver got 3 more reads for similar issues. One was a slightly different case and worth noting:

```
  tw32(RX_CPU_BASE + CPU_STATE,
       0xffffffff);
  tw32(RX_CPU_BASE + CPU_MODE,
       0x00000000);
+
+ /* Flush posted writes. */
+ tr32(RX_CPU_BASE + CPU_MODE);

  return 0;
```

Code after the return was expecting the `CPU_MODE` to have been cleared already. I got lazy and stopped trying to figure out what.

4.4 CPU v.s. IO Timing Trend

The speed of the CPU is getting much faster than the IO path is. HP likes high bandwidth bridges that favor bandwidth over MMIO access. Thus while a problem may not be visible on a 2GHz Pentium, it will show up on on 800 or 1GHz HP ZX1 system. And probably on other systems from SGI, SUN or IBM.

In the case of current HP ZX1 platforms, the *System Bus Adapter* (aka SBA) and *Lower Bus Adapter* (aka LBA, PCI-X Controller) both have FIFOs to queue data in both directions. The fact the a MMIO transaction has to cross 3 busses to get to a device (Central, internal, PCI) is a good hint that timing is going to be longer than on systems with only one or two busses.

One example of different timing was exposed in the tulip driver where it resets the *Phy* (DP83840A or LXT971D). No issue exists with this code using the same 100BT cards on 400 MHz PA-RISC. The issue showed up occasionally on 550MHz PA-RISC and consistently on faster HP ZX1 platforms. HP 100BT products needed the patch that appears in Figure 4.4.

Though this works, I want to be clear the patch is wrong. I discovered this worked and submitted the patch before I found and read the respective product data sheets. The right fix is to poll the phy after the `reset_sequence` until an "in-reset" bit clears. Then one should wait about 500 microseconds before sending the `init_sequence`.

Figure 2: Incorrect patch for HP 100BT products

```
diff -u -p -r1.2 media.c
--- drivers/net/tulip/media.c   25 Jan 2002 20:14:57 -0000      1.2
+++ drivers/net/tulip/media.c   25 Mar 2002 19:57:19 -0000
@@ -284,6 +284,10 @@ void tulip_select_media(struct net_devic
                for (i = 0; i < init_length; i++)
                        outl(init_sequence[i], ioaddr + CSR12);
        }
+
+       (void) inl(ioaddr + CSR6); /* flush CSR12 writes */
+       udelay(500);            /* Give MII time to recover */
+
        tmp_info = get_u16(&misc_info[1]);
        if (tmp_info)
                tp->advertising[phy_num] = tmp_info | 1;
```

4.5 MMIO Reads are expensive

The last time I measured the cost of an MMIO read on a 400MHz PA-8500 system, I got something around 500-600 CPU cycles. The same measurement on an 800 MHz HP ZX1 system was around 900-1000 CPU cycles. PCI bus traces from a 450MHz PII system suggested the MMIO read time was in the same ball park.
Conclusion: **MMIO reads are expensive.**

For an example of MMIO read avoidance, see

```
http://cvs.parisc-linux.org
/linux/arch/parisc/kernel
/sba_iommu.c?rev=1.66
```

and search for `DELAYED_RESOURCE_CNT`. This code only works because MMIO writes are *Post-able*.

4.6 Soft Fail v.s. Hard Fail

The first time we tried the bcm5700 driver it came up and started talking on the LAN. I was impressed until I tried to *ifconfig eth0 down* the NIC. The system MCA'd. Using MCA state dump, I was able to determine the address which failed to respond was a register on the BCM5701 chip.

After tracing through lots of code, we finally figured out what was happening. The bcm5700 driver was resetting the card twice during the `close(2)`. And the bcm5700 chip wasn't being re-enabled on the PCI bus after the second reset. The MCA occurs after the `close(2)` when a request for statistics tries to read data from the now defunct BCM5701 chip. Bad driver. Don't do that. HP implements *Hard Fail* in its chipsets. HP engineers decided it's better to crash a server if it's known the drivers do not properly handle failed reads (return -1 typically).

The Intel Itanium systems don't crash running the bcm5700. I gather tradional PCs implement *Soft Fail* since it seems to be OK to get garbage back from failed MMIO reads. I suspect it's because the problem will look like a SW problem (which it is) and not a HW problem. I.e. the HW vendor doesn't have to take the support call and doesn't look worse than its competitors.

AFAIK, LBA supports this mode of operation as well but can only be enabled by modifying kernel source. I like using HW to expose SW problems. I don't expect this to change.

5 BIOS vs EFI drivers

Some drivers (e.g. VGA, megaraid) depend on expansion BIOS to initialize and fire up the card before the linux driver sees it. The previous Itanium platform EFI emulates x86 and supports the x86 BIOSs. For better or worse, HP decided to drive the migration to EFI at the risk of backwards compatibility. In order to work on HP ZX1 systems, an EFI "driver" must be provided to do the same thing. To date, all the IO card vendors that supply HP have committed to providing such a driver and I know they are delivering or have delivered.

6 iDebugging IO driver crash

You wrote a driver and tried it on an HP ZX1 box. It crashed. Welcome to hell ...just kidding. Like PA-RISC systems, IA64 platforms provide a crash state under several circumstances. MCA and INIT are two of those circumstances that are interesting for developers. For the PA-RISC literate, MCA and INIT roughly equate to *HPMC* and *TOC* respectively.

6.1 Intro to `errdump MCA`

MCAs will occur anytime an error signal is broadcast on the McKinley bus. For driver problems, this is typically a CPU read time out. CPU read timeouts occur when a dereferenced MMIO address don't return before a timer in the CPU expires. Since MMIO writes are *Posted*, normally the *victim* is a MMIO read even if a MMIO write caused the error.

Two cases can cause this: either the PCI device stopped responding (e.g. firmware died, chip locked up, MMIO BAR disabled) or a DMA was attempted to an invalid address. The former cases can typically be debugged with printk and knowing which address caused the dump.

One can view the MCA dump with `errdump MCA` command at the EFI shell. Once the MCA data is captured and saved, it's usually a good idea to `errdump clear`. IA-64 Linux will print this dump on the next boot. That's ˜1000 lines of output in a less friendly format. And make sure to save the matching System.map in order to look up symbols. When loading kernel modules, squirrel away the dynamically linked symbols too.

Here is what some of the fields mean:

- `IIP` is the current Instruction Pointer when the system noticed the error.
- `XIP` is the IIP of the most recent trap or interrupt occurred.
- `Requestor ID` is the ID of the originator of a transaction.
- `Responder ID` is the ID of the device that responded with data
- `Target ID` is IO address we are trying to reach.

6.2 Intro to `errdump INIT`

An INIT is used like an NMI. It resets the machine and saves the current state. HP ZX1 platforms have a small blue button in the back of the box label which can be used to generate an INIT. Like an MCA, similar data gets stored.

To be honest, I've never used an INIT and only know of it. Problems I tend to chase are MCAs and not lockups.

7 Acknowledgements

I've learned a lot from folks in the HPUX community when I worked on it and continue to learn from them. I dare not name names for fear of retribution.

And for the past two years, I've been learning new things from (in no particular order): Lamont "NMU" Jones, Ryan Bradetich, Matthew Wilcox, Martin Petersen, Paul Bame, Bdale Garbee, Jes Sorensen, Dave Miller, and a host of other Open Source kernel and application hackers.

More information about IA64-linux can be found at:

`http://www.linuxia64.org/`
`http://www.hp.com/`

Reverse engineering an advanced filesystem

Christoph Hellwig
LST e.V.
hch@lst.de
http://verein.lst.de/˜hch/

Abstract

The *VxFS* filesystem from VERITAS is an example of a UNIX filesystem that not only offers a broad range of advanced functionality, such as extent based allocation, intent logging, snapshoting, but also has on-disk formats with various differences between the versions and ports.

FreeVxFS is a freely available implementation of the VxFS filesystem format for Linux, implemented only by looking at the very rarely available public documentation and reverse engineering the SCO UnixWare version of the commercially available VXFS driver.

1 Introduction

Today's operating systems feature a wide vary of commercially available advanced filesystems. Only for a small number of such filesystems (for examples IBM's *JFS2* family or SGI's *XFS*) does a freely available Linux kernel implementation exist.

Linux contributors already have implemented support for many simpler proprietary filesystems like *FAT* or *EFS*, but the there are only few independent implementations of complex filesystem designs such as Microsoft's *NTFS*.

The VERITAS Filesystem (VxFS) originated from a joint-venture of VERITAS and the Unix System Laboratories (USL) in the early 90's of the last century to develop an advanced filesystem for System V Release 4 Unix (SVR4), featuring capabilities such as read-ahead logging (commonly called journaling) and copy-on-write snapshots at the filesystem level. Today it has been ported to a great number of Unix derivatives such as Sunsoft Solaris, Sequent (IBM) Dynix/ptx, Hewlet-Packard HP-UX, or Caldera OpenUnix; and non-Unix platforms like Microsoft Windows 2000.

VxFS is a proprietary VERITAS product and delivered in source or binary form to paying customers. There is, on the other hand, no public documentation of the on-disk format used by VxFS, which makes implementations other then VERITAS' difficult to implement. There is need to access VxFS-formated disks from Linux for various reasons, like migration from obsolete Unix platforms or access to foreign files for development. (In the author's case this was the Linux-ABI binary emulation framework).

In the first Quarter 2002, VERITAS announced the commercial availability of the VERITAS Foundation Suite for Linux, featuring a port of their VxFS-implementation. Being tied to obsolete versions of the Red Hat kernel package and priced in 4-digit US-dollar range, it is not an option for most possible FreeVxFS uses, though.

2 Legal Background

When implementing a non-trivial piece of software that inter-operates with another software you either need a written specification of the interface, or you need to look at the other software, using helper tools such as disassemblers; this is called *reverse engineering*. In the VxFS case there is no proper documentation of the filesystem's layout as stored on disk so *reverse engineering* is the only available choice. In the European Union reverse engineering of software products is handled by the Directive on Software Copyright Protection from 14 May 1991. In Article 6 (Decompilation) it states the following:

> The authorization of the rightholder shall not be required where reproduction of the code and translation of its form within the meaning of Article 4 (a) and (b) are indispensable to obtain the information necessary to achieve the interoperability of an independently created computer program with other programs, provided that the following conditions are met:...

As the creation of a Linux driver for a proprietary filesystem matches the terms of interoperability used in this EU law exactly, we are explicitly allowed to examine an existing, licensed installation of VxFS to gain information for implementing a free replacement for Linux.

Another interesting legal problem came up when the released driver was merged into the official Linux kernel tree, as VERITAS claimed the use of "vxfs" as driver name was threatening their Trademark. The driver name was changed to freevxfs to avoid further legal problems.

3 Getting Started

Structure is more important than code—this golden programming rule is even more important when trying to archive format compatibility with an existing software.

To implement an independent driver for a filesystem layout, one needs to know every single structure that is written on disk in detail to archive full compatibility. On the other hand the inner workings of the drivers might be completely different, especially if they were written for different environments (e.g. operating systems in this case).

Thus the first step to produce a free VERITAS filesystem driver for Linux was to create a full description of the disk layout that is not directly derived from the original code. As starting point I used the public available documentation of UNIX vendors shipping VxFS with their products, but the results were discouraging, there were only two documentation sets that contain non trivial information about the VxFS disk on-disk format: first the VxFS System Administrator's [AdmGuide] has a chapter titled "VxFS disk layout" which contains a very high-level documentation of the basic format elements; second the inode_vxfs (4) [Inode] and fs_vxfs (4) [Fs] on HP-UX contain a description of many fields of the VxFS superblock and inode, but neither document defines even one element of the VxFS structure completely.

To get a suitable description despite this lack of human readable-format description, reverse engineering techniques had to be applied. There are three major reverse engineering procedures in the context of software: disassembly, use of symbolic debugging information, and protocol snooping. For the development of FreeVxFS, only the first two were used, as protocol snooping of block storage devices requires special and only rarely available hardware (with hardware-emulators like Bochs this is in the process of becoming easier).

4 Symbolic Debugging

Any modern C compiler has a mode to include information about the source-level representation with a binary program to allow high-level debugging with tools like GNU gdb.

VERITAS' VxFS product does not contain any program with such debug information, but it contains C headers files to allow access to its structures from custom programs.

Technically these files could be directly used by a free implementation of VxFS to use its definition similar to free programs using other system headers on proprietary operating systems. The problem with this approach is that it requires the compilation of the driver to happen only on systems with licensed installations of VERITAS' filesystem product, thus disallowing a free operating system to be used as development host.

So instead of directly using the header files, the structural information is extracted from the them using the aforementioned debugging methods to create a set of new and entirely free headers that define the VxFS on-disk format. The program that pulls in the headers is very simple, as it needs to have no functionality at all—it exists only to allow debugging information to be generated.

```
#include <sys/types.h>
#include <sys/time.h>

/* fix compilation with gcc */
#define uint8_t __junk

#include <sys/fs/vx_machdep.h>
#include <sys/fs/vx_gemini.h>
#include <sys/fs/vx_param.h>
#include <sys/fs/vx_layout.h>

main()
{
}
```

Once compiled with debugging options enabled (e.g. `gcc -g`), we have a binary suitable for attacking with a debugger such as gdb. This process is time-consuming as the names of the different record types have to be guessed from the previously mentioned public documentation and often one of the custom types embeds a number of other such types. In addition all scalar types are shown in form of C language basic types by gdb and all typedef information is lost. To allow a portable format description that can also be used (e.g. on computers with 64-bit wide longwords) all occurrences of types that have different sizes on different Linux ports must be replaced with proper, explicitly sized types. The following example gdb session shows the definition of the on-disk inode used by VxFS (`struct vx_dinode`):

```
(gdb) ptype struct vx_dinode
type = struct vx_dinode {
    struct vx_icommon di_ic;
}
(gdb) ptype struct vx_icommon
type = struct vx_icommon {
    long int ic_mode;
    long int ic_nlink;
    long int ic_uid;
    long int ic_gid;
    vxhyper_t ic_size;
    struct timeval ic_atime;
    struct timeval ic_mtime;
    struct timeval ic_ctime;
    char ic_aflags;
    char ic_orgtype;
    u_short ic_eopflags;
    long int ic_eopdata;
    union vx_ftarea ic_ftarea;
    long int ic_blocks;
    long int ic_gen;
    vxhyper_t ic_vversion;
    union vx_org ic_org;
    long int ic_iattrino;
}
(gdb)
```

The symbolic debugging information was the most useful resource during the FreeVxFS development.

5 Disassembly

The author has examined most of the UnixWare VxFS binary driver module with various disassemblers.

The simplest form of disassembly can be produced by the program `objdump` from the GNU binutils package. Its output for trivial vnode operations appears in Figure 1.

Commercially available disassemblers like DataRescue's IDA Pro (the windows version runs under Linux/wine) offer additional features such as the naming of data types or additional symbol resolving that should not be discussed further here.

Although this uncovered a number of interesting facts like the exorbitant stack usage in VERITAS's driver, the disassembly of the UnixWare VxFS driver did not uncover a notable amount of information useful for the current publicly available read-only FreeVxFS versions. On the other hand, the slowly progressing development of write support would be impossible without the use of disassembly, mostly because a number of bitmap operations in the block/inode allocators is not documented in any other way than the program itself.

6 Implementation

As already mentioned previously, FreeVxFS targets the Linux kernel from version 2.4 upwards. There are various reasons for this choice:

- Linux is the free operating system with the biggest overall user count. The 2.4 kernel was the upcoming stable release when the development was started.
- The Linux kernel allows runtime loading of independently developed filesystem modules. This allowed the early FreeVxFS development to happen without ties to the direct kernel development and with very short turnaround times.
- All recent Linux kernels feature a rich set of generic routines that can be used in filesystem code. Starting with the 2.4 kernel most of the data I/O path is handled by such generic code, thus letting filesystem developers concentrate on difficult aspects of their particular filesystem implementation.

The early FreeVxFS development was done as a project separate from the main Linux kernel tree and supported different kernel versions with the same codebase. Starting with the 2.4.6 prereleases it merged into the official Linux kernel tree and is maintained as part of it.

Like most filesystem drivers, FreeVxFS development was done incrementally in the beginning—that means support for the different parts of the on-disk format was implemented after the previous one was implemented, component-tested, and considered functional. A feature of the VxFS layout (actually only in the > v1 disk layout, but FreeVxFS doesn't support the historic VxFS v1 filesystems anyway) made this traditional approach impossible very early. The issue is that most of the metadata describing a filesystem is not directly stored in the superblock but in data blocks pointed to by regular inodes—this includes the inode table itself! (This is found by its containing extent to avoid endless recursion.)

To get past this point, a huge amount of code (almost half of the FreeVxFS implementation) had to be implemented at once, without previous testing of individual components. Of course this led to a number of very hard-to-debug problems and accounted for more than two-thirds of the development time.

7 Work in Progress and Future Plans

During the last month the FreeVxFS driver in the main kernel tree was steadily improved by bugfixes reported and/or fixed by the users. In addition support for different VxFS variants (block size, superblock locations) was added to support a broader range of target systems.

Ongoing major short-term development includes support for byte-swapping in the filesystem driver, allowing access to filesystems that were created on computers using a different byteorder than the accessing system (a feature VERITAS' driver is still lacking!) and a proper way to handle VxFS filesystems for HP-UX that have various small differences in the layout of important layout elements (e.g. the inode).

The most important long-term development project is to implement support for writing to VxFS filesystems. This feature is already functional in early stages, but fragments the filesystems so badly that it is of no practical use.

References

[AdmGuide] *VxFS System Administrator's Guide* VERITAS Inc. `http://ou800doc.caldera.com/ODM_FSadmin/CONTENTS.html`.

[Inode] *inode (vxfs) - format of a VxFS inode* Hewlett-Packard Company. `http://devresource.hp.com/STK/man/11.00/inode_vxfs_4.html`, (1997).

[Fs] *fs (vxfs) - fs format of VxFS file system volume* Hewlett-Packard Company. `http://devresource.hp.com/STK/man/11.00/fs_vxfs_4.html`, (1997).

Figure 1: Disassembly via `objdump`

```
0000000000075ba0 <vx_open>:
   75ba0:       8b 54 24 04             mov    0x4(%esp,1),%edx
   75ba4:       57                      push   %edi
   75ba5:       33 ff                   xor    %edi,%edi
   75ba7:       56                      push   %esi
   75ba8:       8b 02                   mov    (%edx),%eax
   75baa:       8b 70 28                mov    0x28(%eax),%esi
   75bad:       8b 40 24                mov    0x24(%eax),%eax
   75bb0:       83 f8 01                cmp    $0x1,%eax
   75bb3:       75 57                   jne    75c0c <vx_open+0x6c>
   75bb5:       8b 86 a4 01 00 00       mov    0x1a4(%esi),%eax
   75bbb:       25 00 f0 00 ff          and    $0xff00f000,%eax
   75bc0:       3d 00 90 00 00          cmp    $0x9000,%eax
   75bc5:       74 3d                   je     75c04 <vx_open+0x64>
   75bc7:       8b 44 24 10             mov    0x10(%esp,1),%eax
   75bcb:       a9 00 00 08 00          test   $0x80000,%eax
   75bd0:       75 2a                   jne    75bfc <vx_open+0x5c>
   75bd2:       6a 01                   push   $0x1
   75bd4:       56                      push   %esi
   75bd5:       e8 fc ff ff ff          call   75bd6 <vx_open+0x36>
   75bda:       83 c4 08                add    $0x8,%esp
   75bdd:       8b 46 5c                mov    0x5c(%esi),%eax
   75be0:       85 c0                   test   %eax,%eax
   75be2:       75 0a                   jne    75bee <vx_open+0x4e>
   75be4:       8b 46 58                mov    0x58(%esi),%eax
   75be7:       3d ff ff ff 7f          cmp    $0x7fffffff,%eax
   75bec:       76 05                   jbe    75bf3 <vx_open+0x53>
   75bee:       bf 4f 00 00 00          mov    $0x4f,%edi
   75bf3:       56                      push   %esi
   75bf4:       e8 fc ff ff ff          call   75bf5 <vx_open+0x55>
   75bf9:       83 c4 04                add    $0x4,%esp
   75bfc:       8b c7                   mov    %edi,%eax
   75bfe:       5e                      pop    %esi
   75bff:       5f                      pop    %edi
   75c00:       c3                      ret
   75c01:       83 c7 00                add    $0x0,%edi
   75c04:       bf 59 00 00 00          mov    $0x59,%edi
   75c09:       eb f1                   jmp    75bfc <vx_open+0x5c>
   75c0b:       90                      nop
   75c0c:       5e                      pop    %esi
   75c0d:       5f                      pop    %edi
   75c0e:       33 c0                   xor    %eax,%eax
   75c10:       c3                      ret
   75c11:       83 c7 00                add    $0x0,%edi
   75c14:       81 ff 00 00 00 00       cmp    $0x0,%edi
   75c1a:       81 ff 00 00 00 00       cmp    $0x0,%edi
```

BitKeeper for Kernel Developers

Val Henson *Jeff Garzik*
val@nmt.edu *jgarzik@mandrakesoft.com*

Abstract

BitKeeper[1] is a revolutionary new distributed source control management suite which is ideal for Linux kernel development. BitKeeper provides tools which automate and simplify many common kernel development tasks. In this paper, we describe basic BitKeeper concepts and operations, BitKeeper solutions for common kernel development problems, and a workflow for interacting with other Linux developers using BitKeeper. We also discuss some of BitKeeper's shortcomings and what is being done to correct them. We conclude that BitKeeper can dramatically improve the efficiency of Linux kernel developers.

1 Introduction

A new source control system is available - why should Linux kernel developers care? Because this particular source control system was designed from the ground up to solve exactly the problems inherent in Linux kernel development. Kernel developers need to manage thousands of files, live and work all over the world, often have limited bandwidth and connectivity, and frequently merge large numbers of changes. Older source control systems were designed for a development model where most developers worked in the same physical building and had 24-hour access to a central repository over high bandwidth local networks. The developers, rarely numbering more than 100 per project, normally checked in changes directly to the central repository, and could easily communicate with other developers working on the same part of the code. Unsurprisingly, the source control software written under these assumptions was not very useful for thousands of loosely connected developers distributed world-wide.

The BitKeeper distributed source control system was designed for, written for, and tested by Linux kernel developers. Linux kernel development provided the perfect test case for a truly distributed source control system, and BitKeeper has been and continues to be shaped by input from kernel developers. As a result, it is preeminently useful for kernel development. The purpose of this paper is to familiarize kernel developers with the most useful and time saving features of BitKeeper, so that developers can spend less time on mechanical make-work and more time on development. After reading this paper, developers new to BitKeeper may consider trying BitKeeper for the first time, and developers already using BitKeeper may learn a few new tricks.

First, we'll briefly review basic BitKeeper concepts and operations (experienced BitKeeper users should skip this section). We'll then examine a variety of problems frequently encountered during kernel development and show how BitKeeper solves these problems. Next, we'll review the workflow involved in using BitKeeper for Linux kernel development. Finally, we'll discuss some of the shortcomings of BitKeeper and what is being done to correct

[1] BitKeeper is a trademark of BitMover, Inc.

them.

2 Basic BitKeeper Concepts

This section presumes knowledge of basic source control concepts such as "check in" and "check out." We will instead concentrate on the ways in which BitKeeper is different from traditional source control systems. Some of the major differences between the architecture of traditional source control systems and the architecture of BitKeeper exist in order to satisfy one of its key design requirements: Developers should be able to commit work locally, without accessing a remote repository, until the developer is ready to merge with the remote repository. Some other key design goals were reproducibility, data integrity, and performance.

2.1 Running BitKeeper

First, let's go over the nuts and bolts of using BitKeeper: How do you get it, and how do you run it? Download BitKeeper by going to:

`http://www.bitkeeper.com`

And clicking on "Downloads." All BitKeeper commands are of the form "`bk <command>`" to avoid namespace clashes. BitKeeper has built-in help, just run "`bk helptool`" (for the GUI tool) or "`bk help`" (for the command line tool). While BitKeeper has many useful graphical tools, a developer can work with BitKeeper using only the command line tools - BitKeeper does not require a windowing environment. We also recommend that first-time users run the demo, which is at:

`http://www.bitkeeper.com/Test.html`

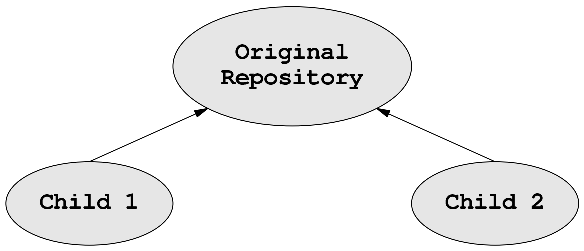


Figure 1: Parent pointers after cloning.

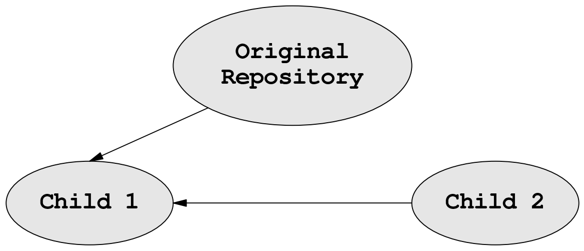


Figure 2: Parent pointers after changing with "`bk parent`".

2.2 Clones, parents, and children

A BitKeeper repository is a collection of source controlled files. To create a working copy or the equivalent of a CVS sandbox, a developer "clones" a repository. The word "clone" was chosen because cloning a repository creates an exact copy of the original repository. All of the information and administrative files in the original repository are included in the new repository, making it possible to work in any repository completely independent of any other repository. After the clone is completed, the new repository regards the original repository as its "parent." The parent of a tree can be changed at any time, to any other related tree, or to no tree at all (see Figures 1, 2). Because each repository is identical, any repository can be cloned, and the child of one repository can also be the parent of another repository (see Figure 3). Note, however, that despite all the parent-child terminology, BitKeeper repositories interact on a peer-

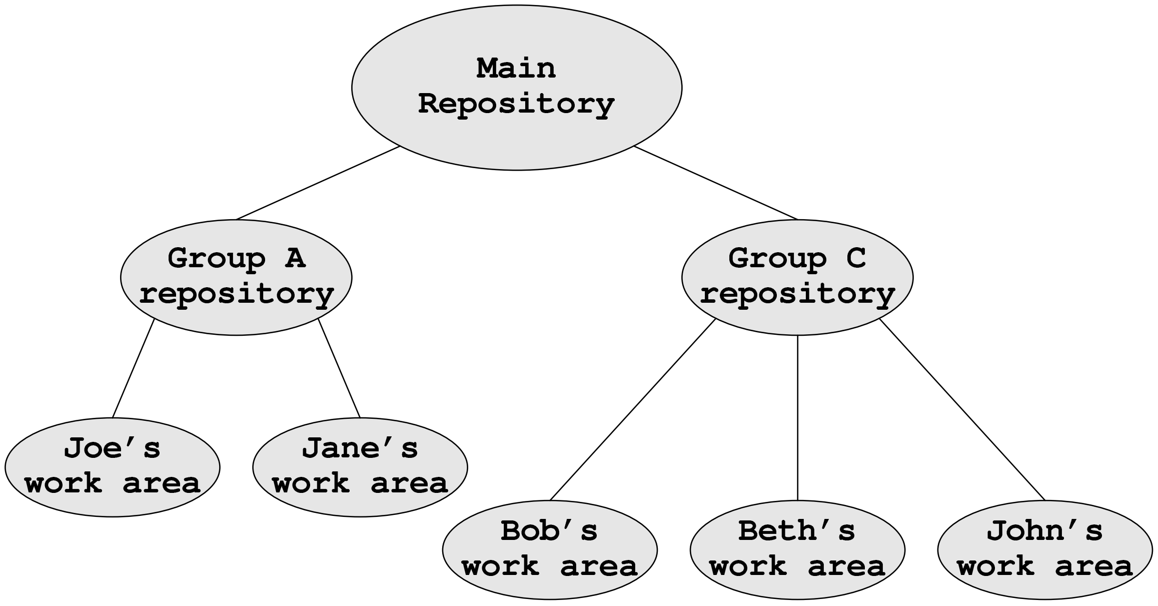


Figure 3: Example BitKeeper repository structure.

to-peer basis, since the relationship between any two trees can be changed at any time.

2.3 Changesets

In BitKeeper, changes to individual files are grouped together into changesets. A changeset is a grouping of one or more deltas to one or more files representing a single logical change. Each changeset can contain multiple deltas to the same file. Each revision to each file in the changeset is commented, as is the changeset as a whole. Logically related changes to separate files can now be explicitly grouped together. For example, if one bug fix requires changes to three different files, all three files' changes can be grouped into one changeset. Being able to explicitly group changes together rather than guessing at their relationships (from last modified date, or location in the same directory) is very useful. Even more useful is that each changeset is an automatic synchronization point, similar to a CVS tag. Users can reproduce the exact state of the repository as of the point that any changeset was committed.

2.4 Push and pull

Changesets are exchanged between repositories using "`bk push`" and "`bk pull`". (Note that commits modify only the local repository, and do not affect the parent repository.) Push will send changesets from the child to the parent, and pull will retrieve changesets from the parent to the child. Each push or pull only sends the changesets which are present in one tree but not in the remote tree (see Figure 4). A push will only send changes which are already merged with the changes in the remote tree, so merging with another tree is done by first pulling the remote tree's changes, merging them in the local tree, and then pushing the merged changes back. Push and pull will by default push to or pull from the parent of the local tree, but these commands can also take an argument specifying a different tree to push to or pull from.

Clones can be thought of as creating a personal, private, unnamed "branch," and pulls as a convenient way of merging with the "trunk"

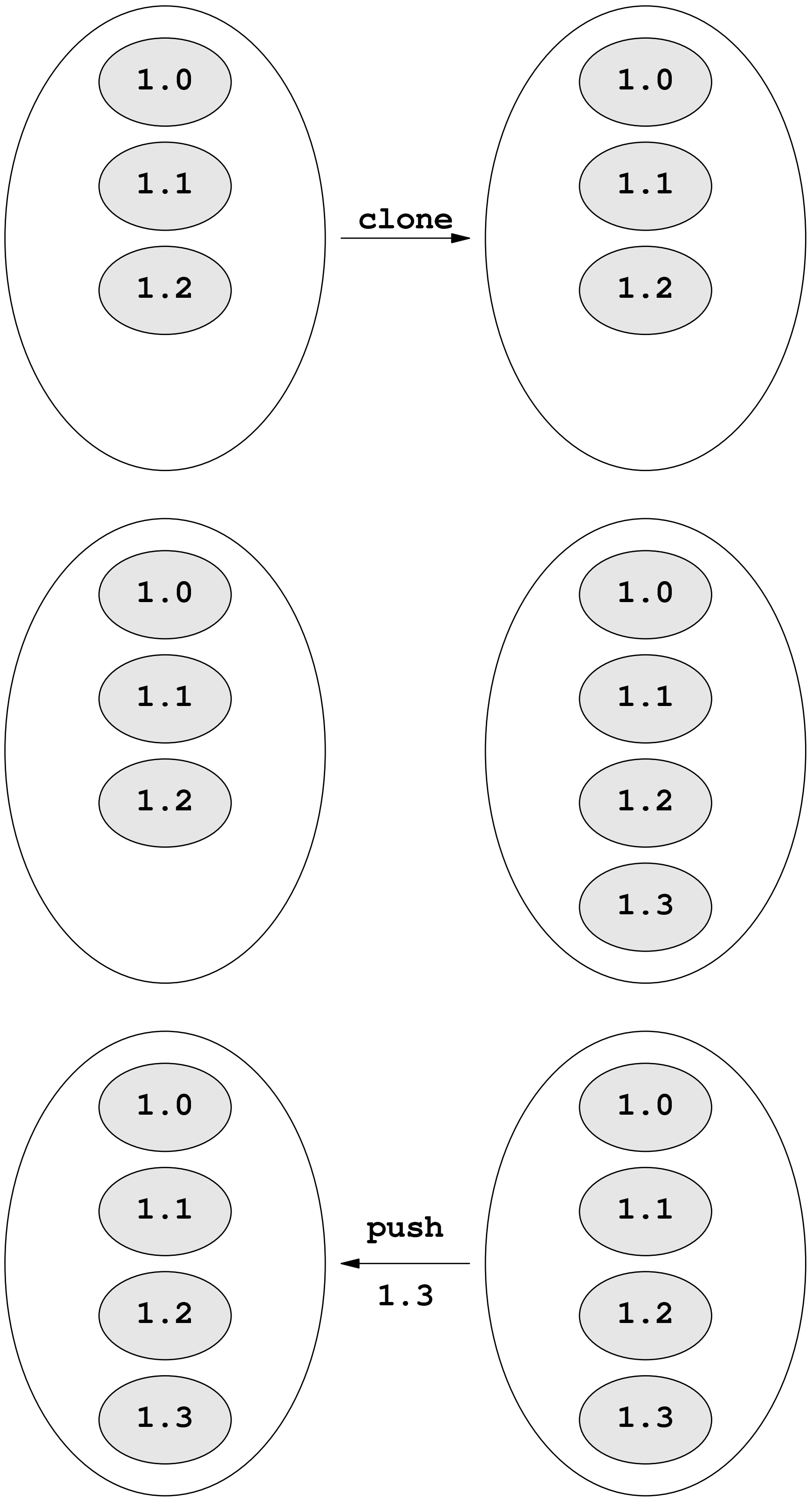


Figure 4: Example of a push: Initial clone, commit a change, push it back.

without pushing the "branch's" changes to the "trunk." The push-pull model gives developers control over whether or not local changes are pushed to other repositories without sacrificing ease of synchronization with other repositories. CVS users will enjoy the freedom of being able to commit half-finished changes without breaking the main tree.

2.5 Conflict resolution

Usually, a push or a pull that changes a locally modified file will be auto-merged by BitKeeper. In typical use, BitKeeper auto-merges approximately 95% of conflicts that would not have been merged by CVS or `diff` and `patch`. The percentage of successful merges relative to CVS actually increases with the number of developers working on the same repository. BitKeeper improves the auto-merge rate in two ways. First, each merge is only done once - CVS remerges from the point where the trunk and the branch first separated every time a branch's changes are pushed back to the trunk. BitKeeper only needs to merge the changes since the last changeset shared by the two repositories. Second, BitKeeper uses a unique merging algorithm that no other source control system implements. The improvement in the success rate of the merge algorithm is made possible by storing certain kinds of meta-data for each file that neither CVS nor `diff` and `patch` can store or generate.

Each pull to a locally modified repository results in the creation of a changeset, which is empty if no files needed to be merged (see Figures 5, 6). Occasionally, a pull will result in a conflict that can't be auto-merged. The BitKeeper command "`bk resolve`" offers a menu of options for each file with conflicts, ranging from "Use local file" to "Merge using graphical three-way file merge tool." Once all the conflicts are resolved, the changes required to resolve the conflicts are saved in the changeset created by the merge, along with your comments. For developers who don't trust auto-merging, "`bk pull`" has an option to disable the auto-merge feature. Each conflict can then be individually hand-merged or auto-merged and the results approved before being committed. We recommend that developers try BitKeeper's auto-merge algorithm even if they have had bad experiences with auto-merging in the past; the new algorithm is an immense improvement over all previous algorithms and, in the authors' experience, always merges changes correctly.

What we just described is only the most common kind of conflict, a conflict in the data of the file itself, or a content conflict. BitKeeper also resolves conflicts in many other file attributes: permissions, ownership, type, pathname, and more. Viewing the pathname of a file as just one more file attribute makes it easy to move files around within a BitKeeper repository.

3 Kernel Development Problems and Solutions

Now that we've explained the basic terminology, let's get to the interesting part: real-life scenarios where BitKeeper makes kernel development less painful. All the scenarios described were experienced first-hand by the authors while actually engaged in useful kernel development. They were not artificially constructed to show off interesting but useless features of BitKeeper but instead are commonly encountered problems solved by using BitKeeper. Some of the described solutions are implemented by other source control systems, but are not easy to do with `diff` and `patch`, the most commonly used tools for working with the Linux kernel source. We'll start with simple scenarios that are relatively easily handled by any source control more complex than

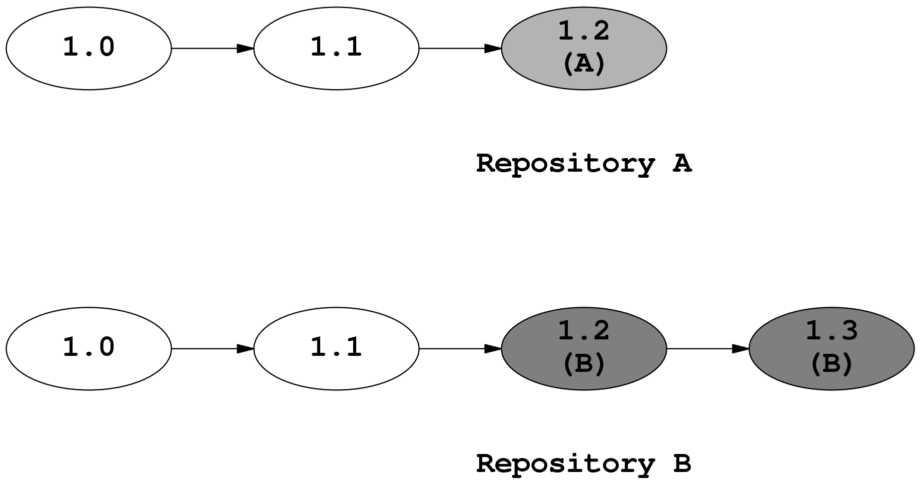


Figure 5: Repositories before merge, shaded changesets were added since clone.

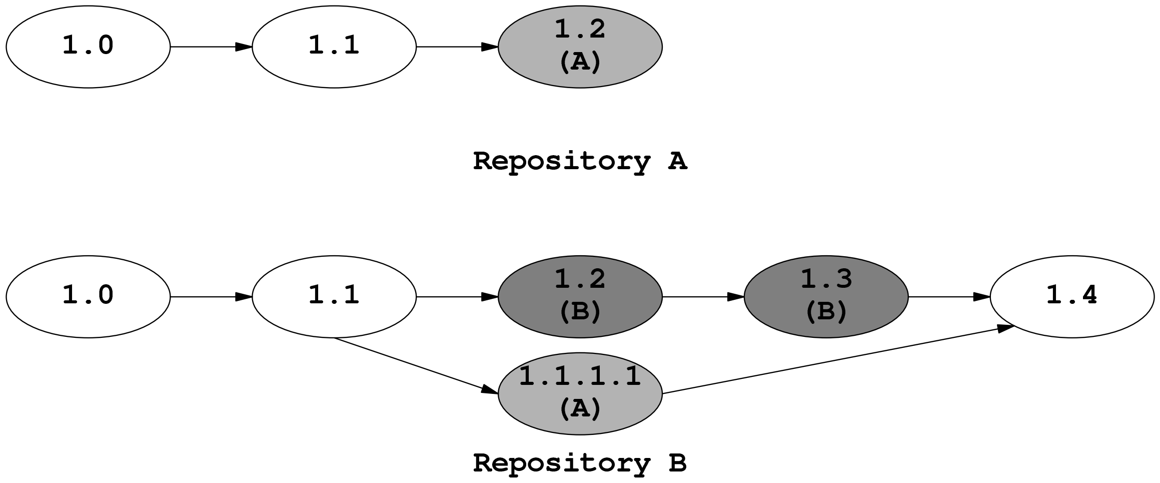


Figure 6: After a pull of A's changes to B, with A's changes merged.

diff and patch, and gradually build up to more difficult scenarios where more advanced source control management systems fail.

3.1 Maintaining different trees

Any serious kernel developer will be familiar with this scenario: You maintain both a stable kernel and a development kernel. The stable kernel contains a few bug fixes and some minor but safe improvements. The development kernel contains some riskier changes, new features that haven't been tested well yet, half-written drivers, and lots of debugging statements. Most likely, it also contains all of the changes in your stable kernel - or it would, if you always remembered to patch it with your latest changes to the stable kernel. But your development kernel is just different enough that patch fails to apply your diffs from the stable kernel cleanly, especially if you have moved a few files around. When you want to transfer your development changes into your stable kernel, parts of the patch usually have to be hand-applied. It's generally a pain to keep your development and stable kernels in sync.

3.1.1 Solution

Clone your development tree from your stable tree. Whenever you make a change in your stable tree, run "bk pull" from your development tree. When your development changes are ready, "bk push" them to your stable tree. BitKeeper's auto-merge algorithm merges the majority of your changes for you, even when you've changed the location of some of the files.

This scenario can be generalized to any number of child repositories, each with their own child repositories. A developer could have "really stable," "stable," "semi-stable," "unstable," and "broken" trees, or one child for each set of experimental changes (see Figures 3, 7). Most developers using BitKeeper have anywhere from 5 to 50 different clones of the same repository, each for for a different set of changes. You might be worried about disk space at this point, but "bk clone" has an option to hard-link the files in the new repository to the files in the old repository if they are both on the same filesystem, so only the files that have actually changed take up any significant amount of disk space.[2] A clone can be thought of as a branch, except that it is far easier to create and merge back to the "trunk" than in most source control systems. Cloning a new repository is so easy that you'll find yourself doing it for the most trivial of purposes.

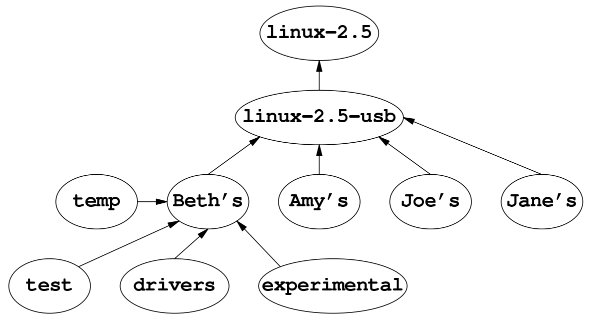


Figure 7: Example BitKeeper repository structure.

3.2 Updating to the latest version

You went on vacation for two weeks, and now you are back and 10 patches are pending for various kernel trees. Your automatic patch application script chokes because the naming convention has changed - again. Plus, one of the patches on your local ftp.kernel.org mirror was corrupted and won't be updated until midnight, local time. You settle down for a long night of painful hand-application of

[2]This is only a partial solution, see the discussion of "lines of development" in the section "BitKeeper Drawbacks."

patches.

3.2.1 Solution

"`bk pull`" downloads and applies the changes for you, regardless of what the latest kernel version was named, Linux 2.3.42, or Linux 2.4.0-test-pre7-sr71-blackbird-unstable. Data integrity checking at every step prevents any part of your tree from getting corrupted. Your update is even faster because BitKeeper compresses the information it sends over the network (it even reports the compression factor when it uncompresses the data locally).

3.3 Merging after long separation

You've decided to concentrate on getting USB working - really working, some major improvements, and you're not going to have time to merge with the vanilla kernel every time a prepatch is released. Two months later, you look up and realize that you now have 2MB of diffs between your tree and the mainline. You apply the patches, run a "`find . -name '*.rej'`" and write off getting any useful work done for the next few hours.

3.3.1 Solution

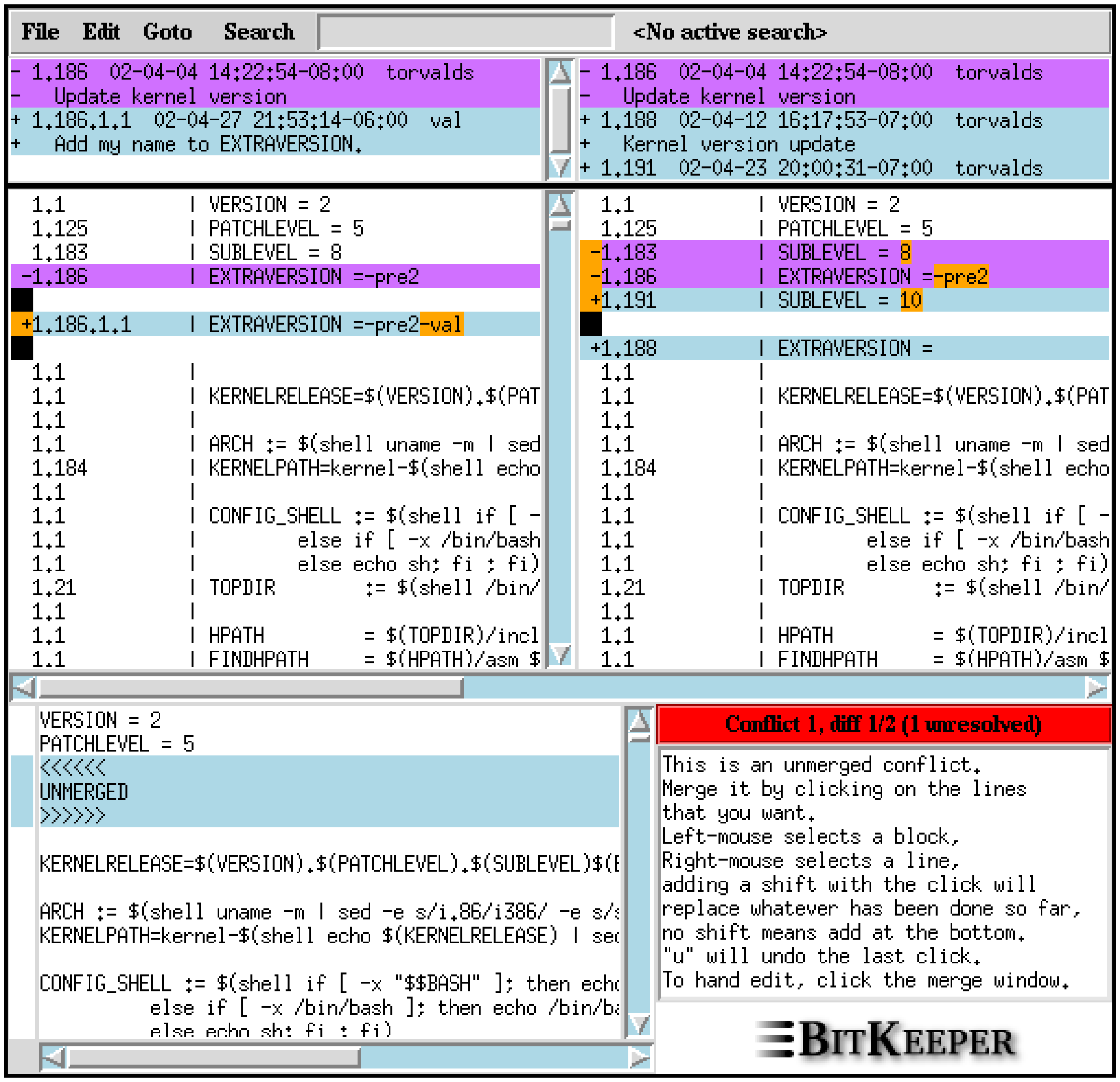
"`bk pull`" applies and auto-merges most of the changes for you. Occasionally, BitKeeper's auto-merge algorithm finds conflicts it can't resolve. At this point, "`bk resolve`" and the graphical three-way file merge tool turn what is usually 3 hours of work with `patch`, `find`, and your favorite editor into 15 minutes of point and click. The three-way file merge tool shows you the local and remote versions of the file side-by-side, with the differences color highlighted (see Figure 8). The changeset comments for each version of the file are shown above each file. The bottom half of the window shows the partially merged file, and navigation keys are described in the lower right-hand corner. When you've finished merging one conflict, by clicking on the lines from each file that you want and/or hand-editing in the merge window, hit the key to jump to the next conflict. When you're happy with the merged file, save the file and go on to the next file with conflicts. Simpler commands exist for simpler problems, for example, "Use remote file" simply replaces the local file with the remote file.

One of the authors recently merged a heavily modified 2.4.12 kernel tree with a 2.4.16-based kernel tree using `diff` and `patch` (no BitKeeper tree was available for the 2.4.16 version). It took her approximately three hours. She routinely merges from a heavily modified 2.4.12 kernel to 2.4.18 in 15 minutes,[3] using BitKeeper. Using `diff` and `patch` instead of BitKeeper wasted several hours that could have been spent fixing a particulary vexing timer interrupt bug.

3.4 Creating a patch

Another developer asks you for your bootloader changes. They're in a tree with several other unrelated projects and a number of other changes that you don't want to send to anyone. You create a patch, hand edit out the "`misc.c~`" file that was accidentally included, and send it off. A few minutes later, the other developer emails you back saying that the kernel no longer boots, but it does print out a whole lot of debugging information. You remember that you forgot to include the changes to `head.S`, and you also forgot to remove that debugging statement triggered by the bug you fixed in `head.S`. Several more iterations and hand-edited patches later, you finally create a working patch.

[3]After a bit of practice. The first few merges took about 30 minutes each.



```
- 1.186   02-04-04 14:22:54-08:00  torvalds
-   Update kernel version
+ 1.186.1.1  02-04-27 21:53:14-06:00  val
+   Add my name to EXTRAVERSION.
```

```
- 1.186   02-04-04 14:22:54-08:00  torvalds
-   Update kernel version
+ 1.188   02-04-12 16:17:53-07:00  torvalds
+   Kernel version update
+ 1.191   02-04-23 20:00:31-07:00  torvalds
```

```
  1.1           | VERSION = 2
  1.125         | PATCHLEVEL = 5
  1.183         | SUBLEVEL = 8
 -1.186         | EXTRAVERSION =-pre2

 +1.186.1.1     | EXTRAVERSION =-pre2-val

  1.1           |
  1.1           | KERNELRELEASE=$(VERSION).$(PAT
  1.1           |
  1.1           | ARCH := $(shell uname -m | sed
  1.184         | KERNELPATH=kernel-$(shell echo
  1.1           |
  1.1           | CONFIG_SHELL := $(shell if [ -
  1.1           |         else if [ -x /bin/bash
  1.1           |         else echo sh; fi ; fi)
  1.21          | TOPDIR        := $(shell /bin/
  1.1           |
  1.1           | HPATH         = $(TOPDIR)/incl
  1.1           | FINDHPATH     = $(HPATH)/asm $
```

```
  1.1           | VERSION = 2
  1.125         | PATCHLEVEL = 5
 -1.183         | SUBLEVEL = 8
 -1.186         | EXTRAVERSION =-pre2
 +1.191         | SUBLEVEL = 10

 +1.188         | EXTRAVERSION =
  1.1           |
  1.1           | KERNELRELEASE=$(VERSION).$(PAT
  1.1           |
  1.1           | ARCH := $(shell uname -m | sed
  1.184         | KERNELPATH=kernel-$(shell echo
  1.1           |
  1.1           | CONFIG_SHELL := $(shell if [ -
  1.1           |         else if [ -x /bin/bash
  1.1           |         else echo sh; fi ; fi)
  1.21          | TOPDIR        := $(shell /bin/
  1.1           |
  1.1           | HPATH         = $(TOPDIR)/incl
  1.1           | FINDHPATH     = $(HPATH)/asm $
```

```
VERSION = 2
PATCHLEVEL = 5
<<<<<<
UNMERGED
>>>>>>

KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(E

ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/s
KERNELPATH=kernel-$(shell echo $(KERNELRELEASE) | sed

CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo
          else if [ -x /bin/bash ]; then echo /bin/ba
          else echo sh; fi ; fi)
```

```
This is an unmerged conflict.
Merge it by clicking on the lines
that you want.
Left-mouse selects a block,
Right-mouse selects a line,
adding a shift with the click will
replace whatever has been done so far,
no shift means add at the bottom.
"u" will undo the last click.
To hand edit, click the merge window.
```

Figure 8: Merging a conflict with the three-way file merge tool.

3.4.1 Solution

Run "`bk revtool`" to find the changeset with the comment, "Fixed the bootloader again" and then run "`bk export -tpatch -r1.203 > ../bootloaderpatch`", which exports that changeset in unified diff format. With Bitkeeper, you naturally group related changes into one changeset with a descriptive comment. Once you've found the changeset(s) you want, BitKeeper automatically converts them into the patch format you prefer. The authors frequently have minor unreleased bugfixes requested by other developers or customers; with BitKeeper, creating and sending the proper patch takes seconds.

3.5 Sharing changes

You'd like to see another developer's changes to the memory management code, but they're not ready to be merged with the main tree. You send email asking for the patch, but the other developer has just gone to sleep. You're in a different timezone and it'll be 16 hours before you get to see the changes. 16 hours later, the two of you go through the usual "Patch doesn't apply" conversation.

3.5.1 Solution

"`bk clone`" the other developer's public BitKeeper repository. Or, if you're both working in a clone of the same repository, just "`bk pull <location of tree>`" to get the other developer's changes. With BitKeeper, developers don't even have to be at the computer to share their latest patches, as long as they have a publicly accessible tree.

This scenario is an example of sideways synchronization, one of the benefits of the peer-to-peer model (see Figure 9). Changes no longer

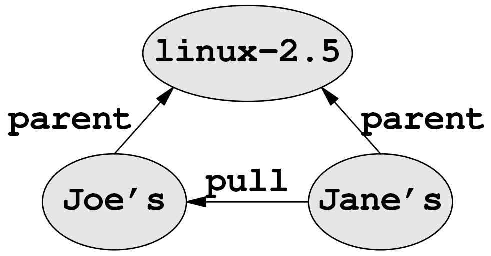


Figure 9: Example of sideways synchronization.

have to committed to the central repository before any other developer can pull them. Now, any clone of the same repository can be merged with any other clone, regardless of when or how it was cloned. Sets of changes can be easily pushed or pulled around a group of related repositories in ways that are extremely useful in the day-to-day life of a kernel developer.

3.6 Moving files

You've just reorganized the `linux/drivers/` hierarchy, again. The patch is huge, so you post to the `linux-kernel` mailing list with a brief description of the change, and the eternal question, "Should I submit the changes to Linus as a patch or as a script?" The inevitable debate ensues, you write at least one buggy script, and Linus eventually gets the changes. The next prepatch is even bigger than usual, and armchair kernel hackers complain bitterly about it for weeks.

3.6.1 Solution

"`bk mv`" the files to their new locations. Since BitKeeper really implements renaming of source controlled files, rather than "abandon the old file and create a new file," the resulting changeset is tiny and almost no one even notices it happened. BitKeeper generates a unique id for each file in the repository at its creation, and will never confuse one file with another just because they happen to have the same pathname. Other developers who pull this changeset will find that their changes to the moved files are "magically" merged into the correct files at their new locations.

3.7 Debugging a patch

You apply the patch for 2.4.18-pre2 and discover that it's broken the NFS server. The patch includes changes to nearly every file in `fs/nfsd/`, and most of the changes appear to be cosmetic or related to that API change last week. You wearily page through the diff, searching for something that actually changes the behavior of nfsd.

3.7.1 Solution

The changesets you pulled are all nicely commented. You start up "`bk revtool`" and type "nfs" into the "Search" field to search the check in comments (see Figure 10), or else you select a file in `fs/nfsd/` and examine the most recent changesets affecting that file. Locating an interesting changeset, you click on "View ChangeSet" and quickly skim the beautiful, easy-to-read graphical diffs (see Figure 11). (While many graphical diff viewing tools exist, this graphical tool is integrated with the changeset viewing tools, which is a significant advantage when trying to understand related changes.) In a couple of minutes, you find a changeset with the comment, "Back out Trond's NFS changes, I don't know what they're for." Since you know Trond is the NFS maintainer, you're a little suspicious of this changeset. To check, you use revtool to find the previous changeset for that file, and you discover that it's a changeset from Trond with the comment, "Fix bug in NFS serving." You quickly run "`bk cset -x<rev>`" to exclude the changeset that reverted Trond's fix, recompile, and have NFS serving working again in 5 minutes flat.

3.8 General debugging

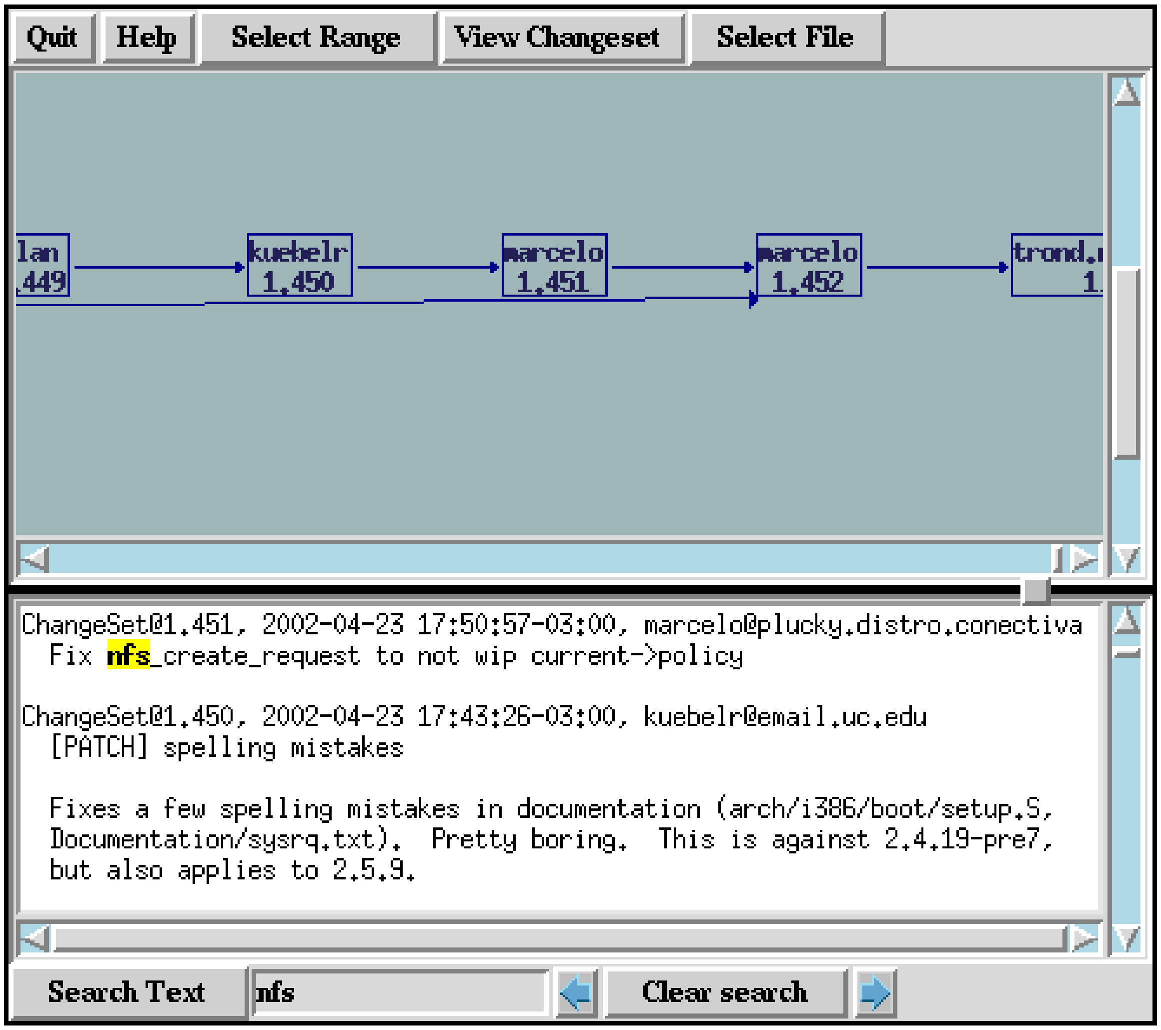
You notice a bug in the yellowfin ethernet driver. It's a minor bug, and it's gone unfixed for quite some time. You try a few different versions but can't quickly find the point where it broke. You type the first of a long series of printks.

3.8.1 Solution

No one has a solution for all bugs, but "`bk revtool`" makes it easy to investigate the changes to a file and the reasons for those changes. Using "`bk revtool drivers/net/yellowfin.c`", you check the history of `yellowfin.c` and find a few suspicious changesets, most notably one from Dave with the phrase, "Totally untested" in the comments. Spending a few minutes with "`bk revtool`" narrows your likely suspects down to a few lines of code and gives you some preliminary ideas of what might have gone wrong. You find that a few endian-ness bugs were introduced during an API change several months ago, and repair the bugs.

3.9 Updating a port

No one has used the Gemini port of the PowerPC branch in 6 months. It doesn't even compile any more. You're new to the PowerPC port and don't know what's changed in the last 6 months. Some major reorganizations have occurred, and it looks like someone attempted to make the required changes for Gemini but never bothered compiling them. You look at other ports but each port varies so wildly that



```
ChangeSet@1.451, 2002-04-23 17:50:57-03:00, marcelo@plucky.distro.conectiva
  Fix nfs_create_request to not wip current->policy

ChangeSet@1.450, 2002-04-23 17:43:26-03:00, kuebelr@email.uc.edu
  [PATCH] spelling mistakes

  Fixes a few spelling mistakes in documentation (arch/i386/boot/setup.S,
  Documentation/sysrq.txt).  Pretty boring.  This is against 2.4.19-pre7,
  but also applies to 2.5.9.
```

Figure 10: Searching for “nfs” in revtool.

you can't find any easy examples to follow. Resigned, you start learning the PowerPC port from first principles, downloading the occasional 2MB diff and sifting through it for clues.

3.9.1 Solution

Use "`bk revtool`" to look at the context of each change to the Gemini port. After clicking on "View ChangeSet" and looking at the graphical diffs (see Figure 11), the source of many of the compilation errors quickly becomes obvious - a major reorganization of SMP support 5 months ago, where the offending code was cut and pasted without changing the variable names. The changeset that accomplished this reorganization gives you many clues about what you need to do to make the Gemini port compatible with the new system. Other bugs become obvious as soon as you look at the history of the relevant files and their associated changesets. Changes that weren't made for the Gemini were made for other ports, providing a model for your bug fixes. Some bugs are more difficult to fix, but in a day or two, you have repaired 6 months of neglect and Gemini is booting again.

We've shown only a few of the more common ways in which BitKeeper can easily save several hours a day for the active kernel developer. BitKeeper goes above and beyond merely archiving old versions of your code, it also provides a powerful set of tools for understanding code and working with other developers.

4 BitKeeper Workflow for the Linux Kernel

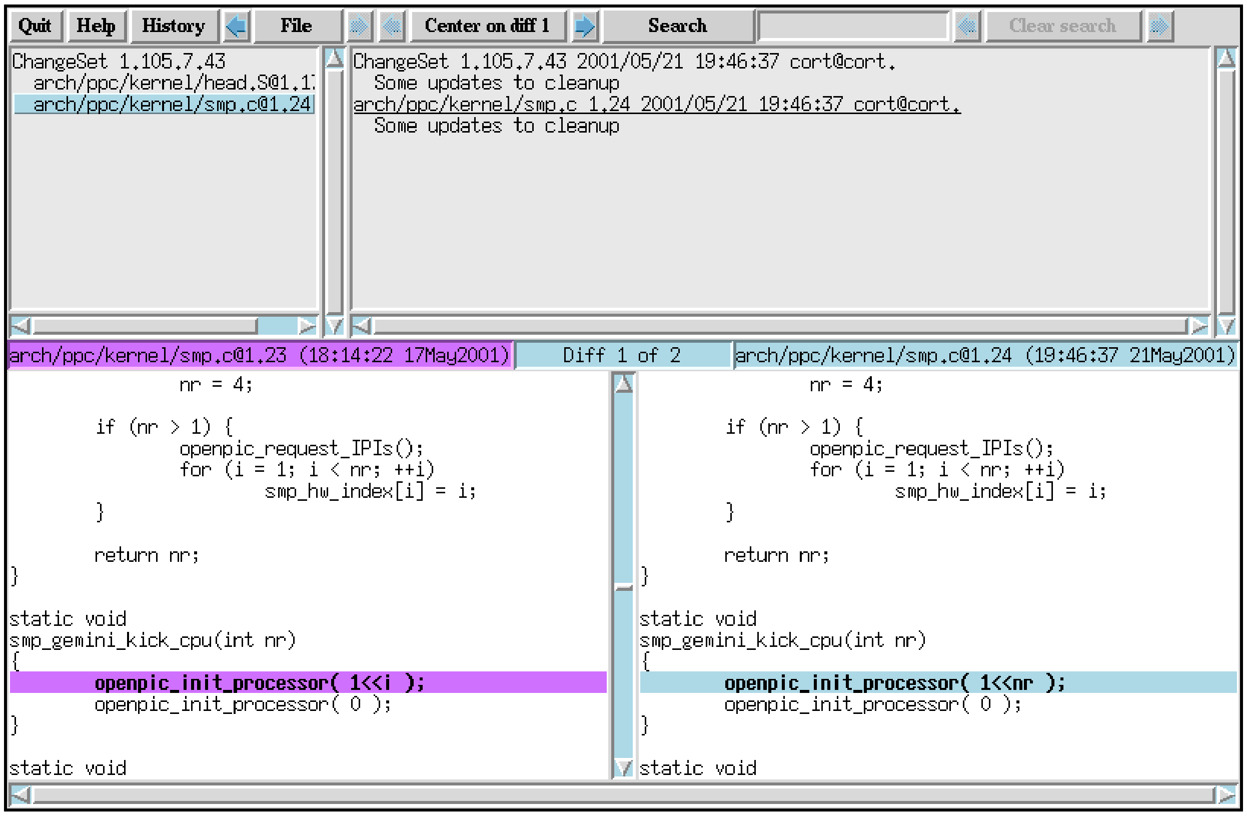
Now that you're using BitKeeper for kernel development, how do you merge your changes with other maintainers and contributors? The Linux development model does not work if all developers are allowed to push to one main repository, which is the only workflow allowed by most other source control systems. Instead, maintainers farther up in the hierarchy pull changesets from those farther down in the heirarchy, creating a series of staging repositories from the lowest levels of development up to the main repository (see Figure 3 for an example with one level of staging). Developers can also push and pull changesets horizontally between any two trees, regardless of where the trees are located in the staging hierarchy. We'll describe the kernel development workflow between a maintainer at the top of the heirarchy and a maintainer one level down.

4.1 Themes

Often the upstream maintainer will happily accept one group of changes but reject another group. Since "`bk pull`" will pull all of the changes that are in the remote tree and not in the local tree, it's important to separate out your changes into logical "themes."[4] For example, you might have the "`network drivers`" theme, the "`vm hacks`" theme, the "`utterly innocuous bug fixes`" theme, and the "`personal hacks`" theme. Each of these themes has its own tree, and for convenience, you may merge all of your theme trees into one local tree. Each of the theme trees will have as its parent the main Linux tree (see Figure 12).

Generally, the theme trees will not be merged directly with each other, but will only be pulled up into the main tree or down into your working tree. Each of these trees must have been originally created by cloning from the main tree (or a clone of the main tree, or a clone's clone, ad infinitum). The upstream maintainer

[4]Many people consider the requirement of "theme" trees to be one of BitKeeper's main drawbacks; see the section "BitKeeper Drawbacks" for information on upcoming BitKeeper features to correct this.



```
ChangeSet 1.105.7.43
  arch/ppc/kernel/head.S@1.1
  arch/ppc/kernel/smp.c@1.24
```

```
ChangeSet 1.105.7.43 2001/05/21 19:46:37 cort@cort.
  Some updates to cleanup
arch/ppc/kernel/smp.c 1.24 2001/05/21 19:46:37 cort@cort.
  Some updates to cleanup
```

```
arch/ppc/kernel/smp.c@1.23 (18:14:22 17May2001)
                nr = 4;

        if (nr > 1) {
                openpic_request_IPIs();
                for (i = 1; i < nr; ++i)
                        smp_hw_index[i] = i;
        }

        return nr;
}

static void
smp_gemini_kick_cpu(int nr)
{
        openpic_init_processor( 1<<i );
        openpic_init_processor( 0 );
}

static void
```

```
arch/ppc/kernel/smp.c@1.24 (19:46:37 21May2001)
                nr = 4;

        if (nr > 1) {
                openpic_request_IPIs();
                for (i = 1; i < nr; ++i)
                        smp_hw_index[i] = i;
        }

        return nr;
}

static void
smp_gemini_kick_cpu(int nr)
{
        openpic_init_processor( 1<<nr );
        openpic_init_processor( 0 );
}

static void
```

Figure 11: Change to fix a compilation error, viewed with BitKeeper's graphical diff viewer, running inside `bk csettool`.

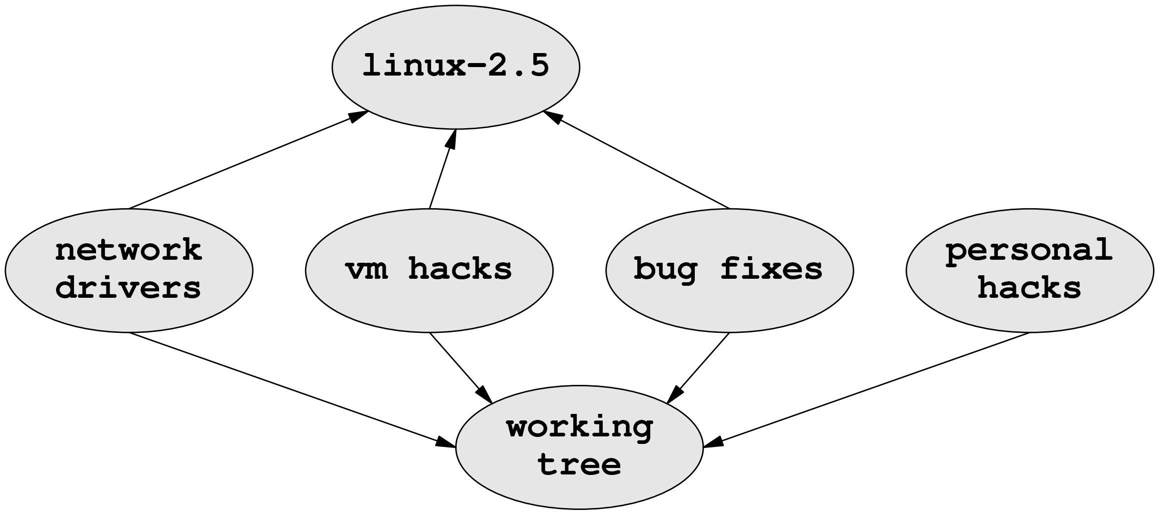


Figure 12: Graph of a typical developer's theme trees. Changes are pulled in the direction of the arrows.

can then pull your changes up with "`bk pull <location of your tree>`".

4.2 Comments

It is essential to write clear, descriptive check in comments. Not only will your comments be publicly archived for all eternity, but the upstream maintainer will want to read your comments before pulling the associated changes and, once pulled, use the comments to help decide whether or not to accept your changes. Good comments are also valuable as debugging tools, or as landmarks for navigating around the tree's history. If your first attempt at commenting your changes is inadequate, you can and should use "`bk comment -C<rev>`" to update and improve your comments later.[5] As an added bonus, very complete and detailed ChangeLogs are easily generated from your comments.

4.3 Internet accessible repository

The upstream maintainer will need access to your repository one way or another. Either give the maintainer ssh access to a machine with a clone of your repositories, or set up world readable repositories by running bkd, the BitKeeper daemon. The bkd can use HTTP ports and proxies, which allows access to your BitKeeper tree through most firewalls. BitMover provides free hosting for many BitKeeper repositories at `http://www.bkbits.net` and already hosts over a hundred personal Linux kernel repositories, including the main 2.4 and 2.5 repositories. For more information on hosting a repository, see `http://www.bitkeeper.com/Hosted.html`. As a last resort, you may also send your changes through email, using "`bk send`" and "`bk receive`".

[5]Comments changed this way don't propagate; if that changeset is pulled into another tree, and you change the comments afterwards, the comments in the other tree will not be updated, even after a push or pull.

4.4 Send a summary of your changes

No maintainer wants to blindly pull a set of changes. At the very least, you should send a summary of the changesets in your repository before asking the upstream maintainer to pull them. "`bk changes -L 2>&1 > ../pending`" will auto-generate a summary of all the pending changesets and their comments.

4.5 Keep your repository up to date

While your upstream maintainer can still pull your changes even if your tree isn't up to date with the upstream tree, you are offloading the work of resolving potential conflicts onto the upstream maintainer. Just like with `diff` and `patch`, your changesets are more likely to be accepted if they merge without conflicts into the main tree. It's good practice to run "`bk pull`" and merge any conflicts yourself before asking the upstream maintainer to pull your changes. Occasionally, the upstream maintainer prefers to do the merging, in which case you should allow the maintainer to pull and merge your changes. You can also perform the equivalent of a "`bk push`" without first doing a "`bk pull`" by using the command "`bk -u<maintainer's tree> send <maintainer's email address>`".

Following these recommendations will result in a smooth flow of patches to the main tree. As long as you comment well, logically separate your changes, and keep your repositories up to date, getting your submissions accepted will be easier than ever.

5 BitKeeper Drawbacks

Like all software, BitKeeper is not perfect. Some commonly requested features are the ability to subdivide repositories, to tell push and pull to send only a subset of new changes instead of all new changes, and to support true lines of development. The main complaint is that changes currently have to be pushed or pulled in an all-or-nothing manner, requiring the creation of theme repositories in order to be able to "cherry pick" subsets of changesets. The "`bk clone`" command has an option to create the new repository by hard linking the files in the new repository to the files in the original repository, which saves a lot of space if the two clones are on the same filesystem. This is only a partial solution to the problem of needing different theme trees for the Linux style of devlopment.

Two new features addressing these problems are currently being developed for BitKeeper. The first feature is nested repositories, which allows any repository or subrepository to contain multiple subrepositories which can be cloned and checked into separately. The other new feature is lines of development, or LODs. This feature allow a single repository to have more than one "tip" to the tree, allowing two or more independent lines of development to coexist in one repository. A developer will be able to cherry pick changes from one LOD and pull them into another LOD without also pulling all the changesets that came before that changeset.

While BitKeeper is both usable and useful in its current state, development on it is not standing still. Frequently requested features are written and added as quickly as possible. In the meantime, it is fairly easy to implement workarounds for unavailable features.

6 Conclusions

BitKeeper can dramatically improve the efficiency of Linux kernel developers working both alone and with other kernel developers. BitKeeper's tools aid in understanding code, debugging problems, and merging with other developers. Common kernel development tasks, such as updating your tree and sending patches, are trivial when using BitKeeper. Most importantly, kernel developers no longer spend hours on boring tasks which can and should be automated. One of the authors estimates that she saves between 2 and 5 hours a week (about 4-10% of total working hours) by using BitKeeper instead of `diff` and `patch`. Developers who integrate a lot of code from other developers would almost certainly save even more time than that. Using BitKeeper will benefit anyone who works with the Linux kernel source, and will benefit active kernel developers most of all.

Developers interested in using BitKeeper for the Linux kernel may find the BitKeeper Linux kernel development FAQ useful:

http://www.bitkeeper.com
/Documentation.FAQS.Linux.html

Linux Advanced Routing & Traffic Control

Bert Hubert
PowerDNS.COM, bv
bert@powerdns.com, http://ds9a.nl/

Abstract

Linux contains a wildly powerful system for shaping traffic and distributing it according to elaborate rules. This paper serves a dual purpose: to explain how to do this as a user and how to write a scheduler in the kernel.

1 Introduction

In the absence of infinite bandwidth there will always be a need to hand out capacity according to rules. Traditionally this has been a main reason to add non-IP technology to a network, like ATM or frame relay. Since IP is steadily taking over the world, Linux is well placed to play a role in enabling IP to take over traffic controlling functions from other technologies.

Traditionally, traffic control has been very difficult to configure and Linux is no different in this respect. In addition to this most of the important bits have not been documented.

About two years ago, the '2.4 Advanced Routing' HOWTO was started, well before the advent of Linux 2.4 in order to rectify this situation. Not hampered by any understanding a lot was written which was already helpful in configuring traffic control under linux.

By now a set of manpages has been written and the HOWTO properly explains most things.

2 Theory

As explained, traffic control is not an easy subject. Its difficulty can be compared to that of a postal service deciding to offer two kinds of service—'fast' and 'slow' where there previously was only 'reasonably fast.' Some kind of system must be devised to prioritize some kinds of traffic, but actively slow down others.

Now, the naive view of traffic shaping ("Hey, just slow the packets down!") corresponds to ordering all mail vehicles to lower speed—which clearly does not solve our problem.

We must be far smarter than that and not resort to wasteful solutions like slowing everything down.

The first thing to realise is that we can only realistically do complicated things with *outgoing* traffic. We have zero control over the rate at which people send us data. Again, this is like receiving (physical) mail. People send it to us and we can only decide not to read it—there is no way to make it come in any slower.

Furthermore even for outgoing traffic we can only treat packets that are *in* a computer we maintain. A prime example which many people encounter is trying to shape traffic going out to a cable modem which is connected via a 10 megabit ethernet. As this 10 megabit connection is lots faster than the cable modem, the Linux machine does not own the queue and hence powerless to prioritize traffic, unless further work is undertaken.

So—the theory is like this. Make sure that if there is a need to prioritize traffic, there is a queue which can be processed. Because there is normally only an outgoing queue configure your traffic control such that the data that needs to be prioritized is outgoing.

3 Verbiage

As with any complicated subject it is important to get the terminology right. I'm much indebted to Jamal who keeps pointing this out to me—his persistence is formidable and my stubbornness only just matches it.

- Queueing Discipline An algorithm that manages the queue of a device, either incoming (ingress) or outgoing (egress).
- Classless qdisc A qdisc with no configurable internal subdivisions.
- Classful qdisc A classful qdisc contains multiple classes. Each of these classes contains a further qdisc, which may again be classful, but need not be. According to the strict definition, pfifo_fast *is* classful, because it contains three bands which are, in fact, classes. However, from the user's configuration perspective, it is classless as the classes can't be touched with the tc tool.
- Classes A classful qdisc may have many classes, which each are internal to the qdisc. Each of these classes may contain a real qdisc.
- Classifier Each classful qdisc needs to determine to which class it needs to send a packet. This is done using the classifier.
- Filter Classification can be performed using filters. A filter contains a number of conditions which if matched, make the filter match.
- Scheduling A qdisc may, with the help of a classifier, decide that some packets need to go out earlier than others. This process is called Scheduling, and is performed for example by the pfifo_fast qdisc mentioned earlier. Scheduling is also called 'reordering,' but this is confusing.
- Shaping The process of delaying packets before they go out to make traffic confirm to a configured maximum rate. Shaping is performed on egress. Colloquially, dropping packets to slow traffic down is also often called Shaping.
- Policing Delaying or dropping packets in order to make traffic stay below a configured bandwidth. In Linux, policing can only drop a packet and not delay it—there is no 'ingress queue.'
- Work-Conserving A work-conserving qdisc always delivers a packet if one is available. In other words, it never delays a packet if the network adaptor is ready to send one (in the case of an egress qdisc).
- non-Work-Conserving Some queues, like for example the Token Bucket Filter, may need to hold on to a packet for a certain time in order to limit the bandwidth. This means that they sometimes refuse to give up a packet, even though they have one available.

Now that we have our terminology straight, let's see where all these things are. Figure 1 shows us.

4 Configuration example

This example is short but useful with actual physical phone modems:

Figure 1: This schematic is due to Jamal as well.

```
           Userspace programs
                   ^
                   |
   +---------------+-----------------------------------------+
   |               Y                                         |
   |    -------> IP Stack                                    |
   |   |              |                                      |
   |   |              Y                                      |
   |   |              Y                                      |
   |   ^              |                                      |
   |   |  / ----------> Forwarding ->|
   |   ^ /                           |                       |
   |   |/                            Y                       |
   |   |                             |                       |
   |   ^                             Y          /-qdisc1-\   |
   |   |                            Egress     /--qdisc2--\  |
--->->Ingress                       Classifier ---qdisc3---- | ->
   |   Qdisc                                   \__qdisc4__/  |
   |                                            \-qdiscN_/   |
   |                                                         |
   +---------------------------------------------------------+
```

```
# tc qdisc add dev ppp0 root \
        sfq perturb 10
```

Ok—what does this do? We've configured ppp0 to have a root queueing discipline called SFQ, which stands for Stochastic Fairness Queue. What this means is that the kernel now assigns each outgoing packet to a 'bucket' and dequeues a packet from each bucket in turn.

This is good for making sure that an outgoing upload does not interfere with, say, ssh traffic.

To inspect the configuration:

```
# tc -s -d qdisc ls dev ppp0
qdisc sfq 800c:  dev ppp0 quantum
   1514b limit 128p flows
   128/1024 perturb 10sec
   Sent 4812 bytes 62 pkts
(dropped 0, overlimits 0)
```

The number 800c: is the automatically assigned handle number, limit means that 128 packets can wait in this queue. There are 1024 hashbuckets available for accounting, of which 128 can be active at a time (no more packets fit in the queue!) Once every 10 seconds, the hashes are reconfigured.

5 Available Queueing Disciplines

The Linux kernel comes with many Queueing Disciplines or qdiscs. Some of there are non-functional or so underdocumented that they are not in use. There are also qdiscs that have not been merged yet.

- pfifo_fast This queue is, as the name says, First In, First Out, which means that no packet receives special treatment. At least, not quite. This queue has 3 so called 'bands.' Within each band, FIFO rules apply. However, as long as there are packets waiting in band 0, band 1 won't be processed. Same goes for band 1 and band 2.

The kernel honors the so called Type of Service flag of packets, and takes care to insert ‘minimum delay’ packets in band 0.

Do not confuse this classless simple qdisc with the classful PRIO one! Although they behave similarly, pfifo_fast is classless and you cannot add other qdiscs to it with the tc command.

- Token Bucket Filter

  The Token Bucket Filter (TBF) is a simple qdisc that only passes packets arriving at a rate which is not exceeding some administratively set rate, but with the possibility to allow short bursts in excess of this rate.

  TBF is very precise, network- and processor-friendly. It should be your first choice if you simply want to slow an interface down!

  The TBF implementation consists of a buffer (bucket), constantly filled by some virtual pieces of information called tokens, at a specific rate (token rate). The most important parameter of the bucket is its size, that is the number of tokens it can store.

  Each arriving token collects one incoming data packet from the data queue and is then deleted from the bucket.

- Stochastic Fairness Queueing

  Stochastic Fairness Queueing (SFQ) is a simple implementation of the fair queueing algorithms family. It’s less accurate than others, but it also requires less calculations while being almost perfectly fair.

  The key word in SFQ is conversation (or flow), which mostly corresponds to a TCP session or a UDP stream. Traffic is divided into a pretty large number of FIFO queues, one for each conversation. Traffic is then sent in a round robin fashion, giving each session the chance to send data in turn.

  This leads to very fair behaviour and disallows any single conversation from drowning out the rest. SFQ is called “Stochastic” because it doesn’t really allocate a queue for each session, it has an algorithm which divides traffic over a limited number of queues using a hashing algorithm.

  Because of the hash, multiple sessions might end up in the same bucket, which would halve each session’s chance of sending a packet, thus halving the effective speed available. To prevent this situation from becoming noticeable, SFQ changes its hashing algorithm quite often so that any two colliding sessions will only do so for a small number of seconds.

- Prio The PRIO qdisc doesn’t actually shape, it only subdivides traffic based on how you configured your filters. You can consider the PRIO qdisc a kind of pfifo_fast on stereoids, whereby each band is a separate class instead of a simple FIFO.

  When a packet is enqueued to the PRIO qdisc, a class is chosen based on the filter commands you gave. By default, three classes are created. These classes by default contain pure FIFO qdiscs with no internal structure, but you can replace these by any qdisc you have available.

  Whenever a packet needs to be dequeued, class :1 is tried first. Higher classes are only used if lower bands all did not give up a packet.

  This qdisc is very useful in case you want to prioritize certain kinds of traffic without using only TOS-flags but using all the power of the tc filters. It can also contain more all qdiscs, whereas pfifo_fast is limited to simple fifo qdiscs.

Because it doesn't actually shape, the same warning as for SFQ holds: either use it only if your physical link is really full or wrap it inside a classful qdisc that does shape. The last holds for almost all cable-modems and DSL devices.

In formal words, the PRIO qdisc is a Work-Conserving scheduler.

- CBQ CBQ is the most complex qdisc available, the most hyped, the least understood, and probably the trickiest one to get right. This is not because the authors are evil or incompetent, far from it, it's just that the CBQ algorithm isn't all that precise and doesn't really match the way Linux works.

  Besides being classful, CBQ is also a shaper and it is in that aspect that it really doesn't work very well. It should work like this. If you try to shape a 10mbit/s connection to 1mbit/s, the link should be idle 90% of the time. If it isn't, we need to throttle so that it IS idle 90% of the time.

  This is pretty hard to measure, so CBQ instead derives the idle time from the number of microseconds that elapse between requests from the hardware layer for more data. Combined, this can be used to approximate how full or empty the link is.

  This is rather circumspect and doesn't always arrive at proper results. For example, what if the actual link speed of an interface that is not really able to transmit the full 100mbit/s of data, perhaps because of a badly implemented driver? A PCMCIA network card will also never achieve 100mbit/s because of the way the bus is designed—again, how do we calculate the idle time?

  It gets even worse if we consider not-quite-real network devices like PPP over Ethernet or PPTP over TCP/IP. The effective bandwidth in that case is probably determined by the efficiency of pipes to userspace—which is huge.

  People who have done measurements discover that CBQ is not always very accurate and sometimes completely misses the mark.

  In many circumstances however it works well. With the documentation provided here, you should be able to configure it to work well in most cases.

- Hierarchical Token Bucket (outside of the kernel) Martin Devera (<devik>) rightly realised that CBQ is complex and does not seem optimized for many typical situations. His Hierarchial approach is well suited for setups where you have a fixed amount of bandwidth which you want to divide for different purposes, giving each purpose a guaranteed bandwidth, with the possibility of specifying how much bandwidth can be borrowed.

  HTB works just like CBQ but does not resort to idle time calculations to shape. Instead, it is a classful Token Bucket Filter—hence the name. It has only a few parameters, which are well documented on his site.

  As your HTB configuration gets more complex, your configuration scales well. With CBQ it is already complex even in simple cases! HTB is not yet a part of the standard kernel, but it should soon be!

  If you are in a position to patch your kernel, by all means consider HTB.

- bfifo/pfifo These classless queues are even simpler than pfifo_fast in that they lack the internal bands—all traffic is really equal. They have one important benefit though, they have some statistics. So even if you don't need shaping or prioritizing, you can use this qdisc to determine the backlog on your interface.

pfifo has a length measured in packets, bfifo in bytes.

- Clark-Shenker-Zhang algorithm (CSZ) This is so theoretical that not even Alexey (the main CBQ author) claims to understand it. From his source:

  "David D. Clark, Scott Shenker and Lixia Zhang Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism."

  As I understand it, the main idea is to create WFQ flows for each guaranteed service and to allocate the rest of bandwith to dummy flow-0. Flow-0 comprises the predictive services and the best effort traffic; it is handled by a priority scheduler with the highest priority band allocated for predictive services, and the rest—to the best effort packets.

- DSMARK Dsmark is a queueing discipline that offers the capabilities needed in Differentiated Services (also called DiffServ or, simply, DS). DiffServ is one of two actual QoS architectures (the other one is called Integrated Services) that is based on a value carried by packets in the DS field of the IP header.

  One of the first solutions in IP designed to offer some QoS level was the Type of Service field (TOS byte) in IP header. By changing that value, we could choose a high/low level of throughput, delay or reliability. But this didn't provide sufficient flexibility to the needs of new services (such as real-time applications, interactive applications and others). After this, new architectures appeared. One of these was DiffServ which kept TOS bits and renamed DS field.

- On the ingress All qdiscs discussed so far are egress qdiscs. Each interface however can also have an ingress qdisc which is not used to send packets out to the network adaptor. Instead, it allows you to apply tc filters to packets coming in over the interface, regardless of whether they have a local destination or are to be forwarded.

  As the tc filters contain a full Token Bucket Filter implementation, and are also able to match on the kernel flow estimator, there is a lot of functionality available. This effectively allows you to police incoming traffic, before it even enters the IP stack.

- Random Early Detection (RED) The normal behaviour of router queues on the Internet is called tail-drop. Tail-drop works by queueing up to a certain amount, then dropping all traffic that 'spills over.' This is very unfair, and also leads to retransmit synchronisation. When retransmit synchronisation occurs, the sudden burst of drops from a router that has reached its fill will cause a delayed burst of retransmits, which will over fill the congested router again.

  In order to cope with transient congestion on links, backbone routers will often implement large queues. Unfortunately, while these queues are good for throughput, they can substantially increase latency and cause TCP connections to behave very bursty during congestion.

  These issues with tail-drop are becoming increasingly troublesome on the Internet because the use of network unfriendly applications is increasing. The Linux kernel offers us RED, short for Random Early Detect, also called Random Early Drop, as that is how it works.

  RED isn't a cure-all for this, applications which inappropriately fail to implement exponential backoff still get an unfair share of the bandwidth, however, with RED they do not cause as much harm to the throughput and latency of other connections.

RED statistically drops packets from flows before it reaches its hard limit. This causes a congested backbone link to slow more gracefully, and prevents retransmit synchronisation. This also helps TCP find its 'fair' speed faster by allowing some packets to get dropped sooner keeping queue sizes low and latency under control. The probability of a packet being dropped from a particular connection is proportional to its bandwidth usage rather than the number of packets it transmits.

RED is a good queue for backbones, where you can't afford the complexity of per-session state tracking needed by fairness queueing.

- Generic Random Early Detection

  Not a lot is known about GRED. It looks like GRED with several internal queues, whereby the internal queue is chosen based on the Diffserv tcindex field. According to a slide found here, it contains the capabilities of Cisco's 'Distributed Weighted RED,' as well as Dave Clark's RIO.

  Each virtual queue can have its own Drop Parameters specified.

  "Ask Jamal"

- Weighted Round Robin (WRR) This qdisc is not included in the standard kernels but can be downloaded. Currently the qdisc is only tested with Linux 2.2 kernels, but it will probably work with 2.4/2.5 kernels too.

  The WRR qdisc distributes bandwidth between its classes using the weighted round robin scheme. That is, like the CBQ qdisc it contains classes into which arbitrary qdiscs can be plugged. All classes which have sufficient demand will get bandwidth proportional to the weights associated with the classes. The weights can be set manually using the tc program. But they can also be made automatically decreasing for classes transferring much data.

  The qdisc can be very useful at sites such as dorms where a lot of unrelated individuals share an Internet connection. A set of scripts setting up a relevant behavior for such a site is a central part of the WRR distribution.

6 Kernel API

Only classless qdiscs are covered here. Writing a classful qdisc is an advanced topic.

To the kernel, a qdisc looks like Figure 2.

Packets are enqueued by the kernel and immediately after as many packets as possible are bursted out of the qdisc to the hardware.

- next Pointer in the linked list – leave alone
- cl_ops NULL for a classless qdisc
- id Name of this interface
- priv_size Size of the private data of this backend
- enqueue Called by the kernel to queue a new packet for transmission
- dequeue Called to get a packet for the hardware to send out now
- requeue Called by the kernel to put back a packet at the head of the queue. The next call to dequeue will most likely return it.
- drop Called by the kernel to indicate that a packet should be dropped & freed from the queue, without returning it

Figure 2: The `Qdisc_ops` structure

```
struct Qdisc_ops
{
    struct Qdisc_ops        *next;
    struct Qdisc_class_ops  *cl_ops;
    char                    id[IFNAMSIZ];
    int                     priv_size;

    int                     (*enqueue)(struct sk_buff *, struct Qdisc *);
    struct sk_buff *        (*dequeue)(struct Qdisc *);
    int                     (*requeue)(struct sk_buff *, struct Qdisc *);
    int                     (*drop)(struct Qdisc *);

    int                     (*init)(struct Qdisc *, struct rtattr *arg);
    void                    (*reset)(struct Qdisc *);
    void                    (*destroy)(struct Qdisc *);
    int                     (*change)(struct Qdisc *, struct rtattr *arg);

    int                     (*dump)(struct Qdisc *, struct sk_buff *);
};
```

- init Called before use
- reset Should purge the queue and reset settings
- destroy Cleanup
- change Accept reconfigured settings over the Netlink
- dump Report statistics over the Netlink

6.1 How the kernel interacts with the qdisc

enqueue is called from dev_queue_xmit() in net/core/dev.c:

```
/* Grab device queue */
spin_lock_bh(&dev->queue_lock);
q = dev->qdisc;
if (q->enqueue) {
    int ret = q->enqueue(skb, q);

    qdisc_run(dev);

    spin_unlock_bh(&dev->queue_lock);
    return ret == NET_XMIT_BYPASS ?
                  NET_XMIT_SUCCESS :
                  ret;
}
```

qdisc_run(), which lives in include/net/pkt_sched.h, is then immediately called to get the packets out on the wire (or ether, for that matter):

```
static inline void
qdisc_run(struct net_device *dev)
{
  while (!netif_queue_stopped(dev)
    && qdisc_restart(dev)<0)
      /* NOTHING */;
}
```

Getting nearer to the wire, qdisc_restart() is in net/sched/sch_generic.c:

```
int
qdisc_restart(struct net_device *dev)
{
  struct Qdisc *q = dev->qdisc;
  struct sk_buff *skb;

  /* Dequeue packet */
  if ((skb = q->dequeue(q)) != NULL)
  {
    if (spin_trylock(&dev->xmit_lock))
    {
```

```
/* Remember that the driver
   is grabbed by us. */
dev->xmit_lock_owner =
  smp_processor_id();

/* And release queue */
spin_unlock(&dev->queue_lock);

if (!netif_queue_stopped(dev))
{
  if (netdev_nit)
    dev_queue_xmit_nit(skb,
      dev);

  if (dev->hard_start_xmit(skb,
      dev) == 0) {
    dev->xmit_lock_owner = -1;
    spin_unlock(&dev->xmit_lock);

    spin_lock(&dev->queue_lock);
    return -1;
  }
}
/* code for when the queue
   IS stopped */
...
```

hard_start_xmit(skb,dev) actually moves (or shakes) the electrons.

6.2 Minimal qdisc

The kernel actually contains a 'noop' qdisc which sees some use in efficiently dropping packets on the floor. Or as Alexey says it:

```
/* "NOOP" scheduler:
the best scheduler,
recommended for all
interfaces under all
circumstances.  It is
difficult to invent
anything faster or
cheaper.  */
```

However, this is too minimal to serve as an example.

We'll look at the pfifo qdisc which performs simple taildrop after n packets. Somewhat simplified and commented source of pfifo_enqueue:

```
int pfifo_enqueue(struct sk_buff *skb,
    struct Qdisc *sch)
{
  /* get our private data */
  struct fifo_sched_data *q =
    (struct
     fifo_sched_data *)sch->data;

  /* is there room for
     another packet */
  if (sch->q.qlen <= q->limit) {
    /* add it at the tail
       end of our q */
    __skb_queue_tail(&sch->q, skb);

    /* accounting - note that this is
       not in the private part */
    sch->stats.bytes += skb->len;
    sch->stats.packets++;
    /* there might be accounting in
       q-> too, but not for pfifo */
    return 0;
  }
  /* if we get here, there is no room
     and we drop & cleanup */
  sch->stats.drops++;

  kfree_skb(skb);
  /* sorry, no room */
  return NET_XMIT_DROP;
}
```

The dequeue function is simpler:

```
struct sk_buff
   *pfifo_dequeue(struct Qdisc* sch)
{
   return __skb_dequeue(&sch->q);
}
```

The pfifo queue cannot be configured—it takes its queuelength from the adapter's txqueuelen.

References

[LARTC] *Linux Advanced Routing & Traffic Control HOWTO* bert hubert. `http://lartc.org/` (2002)

Maintaining the Correctness of the Linux Security Modules Framework

Trent Jaeger   *Xiaolan Zhang*   *Antony Edwards*
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
Email: {jaegert,cxzhang}@us.ibm.com

Abstract

In this paper, we present an approach, supported by software tools, for maintaining the correctness of the Linux Security Modules (LSM) framework (the LSM community is aiming for inclusion in Linux 2.5). The LSM framework consists of a set of function call hooks placed at locations in the Linux kernel that enable greater control of user-level processes' use of kernel functionality, such as is necessary to enforce mandatory access control. However, the placement of LSM hooks within the kernel means that kernel modifications may inadvertently introduce security holes. Fundamentally, our approach consists of complementary static and runtime analysis; runtime analysis determines the authorization requirements and static analysis verifies these requirements across the entire kernel source. Initially, the focus has been on finding and fixing LSM errors, but now we examine how such an approach may be used by kernel development community to maintain the correctness of the LSM framework. We find that much of the verification process can be automated, regression testing across kernel versions can be made resilient to several types of changes, such as source line numbers, but reduction of false positives remains a key issue.

1 Introduction

The Linux Security Modules (LSM) project aims to provide a generic framework from which a wide variety of authorization mechanisms and policies can be enforced. Such a framework would enable developers to implement authorization modules of their choosing for the Linux kernel. System administrators can then select the module that best enforces their system's security policy. For example, modules that implement mandatory access control (MAC) policies to enable containment of compromised system services are under development.

The LSM framework is a set of authorization hooks (i.e., generic function pointers) inserted into the Linux kernel. These hooks define the types of authorizations that a module can enforce and their locations. Placing the hooks in the kernel itself rather than at the system call boundary has security and performance advantages. First, placing hooks where the operations are implemented ensures that the authorized objects are the only ones used. For example, system call interposition is susceptible to time-of-check-to-time-of-use (TOCTTOU) attacks [2], where another object is swapped for the authorized object after authorization, because the kernel does not necessarily use the object authorized by interposition. Second, since the authorizations are at the point of the operation, there is no need to redundantly transform system call arguments to kernel objects.

While placing the authorization hooks in the kernel can improve security, it is more difficult to determine whether the hooks mediate and authorize all controlled operations. The system call interface is a nice mediation point because all the kernel's controlled operations (i.e., operations that access security-sensitive data) *must* eventually go through this interface. Inside the kernel, there is no obvious analogue for the system call interface. Any kernel function can contain accesses to one or more security-sensitive data structures. Thus, any mediation interface is at a lower-level of abstraction (e.g., inode member access). Also, it is necessary to link these operations with their access control policy (e.g., write data) to ensure that the correct authorizations are made for each controlled operation. If there is a mismatch between the policy enforced and the controlled operations that are executed under that policy, unauthorized operations can be executed. We believe that manual verification of the correct authorization of a low-level mediation interface is impractical.

We have examined both static and runtime analysis techniques for verifying LSM authorization hook placement [6, 20]. Our static analysis approach identifies kernel variables of key data types (e.g., inodes, tasks, sockets, etc.) that are accessed prior to authorization. The advantage of static analysis is that its complete coverage of execution paths (both data and control) enables it to find potential errors more easily. Many successes with static analysis have been reported recently [7, 11, 16]. The effectiveness of static analysis is limited by the manual effort required for annotation and the number of false positives that are generated [1]. Also, some tasks are very difficult for static analysis. However, runtime analysis requires benchmarks that provide sufficient coverage and also creates false positives that must be managed. Thus far, our experience has been that runtime analysis provides a useful complement for static analysis, so both types of analyses need to be performed to obtain effective verification.

While our initial results have been positive [2], ultimately, we believe that it is necessary that such analysis become part of the kernel development process to really maintain the effectiveness of the LSM framework. As the Linux kernel is modified, the LSM authorization hooks may become misplaced. That is, some security-sensitive operations that were previously executed only after authorization may now become accessible without proper authorization. Since the subtleties of authorization may be non-trivial, the kernel developers need a tool that enables them to verify that the authorization hooks protect the system as they did before or identify the cases that need examination. Further, kernel developers need a way of communicating changes that need to be examined by the LSM community.

In this paper, we outline the analysis capabilities of our static and runtime tools and describe how they are used together to perform LSM verification. We do not provide a detailed discussion of the analysis tools, so interested readers are directed elsewhere for that information [6, 20]. We would also like to make such tools available and practical for the kernel development community, so we examine how effectively the analysis steps can be automated and what issues the users of the analy-

[1]Static analysis is overly conservative because some impossible paths are considered which can lead to some false positives.

[2]Five LSM authorization hooks have been added or revised due to the results of our analysis tools.

sis tools must resolve in order to complete the analysis. We find that much of the verification process can be automated, regression testing across kernel versions can be made resilient to minor changes, such as source line numbers, but reduction of false positives remains a key issue. While the analysis tools are not yet available as open source, we are working to obtain such approval.

The remainder of the paper is structured as follows. In Section 2, we review the goals and status of the LSM project. In Section 3, we define the general hook placement problem. In Section 4, we review the static and runtime analysis verification approaches. In Section 5, we outline how LSM verification experts use the static and runtime analysis tools in a complementary fashion to perform a complete LSM verification. In Section 6, we examine how the analysis tools can be made practical for use by the kernel development community. In Section 7, we conclude and describe future work.

2 Linux Security Modules

The Linux Security Modules (LSM) framework is being developed to address insufficiencies in traditional UNIX security. Historically, UNIX operating systems provide a single authorization mechanism and policy model for controlling file system access. This approach has been found to be lacking for a variety of reasons, and these inadequacies have been exacerbated by emerging technologies. First, the UNIX policy model lacks the expressive power necessary for some security requirements. UNIX file mode bits enable control of file accesses based on three types of relationships that the subject may have with the file: file owner, file group owner, and others. Some reasonable access control combinations cannot be expressed using this approach, so extension have been created (e.g., access control lists (ACL)). Second, the UNIX access control model provides discretionary access control (DAC) whereby the owner of the objects controls the distribution of access. Thus, users can accidentally give away rights that they did not intend, and the all-powerful user *root*, as which a wide variety of diverse programs run, can change access control policy in the system arbitrarily. Third, with the advent of new programming paradigms, such as mobile code, the UNIX assumption that every one of the users' processes should always have all of the users' rights became flawed [3], and it was found that the UNIX access control model was too limited to enable the necessary level of flexibility [10, 1, 9]. Fourth, controlling access to a variety of other objects besides files was also found to be necessary, and, in some cases, restricting the relationships that objects may enter is necessary [17]. For example, the ability to mount one file system on another is a controlled operation on the establishment of that relationship between the two file systems.

Initially, the authorization mechanisms proposed to address these limitations were inserted at the user-system boundary (e.g., by wrapping system calls [1] or callbacks [9]). By not integrating the authorization mechanisms within the kernel, the authorization mechanism lacks the kernel state at the time that the operation is performed. Attacks have been found that can take advantage of the interval between the time of the authorization and the time at which the operation is invoked [2]. Further, the performance of the system is degraded because the kernel state must be computed twice if the authorization mechanism is placed at the system call interface. Recent research work on improving the UNIX authorization mechanism in Linux has focused on inserting hooks to the authorization mechanism in the kernel directly [4, 13, 14, 15, 18]. However, the variety of authorization hook placements and styles resulted in ad hoc modifications to the Linux kernel.

Another major advancement has been the separation between the authorization mechanism and the policy model used. The work on DTOS and Flask security architectures demonstrated how the authorization policy server can be separated from the authorization mechanism [12, 17]. Thus, a variety of access control policies can be supported. In particular, a variety of mandatory access control (MAC) policies can be explored. An advantage of MAC policies is that provable containment of overt process actions is possible, so protection of the TCB and key applications can be implemented. Various flavors of MAC policy models have been examined, but no one approach has been shown to be superior. The design of effective policy models and policies themselves remains an open research issue.

The LSM project includes several of the parties working on independent Linux kernel authorization mechanisms, in particular Security-Enhanced Linux (SELinux) and Immunix SubDomain, to create a generic framework for calling authorization modules from within the kernel. Motivation to unite these mechanisms came when Linus Torvalds outlined his goals for such a framework [19]. Linus stated that he wants authorization to be implemented by a module accessible via generic hooks. The hope that an acceptable authorization framework would be integrated with the mainline Linux kernel has resulted in a comprehensive LSM implementation.

As of Linux 2.4.16, LSM consists of 216 authorization hooks inserted in the kernel that can call 153 distinct authorization functions defined by the authorization modules (i.e., loadable kernel modules). The authorization hooks enable authorization of a wide variety of operations, including operations on files, inodes, sockets, IPC messages, IPC message queues, semaphores, tasks, modules, skbuffs, devices, and various global kernel variables. Authorization modules for SELinux, SubDomain, and OpenWALL have been built for LSM, so LSM is capable of enforcing MAC policies already.

3 General Hook Placement Problems

3.1 Concepts

We identify the following key concepts in the construction of an authorization framework:

- **Authorization Hooks:** These are the authorization checks in the system (e.g., the LSM-patched Linux kernel).
- **Policy Operations:** These are the operations for which authorization policy is defined in the authorization hooks. Because we would like to identify code that is representative of the policy operation, they are practically defined as the first controlled operation (see below) requiring this policy.
- **Security-sensitive Operations:** These are the operations that impact the security of the system.
- **Controlled Operations:** A subset of security-sensitive operations that mediate access to all other security-sensitive operations. These operations define a *mediation interface*.

The definition of these concepts is made clear by a comparison between system call mediation and the in-kernel mediation used by LSM shown in Figure 1. When authorization hooks are placed at the system call interface, the policy operations (e.g., the conceptual operation

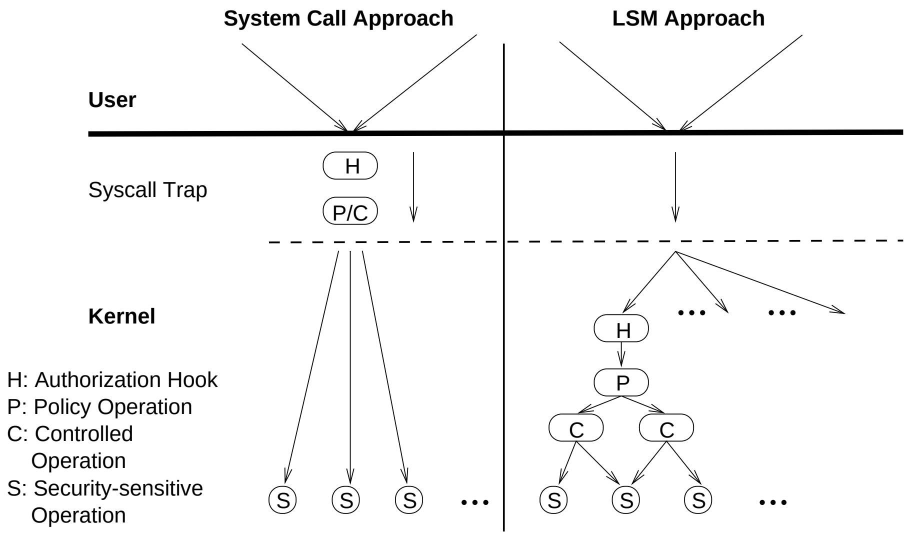


Figure 1: Comparison of concepts between system call interposition framework and LSM.

`write`) and controlled operations (e.g., where mediation of all file opens for write access occur at the system call `sys_open` with the access flag `WRONLY`) are effectively the same. This is because policy is specified at the system call interface, and the system call interface also provides complete mediation. The security-sensitive operations in both cases are the data accesses made to security-relevant kernel data, such as files, inodes, mappings, and pages.

When authorization hooks are inserted in the kernel, the level of complete mediation is the kernel source code, so the policy operations and controlled operations are no longer necessarily the same. For example, rather than verifying file open for write access at the system call interface, the LSM authorizations for directory (exec), link (follow link), and ultimately, the file open are performed at the time these operations are to be done. This eliminates susceptibility to TOCTTOU attacks [2] and redundant processing. The kernel source is complex, however, so it is no longer clear that all security-sensitive operations are actually authorized properly before they are run.

Given the breadth and variety of security-sensitive operations, we would like to identify a higher-level interface for verifying their proper LSM authorization. This interface must mediate all access from the authorization hooks to the security-sensitive operation. This interface is referred to as the *mediation interface* and is defined by a set of controlled operations.

3.2 Relationships to Verify

Figure 2 shows the relationships between the concepts.

1. **Identify Controlled Operations:** Find the set of operations that define a mediation interface through which all security-sensitive operations are accessed.

2. **Determine Authorization Requirements:** For each controlled operation, identify the policy operations that must be authorized by the LSM hooks.

3. **Verify Complete Authorization:** For each controlled operation, verify that the policy operations (i.e., authorization

requirements) are authorized by LSM hooks.

4. **Verify Hook Placement Clarity:** Policy operations should be easily identifiable from their authorization hooks. Otherwise, even trivial changes to the source may render the hook inoperable.

The basic idea is that we identify the controlled operations and their authorization requirements, then we verify that the authorization hooks mediate those controlled operations properly. This verifies, that the LSM hook placement is correct with respect to this set of controlled operations and authorization requirements. When the mediation interface is shown to be correct, it verifies LSM hook placement with respect to all security-sensitive operations. These tasks are complex, so it is obvious that automated tools are necessary.

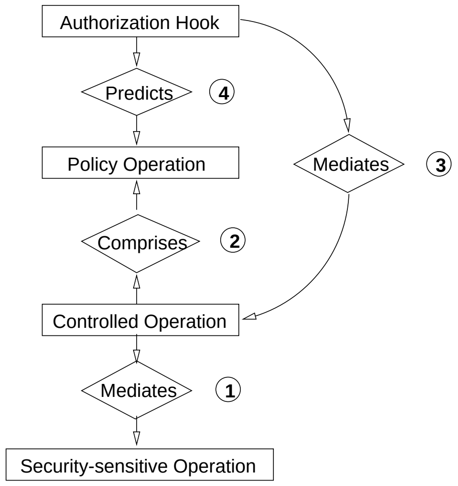


Figure 2: Relationships between the authorization concepts. The verification problems are to: (1) identify controlled operations; (2) determine authorization requirements; (3) verify complete authorization; and (4) verify hook placement clarity.

In addition, we found that additional automated support is necessary to identify the controlled operations and their authorization requirements. First, manual identification of the controlled operations is a tedious task. We must develop an approach by which controlled operations can be selected from the set of security-sensitive operations. Once this approach has been determined automated techniques are needed to extracted these operations from the kernel source. Second, because the controlled operations are at a lower level than the policy operations, we need to determine the policy operations (i.e., authorization requirements) for a controlled operation. Since we expect a large number of controlled operations, it is necessary to develop an approach to simplify the means for identifying their authorization requirements.

Lastly, to ensure maintainability of the authorization hooks we can verify that the policy operations can be easily determined from the authorization hook locations. This work is has been done, but in interest of focus it is outside the scope of this paper. This is work is presented elsewhere [6].

3.3 Related Work

We are not aware of any tools that perform any of the tasks outlined above. While static analysis has had some promising results lately [7, 11, 16], the problems upon which they have been applied have been different and narrower in scope (e.g., buffer overflow detection). We believe that static analysis tools will eventually provide some important improvements in the verifications described above, but some analyses will be easier to do with runtime tools (e.g., due to reduced specification for comprehensive tests).

4 Solution Background

In this section, we review the approaches we devised for using static and runtime analysis to verify the placement of LSM authorization hooks.

4.1 CQUAL Static Analysis

We use the CQUAL type-based static analysis tool as the basis for our static analysis [8]. CQUAL supports user-defined *type qualifiers* that are used in the same way as the standard C type qualifiers such as `const`. We define two type qualifiers, `checked` and `unchecked`. The idea is that a variable with a `unchecked` qualifier cannot be used when a variable with a `checked` qualifier is expected. This simulates the need to authorize variables before they are used in controlled operations.

The following code segment demonstrates the type of violation we want to detect. Function `func_a` expects a `checked` file pointer as its parameter, but the parameter passed is of type `unchecked` file pointer.

```
void func_a(struct file
             *checked filp);

void func_b( void )
{
        struct file * unchecked filp;
        ...
        func_a(filp);
        ...
}
```

As input to CQUAL, we define type relations between the `checked` and `unchecked` type qualifiers that represent the requirement that a `checked` type cannot be used when an `unchecked` is expected. Using its inference rules, CQUAL performs *qualifier inference* to detect violations against these type relations. These violations are called *type errors*. CQUAL reports both the variables involved in the type errors and the shortest paths to type error creation for these variables. For a more detailed description of CQUAL, please refer to the original paper on CQUAL [8].

In order to do this analysis, CQUAL requires that the target source be annotated with the type qualifiers. This is an arduous and error-prone task for a program like the Linux kernel, so we use GCC analysis to automate the annotation process. There are three GCC analyses we perform to prepare the source code for CQUAL processing.

1. All controlled object must be initialized to `unchecked`.
2. All function parameters that are used in a controlled operation must be marked as `checked`.
3. Authorizations must upgrade the authorized object's qualified type to `checked`.

In order to ensure that static analysis is sound (i.e., no type errors are missed by the analysis), we perform some additional GCC analyses. For example, we verify no reassignments of variables and check for intra-procedural type errors. These analyses are sometimes primitive, but they have limited the amount of manual work required sufficiently. We are working with the CQUAL community and others to improve the effectiveness of static analysis for this purpose. For a more detailed description of our static analysis, see our paper [20].

With our static analysis, we have identified some LSM vulnerabilities in Linux 2.4.9, but since the runtime analysis tool was done first we have found only one new, exploitable vulnerability. It has since been fixed in later versions of LSM [5]. Figure 3 shows the vulnerability.

The code fragment demonstrates a time-of-check-to-time-of-use [2] (TOCTTOU) vulnerability. In this case, the `filp` variable is authorized in `sys_fcntl`. However, a new version of `filp` is extracted from the file descriptor and used in the function `fcntl_getlk`.

Since a user process can modify its mapping between its file descriptors and the files they reference this error is exploitable.

4.2 Vali Runtime Analysis

We have developed a tool, called Vali [3], for collecting key kernel runtime events (Vali runtime) and analyzing this runtime data (Vali analysis) to determine whether LSM authorization hooks are correctly placed [6]. The key insight of the Vali analysis is that most of the LSM authorization hooks are correctly placed, so it is anomalies in the authorization results that enables us to identify errors. Using this approach, we have found 5 significant anomalies in LSM authorization hook placement, 4 of which have been identified as bugs and fixed.

Vali consists of kernel instrumentation tools, kernel data collection modules, and data analysis tools. The kernel instrumentation tools build a Linux kernel for which kernel events (e.g., system calls and interrupts), function entry/exits, LSM authorizations, and controlled operations can be logged by the data collection modules. We use the same kind of GCC analysis as we did for the static analysis to find controlled operations in the kernel. Other events are easily instrumented through GCC instrumentation (functions), breakpoints on entry addresses (kernel events), and the LSM authorization hooks themselves (authorizations).

Loadable kernel modules for each type of instrumentation collect these events. The main problem with data collection is not the performance overhead, but the data collection bandwidth. The performance overhead of instrumentation is only about 10%, and since the analysis kernel is not a production kernel this is quite acceptable. However, the rate at which data is generated can exceed the disk through-

[3]Vali is the Norse God of Justice and the first four letters in "Validate."

```
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
               unsigned int cmd,
               unsigned long arg)
{
  struct file * filp;
  ...
  filp = fget(fd);
  ...
  err = security_ops->file_ops
        ->fcntl(filp, cmd, arg);
  ...
  err = do_fcntl(fd, cmd, arg,
                 filp);
  ...
}
static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETLK:
      err = fcntl_setlk(fd, ...);
    ...
  }
  ...
}
/* from fs/locks.c */
fcntl_getlk(fd, ...) {
  struct file * filp;
  ...
  filp = fget(fd);
  /* operate on filp */
  ...
}
```

Figure 3: Code path from Linux 2.4.9 containing an exploitable type error.

```
# open for read access
1 = (+,id_type,CONTEXT)
(+,di_cfm_eax,sys_open)
(+,co_ecx,RDONLY)
2 (D,1) = (+,ALL,0,0)
# open for read-write
1 = (+,id_type,CONTEXT)
(+,di_cfm_eax,sys_open)
(+,co_ecx,RDWR)
2 (D,1) = (+,ALL,0,0)
```

Figure 4: Filtering rules for open system call (sys_open) with read and read-write access flags. The (D,1) means that the rule should use only the records that have matched this rule number in the second argument. There is also a negation counterpart (N,x) where the specified records are excluded.

put rate, so we enable event filtering. Currently, this is simply collecting 1 out of n events where n can be tuned. Ultimately, we would like to be able to control the types of events collected to ensure that rarer events are not missed.

The logged data identifies the objects used in controlled operations and the authorizations made upon those objects. While different object instances are used in different system call instances, objects referenced by the same variable (i.e., used in the same controlled operations) should normally have the same authorizations. This is not entirely true as some system calls (e.g., `open`, `ioctl`, etc.) may imply different authorizations based on the flags that are sent. Therefore, we have defined a simple filtering language to identify the kernel events that should have the same authorizations for all objects (see Figure 4 for examples). Per filter, all objects should have the same authorizations. Therefore, we can identify anomalous cases that do not have the expected authorizations, and these cases are often errors. Because the filters enable focusing on a small set of operations, we have had more success finding problems using the runtime analysis than the static analysis. We have found errors ranging from missing authorizations for an obscure system call `getgroups16` to a missing authorization for resetting the fowner of a file using one of the flag variants of `fcntl` (see Figure 5).

The runtime analysis also does something that the static analysis does not: it identifies the expected authorizations for an object in a system call. Cases that are consistent identify a belief in the set of authorizations that are required on an object. These authorization requirements can be used as input to the static analysis tool which can then be used to verify the correct authorizations, not just the existence of an authorization. While most of the controlled operations require just one authorization, the error in the `fcntl` case above was in the lack of a second authorization to check the permission to set the owner.

5 LSM Community Analysis Approach

As might be gathered by the previous section, we find that the two analysis approaches are quite complementary. In this section, we outline the approach intended for use by LSM verification experts to verify LSM authorization hook placement using our analysis tools. We discuss how the kernel development community might use the analysis tools to maintain LSM correctness in the following section.

Verification of LSM authorization hook placement involves the following steps:

1. **Checked/Unchecked Static Analysis**: We first apply our static analysis approach to find variables for which no authoriza-

```
/* from fs/fcntl.c */
static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETOWN:
     /* set fowner is authorized
        for filp */
     err = LSM->file_ops->set_fowner(
           filp);
     ...
     filp->f_owner.pid = arg;
     ...
    case F_SETLEASE:
      err = fcntl_setlease(fd, filp,
            arg);
    ...
  }
  ...
}
/* from fs/locks.c */
fcntl_setlease(unsigned int fd,
       struct file *filp,
       long arg) {
  struct file_lock *my_before;
  ...
  if (my_before != NULL) {
    error = lease_modify(my_before,
                   arg, fd, filp);
  ...
}
lease_modify(struct file_lock
             **before,
             int arg, int fd,
             struct file *filp) {
  ...
  if (arg == F_UNLCK) {
     /* ERROR: could set active
        fowner to 0 */
     filp->f_owner.pid = 0;
     ...
  }
...
}
```

Figure 5: Code path from Linux 2.4.16 containing an exploitable error for the system call `fcntl(fd, F_SETLEASE, F_UNLCK)` whereby the `pid` of an active lock can be set to 0.

tions are performed. Since there are a significant number ($\tilde{3}0$ for the VFS layer alone for $\tilde{2}50$ controlled variables) of type errors, these must be further classified to eliminate all those that are known not to be a real error.

2. **Runtime Analysis for Authorization Requirements**: Using benchmarks that cover as much of the kernel source as possible plus potential exploits derived for testing the remaining static analysis type errors, perform the runtime analysis to derive the kernel authorization requirements.

3. **Static Analysis Using the Authorization Requirements**: More complete coverage of the kernel is possible using static analysis, so repeat the static analysis using the authorization requirements.

4. **Runtime Verification Using All Exploits**: Repeat the runtime analysis using any newly derived exploits from the second static analysis.

5.1 Checked/Unchecked Analysis

In the first step, the static analysis is applied to the kernel source, and some number of type errors are identified by CQUAL. We have fully automated this process, but the problem of examining type errors and determining whether they are exploitable must ultimately be done by an expert. The type error rate (type errors per variable of a controlled data type) varies from subsystem to subsystem, but it is 12% for the VFS layer (higher than usual) in Linux 2.4.9. Therefore, we have 30 variables in the VFS layer that are not explicitly authorized before they are used in a controlled operation.

Many of these type errors are not exploitable errors, however. In the VFS layer, these type errors come in three kinds: (1) use in initialization or other "safe" functions; (2) extraction of inodes from checked dentries; and (3) unknown type errors. The first two kinds of errors are not exploitable errors, so we need to change our analysis to prevent them from being generated. In the first case, we can relabel these functions, so they no longer require a `checked` variable. Since some functions are not obviously "safe," so there is some possibility for error here. When these functions are modified, some re-evaluation is necessary to maintain is status. In the second case, we need to change CQUAL to infer a `checked` variable if it comes from a `checked` field, and no user process can modify this relationship. For example, if we check a dentry then later extract the inode from this dentry, the LSM hooks believe that the inode is authorized also. This is because the dentry inode relationship is fixed (i.e., not modifiable by user processes). For situations of this type, we can infer the variable is `checked`. CQUAL does not support this kind of reasoning, but we are working with the CQUAL community to do this.

For other type errors, we need to find other means to distinguish whether they are real errors or not. For many of these cases, we develop exploit programs to try to find vulnerabilities with respect to the LSM authorizations. At present, this is a manual process, but we would like to automate some aspects of this process based on the nature of the type error. Ultimately, some degree of manual effort will always be required in processing type errors.

5.2 Authorization Requirements Generation

Second, we then perform the runtime analysis given the benchmark and exploit programs to discover vulnerabilities, identify anomalies in authorizations, and determine the authorization requirements of the controlled operations. The exploit programs identify vulnerabilities automatically, so we do not detail them further here (see the detailed static analysis paper [20] for the program that exploits the vulnerability in Figure 3). We discuss anomaly identification and the determination of authorization requirements here.

The Vali runtime analysis identifies the authorizations that are made on each object in a kernel event (i.e., system call with the same expected authorizations). Any variations in the authorizations signal either a miss definition a kernel event (i.e., the authorizations really change when we did not expect) or an anomaly in authorization. Recall that kernel events are defined by filter rules. Since writing these rules depends on deep knowledge of the kernel and LSM authorization, we expect that the LSM verification experts will write such filter rules to correctly generate anomalies. For example, we defined rules for open read-only and read-write cases in Figure 4.

An authorization anomaly is a case where an authorization only occurs in some of the cases of the kernel event. In Figure 5, the `set_owner` authorization was missing even though the inode field `f_owner` was accessed in a `fcntl` system call. While different flags to `fcntl` may result in different authorizations, we found that because the same fields were accessed with different authorizations, there could be a potential problem. Thus, this error was found in trying to write an appropriate filter for the different variants of `fcntl` invocations. A complete discussion of the different types of anomalies and their use in the classification of kernel events is provided in our runtime analysis paper [6].

Once the filters are written, they can be used by the Vali analysis tool to generate the object authorizations and any anomalous authorizations. In general, an object's authorizations may vary depending on the operations performed (i.e.,

```
DFN d 0 0 384 -1
DFN d 0 0 384 1
DFN d 0 0 400 -1
...
SFN(ALWAYS) d 0 0xc
```

Figure 6: Vali analysis output aggregating all inode and file operations with the same authorization. The DFN fields indicate: (1) "d" is *datatype-insensitive*, meaning all operations on the datatype have the same requirements; (2) first 0 is aggregate id; (3) second 0 is the class id for "inode"; (4) next number is the member id accessed; (5) last is the access type code.

field and access type) and the functions in which the operations are performed. The Vali analysis tool aggregates the common authorizations first by operation type, if their authorizations are always the same, then by function. That is, we hope that all operations in an event have the same authorizations. If not, then other operation attributes are used to aggregate when the authorizations are the same for operations with the same attribute value. We use operation datatype, object, member access, and access+function as the aggregation attributes. Figure 6 shows a datatype aggregation for inodes in the `read` system call (files are also aggregated). Figure 7 shows that authorizations may vary depending on the member access or the function in which the access is performed . The aggregation attributes are totally-ordered, so we try to aggregate at the attribute that yields the largest aggregate.

Maximizing aggregation also has the positive outcome that it reduces the number of regression differences. For example, if a controlled operation has the same authorization requirements regardless of the function in which it is run, then moving or adding the operation to a new function does not signal a regression difference.

```
DFN f 0 0 320 -1
SFN(ALWAYS) f 0 0x37

DFN f 1 0 1152 1
SFN(ALWAYS) f 1 0x13

DFN i 0 0 1216 1 ext2_lookup
...
SFN(ALWAYS) i 0 0x1a

DFN i 1 0 1216 -1 find_inode
SFN(ALWAYS) i 1 0x37
```

Figure 7: Vali analysis output showing four groups of *function-insensitive* ("f") and *function-internals-sensitive* ("i") operations for stat64. Function-insensitive accesses have the same authorizations regardless of the function in which the dangerous operation appears. Function-internals-insensitive operations have different authorizations as accesses to member 1216 in the two functions `ext2_lookup` and `find_inode` identify.

When all anomalies have been resolved, then the output defines the authorization requirements for the controlled operations in the kernel events in which they are run. Of course, some authorizations could be missing entirely for all runs, but we expect that the aggregated requirements will make it possible to verify this with reasonable effort.

5.3 Static Analysis Using Requirements

The authorization requirements found using the Vali runtime analysis can be used as input to the CQUAL static analysis. Three changes must be made to use the authorization requirements:

1. Authorizations must result in variables of a qualified type of the authorization made.

2. Functions annotations must be changed to

expect parameters of qualified types depending on the authorizations expected.

3. A type qualifier lattice must be built that represents the legal relationships between type qualifiers.

In the first case, we must now change `unchecked` variables to a qualified type commensurate with the authorization (e.g., `read_authorized`). Given that a variable can have multiple authorizations that depends on the kernel's control flow, such annotation itself is a subject of static analysis. CQUAL does not help with annotation (i.e., it is an input to CQUAL), so we must devise another technique for proper annotation. Fortunately, objects almost always have only one check, and no more than three, so this problem is handled manually at present.

The Vali authorization requirements for controlled operations generated from the Vali runtime analysis are used to identify the type qualifier requirements of functions. Figure 8 displays the output data from Vali used as input to this process. Variables are identified by their line number, data type, member access, and access type. While this is not completely unambiguous, it is sufficient for the kernel currently and we can identify ambiguities that cannot be resolved automatically. The SFNs identified as ALWAYS indicate the authorization requirements to be enforced on this variable.

Since multiple kernel events may use the same functions, the type qualifiers are, in general, the OR of these cases. This is represented using CQUAL's type qualifier lattice [8]. Since CQUAL's granularity is a function, code within a function that is called only when different authorization requirements are expected will not necessarily be handled properly by CQUAL. An example of this is `lease_modify` in Figure 5 where the authorization for `set_owner` is only necessary if `(arg == F_UNLCK)`. Where a combination of authorizations to a function are of the form $A \vee (A \wedge B)$ where A and B are authorization types, then we may need to manually annotate the code where $A \wedge B$ is required. We can do this by creating a dummy function call that requires $A \wedge B$. Ultimately, better intra-procedural analysis is required to find blocks of code within functions that require different authorizations, however, because such manual annotation limits regression testing (see Section 6.1).

```
DFN(namei.c, 207)(OT_INODE, 1152, -1)
SFN(ALWAYS): SCN_INODE_PERMISSION_EXEC
SFN(ALWAYS): SCN_INODE_PERMISSION_WRITE
SFN(ALWAYS): SCN_INODE_UNLINK_DIR
SFN(NEVER): SCN_INODE_UNLINK_FILE
SFN(NEVER): SCN_INODE_DELETE
```

Figure 8: Vali runtime output for the authorization requirements for a controlled operation in the `unlink` system call. The DFN indicates the line number, variable type, operation, and access which can be used to identify the variable in most cases. The ALWAYS SFNs indicate the authorization requirements.

5.4 Further Exploit Verification

Any exploit programs derived from the second static analysis are added to the Vali runtime analysis benchmarks, and runtime analysis is rerun. Since these programs are mainly looking for vulnerabilities, rather than anomalies, the output will be largely unchanged from step 2.

6 Kernel Community Approach

As the kernel evolves, the placement of the LSM authorization hooks may be invalidated. Since the kernel development community at large will modify the kernel, we need an approach in which the kernel modifications can proceed while maintaining the verification status of the LSM authorization hooks. Clearly, the kernel development community will not be inclined to perform the verification process of the LSM community as described above. However, it is possible for the kernel development community to leverage this work to maintain LSM verification.

Basically, we envision that kernel developer's task in maintaining the LSM authorization hook verification will involve regression testing on the static and runtime analyses. As part of an extended kernel build, the static analysis process can be run as in step 3 of the LSM community process above. The type errors generated above can be compared to the existing classifications to verify that no new type errors or type error paths are created.

Since some unresolved type errors are likely to remain for a while, it is ultimately necessary to perform runtime regression testing. While this task requires more work because the new kernel must be run, much, if not all, of this task can also be automated. In this case, the goal of the kernel development community is to identify any new anomalies or new authorization requirements (e.g., if a new object is added) to the LSM community. As described below, the Vali runtime analysis tool can identify such differences automatically.

6.1 Static Regression Testing

Since the GCC annotation process, CQUAL analysis, and output classification can be performed automatically, static LSM regression testing can be integrated as an extension to the automated build process. We first describe the tasks that are necessary to automate static analysis as part of the build process. While the analysis will generate the output automatically, some situations arise in manual analysis is currently required. We list these cases and examine their implications.

The build process for static analysis consists of the following steps:

- **GCC analysis**: Our extended GCC compiler must be used to build the kernel. The compiler creates a log of controlled operations, controlled variable declarations, and LSM authorization hooks. The following Makefile modifications are necessary:

  ```
  CC = $(CROSS_COMPILE)/\
  $(VALI_GCC)/gcc \
      --param ae-analyses=8243
  ```

  The parameter `ae-analyses` indicates the types of information that our extended GCC compiler gathers.

- **Perl annotation**: Perl scripts have been written to pre-process the GCC analysis output into a form that is then used by a second set of Perl scripts to automatically annotate the Linux source code with CQUAL type qualifiers.

  The GCC analysis generates output for each controlled operation, such as seen in Figure 9.

  The first set of Perl scripts processes such output into the form:

  ```
  /usr/src/linux-2.4.9-\
      .../fs/attr.c:61 \
      inode_setattr *inode
  ```

- **CQUAL Linux build**: CQUAL requires some pre-processing of the Linux source code before it can perform the analysis (e.g., removal of blanks and comments). The standard GCC compiler can be used for this step.

```
DEBUG_ACCESS: controlled operation:
   file = /usr/src/linux-2.4.9.../fs/attr.c
   current_function = inode_setattr.
   function_line = 63.
   current_line_number = 66
   access_type = write
   name = (*inode)
   member = i_uid (384)
   is_parameter = 1.
DEBUG_ACCESS: controlled operation:
   file = /usr/src/linux-2.4.9.../fs/attr.c
   current_function = inode_setattr.
   function_line = 63.
   current_line_number = 68
   access_type = write
   name = (*inode)
   member = i_gid (416)
   is_parameter = 1.
```

Figure 9: The GCC analysis output.

```
$(CC) $(CFLAGS) -E $< | \
  $(CQUALBINDIR)/remblanks >$*.ii
```

- **CQUAL analysis**: CQUAL can then be used to perform the static analysis. The first step runs the CQUAL analysis. The second step generates the type error path information. See Figure 10.
- **Authorization requirement annotation**: Vali runtime analysis generates authorization requirements per controlled operation as shown in Figure 8. We are in the process of writing Perl scripts to apply these requirements to the annotation of authorizations specific to the requirements described above. We hope to be able to report on this at the symposium.

While we have not done detailed time measurements, we have found that the entire analysis adds about 10 minutes to the build time. GCC analysis adds little overhead to the kernel build. Perl processing takes about 5 minutes for the kernel. CQUAL analysis takes approximately another 5 minutes for the kernel.

The current static analysis process verifies that the LSM authorization hook placement is correct, but some situations need further, manual examination. These cases are listed below:

1. **Changes to “safe” functions**: The GCC analysis identifies the addition of a new controlled operation to a function formerly classified as “safe,” see Section 5.1.
2. **Changes to manually annotated functions**: A source comparison detects a change to a function with any manual annotation (e.g., the addition of dummy functions for ORed authorization requirements, see Section 5.3).
3. **New type error variables**: The CQUAL analysis identifies any new variables that have type errors.
4. **New shortest type error paths**: The

```
$(CQUALBINDIR)/cqual -prelude \
    $(CQUALDIR)/config/prelude.i.security -config
$(CQUALDIR)/config/lattice.security attr.ii 2>attr.path
```

Figure 10: Performing static analysis with CQUAL.

CQUAL analysis identifies any new shortest type error path for a variable with an existing type error.

The first two situations are cases where the dependencies of the analysis have changed, such that the analysis may no longer be sound. While we hope to eliminate such dependencies through further analysis, we expect the analysis will always be subject to some number of dependencies. In fact, as the analysis becomes more elaborate, the complexity of dependencies increases, so the current set may prove to be the best option.

The second two situations are the identification of a new type error that may indicate a real vulnerability. In order to reduce the number of false positives, secondary analyses are necessary to identify them. These analyses may have dependencies (e.g., that is the cause of case 1), so the cost of managing the dependencies must be less than the value of removing the false positives.

While it is not completely clear where the balance between manual effort on the part of the kernel developer and LSM community is in this process, we anticipate the following. Our goal is that most notifications of case 1 and 2 can be handled trivially by the kernel development community and the LSM community can verify. Errors of case 3 and 4 may also be handled by the development community in many cases, but again the LSM community may do deeper verifications and develop classifiers to eliminate identifiable false positives.

6.2 Runtime Regression Testing

Since we expect that there will always be some number of type errors for which exploits can become possible and some tasks that are more easily or better done by runtime analysis, we strongly recommend performing the runtime regression analysis. However, this analysis is more time-consuming than the static analysis in two key ways: (1) the instrumented kernel must be built and (2) the runtime analysis benchmarks must be executed on the instrumented kernel.

At present, the build process for a Vali-instrumented kernel, runtime logging modules, and analysis tools is completed automated. However, the execution of the analysis is not automated at present. The main tasks that are not automated are: (1) the collection of instruction pointer locations for kernel entry/exit points used to identify the kernel events and (2) the runtime execution. The first task only involves "grepping" the generated object dump for a few well-known instruction, so it appears straightfoward to automate this. We are looking into how to automate the runtime data collection using VMware [4].

In order to enable regression testing, the Vali runtime analysis tool generates output that does not include line number or instruction pointer information, so that regression can be done across minor kernel modifications. Further, aggregation of controlled operations that are not function-sensitive enables regression across kernel modifications regardless of func-

[4]VMware is a trademark of VMware, Inc.

tions executed in a kernel event.

Figures 6 and 7 give an idea of how the output from the Vali runtime tool enables regression. The output shows the controlled operations with the same authorization requirements. In cases where the authorization requirements of controlled operations are sensitive to the function in which the operation is run, more information is displayed. In this case, if the controlled operation is moved from one function to another, the regression test identifies the change.

Given aggregation, the following types of changes between regression tests are possible:

- **New controlled operation in an aggregate**: An operation has been added, and it has been classified with an existing aggregate.
- **Remove controlled operation from an aggregate**: An operation has been deleted, so it no longer appears.
- **Move controlled operation to another aggregate**: Either an authorization or an operation has been moved such that a different set of authorizations are active when the operation is performed.
- **Create a new aggregate**: A new set of authorizations has been created or a new sensitivity has been triggered such that a new aggregate of operations and permissions has been created.

The addition and removal of controlled operations is not a major change if they adhere to the existing aggregates. However, it is always wise to verify that the operations are consistent with the aggregations assigned to them. The move of operations to other aggregates or the creation of new aggregates are significant changes that warrant review.

7 Conclusions and Future Work

In this paper, we outline static and runtime analysis tools that we have developed to verify the correctness of LSM authorization hook placement. These tools have been used to find five, since fixed, errors in LSM hook placements. We believe that such verification should not be a one-time process, but rather it should practical for kernel developers to perform regression testing as the kernel is modified. The problem is to automate the analysis process as much as possible and only provide test results that really require examination by the development community, as much as possible. We demonstrate that static analysis process and most of the runtime analysis process are automated already. We also identify the types of analysis results that the tools will report to the developers. While it is nice to eliminate as many false positives as possible, we are limited by the Halting Problem as to how many can be removed in general and the means for identifying false positives introduces dependencies that also must be verified. At present, we do not eliminate most false positives automatically, but expect that the LSM community will identify them as such and regression over these will be sufficient (i.e., as long as few, new false positives are introduced little effort will be required to handle them). The generated output is low-level which enables quick comparison, but still makes is difficult for developers. Interfaces for handling this information are a significant area of future work.

References

[1] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[3] N. S. Borenstein. Computational mail as a network infrastructure for computer-supported cooperative work. In *Proceedings of the Fourth ACM CSCW Conference*, pages 67–74, 1992.

[4] Wirex Corp. Immunix security technology. Available at `http://www.immunix.com/Immunix/index.html`.

[5] A Edwards. [PATCH] add lock hook to prevent race, January 2002. Linux Security Modules mailing list at `http://mail.wirex.com/pipermail/linux-security-module/2002-January/002570.html`.

[6] A. Edwards, T. Jaeger, and X. Zhang. Verifying authorization hook placement for the Linux Security Modules framework. Technical Report 22254, IBM, December 2001.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*, October 2000.

[8] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 192–203, May 1999.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *The Sixth USENIX Security Symposium Proceedings*, pages 1–12, 1996.

[10] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[11] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, 2001.

[12] S. Minear. Providing policy control over object operations in a Mach-based system. In *Proceedings of the Fifth USENIX Security Symposium*, 1995.

[13] NSA. Security-Enhanced Linux (SELinux). Available at `http://www.nsa.gov/selinux`.

[14] LIDS organization. Linux intrusion detection system. Available at `http://www.lids.org`.

[15] A. Ott. Rule set-based access control (RSBAC) for Linux. Available at `http://www.rsbac.org`.

[16] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–216, 2001.

[17] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and J. Lepreau. The Flask security architecture: System support for diverse policies. In *Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[18] Argus Systems. Argus PitBull LX. Available at `http://www.argus-systems.com`.

[19] L. Torvalds and C. Cowan. Greetings, April 2001. Linux Security Modules mailing list at `mail.wirex.com/pipermail/linux-security-module/2001-April/000005.html`.

[20] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002. To appear.

Buried alive in patches: 6 months of picking up the pieces of the Linux 2.5 kernel

Dave Jones
SuSE Labs
davej@suse.de, http://www.codemonkey.org.uk

Abstract

When development began on the 2.5 Linux kernel, I volunteered to forward port 2.4 fixes to 2.5, and keep them in sync ready for when Linus was ready to accept them. Additionally, I collected the small bits Linus frequently overlooked, and perhaps more importantly, tried to keep the 2.5-dj tree in a usable, bootable state even when 2.5 mainline wasn't.

1 Introduction

With the advent of a new development series of the Linux kernel, Linus Torvalds typically hands off the current tree to someone else, who becomes maintainer of that stable series, and work on the development branch accelerates, whilst the stable branch collects fixes and updates rather than new features.

Typically, as focus on the development branch is in other areas, important fixes continue to pour into the stable series which are sometimes not picked up in the development branch until much later, or sometimes, not at all.

In the past Alan Cox has done sterling work in picking up the fixes that go into stable series, and collecting them together in his regularly released -ac patches. At various points, Alan would then push known good parts to Linus at such a time that he is ready for them.

When the 2.4 kernel diverged to the 2.5 development branch, there was still a considerable number of fixes going into 2.4. Alan was busy with other projects at the time, and so it was suggested that 'someone' should pick up the bits going into 2.4 and make sure they don't get left behind for 2.5.

Without fully thinking through the consequences, I decided to step forward, and got a lot more than I bargained for, but learned a lot, and had a lot of fun on the way.

2 The problems

The easy part of the job looks something like this:

```
repeat:
$ cd linux-2.5
$ cat ../patch-2.4.18-pre1.diff \
  | patch -p1 -F1 --dry-run

(note rejects)

$ vi patch-2.4.18-pre1.diff

(chop out rejects)
until applies
```

The same process applies whenever Linus releases a new 2.5 kernel.

This is however just a fraction of what the job entails.

The first thing to be aware of is that a lot of the fixes going into 2.4 may not be relevant to what's happening in 2.5. For example, maybe the maintainer of relevant code wants things fixed cleaning in 2.5, whereas a band-aid is acceptable in 2.4, or maybe large restructuring of the code is planned for 2.5, so the fix is irrelevant. Sometimes development of a driver continues actively in 2.4 and 2.5, and takes wildly different directions.

Keeping up-to-date with what every maintainer is doing with their subsystem is a tricky task that involves lots of mindreading, guesswork, and occasionally email to ask, "What exactly is going on with xxx?"

Another tricky part of the job is making sure the tree is still in a state where you can test that what you've just merged actually works. Not easy at times in a development series when there are bits of core functionality being ripped apart. Sometimes this results in compromises (not merging certain parts until they compile), sometimes getting ahead of mainline (where fixes appear faster than Linus merges them), and sometimes by means of adding really ugly hacks that don't stand a chance to be accepted for mainline, but "do the job" for most people who are more concerned about their own specific part of the kernel.

Perhaps by far trickiest part of all however is trying to split things up into Torvalds-size pieces so that every so often, a resync can occur to push some of the more obviously correct, and well-tested bits back to the mainline tree. As I found my feet with syncing, I tried various approaches, some of which worked out better than others.

There are several reasons for the difficulty.

- In the case of merging a 2.4pre to 2.5, using the above method means I'm left with a several MB patch which originally consisted of perhaps dozens of smaller patches. Whilst some of these are sent to the Linux kernel mailing list, not all are visible until they show up in Marcelo's tree.
- Maintainer issues. Sometimes it's not immediately obvious from reading the diff (or even the code in a before and after state) why a patch is needed. The maintainer however knows (or at least should know) his/her own code inside out, and know the precise reasoning behind every diff going into Marcelo/Linus' kernel. In these circumstances, it's often the best policy to let them take care of merging such patches, as they can explain to Linus in much better terms why he needs to take the patch instead of my guesswork and hand waving.
- Patch drift. Patches I did manage to pick up from the kernel mailing list, or were Cc:'d to me were a little easier, as they tended to come with good descriptions by the patch author. The only problem with these was that over time, the patch would no longer apply to Linus' vanilla tree, so the patch would have to be kept up to date. With so many patches applied, keeping them all up to date seperately became harder and harder, especially if several patches wanted to touch the same files.
- Conflicts. When two or more patches are touching the same file, what happens next depends on the level of change in the various parts. For example, if I had several patches touching the tulip network driver, one fixing an obvious bug, one fixing a spelling mistake, and another adding

a MODULE_LICENSE tag, the latter two are trivial enough that the patch doesn't need splitting.

Where there are two parts to the patch fixing different problems, or perhaps adding functionality, things get more complicated, and tools such as editdiff become huge timesavers.

It became apparent very quickly that splitting up the several MB patches from 2.4 back into their component parts for each release wasn't feasible due to the amount of time it took. Each time a new Marcelo prepatch appeared, it was merged wholesale after removal of unneeded parts. When the periodic resyncs with Linus then occured, there were in many cases quite a few trivial patches to the same file, making it easy to get rid of lots of the smaller parts of my tree.

3 Patch Rejection

It would be an easy job to simply apply every patch that ever gets sent either directly, or to the kernel mailing list. However things are never simple, and a number of factors have to be taken into consideration.

Chances of Linus ever accepting them. Some patches are just too ugly to live. Various people sent me patches that Linus had rejected, in the hope that as it was coming from me, Linus would somehow take a different view. Somewhat amused by this, most of these patches are either memorable, or discussion with the relevant maintainer is usually enough to get a "don't apply" message back. If there's no chance of Linus ever taking it, then keeping patches of this kind in my tree was deemed pointless.

Controversial patches. A good example of this case was Eric Raymond's CML2 patch. A very large patch that touched the configuration file of every part of the kernel. It's hard to imagine a more far-reaching change. At one point, Linus even made claims that he had no interest in the kernel configuration language, and that it would be probably better maintained outside the kernel tree. So this example also falls into category 1. Had I merged CML2 at any point, this would have made it impossible to merge any configuration updates from mainline without first rewriting them as CML2 rules, which was unacceptable. Likewise, any changes made could not be sent back to mainline.

Orthoganal works. Sometimes patches appear, and there will be nothing technically wrong with them, but perhaps the timing is wrong due to someone else working in the same area on maybe a larger scale. An example of this was the i386 sub-arch patches James Bottomley did, which unfortunatly clashed with the considerable rewrites Randy Dunlap did to i386-specific drivers such as MTRR.

4 Timeline of events.

4.1 December 2001

At the beginning of December, Linus had put out 2.5.0, and was concentrating almost solely on merging Jens Axboe's block layer rewrite. At the same time, Marcelo had begun his first 'real' patch merging, after having put out the rush-released 2.4.16. It was noted that these fixes were not getting merged into 2.5, and Dave Miller suggested that someone collect them, and keep them up to date until such a time that Linus was ready to accept them. I had been doing this partially at the time for my own use anyway, so I decided to take it on.

Towards the end of the month, when Linus was up to 2.5.1pre11, I made my first release of -dj, which was around a 1MB diff against Linus' current tree.

4.2 January 2002

In January, Linus had got up to 2.5.2pre, and Marcelo had accelerated in patch merging as he got used to his new role. The diffsize between my tree and Linus' had started to increase dramatically, and so the first resync of the trees was planned. After pushing a considerable amount of the more obvious fixes to Linus, only some of the trickier-to-merge bits remained, although still a sizable amount.

It was at this time that Linus started breaking things dramatically in 2.5. First came the big kdevt redesign, which broke compilation of many parts of the kernel. The linux-kernel mailing list was awash with many fixes for these compile errors, although it took Linus some time to get many of them merged.

Another key point of note in January was the introduction of the new Framebuffer API into my tree. James Simmons wanted a 2.5 that stood more chance of being able to compile, and decided to use my tree as a basis for development.

A great supplement to me picking up various small patches on the Linux kernel mailing list was Rusty Russell's trivial patchbot. Patches sent to the robot get archived, and automatically retransmitted on the sender's behalf, and do all kinds of magic like automatically checking that they still apply when a new kernel appears, and bouncing a "doesn't apply any more—please rediff" back to the patch author. With both Rusty and myself picking up and retransmitting these on the patch-author's behalf, the "Linus isn't taking my patches" arguments for trivial patches all but disappeared.

During pushing bits to Linus, I discovered Tim Waugh's patchutils, which is a set of tools for manipulating diffs. Until this point, when sending a patch to Linus, I had been doing pretty much everything in vim, and chopping out unnecessary parts of the rest of my tree from the patch. Quite a time-consuming process. With patchutils, things got a lot easier as I could now with a 'simple' command line extract all the related parts of a patch from my tree. For example,

```
grepdiff pf_gfp_mask dj1.diff \
        | xargs -n1 \
        filterdiff dj1.diff -i
```

would extract all the diffs from my tree that touch pf_gfp_mask. After doing this, and deleting the irrelevant hunks of the the diff from the output, the patch was more or less ready to go to Linus.

As well as these, Patchutils also includes many other useful tools that save more time to a patch-merger than any other tool I've yet to run.

Toward the end of the month, my tree was around 2MB away from Linus.

4.3 February

At the beginning of February, Linus was up to 2.5.5, and with the block layer taking shape, he had started merging some other large new features. This kernel saw the introduction the x86-64 port, the ALSA merge, and the first parts of the new input layer (all of which had been in -dj for a month).

On the downside, some other things had continued to break dramatically. Lots more drivers no longer compiled due to the virt_to_bus macros being changed in an attempt to get more portable drivers. I made a compromise in my tree and wrapped this change in an option called CONFIG_DEBUG_OBSOLETE. When not selected, the old behaviour occurred, and the drivers continued to compile. Crude, but effective.

Other notable changes this month included the various IDE cleanups by Martin Dalecki, Vojtech Pavlik, Pavel Machek, and others. As a result of this work, Andre Hedrick stood down as 2.5 IDE maintainer.

It became aparent that I hadn't pushed to Linus for a while, as by now, my tree was 4MB away from Linus, with quite a lot of patches pending. The new framebuffer API work was taking up a large percentage of this, as was the new input layer work.

Christoph Hellwig had been regularly looking through my patches, and with nearly every release he would spot something really dumb that I did, like reintroducing calls to a now-dead API, or CVS $ID: tag damage. (I use CVS and frequently forget to add files with -ko). These silly mistakes happened often enough that I decided to write a simple perl script to check for silly mistakes like this.

```
#!/usr/bin/perl -w
# checkdiff.pl -- 2.5-dj kernel patch checker.
#
# I’m stupid.
# This script sanity checks diffs before I put them out, so
# that hch doesn’t have to remind me I goofed.

use strict;

foreach (@ARGV) {
    process ($_) or warn "Couldn’t check file $_: $!";
}

sub process {
    my $filename=shift;

    open INPUT, $filename or return undef;
    my @lines=<INPUT>;
    close INPUT;
    chomp @lines;

    my $linenr=0;

    foreach my $line (@lines) {
        $linenr++;

        if ($line=~/davej/ and $line=~/\$Id:/) {
            print "Found davej CVS damage at line $linenr\n$line\n\n";
        }

        # We are adding a line, check its not obsolete.
        if ($line=~/^\+/) {
            if ($line=~/iorequest_lock/) {
                print "Adding code to frob iorequest_lock" .
                      " at line $linenr\n$line\n\n";
            }
            if ($line=~/[        ]MAJOR[ (]/) {
                print "Adding code to frob MAJOR at line $linenr\n$line\n\n";
            }
            if ($line=~/get_fast_time/) {
                print "Adding get_fast_time at line $linenr\n$line\n\n";
            }
            if ($line=~/strtok/) {
                print "Adding strtok at line $linenr\n$line\n\n";
            }
            # ... other rules here ...
        }
    }
    return 1;
}
```

Cut-down version of the Perl script used for checking patches before uploading to kernel.org

4.4 March

By March, mainline was up to 2.5.7. Probably the biggest change was the introduction of Robert Olsen and co's NAPI work.

In an attempt to get the diff size between my tree and Linus' down, I decided to drop the arch updates for architectures like S390, m68k, etc. The reasoning behind this was that even with the updates from my tree, they never compiled in 2.5 anyway, due to the lack of other necessary changes, such as those imposed by the introduction of Ingo Molnar's O(1) scheduler.

Linus had at this point been using bitkeeper for a few weeks, and was just starting to get comfortable with it. Patches seemed to be getting applied at a much more accelerated rate than previously. I wondered if it could help out with the merging of my-tree to Linus', and played with it for a week or so, often exchanging ideas/queries with Larry McVoy, but ultimately, it didn't work out for me. The only way bitkeeper would work for such a large set of diffs (at the time, up to 7MB away from mainline) would have been to have many trees for all the different patches, all combined into one union tree. There was however a problem with this. If I go to the extreme of splitting up my tree into component parts to feed into bitkeeper trees, I may as well just feed those small bits to Linus as regular GNU patches.

Linus agreed with this, and this is how we continued the merging.

I started to fall behind a little in this month with syncing both from my tree to Linus and vice versa. Mostly due to me moving house, and having to make do without life's essentials (like ADSL) for a while.

After getting back online, and spending a few days getting everything back up to date, Linus went on holiday for a few weeks, with the parting note "Jeff Garzik and Dave Jones will be taking care of patches whilst I'm gone." The following two weeks were mostly quiet fortunately, which gave me great opportunity to split up patches ready for the largest resync so far.

4.5 April

April began with 2.5.8 still being current. Linus had returned from his holiday, and 'The big resyncing' ensued. With 7.5MB of diff between his tree and mine, this was no small task, and not one that would be over any time soon.

I pushed 128 patches to Linus, 493KB worth, and a further 474KB in 50 patches to Jeff Garzik (network drivers and the like). 6 hours later, 126 of the patches I sent direct to Linus were applied. An hour later, most of what I sent to Jeff also showed up in Linus' tree. In a state of shock, I wondered how I could "shovel faster." By the time Linus put out pre1, my tree was 6MB away from mainline.

pre2 was another few hundred KB (although mostly fixing wrong merges from pre1). Whilst splitting up various bits for Linus, and looking through the patches in my tree, I also found a lot of other patches that really didn't make sense to continue carrying (whitespace differences, superseded patches, updated CVS idents, etc.).

A mistake that happened during merging here was that CONFIG_DEBUG_OBSOLETE slipped in as a Config.help text. Interestingly, Linus decided to drop what this option was wrapping in my tree, and make it unconditional. Whilst having more portable drivers was an admirable goal, it hadn't really prompted any of the maintainers to fix their drivers, causing more headache than anything productive.

After this, I spent a week at Linuxworld—i.e., where I had hoped to do more patch splitting, but it didn't happen due to time constraints. What did happen was the beginning of more catch up work. By the time I returned, I had to resync 2.4.19pre4,pre5,pre6 & 2.5.8pre2 & pre3 to my tree.

In addition to this, there were now 113 patches in my inbound queue. At this point it was realised that I couldn't 'stop to do a resync' every so often, and that syncing a "moving target" was considered only viable option.

Toward the end of the month, Linus stopped doing -pre patches, and instead started doing full releases more often. At the end of the month, 2.5.11 was current, which featured another large number of merges, including the beginnings of the new framebuffer code. At time of writing this paper, resync against 2.5.11 hadn't been done, but estimated diffsize was between 4-5MB.

5 Future Plans

At the time of writing, there are over two months remaining before OLS, during which much more resyncing will happen, and conversely, many more patches are likely to get applied to my tree. The larger parts of my tree (the Framebuffer code, new input API, ALSA OSS updates) are slowly starting to get merged into mainline. If by some freak opportunity my tree manages to get down to a reasonable size again, I'll take a look at using bitkeeper for merges once again.

Given a hypothetical perfect world, the plan is that by the time 2.6 is ready, the -dj series of kernel patches will become less necessary. Depending on rates of merging, positions of moon, and other acts of randomness, it may be that 2.6-dj patches will become necessary until fully merged, and some of the more experimental bits may end up being put back until 2.7.x opens.

6 Links

`http://www.kernel.org/pub/people/davej/patches/2.5/`
Where my -dj patches are to be found.

`http://www.codemonkey.org.uk/Linux-2.5.html`
A list of known remaining problems with current/recent 2.5's.

`http://diary.codemonkey.org.uk/`
My online diary where I frequently write about progress on 2.5

`http://cyberelk.net/tim/patchutils/`
Tim Waugh's patchutils

`trivial@rustcorp.com.au`
Rusty Russell's trivial patch robot.

7 Acknowledgments

The work of patch integrator is largely a solitary task, but there are countless people worthy of gratitude, from the many people who were prepared to download and try out my -dj patches, especially those who sent feedback/reports/patches.

Tim Waugh for patchutils, which made merging so much easier.

Larry McVoy for making Linus scale despite the critics.

And finally, Alan Cox for being busy with other things in December 2001. Without which, I'd probably have found something much more boring to have spent the last 6 months hacking.

Documentation/CodingStyle and Beyond

Greg Kroah-Hartman
IBM Linux Technology Center
greg@kroah.com | gregkh@us.ibm.com

Abstract

With more companies starting to write Linux kernel code, an understanding of what is the acceptable kernel coding style and conventions is becoming a necessity. The goal of this paper is to explain both the written and unwritten Linux kernel programming style. It explains why a consistent coding style and rules are a requirement for the kernel. It discusses the basic kernel style rules as outlined in `Documentation/CodingStyle` and explains the large number of style rules that are not documented. Each of these rules is documented with existing code, and why the rule is considered a "good thing."

1 Why rules?

Why are there kernel programming style rules in the first place? Why not just let every author code in whatever style they want to, and let everyone live with it? After all, code formatting does not affect memory use, execution speed, or anything else a normal user of the kernel would see. The reason can be summed up with this quote from Elliot Soloway and Kate Ehrlich in 1984[1]

> It is not merely a matter of aesthetics that programs should be written in a particular style. Rather there is a psychological basis for writing programs in a conventional manner: programmers have strong expectations that other programmers will follow these discourse rules. If the rules are violated, then the utility afforded by the expectations that programmers have built up over time is effectively nullified.

A number of other studies and research has proven that if a large body of code is written in a common style, it directly affects how easy it is to quickly understand the code, review it, and revise it.

Since the number of developers that look at the Linux kernel code is very large, it is in the best interest for the project to have a consistent style guideline. This allows the code to be more easily understood either by someone reading it for the first time, or by someone revisiting their old code later. It also allows someone else to more easily read, understand, and potentially fix and enhance your code, which is one of the greatest strengths of open source code.

2 What are the rules?

Now that we have an understanding that there should be some rules, what are they? Linus Torvalds and other kernel programmers have written a short document that details some of the kernel programming rules. This document is located in the *Documentation/CodingStyle* file in the kernel source tree. It is required reading for anyone who wants to contribute to the

Linux kernel. Here is a summary of these rules.

2.1 Indentation

All tabs are 8 characters[1], and will be the `<TAB>` character. This makes it easy to quickly locate where different blocks of code start and end. If you find your code is being indented too deeply, with more than three levels of indentation causing the code to shift off to the right of the screen, then you should fix the code. It is a good warning.

2.2 Placing Braces

The original authors of UNIX placed their braces with the opening brace last on the line, and the closing brace first on the line, like:

```
if (x is true) {
        we do y
}
```

Because of this, the kernel shall be written in this style.

The exception to this rule are functions, which have the opening brace at the beginning of the line, like:

```
int function(int x)
{
        body of function
}
```

Again, this is how Kernighan and Ritchie wrote their code.

For good examples of the proper indentation and braces style, look at any of the *fs/*.c* files, or anything in the *kernel/*.c* files. Generally, most of the kernel is in the proper indentation and brace style, but there are some notable exceptions. The code in *fs/devfs/*.c* or *drivers/scsi/qla1280.** are good examples of how **not** to do indentation and braces.

There is a script that can be used to run the `indent(1)` program in the proper kernel indentation and braces style. It is useful if you have to convert a large amount of code to the correct format. This file is located at *scripts/Lindent* in the kernel source tree.

2.3 Naming

Your variables and functions should be declared descriptively and concisely. You should not use long flowery names like `CommandAllocationGroupSize` or `DAC960_V1_EnableMemoryMailboxInterf()`, but rather, `cmd_group_size`, or `enable_mem_mailbox()`. Your names need to be descriptive, and easily recognized. Mixed case names are frowned upon and encoding the type of the variable or function in the name (like "Hungarian notation") is forbidden.

Global variables should be only used if they are absolutely necessary. Local variables should be short and to the point. `i` and `j` are valid local loop variable names, while `loop_counter` is too verbose. `tmp` is allowed to be used for any short-lived temporary variable.

Again, good examples of proper names can be found in *fs/*.c*. Lots of driver code has bad variable names, as they have been ported from other operating systems. *drivers/block/DAC960.** and *drivers/scsi/cpqfc** are examples of how to **not** name functions and variables.

[1]To fit within a column, some of the examples have been forced to deviate from this. *—OLS Formatting Team*

2.4 Functions

Functions should only do one thing, and do it well. They should be short, and contain one or two screens of text. If you have a function that does lots of small things for different cases, it is acceptable to have a longer function. If you have a complex long function, it should be rewritten to be simpler.

If you have a large number of local variables within a function, it is also a measure of the complexity. If there are more than 10 local variables, it is too complex.

There are lots of good examples of nice sized functions in the *fs/*.c* and other kernel core code. Some bad examples of functions can be found in *drivers/hotplug/ibmphp_res.c* where one function is 370 lines long, or *drivers/usb/usb-uhci.c* where one function has 18 local variables.

2.5 Comments

Comments are very good to have, if they are good comments. Bad comments explain how the code works, who wrote a specific function on a specific date, or other such useless things. Good comments explain what the file or function does, and why it does it. They should be at the beginning of the function, and not necessarily embedded within the function. You are writing small functions, right?

There is now a standard format for function comments. It is a variant of the documentation method used by the GNOME project for their code. If you write your function comments in this style, the information in them can be extracted by a tool and made into standalone documentation. This can be seen by running `make psdocs` or `make htmldocs` on the kernel tree to generate a *kernel-api.ps* or *kernel-api.html* file containing all of the public interfaces to the different kernel subsystems.

```
/**
 * function_name(:)? (- short description)?
(* @parameterx: (description of parameter x)?)*
(* a blank line)?
 * (Description:)? (Description of function)?
 * (section header: (section description)? )*
(*)?*/
```

Figure 1: The format of a block comment

This style is documented in the file *Documentation/kernel-doc-nano-HOWTO.txt* and *scripts/kernel-doc*. The basic format can be seen in Figure 1.

The short function description cannot be multi-line, but the other descriptions can be, and they can contain blank lines. All further descriptive text can contain the following markups:

`funcname()` - name of a function

`$ENVVAR` - name of a environment variable

`&struct_name` - name of a structure (up to two words including 'struct')

`@parameter` - name of a parameter

`%CONST` - name of a constant.

A simple example of a function comment with a single argument looks like:

```
/**
 * my_function - does my stuff
 * @my_arg: my argument
 *
 * Does my stuff explained.
 **/
void my_function (int my_arg)
{
        ...
}
```

Comments should be written for structures, unions and enums. The format for them is much like the function format:

```
/**
 * struct my_struct - short description
 * @a: first member
 * @b: second member
 *
 * Longer description
 */
struct my_struct {
        int a;
        int b;
};
```

Some good examples of well commented functions can be found in the *drivers/usb/usb.c* file, where all global functions are documented. The file *arch/i386/kernel/mtrr.c* is a good example of a file with a reasonable amount of comments, but they are in the incorrect format, so they can not be extracted by the documentation tools. *drivers/scsi/pci2220i.c* is also a good example of how **not** to create the comment blocks for your functions.

2.6 Data Structure requirements

The addition of a chapter on data structures, showed up in the 2.4.10-pre7 kernel release. It describes how every data structure that can exist outside of a single-threaded environment, needs to implement reference counting to properly handle the memory management issues. If you add reference counting to your structure, you can avoid lots of nasty locking issues and race conditions. Multiple threads can access the same structure without having to worry that a different thread will free the data from under it.

The last sentence in this chapter is required reading by any kernel developer:

> Remember: if another thread can find your data structure, and you don't have a reference count on it, you almost certainly have a bug.

A good example of why reference counting is necessary can be found in the USB data structure, `struct urb`. This structure is created by a USB device driver, filled with data, sent to a USB host controller where it will be processed and eventually sent out over the wire. When the host controller is finished with the urb, the original device driver is notified. While a host controller driver is processing the urb, the original driver can try to cancel the urb, or even free it. This led to long detailed arguments on the *linux-usb-devel* mailing list about when in the life span of a urb it was allowed to be touched by either driver, and numerous bugs in the core USB subsystem and different USB drivers.

In the 2.5 kernel series, `struct urb` had a reference count added to it, and the USB core and USB host controller drivers had a small amount of code added to properly handle the reference count. Now whenever a driver wants to use the urb, a reference count is incremented. When it is finished, the reference count is decremented. If this was the last user, the memory is freed, and the urb disappears. This allowed the USB device drivers to vastly simplify their urb handling logic and fixed lots of different race condition bugs. It also made all of the developer's lives simpler by quieting all arguments about the topic.

3 Unwritten rules

If you follow the above set of rules, your code looks like good Linux kernel code. There are quite a few unwritten rules and style guidelines that good kernel code follows. Here are some of them.

3.1 Avoid NIH syndrome

There are a wide variety of well designed, well documented, and well debugged functions and data structures within the kernel. Take advantage of them rather than reinventing your own version. Among the most common of these are the string functions, the byte order functions, and the linked list data structure and functions.

3.2 String functions

In the file, *include/linux/string.h*, a number of common string handling functions are defined. These include:

```
strpbrk
strtok
strsep
strspn
strcpy
strncpy
strcat
strncat
strcmp
strncmp
strnicmp
strchr
strrchr
strstr
strlen
strnlen
memset
memcpy
memove
memscan
memcmp
memchr
```

And in the file, *include/linux/kernel.h*, a number of "simple" string functions are defined:

```
simple_strtoul
simple_strtol
simple_strtoull
simple_strtoll
```

If you need any type of string functionality in your kernel code, use the built in functions. Do not try to rewrite the existing functions accidentally.

3.3 Byte order handling

Do not rewrite code to switch data between different endian representations. The file *include/asm/byteorder.h* (*asm* will point to the proper subdirectory, depending on your processor architecture) brings in a wide range of functions that allow you to do automatic conversions, no matter what the endian format of your processor or your data.

3.4 Linked Lists

If you need to create a linked list of any kind of data structure, use the code that is in *include/linux/list.h*. It contains a structure, `struct list_head`, that should be included within the structure for the new list. You can easily add, remove, or iterate over a list of data structures, without having to write new code.

Some good examples of code that uses the list structure can be found in *drivers/hotplug/pci_hotplug_core.c* and *drivers/ieee1394/nodemgr.c*. Some code in the kernel that should be using the list structure, can be found in the ATM core, within the `struct atm_vcc` data structure. Because the ATM code did not use `struct list_head`, every ATM driver needs to walk the lists of data structures by hand, duplicating lots of code.

3.5 `typedef` is evil

`typedef` should not be used in naming any of your structures. Almost all main kernel structures do not have a `typedef` to shorten their usage. This includes the following:

```
struct inode
struct dentry
struct file
struct buffer_head
struct user
struct task_struct
```

Using `typedef` tries to hide the real type of a variable. There have been records of some kernel code using typedefs nested up to 4 layers deep, preventing the programmer from telling what type of variable they are really using. This can easily cause very large structures to be accidentally declared on the stack, or to be returned from functions if the programmer does not realize the size of the structure.

`typedef` can also be used as a crutch to keep from typing long structure definitions. If this is the case, the structure names should be made shorter, according to the above listed naming rules.

Never define a `typedef` to just signify a pointer to a structure, as in the following example:

```
typedef struct foo {
        int bar;
        int baz;
} foo_t, *pfoo_t;
```

This again hides the true type of the variable, and is using the name of the variable type to define what is is (see the comment about Hungarian notation previously.)

Some examples of where `typedef` is badly used are in the *include/raid/md*.h* files where every structure has a `typedef` assigned to it, and in the *drivers/acpi/include/*.h* files, where a lot of the structures do not even have a name assigned to them, only a `typedef`.

The only place that using `typedef` is acceptable, is in declaring function prototypes. These can be difficult to type out every time, so declaring a typedef for these is nice to do. An example of this is the `bh_end_io_t typedef` which is used as a parameter in the `init_buffer()` call. This is defined in *include/fs.h* as:

```
typedef void (bh_end_io_t)
        (struct buffer_head *bh,
        int uptodate);
```

3.6 No magic numbers

The Jargon file[2] describes a magic number within source code as:

> In source code, some non-obvious constant whose value is significant to the operation of a program and that is inserted inconspicuously in-line (hardcoded), rather than expanded in by a symbol set by a commented `#define`. Magic numbers in this sense are bad style.

Fortunately the kernel does not have many instances of code that uses magic numbers. The *drivers/usb/serial/pl2303.c* driver used to have the code shown in Figure 2 in the `open()` function. This code contains a lot of of different magic numbers. The current version of the file can be seen in Figure 3. Even with the odd use of the macros `FISH()` and `SOUP()`, some of the magic numbers have been replaced with the more descriptive `VENDOR_READ_REQUEST_TYPE`, `VENDOR_READ_REQUEST`, `VENDOR_WRITE_REQUEST_TYPE` and `VENDOR_WRITE_REQUEST`. This code could be cleaned up a lot more, detailing what the other magic numbers mean. Unfortunatly the driver was written by reverse engineering a protocol stream captured from a computer running a different operating system. Most of these numbers' true purpose are not known, only that they are necessary.

```
#define FISH(a,b,c,d) \
 i = usb_control_msg (serial−>dev, \
   usb_rcvctrlpipe(serial−>dev,0), \
     b, a, c, d, buf, 1, 100); \
 dbg("0x%x:0x%x:0x%x:0x%x  %d - %x",\
     a,b,c,d,i,buf[0]);

#define SOUP(a,b,c,d) \
 i = usb_control_msg(serial−>dev, \
   usb_sndctrlpipe(serial−>dev,0), \
     b, a, c, d, NULL, 0, 100); \
 dbg("0x%x:0x%x:0x%x:0x%x  %d", \
     a,b,c,d,i);

  FISH (0xc0, 1, 0x8484, 0);
  SOUP (0x40, 1, 0x0404, 0);
  FISH (0xc0, 1, 0x8484, 0);
  FISH (0xc0, 1, 0x8383, 0);
  FISH (0xc0, 1, 0x8484, 0);
  SOUP (0x40, 1, 0x0404, 1);
  FISH (0xc0, 1, 0x8484, 0);
  FISH (0xc0, 1, 0x8383, 0);
  SOUP (0x40, 1, 0, 1);
  SOUP (0x40, 1, 1, 0xc0);
  SOUP (0x40, 1, 2, 4);
```

Figure 2: Original version of *pl2303.c*

```
#define FISH(a,b,c,d) \
 i = usb_control_msg (serial->dev, \
    usb_rcvctrlpipe(serial->dev,0), \
     b, a, c, d, buf, 1, 100); \
 dbg("0x%x:0x%x:0x%x:0x%x  %d - %x", \
     a,b,c,d,i,buf[0]);

#define SOUP(a,b,c,d) \
 i = usb_control_msg(serial->dev, \
    usb_sndctrlpipe(serial->dev,0), \
     b, a, c, d, NULL, 0, 100); \
 dbg("0x%x:0x%x:0x%x:0x%x  %d", \
     a,b,c,d,i);

FISH (VENDOR_READ_REQUEST_TYPE,
     VENDOR_READ_REQUEST, 0x8484, 0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
     VENDOR_WRITE_REQUEST, 0x0404, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
     VENDOR_READ_REQUEST, 0x8484, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
     VENDOR_READ_REQUEST, 0x8383, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
     VENDOR_READ_REQUEST, 0x8484, 0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
     VENDOR_WRITE_REQUEST, 0x0404, 1);
FISH (VENDOR_READ_REQUEST_TYPE,
     VENDOR_READ_REQUEST, 0x8484, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
     VENDOR_READ_REQUEST, 0x8383, 0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
     VENDOR_WRITE_REQUEST, 0, 1);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
     VENDOR_WRITE_REQUEST, 1, 0xc0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
     VENDOR_WRITE_REQUEST, 2, 4);
```

Figure 3: Current version of *pl2303.c*

3.7 No `ifdef` in .c code

With the wide number of different processors, different configuration options, and variations of the same base hardware types that Linux runs on, it is very easy to start having a lot of `ifdef` statements in your code. This is not the proper thing to do. Instead, place the `ifdef` in a header file, and provide empty inline functions if the code is not to be included.

As an example, consider the code in *drivers/usb/hid-core.c* as shown in Figure 4.

Here the author does not want to call `hiddev_hid_event()` if a specific configuration option is not enabled. This is because that function is not present if the configuration

```
static void hid_process_event (struct hid_device *hid,
                               struct hid_field *field,
                               struct hid_usage *usage, __s32 value)
{
        hid_dump_input(usage, value);
        if (hid->claimed & HID_CLAIMED_INPUT)
                hidinput_hid_event(hid, field, usage, value);
#ifdef CONFIG_USB_HIDDEV
        if (hid->claimed & HID_CLAIMED_HIDDEV)
                hiddev_hid_event(hid, usage->hid, value);
#endif
}
```

Figure 4: Original version of `drivers/usb/hid-core.c`

include/linux/hiddev.h:

```
#ifdef CONFIG_USB_HIDDEV
        extern void hiddev_hid_event (struct hid_device *,
              unsigned int usage, int value);
#else
        static inline void hiddev_hid_event (struct hid_device *hid,
              unsigned int usage, int value) { }
#endif
```

drivers/usb/hid-core.c:

```
static void hid_process_event (struct hid_device *hid,
                               struct hid_field *field,
                               struct hid_usage *usage, __s32 value)
{
        hid_dump_input(usage, value);
        if (hid->claimed & HID_CLAIMED_INPUT)
                hidinput_hid_event(hid, field, usage, value);
        if (hid->claimed & HID_CLAIMED_HIDDEV)
                hiddev_hid_event(hid, usage->hid, value);
}
```

Figure 5: After removal of `ifdef` in *drivers/usb/hid-core.c*

option is not enabled.

To remove this `ifdef`, the changes shown in Figure 5 were made.

If `CONFIG_USB_HIDDEV` is not enabled, the compiler replaces the call to `hiddev_hid_event()` with a null function call, and then optimizes away the if statement entirely. This keeps the code readable and is much easier to maintain.

3.8 Labeled Elements in Initializers

`gcc` allows the use of labeled elements in initializers. This means that structures that are initialized at compile time can have the individual field names used to specify what fields to set. For example, if the `struct foo` structure was defined as:

```
struct foo {
        int a;
        int b;
        int c;
};
```

any static definition of a variable of this type would traditionally be written as:

```
static struct foo bar =
    {A_INIT, B_INIT, C_INIT};
```

With the `gcc` extension, this initialization could also be written as:

```
static struct foo bar = {
        a: A_INIT,
        b: B_INIT,
        c: C_INIT,
};
```

which is a lot more descriptive. If a field is not specified with a specific value, the compiler sets that field to zero.

The kernel is filled with large structures, and lots of them are initialized at compile time. Previously, if someone added a new field in a structure, any variables that were declared like the previous example would break. For example, if the `struct foo` structure was changed to be:

```
struct foo {
        int a;
        char a1;
        int b;
        int c;
};
```

Any place that did not use labeled elements would break.

A good example of this, is any file that declares a `struct file_operations` variable. Generally, you do not want to define all fields of this structure, but rely on the VFS core to handle the majority of operations. The file *drivers/char/raw.c* has two good examples of named initializers:

```
static struct file_operations
raw_fops = {
    read:     raw_read,
    write:    raw_write,
    open:     raw_open,
    release:  raw_release,
    ioctl:    raw_ioctl,
};

static struct file_operations
    raw_ctl_fops = {
        ioctl:  raw_ctl_ioctl,
        open:   raw_open,
};
```

The code in Figure 6 from *arch/ia64/sn/io/hcl.c* is a good example of how much overhead is involved if you do not use this style of code.

This file will have to be updated every time the `struct file_operations` structure changes in the future.

4 Conclusion

The Linux kernel consists of a very large amount of source code, contributed by hun-

```
struct file_operations hcl_fops =
{
  (struct module *)0,
  NULL,         /* lseek - default */
  NULL,         /* read */
  NULL,         /* write */
  NULL,         /* readdir - bad */
  NULL,         /* poll */
  hcl_ioctl,    /* ioctl */
  NULL,         /* mmap */
  hcl_open,     /* open */
  NULL,         /* flush */
  hcl_close,    /* release */
  NULL,         /* fsync */
  NULL,         /* fasync */
  NULL,         /* lock */
  NULL,         /* readv */
  NULL,         /* writev */
};
```

Figure 6: *arch/ia64/sn/io/hcl.c*

dreds of developers over many years. Since the majority of this code follows some simple and basic style and formatting rules, the ability for people to quickly understand new code has been greatly enhanced. If you want to contribute to this code, please read the *Documentation/CodingStyle* guidelines and follow them in your patches and new code. The "unwritten" rules can be just as important as the written ones, when you are trying to convince people to accept your contributions, and should be followed just as closely.

5 Trademarks

IBM is a trademark of International Business Machines Corporation.

Linux is a trademark of Linus Torvalds.

Other company, product or service names may be trademarks or service marks of others.

This work represents the view of the author and does not necessarily represent the view of IBM.

References

[1] Soloway, Elliot, and Kate Ehrlich. 1984. "Empirical Studies of Programming Knowledge", IEEE Transactions on Software Engineering SE-10, no. 5 (September): 595-609

[2] `http://www.tuxedo.org/~esr/jargon/`

An AIO Implementation and its Behaviour

Benjamin C. R. LaHaise
Red Hat, Inc.
bcrl@redhat.com

Abstract

Many existing userland network daemons suffer from a performance curve that severely degrades under overload conditions to the point of collapse. The design of AIO was such that it should maintain a steady response rate when faced with a multitude of outstanding connections. With AIO for Linux becoming ready for more widespread use, the real world performance characteristics of the design and its shortcomings needs to be examined. What follows is an attempt to characterise the behaviour of async poll and read under various conditions, contrasting with poll(), and /dev/epoll.

1 Introduction

With the maturing of the Linux kernel, the scalability of various subsystems is becoming a greater concern for vendors in the quest for enterprise adoption. The advent of TUX [TUX] has shown that very high thruput content delivery systems can be built on top of the kernel's internal infrastructure, yet implementing these servers in userspace frequently exacts a large hit in performance.

Traditional IO models based on non blocking IO using select() or poll() have serious shortcomings when faced with loads that include many idle connections (HTTP, FTP, LDAP servers), or attempt to extract increased parallelism from slow subsystems (filesystem and disk IO). For example, the poll() syscall is at best O(n), where n is the number of file descriptors on which events are being monitored. Issuing parallel disk IO requests requires the use of threads, which have their own overhead, in addition to adding a significant amount of debugging effort to the application.

The Asynchronous IO implementation presented attempts to address these concerns twofold: by providing asynchronous operations that can proceed concurrently with the application, as well as utilising an event based completion mechanism to return the results of those operations in an efficient manner.

Previous work under Linux in this area includes the addition of SIGIO [SigIO] based readiness notification, reductions in the overhead of the poll() interface through the creation of /dev/poll [devpoll], and further optimizations with the event based /dev/epoll [EPoll]. The AIO implementation presented here should have similar performance characteristics to the event interface that /dev/epoll uses, as both models have a 1-1 correlation between events being generated and the potential for progress to be made.

One area where AIO poll differs significantly from /dev/epoll stems from readiness vs ready state notification: an async poll is like poll in that the operation completes when the descriptor has one of the specified events pending. However, /dev/epoll only generates an event when the state of the monitored events changes. Further differences emerge once the async read and write operations are introduced.

2 AIO Design and API

The basic design of AIO for Linux is based on the POSIX AIO [PosixAIO] specification and NT's completion port mechanism [Russinovich]. Primary design goals included:

1. lightweight completion events
2. usable for libraries as well as applications
3. support for general disk and network IO
4. scalable for servers handling many connections
5. the desire to eventually suppport zero copy io (O_DIRECT disk io, and hardware checksum assist for TCP transmit)
6. a 64 bit kernel should be able to process structures from both 32 and 64 bit processes with minimal additional code

POSIX AIO fails to meet several of these criteria, in part due to its reliance on signals, as well as the nature of its io wait mechanism (aio_suspend is O(n) where n is the number of outstanding ios).

The core of this implementation centers around the io context which specifies a given completion queue. io contexts are created by io_queue_init and destroyed via io_queue_release. New ios are submitted via io_submit (which is similar to lio_listio), but can only be queued if there is sufficient space in the completion queue to receive any resulting io_event.

Events are read by means of io_getevents. One of the features of the design is that io_getevents can be implemented as a vsyscall, which reduces the overhead of receiving completion events under load.

3 Testing Methodoloy

The goals of testing are to highlight the strengths and shortcomings of the various IO models. To this end, the areas of interest examined include thruput under varying numbers of open file descriptors, thruput with differing message sizes, and the effects of increasing the parallelism in request processing.

To demonstrate the scaling issues involved when dealing with many file descriptors, a simple test application [PipeTest] was developed. Pipetest attempts to measure the number of token passes per second that a given io model can obtain under a set of conditions. The number of idle file descriptors, parallel tokens passes and message size are all parameters. In operation, pipetest opens a specified number of pipes, begins transmitting one or more seed tokens, then proceeds to receive and transmit the tokens for a number of repetitions.

Initial plans were to use TCP network connections between a set of clients and a server, but due to code maturity and other issues, the decision to use a pipe based test was made. Thankfully, use of the pipe mechanism eliminates several potential bottlenecks (including IO bandwidth and driver performance), and restricts measurements to the actual overhead of the code under test. For all test runs, pipetest was able to run at 100% CPU usage.

For the sake of simplicity, and to avoid SMP scaling issues, all tests were run on the 2.4.19pre5 kernel with /dev/epoll and AIO patched in.

4 File Descriptors vs Thruput

It is well known that one of the crippling factors for heavily used server processes comes from the number of active client connections

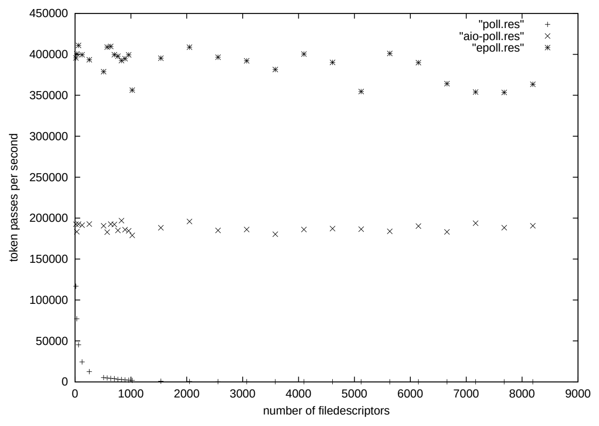


Figure 1: baseline performance vs number of file descriptors

being serviced. To demonstrate this factor on scaling, tests were run where the number of open pipes increased while other factors were held constant. Using Pipetest, the number of token passes per second was measured as a function of the file descriptor count and IO model.

The baseline performance as shown in Figure 1 contains several striking features, most notably that the existing poll() model exhibits a rapid decay as the number of active file descriptors increases. This stems from poll()'s O(n) workload in searching for active file descriptors. Async poll remains flat as the number of fds increases, as does epoll. The overhead of epoll appears to be about half of async poll, which points to a few shortcomings in the allocation and initialization of the async poll structures.

5 A few optimizations

In an attempt to reduce the overhead present in async poll relative to epoll, a fastpath that does not allocate any control data structures leads was created. In Figure 2 we can see this leads to an approximately 20% improvement in thruput. The addition of the io_getevents

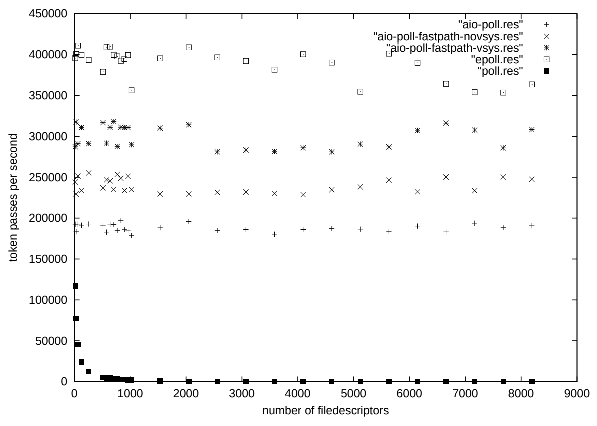


Figure 2: addition of poll fastpath and vsyscall mechanism to aio

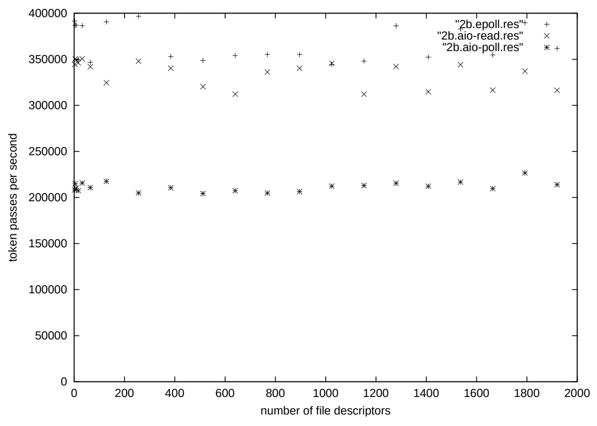


Figure 3: async read implemented

vsyscall lead to 20% increase in performance, bringing async poll to roughly three quarters of epoll thruput.

6 Async read

Figure 3 compares async read to epoll and async poll thruput. It should be noted that async read overhead includes walking the page tables to find the underlying kernel pages for the user virtual address. Since the token write() is performed after the async read is posted, async read benefits from a single copy of the data. This brings async read throughput to

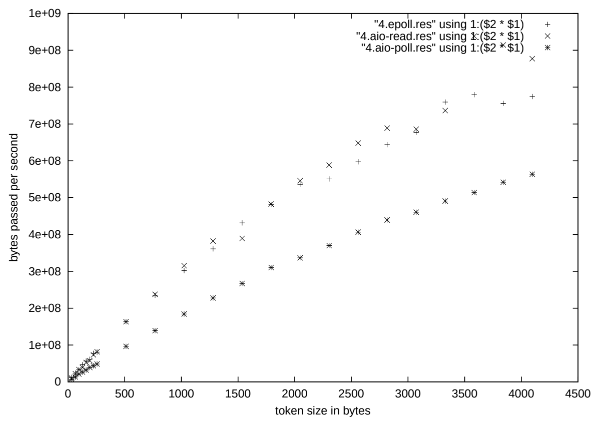


Figure 4: thruput as message size increases

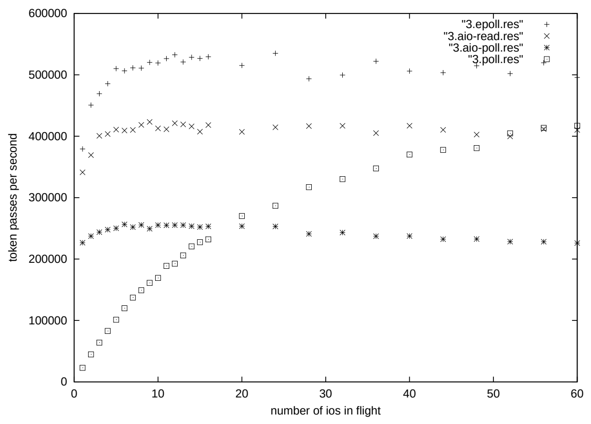


Figure 5: thruput while increasing in flight ios

within 15of epoll in the worst case. Cache effects are much more apparent for async read and epoll, probably owing to the differences in physical page colours between runs. As expected, async read also maintains a flat response when the number of open file descriptors increases.

7 Message Sizes

While epoll/read is faster than async reads for small message sizes, async read should become more efficient as the size increases and the benefits of the single copy begin to outweigh the static setup costs.

In Figure 4 it is apparent that for message sizes of 256 bytes or less, the async read overhead outweighs the effects of single copy. However, for message sizes of 512 bytes and greater, async read is able to exceed epoll thruput, but only by a small margin.

8 Multiple IOs in flight

The tests presented thus far only deal with one in flight IO through each iteration of the event loop. In real world servers, the number of IOs in flight will tend to increase with the number of active clients.

Figure 5 shows the results of a run with the number of IOs through to 60, 128 file descriptors and a 12 byte message size. The event driven models show a small increase in thruput up to 122 IOs, then remain fairly flat. Poll is included to show that it takes at least 15% of the file descriptors to have IOs in flight to match async poll performance, and almost half to match async reads.

9 Conclusions

The first generation of AIO for Linux shows that the event driven model is able to provide significantly improved performance in cases where poll() degrades severely. Comparisons with /dev/epoll show that further work needs to be done to mitigate the cost of mapping userspace memory into kernel structures for small message sizes, but that the overhead is outweighed by the support of single and zero copy as the amount of data transferred increases.

10 Future work and directions

Much work remains to complete the implementation, as many existing parts of the kernel were not designed with asynchronous operation in mind. The first year of development has yielded significant insight into the benefits and drawbacks of various in kernel techniques for the AIO implementation.

For example: during the recent addition of a cancellation API for iocbs, the limitations of using tqueues as the basis for the worktodo helpers became apparent. It turns out that tqueues cannot be cancelled safely in an SMP environment. One option is to make use of tasklets, but that path is hindered by the fact that many internal APIs cannot be called from bottom half context.

References

[TUX] *TUX Web Server Manuals*, Red Hat, Inc. `http://www.redhat.com/docs/manuals/tux/`, (2002).

[RTSig] *Analyzing the Overload Behavior of a Simple Web Server* N. Provos, C. Lever, S. Tweedie, `http://www.usenix.org/publications/library/proceedings/als2000/full_papers/provos/provos_html/index.html`, (ALS 2000).

[EPoll] *Improving (network) I/O performance*, Davide Libenzi `http://www.xmailserver.org/linux-patches/nio-improve.html`, (2001).

[Russinovich] *Inside I/O Completion Ports*, Mark Russinovich `http://www.sysinternals.com/ntw2k/info/comport.shtml`, (1998).

[SigIO] To Be Added.

[devpoll] To Be Added.

[PosixAIO] To Be Added.

[PipeTest] To Be Added.

Testing Linux® with the Linux Test Project

Paul Larson
Linux Technology Center
International Business Machines
plars@us.ibm.com

Abstract

The Linux Test Project is an organization aimed at improving the Linux kernel by bringing test automation to the kernel testing effort. To meet this objective, the Linux Test Project develops test suites that run on multiple platforms for validating the reliability, robustness, and stability of the Linux kernel. The LTP test suite is designed to be easy to use, portable, and flexible enough that tests could be added without requiring the developer to use functions provided by the LTP test driver. This paper covers what the Linux Test Project is and what we are doing to help improve Linux. I also plan to cover the features provided by the test harness, the structure of the test cases, and how test cases can be written to contribute to the Linux Test Project.

1 Introduction

Through the years of Linux development, many people have asked the question, "What is being done to test Linux?" Historically, Linux testing efforts have been primarily informal and ad-hoc in nature. Users of Linux simply use it for their own normal purposes and report any problems they find. Little has been done to bring any organized testing effort to Linux. This matter improved somewhat in May 2000 when Silicon Graphics Inc. (SGI™) introduced the first version of the Linux Test Project (LTP). Since that time, many individuals in the open source community as well as companies such as IBM®, OSDL™, and BULL® have contributed to the LTP.

2 LTP Test Scripts

One of the design goals of the Linux Test Project was to make it easy to use. To facilitate this, the LTP includes three scripts for executing subsets of the automated tests. They are:

- runalltests.sh – runs all the automated kernel tests in sequential order
- network.sh – runs all the automated network tests in sequential order
- diskio.sh – runs the stress_floppy and stress_cdrom tests

The runalltests.sh script can be executed with little or no manual setup required by the user. Even though the script is named "runalltests" it does not really run every test in the LTP, but it does run almost all of them. It runs all of the non-destructive and completely automated tests that do not require the user to perform manual setup tasks. Destructive tests and tests that consume so many system resources that they are designed to be run independently, such as a few of the memory test programs, are not included in the runalltests script.

The network.sh script includes most of the network tests. These are grouped separately because additional setup is required for these tests to function correctly. Two test machines are necessary to run all of the network tests. Both machines should have the same version of LTP compiled and installed in the same location. The client machine is the one where the network.sh script is actually executed. On the server machine, a .rhosts entry must be created for the root user to allow connections from the client machine. The following services will need to be running for successful execution of the network test suite: rlogind, ftpd, telnetd, echo (stream), fingerd, and rshd. More detailed information about the setup for the machines running LTP may be found in the document "How To Run the Linux Test Project(LTP) Test Suite" [RunLTP].

The diskio.sh script is a small test set that runs two IO-intensive tests. One of these targets the cdrom drive and the other targets the floppy drive. For the cdrom test to run, a cdrom with data on it must be inserted in the cdrom drive. For the floppy stress test to run, a blank formatted floppy disk must be in the floppy drive.

3 Pan: The LTP Test Driver

The test driver for the Linux Test Project test suite is called pan. Pan can parse a file that lists the tests to be executed, execute them, and exit with 0 if all tests passed, or with a number indicating how many tests failed. The line from runalltests.sh that executes pan looks like this:

```
${LTPROOT}/pan/pan -e -S -a $$ -n $$
                    -f ${TMP}/alltests
```

The *-e* is necessary to tell pan to exit with the number of test programs that failed. By default it will ignore exit statuses, but it is generally useful to have pan run this way.

The *-S* option tells pan to run test programs sequentially as they are read from the command file. If this option is not specified, it will select test programs at random to run.

The *-a $$* in the command line tells pan the name of a file to use to store the active test names, pids, and commands being executed. The $$ is used here to have it use the current pid so that a unique file is used to store this information. This file is often useful for determining which test program was running last if the test machine hangs or crashes.

The *-n $$* in the command line is a tag name by which this pan process will be known. It is required and should be unique so $$ is convenient to use again.

The *-f* option is used to tell pan the name of a command file to execute test programs from. The command file is a text file containing one test per line. The first item on the line is the tag name of the test, by which pan will know it. Usually this should match the TCID, or test case ID, of the test program. After the tag name and a space should be the executable with any necessary arguments. These files are usually stored under the runtest directory of LTP, but in the case of runalltests, several have been concatenated together into a file called alltests. These command files are a convenient way of grouping test programs together to create custom test suites.

Another useful option for pan that is not used in the provided scripts is *-s*. The *-s* option tells pan the number of test programs to run before exiting. If 0 is used here, pan will keep executing tests until it is manually stopped.

The *-t* option can be used to specify the amount of time pan should run test programs. This time can be specified in seconds (s), minutes (m), hours (h), or days (d). For instance, *-t 12h* would tell pan to stop executing tests after 12 hours.

A complete list of options for pan can be found in the man page for pan in the /doc/man1 [LTPMan] directory under LTP.

4 Organization of Test Programs

Test programs may also be executed individually without the need for running them under pan or from a script. Once compiled, the test programs are linked to under the /testcases/bin directory from the top of the LTP source tree. Test programs may be executed directly from here with any valid command line options. This feature is very useful when a particular test program is observed to cause an error. The test program can be executed alone to reproduce the error without having to wait for the entire test suite to run.

Sometimes it is desirable to modify test programs slightly for debugging purposes, or to add additional testing to them. To help make it easier to find test programs, they have been organized under the testcases directory into four main categories.

- Kernel – Kernel-related tests such as filesystems, io, ipc, memory management, scheduler, and system calls
- Network – Network tests including tests for ipv6, multicast, nfs, rpc, sctp, and network related user commands
- Commands – Tests for user level commands commonly used in application development such as ar, ld, ldd, nm, objdump, and size
- Misc – Miscellaneous tests that do not fit into one of the other categories such as crash (an adaptation of the well-known crashme test), f00f, and a floating point math set of tests

Other test programs that are not part of the automated test scripts previously mentioned can also be found under this directory tree.

5 Developing Tests for LTP

The Linux Test Project was designed to be flexible enough to allow test programs to be added to it without requiring the use of any cumbersome features that are specific to a certain test driver. The LTP does provide a small set of functions that can be used to help with the consistency of test programs and to act as a convenience for the developer, but the driver does not require their use. Test programs written to be executed under the LTP should be self-contained so that they can be executed under pan or separately. They should be able to detect within the test program itself whether or not each test case passed or failed. The test program should return 0 if all test cases in it pass or anything else if any of them fail. The exact nature of the failure indicated by return codes other than 0 may be different from one test program to another. Most of the test programs in LTP are written in C, but they may be written in perl, shell scripting languages, or anything else as long as appropriate return values are preserved. This flexibility allows developers to take any quick test program they have written to test something, make sure it returns 0 if it passes or anything else if it doesn't, and submit it for inclusion in the LTP.

Some of the functions in LTP make use of global variables that define various aspects of the test case. Even if it is unknown whether or not these functions will be used, it is a good idea to define these variables in order to be consistent with other test cases in the LTP.

The TCID variable should be defined in a way similar to the example below:

```
char *TCID="test01";
```

The convention that has been used in other test cases in the LTP is the system call name, or some other name representing the test, followed by a two digit number. The TCID should be different from that used by any other LTP test case or results may be confusing after executing all the tests in the test suite. It is also a good idea to make the TCID be the same as the name of the source code file for the test program. In this example, the file name should be something like *test01.c*.

The global variable `TST_TOTAL` is of type int and should be used to specify the number of individual test cases within the test program. A test program should output a line for each test case declaring whether the test passed or failed.

The `Tst_count` variable is used as a test case counter in the main test loop:

```
extern int Tst_count;
```

The output functions provided by LTP use `Tst_count` to get the number of the test case currently being executed. This should be automatically incremented each pass through the test loop.

The main test loop is just a for loop, but it implements a macro called `TEST_LOOPING()` to control the number of iterations through the loop.

```
for (lc=0; TEST_LOOPING(lc);
     lc++) {
...
}
```

Standard command line options for LTP test cases allow the user to set a certain number of iterations or an amount of time to run each test program. `TEST_LOOPING()` handles making sure that the test program is executed for the correct number of iterations, or for the correct amount of time.

The actual test itself should be wrapped in the `TEST()` macro. `The TEST()` macro starts by resetting the errno variable to 0 to ensure that the correct errno is detected after the test is complete. After executing the system call passed to it, `TEST()` sets two global variables. `TEST_RETURN` is set to the return code and `TEST_ERRNO` is set to the value of errno upon return. There is also a variation of the `TEST()` macro called `TEST_VOID()` for use with testing system calls that return void.

Test programs that require little or no manual setup are preferred. Usually setup can be performed within the test program itself, or with command line options that can be passed from the execution script. If manual setup is required, the test program may be left out of automated execution scripts, or grouped with other test programs that have similar setup requirements such as those found in the network test suite.

Many test programs require a temporary directory to store files and directories created during the test. This is especially true of filesystem tests, and tests of system calls that operate on files and directories. The `tst_tmpdir()` and `tst_rmdir()` functions provide a convenient method of creating and cleaning up a temporary area for the test program to use.

The `tst_tmpdir()` function creates a unique, temporary directory based on the first three characters of the TCID global variable. Once the directory is created, it makes it the current working directory and returns to continue execution of the test program. The name of the directory created will be saved in an extern char* variable called `TESTDIR` for possible use by the test case, and for later removal by the `tst_rmdir()` function. If it is unable to create a unique name, unable to create the directory, or unable to change directory to the new location `tst_tmpdir()` will use `tst_brk()` to output a `BROK` message for all test cases in the test program and exit via the `tst_exit()` function. Since no cleanup function will be performed automatically in this situation, `tst_tmpdir()` should only be used at the beginning of the test program before any resources that would require a cleanup function have been created.

The `tst_rmdir()` function will remove the temporary directory created by a call to `tst_tmpdir()` along with any other files or directories created under the temporary directory. The `system()` function is used by `tst_rmdir()` so the test case should not perform unexpected signal handling on the `SIGCHLD` signal.

One of the biggest conveniences provided by using the Linux Test Project API is `parse_opts()`. The `parse_opts()` function provides a consistent set of useful command line options for test cases, and allows the developer to easily add more options.

```
#include "test.h"
#include "usctest.h"

char *parse_opts(int argc,
     char *argv[],
     option_t option_array[],
     void (*user_help_func)());

typedef struct {
    char *option;
    int  *flag;
    char **arg;
} option_t;
```

Option_array must be created by the developer to contain the desired options in addition to the default ones. `User_help_func()` is a pointer to a function that will be called when the user passes `-h` to the test case. This function should display usage information for the additional options added only. If you do not wish to specify any additional command line options, `parse_opts()` should be called with NULL for *option_array* and `user_help_func()`.

The default options provided by parse_opts are:

-c *n* – Fork n copies of this test and run them in parallel. If -i or -I are also specified, each forked copy will run for the given number of iterations or amount of time respectively.

-e – Log all errnos received during the test.

-f – Suppress messages about functional testing

-h – Print the help message listing these default options first, then call `user_help_func()` to display help for any extra options the developer may have added.

-i *n* – Run the test for n consecutive iterations. Specifying a 0 for n will cause the test to loop continuously.

-I *x* – Run the test loop until x seconds have passed.

-p – Wait to receive a `SIGUSR1` before beginning the test. `TEST_PAUSE` must be used in the test at the point you want it to wait for `SIGUSR1`.

-P *x* – Delay x seconds between iterations.

Another useful feature of the Linux Test Project API is that it provides functions to output results and give test status in a consistent manner, and exit the test program with an exit code consistent with the results from that output. This paper will not cover all of the functions to do this but will briefly discuss the most common ones.

All of these functions need to be passed a *ttype* that specifies the type of message that is being sent. The available values for ttype are:

- TPASS – Indicates that the test case had the expected result and passed
- TFAIL – Indicates that the test case had an unexpected result and failed
- TBROK – Indicates that the remaining test cases are broken and will not execute correctly because some precondition was not met such as a resource not being available.
- TCONF – Indicates that the test case was not written to run on the current hardware or software configuration such as machine type, or kernel version.
- TRETR – Indicates that the test case has been retired and should not be executed any longer.
- TWARN – Indicates that the test case experienced an unexpected or undesirable event that should not affect the test itself such as being unable to clean up resources after the test finished.
- TINFO – Specifies useful information about the status of the test that does not affect the result and does not indicate a problem.

The first result output function is `tst_resm()`.

```
void tst_resm(int ttype,
        char *tmesg,
        [arg ...])
```

This function will output *tmesg* to `STDOUT`. The *tmesg* string and associated arguments can be given to `tst_resm()` and the other functions listed here in the same fashion as strings with arguments can be passed to `printf()`. After outputting the message, the test case will resume.

```
void tst_brkm(int ttype,
        void (*func)(),
        char *tmesg, [arg ...])
```

The `tst_brkm()` function prints the message specified by *tmesg*, calls the function pointed to by *func*, and exits the test program breaking any remaining test cases.

```
void tst_exit()
```

The `tst_exit()` function exits the test program with status depending on *ttype*s passed to previous calls to functions such as `tst_brkm()` and `tst_resm()`. For `TPASS`, `TRETR`, `TINFO`, and `TCONF` the exit status is unaffected and will be 0 indicating that the test passed. `TFAIL`, `TBROK`, and `TWARN` all indicate that something went wrong during the test or that the test failed. Using these values at any point in the test program before `tst_exit()` is called will cause `tst_exit()` to exit the test program with a non-zero status.

When a test cases receives an unexpected signal, it is useful to provide a means of making it exit gracefully. The LTP provides a convenient way of doing this through the `tst_sig()` function.

```
#include "test.h"

void tst_sig(fork_flag,
     handler, cleanup)
```

```
char *fork_flag;
int (*handler)();
void (*cleanup)();
```

If the test case is creating child processes through functions such as `fork()` or `system()`, then `tst_sig` needs to know to ignore `SIGCHLD`. This can be accomplished by setting *fork_flag* to `FORK`. If the test case is not creating child processes, *fork_flag* should be set to `NOFORK`. Keep in mind that if the test program uses `tst_tmpdir()` and `tst_rmdir()`, the *fork_flag* should be set to `FORK` because `tst_rmdir()` uses the `system()` library call.

The handler parameter of `tst_sig()` represents the function that will be called when an unexpected signal is intercepted. The developer may provide a custom signal handler function here that returns int, or the default signal handler may be used. To use the default signal handler for `tst_sig()`, pass `DEF_HANDLER` as the *handler* parameter to `tst_sig()`. If the default handler is used, then the *TCID* and *Tst_count* variables must be defined. The default handler will use `tst_res()` to output messages for all remaining test cases that were incomplete when the signal was received.

The cleanup parameter is used to specify a cleanup function. After the handler has been executed, `tst_sig()` will execute the cleanup function. The cleanup function should take care of removing any resource used by the test program such as files or directories that were created to facilitate testing. If nothing is required for cleanup, `NULL` can be passed to `tst_sig()` in place of a cleanup function.

6 The Future of LTP

Most of the future plans for the Linux Test Project focus on expanding test coverage. The majority of test cases in the LTP today test system calls. This is, of course, a very important part of testing Linux, but not the only thing that should be addressed. Some test programs have already been added for things such as networking, memory management, scheduling, commands, floating point math, and databases, but the breadth of test coverage should continue to expand. As the variety of tests increases, it may one day become necessary to modularize the LTP test programs into separate suites that can be executed and even downloaded separately. The LTP was reorganized to make this easier if and when it becomes desirable to do so.

In addition to broader coverage, a desirable feature is a wider range of test subsets. One example of this that has often been discussed is a predefined set of test programs and different options passed to test programs to create a stress suite. This could be further subdivided into components such as a memory stress suite, a scheduler stress suite, a filesystem stress suite, and so on.

Another project for future development is a front end for defining a customized set of test programs and executing them. There is already a simple menu for launching some of the test suites under the LTP, but something more advanced is desired. Ideally, a good front end should allow the user to select from all available test programs and see a description of all of them. It should also allow the user to define how long to run the suite, how many instances to have running at once, and other options. Finally, it should allow exporting and importing of profiles so that customized test executions can be reproduced later.

Currently, functions are provided by LTP to create test cases with a consistent output format, style, and command line options. These functions are only provided for C though. Tests may be developed in other languages, or even in C, but not make use of these functions and still work under the LTP test suite. The functions are provided as a convenient way of gaining a consistent set of features found throughout most of the test programs in the LTP test suite. To encourage more test development by developers who prefer other languages and want to make use of these functions, they may be implemented in languages such as perl, python, or others upon popular request.

The LTP has recently made significant progress towards running on other architectures besides x86. The majority of test programs in the LTP test suite have been made to work on architectures such as IA64, PPC, and S390. A small amount of work may still be needed for the LTP on a few of these architectures, but the LTP test suite is executed frequently on all of these architectures. As additional equipment becomes available, this list may be expanded to include other architecture targets that are capable of running Linux. Some changes may be necessary to make the LTP test suite run cleanly on these other systems.

The completeness of current test cases should also be analyzed and improved upon if necessary. We are currently looking at code coverage analysis tools to determine how much of the target kernel code is being executed by test cases in the LTP. As we find areas of kernel code that are not adequately covered by test cases in the LTP, new test cases are written or existing ones are modified to expand coverage to these areas. The tools that are being developed to do this will not only allow us to see what areas of the kernel are covered by each test program, but will also allow kernel developers to find test cases that target a specific area of the kernel. So, if changes are made in the kernel and the developer wants to find test programs that will target that area of the code, they can search for the specific test programs that will do that.

7 Enhancing the LTP

Additional test programs are of course critical to the test suite, but for the Linux Test Project to be truly effective, people must use it. The Linux Test Project encourages and appreciates the contributions of anyone in the open source community who wishes to participate. It would be nice to see the LTP test suite run as part of the exit criteria for releasing new kernels in the stable and development trees. In addition to this, it would be useful for kernel developers to execute the test suite against patches before submitting them. The LTP test suite will not find all problems but may reduce the number of errors in new code if used properly. If kernel developers and testers diligently submit test programs for defects as they are found, the test suite could even help reduce the number of regressed defects found in Linux.

The LTP is taking steps to encourage more community involvement. Results of testing done by the LTP are posted on the LTP website at http://ltp.sourceforge.net and on the ltp-results mailing list. Requests for testing can also be submitted on the ltp-results list. An additional mailing list exists for the purpose of discussing development of the LTP test suite. The LTP test suite is also available as a testing tool inside the STP test tool at OSDL (http://www.osdl.org/stp).

References

[LTPMan] *The Linux Test Project Man Pages* Linux Test Project. `http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/ltp/ltp/doc/` (2000)

[Howto] Nate Straz, *Linux Test Project HOWTO* Linux Test Project. `http://cvs.sourceforge.net`

`/cgi-bin/viewcvs.cgi/ltp/ltp/doc/` (2000)

[RunLTP] Casey Abell and Robbie Williamson, *How To Run the Linux Test Project(LTP) Test Suite* Linux Test Project. `http://ltp.sourceforge.net/ltphowto.php` (2001)

Disclaimer and Trademarks

Security Policy Generation through Package Management

Charles Levert
Open Systems Lab
Ericsson Research
8400 Décarie Blvd.
TMR (Qc) Canada, H4P 2N2
Charles.Levert@ericsson.ca

Michel Dagenais
Dept. of Computer Eng.
École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal (Qc) Canada, H3C 3A7
Michel.Dagenais@polymtl.ca

Abstract

Generation and maintenance of security policies is too complex and needs simplification for it to be widely adopted and thus truly make a difference in delivering the promise of more secure computing systems (rather than just being ignored by administrators).

In practice, one of the great obstacles to the adoption of security measures in system software is the complexity of configuration that it entails. Yet, information captured by software package management systems is mostly not relayed to security configuration.

This paper covers the investigation to:

- Identify useful information already coded in packages from various package management systems (RPM, dpkg), as well as translation mechanisms to reuse this information.
- Identify missing information that would best be specified by the package integrator and included in each package.
- Identify the remaining information that is mostly site-specific and that would best be specified by a local administrator.
- Prototype the coding of the resulting design ideas.

The approach taken follows these principles: simplicity of design, best security practices as default behavior (i.e., no or minimal configuration/specification required, use of common patterns), flexibility, and least privilege (at each phase: installation, configuration, activation, and execution). It builds on existing parts of the Linux system landscape, without imposing a total revolution: package management systems, the init process and init script system, file system standards and file placement conventions, as well as current security efforts such as SE Linux (to express and enforce the policy).

1 Introduction

"Complexity is the worst enemy of security." [18]

A package management system provides a structured way to install and de-install software on a computer system. It maintains a database that accounts for (ideally) all files that are not user data on the system. Package management really came of age with operating systems built around the Linux kernel, although other systems with less functionalities, such as the `pkg` system available on Solaris, predate them.

A security policy is an explicit set of rules that govern (the configurable part of) the behavior of a system's security features.

The currently unresolved problem that is the subject of this paper is the following. Software package management systems, such as RPM, already capture much information about the nature of the files that make up a package and about the interactions of a package with other packages, yet this information is mostly not relayed to security configuration. It is quite possible that the complexity of configuration stems from the current requirement to specify, by hand, configuration information that is redundant with what could already be gleaned from packaging information.

Specific parts of the configuration information naturally correspond to the software itself, other parts to its inclusion via a package in an operating system, and yet other parts to a site-local installation. Currently, in the Linux and open source software world, too much of this configuration is unnaturally pushed to the local installation and its human manager. Since security configuration information is not strictly needed for software to perform its main task, it is often left unspecified, which in practice leads to a wide open system from a security standpoint.

It is assumed that the roles of system administrator and security administrator are distinct. Hence, the responsibilities for each of these roles should, as much as possible, center around the very nature of each. For example, software installation by itself should not grant the necessary privileges to activate services with a security impact. Conversely, security administration should not be concerned with the cumbersome details of software installation. In practice, total separation between the two roles may be impossible. However, there is still a gain in security from attempting to separate the two, if only because the human beings that assume these roles are better sensibilized about the responsibility and possible consequences for every action they take.

Note that this paper is not about packaging a security framework in itself. This other important issue is being addressed elsewhere [4].

This paper is organized as follows. The first few sections review existing parts of Linux systems that are pertinent to our endeavor. Section 2 reviews pertinent features of package management systems. Section 3 reviews existing execution control schemes for server processes. Section 4 reviews file system standards and file placement conventions. Section 5 reviews how existing security frameworks are configured. The remaining sections explore how we approach the problem stated in this introduction. Section 6 exposes proposed additions to per-package information. Section 7 then does the same for site-specific information. Section 8 suggests modifications to existing software. Finally, Section 9 covers a prototype implementation.

This work is done in support of the development of the Distributed Security Infrastructure (DSI) [17, 20] open-source effort that is targeted for use in carrier-class (telecom) clusters.

1.1 Basic Principles

- **Simplicity.** All design and code, whether they implement security or non-security features, contribute to the total security of a product. Security vulnerabilities creep in with ordinary bugs when design or code are not produced and then verified (audited) with a security-minded approach. By far the best way to ease this process is to keep things as simple as possible: to have design and code that do only one thing at once, to limit the size and number of functions and modules, to specify things in only one place, etc. This applies to the definition of a security policy framework.

- **Default behavior.** Whenever possible, no explicit security policy rule should have to be specified when a situation is the most typical one. Furthermore, custom startup scripts for server programs with typical behavior should not have to be provided. Good practices, from a security standpoint, should be used as default. On the other hand, bad practices should require explicit manual configuration from the security administrator, so as to discourage them. Security relevant effects (e.g., permission modifications) should be made to be the result of existing actions when that is what one would expect from these actions (e.g., service activation).

- **Flexibility.** If a system prevents its users from accomplishing their tasks, it won't be used. For the security features of a system, it means being turned off, which cancels out all their usefulness. Therefore, there must always be a way to specify behavior that deviates from the default.

  Lack of flexibility can also impose an abrupt transition from an old way of doing things to a new one. This is turn can cause this new way to never take off the ground. Deployment of package managers on Linux systems is pervasive. For the security related package management changes that will be advocated later in this document to be adopted, they must account for flexibility.

- **Least privilege.** The different tasks that are accomplished in relation to a given software package each require their own minimal set of security privileges. These tasks (or phases of system activity) include: installation/De-installation/upgrade, configuration, service assignment/activation, and regular use/execution.

2 Features of Existing Package Management Systems

Package management systems already carry information about their content (installed files, mainly) that can be relevant to the generation of a security policy. Although several package managers are considered in this section, most of the rest of this paper will focus more on RPM.

2.1 Red Hat Package Manager (RPM)

RPM [13] is used by Red Hat distributions, as well as others such as SuSE and Mandrake. All information specified by the maintainer of an RPM package is included in a `*package*.spec` file inside the `.src.rpm` source version of the package. Configuration files can be explicitly designated as such in the `.spec` file of a package. There are several possible declarations (or directives, or file attributes):

- The `%config` directive is used to flag the specified file as being a configuration file.

- `%config(missingok)` indicates that the file need not exist on the installed machine. It is frequently used for files like `/etc/rc.d/rc2.d/S55named` where the existence of the symbolic link is part of the configuration in `%post`, and the file may need to be removed when the package is removed. The file is not required to exist at either install or de-install time.

- `%config(noreplace)` indicates that the file in the package should be installed with extension `.rpmnew` if there is already a modified file with the same name on the installed machine.

(Parts of these descriptions are plain transcripts from [1].)

There is also the `%ghost` file attribute. It indicates that the file is not to be included in the package. It is typically used when the attributes of the file are important while the contents is not (e.g., a log file).

The package format itself is a binary that can be handled using the `rpm` library (and the include file `<rpm/rpmlib.h>`). It begins with a header composed of several tag-and-value pairs. Headers tags are 32-bit integers (`RPMTAG_*`), so extensions should be possible. File tags (`RPMFILE_*`) appear as individual bits in an integer, so extensions should also be possible, but there are much less free bits available and conflicts are likely with future versions.

Each package has the opportunity to provide various scripts to be run before and after the installation and de-installation steps.

2.2 Debian's dpkg

Under the Debian Packaging scheme, binary packages are distributed in a single file with a `.deb` extension. This file is an ar archive that itself contains a file named `control.tar.gz`. This file is in turn an archive that contains plain text files, including one named `control` and possibly one named *`package`*`.conffiles`. This last file contains a list of installed files, one per line, that are configuration files.

The specification for the `.deb` file allows for the future inclusion of new members and explicitly defines the behavior that current programs that manipulates those files should take in order to maintain backward and forward compatibility. This makes it easier to add features to dpkg while allowing for a smoother transition.

As for RPM, each package has the opportunity to provide various scripts to be run before and after the installation and de-installation steps.

2.3 OpenPKG

OpenPKG [6] is somewhat a clone of RPM. It targets systems that are not Linux based such as Solaris and FreeBSD, as well as Debian Linux. Support for some `.spec` tags has been left out, but that does not include any of the tags that are of interest in RPM for the purpose of this investigation.

OpenPKG otherwise includes as an extension an "`rc` script" that provides centralized application control.

3 Existing Execution Control Schemes for Server Processes

Most Linux distributions rely on an execution control scheme for server processes that is inherited from System V Unix. Under that scheme, the system operates at any given time under a specific run level, which is represented by a small integer that takes its value from a predefined set of values with specific meanings. Each run level defines which services should be activated and which should not. Each service is expected to provide a script that can be instructed to start the service, to stop it, and possibly to inquire about the current status of the service, to restart it, etc. Each script also usually provide suggested run levels under which the service should run, a 2-digit sequence number for service startup, and a 2-digit sequence number for service shutdown. These are provided under a `chkconfig:` field that is located in a commented-out line at the beginning of the script. Every time the run level changes, service are started and stopped in the order that is defined by those sequence numbers. (Often, both sequence numbers are chosen such that they add up to 100, so that shutdown is done in the reverse order as startup). The `chkconfig` utility command is used to configure the activation of services at various run levels. To facilitate writing these scripts, commonly used shell functions are available from a single file that can be sourced. There is no default behavior facility that would prevent having to provide these scripts and their suggested information, even for simple services that fit a common pattern.

These scripts are usually stored in the `/etc/rc.d/init.d` or `/etc/init.d` directory.

As these scripts are provided by the package that implements a given service, they are related to package management. An init script is not always provided by the author of the software that is started by the script. Indeed, this software may have initially been targeted at another type of system, such as BSD, that does not rely on these scripts. In such a case, the init script for that software is contributed by the distributor or third-party packager.

At this point in time, there is no strong coordination between the various distributors as to the meaning of the various run level values, the choice of sequence numbers, the utility shell functions that are provided, or even the instructions beyond "start" and "stop" that can be given to the scripts. As a result, the scripts for the same service that come with different distributions (e.g., Red Hat and SuSE) will not be compatible, and hence the packages themselves won't be compatible (even if they agree to use the same dynamic libraries and file locations).

The Linux Standard Base (LSB) effort [11] now attempts to standardize many aspects of system initialization in general, init scripts in particular. The LSB standardizes run level definitions and init script actions. It also introduces a smarter way to determine the order in which the scripts should be run when the run level changes. It is based on "Provides: " and "Required-*: " declarations, and the new notion of facility names that refer to generic services rather than specific ones provided by a package. Some commonly used functions and the location of the file that scripts should source has also been standardize. A few other details are standardized by LSB, but they are not related to security. The recommendations of the current version of the standard, 1.1.0, are not currently implemented by major distributors such as Red Hat.

The Linuxconf configuration and activation system [8] adds a few conventions for init scripts. It supports the following information fields: autoreload, processname, pidfile, config, probe, description, and override [9]. Note that the config field is then another, redundant way in which configuration files are explicitly identified to the system (see Section 2).

This init scripts way of doing service startup and shutdown is also beginning to show its age. For instance, there is now a need for tight packet filtering rules that need to be changed dynamically at service startup and shutdown. This information is currently not provided in the package, be it in the startup script or elsewhere.

Moreover, it is less than obvious that the init script provided with a package should be trusted to actually perform a stop order. If this is a security concern, a framework needs to be devised to make sure that the service is actually stopped, and perhaps even that its permissions are revoked.

The init process also has the capability to directly launch services. It has the additional ability to monitor and restart them if necessary. Behavior of the init process is configured through the `/etc/inittab` file. Typical services that are put under init control include:

- getty processes that are started on consoles and serial lines to display a login prompt;
- the xdm process that is started on an X Window System console.

The init process is what actually manages the run level on a Linux system.

On Debian, dpkg includes a wrapper named `start-stop-daemon` that features the following security-relevant command-line options:

- `-chuid` changes the user ID before executing the daemon process.
- `-chroot` changes the current directory and then changes the root of the file system to it so that the daemon process is jailed.

This wrapper is commonly used by Debian startup scripts.

Another system is D. J. Bernstein's daemontools and the `/service` directory on which it relies [2]. It obviates the need for a `/var/run/`*`name`*`.pid` file and has the ability to monitor and restart services to insure higher availability. It relies on standard UNIX features to accomplish its task.

Service availability monitoring can be performed at different levels, but it only needs to be performed once. These possible levels are the process such as `init` that starts the service, the init script that wraps around the service, or the service itself (by forking into a monitoring process and a service process).

4 File System Standards and File Placement Conventions

Various Linux distributions and other UNIX and UNIX-like systems reserve directories for specific purposes. They have naming conventions for sub-directories and files within those directories. The conventions also cover the kind of files (i.e., their purpose) that should be stored in these directories as well as the ownership and access rights that they should have. Depending on the operating system, these conventions are more or less stated explicitly.

In the Linux world, there exists a common standard for this known as the Filesystem Hierarchy Standard (FHS) [10].

There are also other, independent proposals. D. J. Bernstein's `/package` hierarchy [3] goes all the way and reuses the file system itself as the database for package management.

This is related to conventions in package management and security policy configuration. Indeed, if

- a given directory serves a very specific purpose,
- conventions related to security policy are

in place for this directory, and

- there is the notion that a package's name automatically reserves a subset of the naming space for subdirectories and files within that directory,

then this in itself defines default, clear rules for security policy that can be made to embody good security practices. This removes complexity as there is then no need to specify package-specific rules for the purpose served by that directory, for most packages.

There is an opportunity to extend the set of such directories. For instance, creation of temporary files or named sockets in the `/tmp` directory has historically been the source of many security vulnerabilities. This is because this directory is a public space, no subset of it is reserved to a specific package, and a complex set of steps is then required to make sure that the temporary resource is created securely. These `/tmp` problems could all be avoided by the introduction of conventions that are properly enforced by default on the system.

The downside of such conventions is that they are difficult to adopt instantaneously. The old way of doing things must be supported for some time, while still providing incentives to move to the new, provably secure way.

5 Configuration of Existing Security Frameworks

There are two extreme approaches to specifying the security privileges that a software package (and its various components) may enjoy.

- The privileges are entirely specified by the package itself. The act of installing the package by the system administrator implicitly carries the approval of these stated privileges. The problem is that those privileges can be complex and are not restricted to anything. They are unlikely to be reviewed by the administrator. Moreover, they have to be expressed in the terms of each specific security framework that is to be supported.
- The privileges are entirely specified outside of the package by the security framework. The problem is that all possible packages have to be accounted for in advance. If they are not for a given package, the security administrator has to specify privileges for it by hand. This either lacks flexibility or is unrealistic.

The solution lies in abstracting the whole set of permissions that a package requests in a form that fits in a very small space (e.g., less than a line) and have the system or security administrator approve that explicitly. To that end, there must be a way to express these abstractions and they must be installed/activated beforehand by the administrator.

In order to express them, however, we must gain an idea of the kind of permissions that different existing security frameworks provide. We will examine two popular frameworks, but there are others [16, 21, 19, 7, 14, 12] (these are the ones for Linux).

5.1 Security Enhanced (SE) Linux

SE Linux [15] configuration is performed in two steps. First, an utility named `setfile` is used to assign a security context to every file on the system. This is done using a configuration file (`file_contexts`) that uses regular expressions to full paths of file. This configuration file has been broken down into several *`name`*`.fc` files, one for each covered package, and a `types.fc` file for all other patterns. This configuration operation can be performed during the initial installation of SE Linux, before the system is actually running under it. It can also be performed while running under SE Linux. It is easily conceivable that the operation could be customized to only touch files that are part of a package at package installation time. Note that file names are no longer used once security contexts are assigned to all inodes when the system is running.

The second part of SE Linux configuration is the security policy itself (`policy.conf`), which is then compiled into a binary form under `/ss_policy` and read by the kernel. Type Enforcement rules have been broken down into several files (*`name`*`.te`), one for each covered package, and is combined with other files to form the whole security policy. The security policy has to be reloaded as a whole. This complicates (or at least make more heavyweight) what can be done at package installation time.

5.2 SubDomain

SubDomain [5] relies on a configuration that directly uses the full path names of files. Configuration profiles are stored in the `/etc/subdomain.d/` directory under the name of the program that is executed and subject to control. Sub-processes are covered in a recursive fashion by the syntax. A user-space utility relies on a `sysctl()` interface to feed these rules to the kernel-space part of SubDomain. Add, delete, and replace operations are supported, which means that updates to the in-kernel policy should be possible at package installation time.

6 Proposed Additions to Per-Package Information

6.1 Implicit, Enforced Conventions

As seen in Section 2, package management systems support many file tags to distinguish, e.g., configuration files, from other files. From a security standpoint, one may wish to introduce more tags to identify files that are specifically related to other phases of system activity, such as service activation, as detailed in Section 6.2.

An alternative to this approach is to designate specific locations (typically directories) to necessarily contain files of a given type (i.e., corresponding to what would have been new tags).

In both case, extension mechanisms would be desirable to either add new custom tags, or equivalently to designate new locations. This idea is further expanded upon in Section 6.3 with the idea of generic “abstract” packages.

6.1.1 Naming of Packages

The naming of packages tends to follow some unwritten conventions.

- Packages in a group of related packages normally share a common prefix, although there is nothing to formally separate the prefix from the rest of the name. (Hyphens can be used anywhere else inside a name, be it in a prefix or in a suffix, if any.)
- For packages within a group with that share the same prefix, commonly used suffixes include: `-common`, `-util`, `-apps`, `-tools`, `-extra`, `-devel`, `-client`, `-server`, `-lib`, `-contrib`, `-doc`, `-perl`, `-python`,

and `-X11`. Some of these suffixes are sometimes found in a plural form. Some packages even have several suffixes (e.g., `openssh-askpass-gnome`).

- To complicate matters, some suffixes are not preceded by a hyphen, e.g., `kdebase` and `kdelibs`. Fortunately, these specific suffixes do not appear to be relevant to the security nature of a package.

There is no internal representation of this inside the package.

If these naming conventions were clearly represented, it would be possible to assign a security policy semantic meaning to them. Some package name suffixes, such as `-util`, imply that the executables contained in the package must not carry or be given any special privileges. Conversely, a package with `-server` as a suffix contains executables that should be given specific privileges when activated to provide the service for which they were written. Although this was originally done to enable only the client or only the server to be installed, it is a good security practice to isolate a server executable and its related files in such a package, provided that the opportunity to assign specific privileges is taken. It is possible that several executable files be included in a server package, one being the main server and the others being there for support (to be executed as sub-processes of the main server). In anticipation of such a case, there must be a clear way to tell which executable is the main server.

6.2 Explicit New Facilities

"Any problem in computer science can be solved with another layer of indirection."
— David J. Wheeler.

An indirect mapping is introduced between the actual package name and the service name (e.g., TCP port), to which other information can be coupled (interface, address subset, etc.).

This separates the act of installing a package that can implement a service from the act of designating it as being currently responsible for doing so (and thus receiving the necessary privileges for doing so). Conceptually, this designation can be done in a finer grained manner. For instance, different providers for a given Internet service could be enabled for each network interface (internal/trusted and external/untrusted).

Each `(transport_protocol, port_number)` pair should have its own set of security contexts by default. Explicit configuration can be useful to put a group of ports in a single set of security contexts. SE Linux [15] relies on statements like

```
tcp 25 system_u:object_r:smtp_port_t
```

that have to be configured by hand. This could be generated automatically from the `/etc/services` file. The SE Linux syntax also allow for a range of ports (e.g., `22-23`) to be specified. Specifying a port range requires explicit configuration in all cases, but it is not a frequent occurrence.

The tasks (or phases of system activity) include the following.

- **Installation/de-installation/upgrade.** This is performed by the package manager itself. When installing files and running package-provided scripts, the package manager should minimize its permissions (e.g., by forking a subprocess) so as to only be able to modify the sub-part of the system that is appropriate for the package. A strong file placement standard can tremendously simplify the interpretation of this statement. An installation should not interfere with other packages that are already installed or that could be installed in a sub-part of the system that is reserved for them. Package-provided scripts should not be able to use services that are not strictly related to installation, such as network ports.

A package that implements a given service (e.g., SMTP) should not, just by virtue of it being installed, be able to activate (designate) itself at the provider for this service. This is a separate action to be performed by the system administrator.

Typical steps for installation are:

  - Check for validity of package name, type, etc. The package type may require to be specified explicitly by the administrator to signify approval of the permissions that are inherent to it.
  - Add package-specific installation rules to the security policy;
  - perform actual install under the security context that is defined by those rules (including file copy and script execution).
  - Add package-specific security rules for other tasks.

The security policy rules are inferred from the package name and type. They are not specified by the package itself.

- **Configuration.** Globally, the responsibility for configuration can be assigned to a dedicated management package. This package can then delegate its authority to application-specific configuration packages. This means that a package is not automatically responsible for its own configuration. By default, it should not even be able to probe various unrelated part of the system during installation and execution to adjust its behavior accordingly. Configuration packages may need and be given such permissions, though.

- **Service assignment/activation.** Several packages can implement the same service (e.g.: sendmail, qmail, and postfix are all SMTP mail transport agents). It may be desirable to have more than one installed at once (testing, transition), yet at most one can be assigned the same responsibility. Service activation can be performed by the package manager (by explicit instruction from the administrator, not from the packager) or it can be performed by a service activation manager software, with can possibly delegate its task to more specialized service activation packages (e.g., Internet service activation manager).

- **Regular use/execution.** During regular use, software from a package should be able to read (and only read) its configuration (in `/etc`, possibly store some state in `/var`, etc. Depending on whether it is a service package, an utility package, or other, it can also get other specific permissions, or inherit those of its invoking process. It should not implicitly be able to modify its own installation, configuration, or activation, though.

6.3 Generic Package Types

Examples of generic package types include the following:

- network server programs
- local service programs (e.g., gpm)
- utilities (which require no special permissions other than those passed by their parent process)
  - read-only viewer/browser

  - strict filters

- installer programs
- configurator programs
- activator programs
- security session managers (program that set up a specific security context for others)
- etc.

6.3.1 Generic Abstract Packages

Generic "abstract" packages (named after the object-oriented concept of abstract class) are incomplete packages that merely include a default init script or configuration files. In effect, they define a generic package type. Using this facility, a generic service package could be able to provide an activation script (or declaration), whereas a specific service package could not because such scripts implicitly carry the definition of access rights to be handed to the specific package. Instead, the specific package can be declared as being of the generic package type defined by the generic service package.

"Multiple inheritance" of generic package types by specific packages should be disallowed by default as it can cause problems related to the combination of specific power. Installation of generic packages should stand out and require special attention from system administrators as they effectively imply the permission to install packages that follow the pattern they describe. This, in turn, means that there must be a way to explicitly identify these package as such.

6.3.2 Framework Packages

Some packages specify a framework (e.g., logrotate) under which other packages can register, but only under their own name. This is traditionally done by a `/etc` subdirectory (e.g., `/etc/logrotate.d`). Other frameworks could introduce a `/var` subdirectory instead.

From a security standpoint, a method is required to explicitly label this subdirectory. The package manager must then only allow packages to register there under their own name.

6.3.3 Sample Packages Types

Here are typical permissions that are needed by two sample package types.

The execution of a network server requires the permissions to (among others):

- read its own configuration file(s)
- produce its own pid file (that can also be handled by the availability monitor)
- listen to its assigned service (protocol/port)
- append to own log file (or use log service under its own name)
- spawn modules (possibly under another security context)

Software installation requires the following permissions (among others):

- to install program under same *`name`* or *`name-*`*
- to create and populated subdirectories of same name under `/usr/lib`, `/usr/share`, etc.

- to install an initial configuration file named `/etc/`*`name`*`.conf` or placed under *`name`* in `/etc/sysconfig/`

6.4 Self Restrictions

A package should be able to manage its own private space, such as private directories, by imposing additional restrictions through the use of policy rules. In order to do so, the rules should be expressible in a relative syntax that does not require the redundant mention of the package name. Conversely, it should not be possible to specify rules outside of that package scope. This applies to all phases of system activity for that package.

7 Site-Specific Information

7.1 Default Policy for Generic Package Types

In order to reduce the size of the site-local security configuration, each generic package type must be configurable.

For comparison purposes, SE Linux [15] relies on a system of macros to reduce the complexity of the type enforcement policy files it includes for each special user program and server program.

7.2 Per-Package Configuration

Additional restrictions (e.g., read-only server, local non-networked server) should be easily configurable as site-local options.

8 Proposed Modifications to Existing Software

The previous sections have pinpointed their requirements for many modifications to existing software. These are gathered here for every piece of software that is involved. We try not to introduce new programs, but rather to push the additional security checks into existing programs, at the point where they naturally belong.

The following assumes that dynamic updates to the security policy are possible.

- `/bin/rpm` (and other package managers). Explicit service activation scripts (default preferred), distinct from installation scripts, should be introduced, along with command-line options to specify that a service should be activated at installation time. Naming conventions for packages should be enforced; e.g., utils packages should not include executables with special permissions. Special security attributes for files included in a package should be supported; e.g., the server executable in a server package should be identified as such. Namespace conventions in the file system, as well as conventions introduced by framework packages, have to be enforced. New meta-information can be supported by introducing a new, extended,backward compatible, version of the package format. Alternatively, separate package-specific meta-information files can be used to augment the information present in existing packages. Either way, the package manager has to be able to interpret the new information.

- `/sbin/init`. Provide default init script behavior based on package declarations, process tracking, dependable stops (and restarts), and automatic cleanup of temporary storage (e.g., `/var` storage) to prevent keeping state across invocations (if appropriate). This can involve management of `/var/lock/subsys/`*`name`* files.

- `/sbin/service`. This program should have its own security context and it should perform the task that is currently handled by `run_init` in SE Linux. Init scripts should no longer be run by specifying their full path from anywhere in a distribution, but rather by invoking this program systematically. The program should be rewritten in C, rather than being an interpreted script.

- `/sbin/chkconfig`. Activating a service means enabling the initial transition into a package-specific security context (as is currently done in SE Linux with the `domain_auto_trans()` macro and `type_transition` rules). To guard against cooperating malicious packages where one transition into the other, the notion of defined but forbidden security context could be introduced for de-activated services (SE Linux has `neverallow`, but it is an assertion that can cause a policy to be rejected at compilation, and not an actual rule).

- `/sbin/telinit`. Since some services are only activated for a subset of the available run levels, their associated security context will need to be allowed or forbidden according to the current run level. Note that `/sbin/telinit` and `/sbin/init` are usually the same binary.

- Configuration programs (such as `/sbin/linuxconf`). Configuration files for a program should be put in a security context that is not accessible for modification by the program itself. Configuration programs should be able to transition a subprocess into a security context that can modify those files, as well as restart the program if it is an activated service.

9 Prototype Implementation

The implementation of the ideas exposed in this paper is at a very early stage. Since this work is done in support of the Distributed Security Infrastructure (DSI), which is an open source project, feedback from the community is important before work proceeds to actual implementation. (As of writing, the actual presentation for this paper is two months in the future and progress will have been made on the prototype by then.)

The goals of the prototype implementation are to assess the practicality of the proposed changes and to measure their performance impact on the system.

10 Conclusion

We have explored the possibility of generating a system's security policy, or at least part of it, from the information that is or can be encoded in software packages that are installed on the system.

The approach described in this paper impacts many people: distribution makers, packagers, software designers and implementers, security framework developers, and system administrators.

Since security is a very sensitive subject, community review of this kind of work is primordial. Also, since this work involves modifying many existing subsystems, building commitment from the community is essential.

This work highlights the following requirements on the security policy:

- Dynamic updates. Long running systems cannot afford to be rebooted. A complete reload of the security policy at every one of its modifications also doesn't scale well with the size of the policy itself (which is proportional to the number of packages that are installed on the system).

- Elaborate security contexts. Associated package, run levels, etc., need to be represented in the security contexts to avoid overly complex rules or unnecessary updates to the policy. A balance must be achieved between those concerns and the complexity of the security contexts themselves (and of their evaluation).

References

[1] Edward C. Bailey. *Maximum RPM*. Sams, 1997. `http://www.rpm.org/local/maximum-rpm.tar.gz`.

[2] D. J. Bernstein. Daemontools and its `/service` directory. `http://cr.yp.to/daemontools.html`.

[3] D. J. Bernstein. The `/package` hierarchy. `http://cr.yp.to/slashpackage.html`.

[4] Russell Coker. Packaging NSA SE Linux for Debian. In *Proceedings of 2002 Ottawa Linux Symposium*, Ottawa (On) Canada, June 2002. `http://www.linuxsymposium.org/2002/`.

[5] Wirex Communications. SubDomain. `http://www.immunix.org/subdomain.html`.

[6] Ralf S. Engelschall and Michael Schloh von Bennewitz. OpenPKG. `http://www.openpkg.org/`.

[7] Tal Garfinkel and David Wagner. Janus. `http://www.cs.berkeley.edu/~daw/janus/`.

[8] Jacques Gélinas. Linuxconf. `http://www.solucorp.qc.ca/linuxconf/`.

[9] Jacques Gélinas. Linuxconf Enhanced System V Init Script. `http://www.solucorp.qc.ca/linuxconf/tech/sysvenh/index.html`.

[10] Free Standards Group. Filesystem Hierarchy Standard (FHS). `http://www.pathname.com/fhs/`.

[11] Free Standards Group. Linux Standard Base (LSB). `http://www.linuxbase.org/spec/`.

[12] Serge Hallyn. DTE for Linux. `http://www.cs.wm.edu/~hallyn/dte/`.

[13] Red Hat. Red Hat Package Manager (RPM). `http://www.rpm.org/`.

[14] NAI Labs. Low Water-Mark Integrity Protection for Linux (LOMAC). `http://www.pgp.com/research/nailabs/secure-execution/lomac.asp`.

[15] National Security Agency (NSA). Security Enhanced (SE) Linux. `http://www.nsa.gov/selinux/`.

[16] Amon Ott. Rule Set Based Access Control (RSBAC). `http://www.rsbac.org/`.

[17] Makan Pourzandi, Ibrahim Haddad, Charles Levert, Miroslaw Zakrzewski, and Michel Dagenais. A Distributed Security Infrastructure for Carrier Class

Linux Clusters. In *Proceedings of 2002 Ottawa Linux Symposium*, Ottawa (On) Canada, June 2002. `http://www.linuxsymposium.org/2002/`.

[18] Bruce Schneier and Adam Shostack. Results, Not Resolutions. `http://www.securityfocus.com/news/315`.

[19] Huagang Xie, Philippe Biondi, and Steve Bremer. Linux Intrusion Detection System. `http://www.lids.org/`.

[20] Miroslaw Zakrzewski. Mandatory Access Control for Linux Clustered Servers. In *Proceedings of 2002 Ottawa Linux Symposium*, Ottawa (On) Canada, June 2002. `http://www.linuxsymposium.org/2002/`.

[21] Marek Zelem, Milan Pikula, and Martin Ockajak. Medusa DS 9 Security System. `http://medusa.formax.sk/`.

Scalability of the Directory Entry Cache

Hanna Linder
IBM Linux Technology Center
hannal@us.ibm.com   http://www.ibm.com/linux

Dipankar Sarma
IBM Linux Technology Center
dipankar@in.ibm.com   http://www.ibm.com/linux

Maneesh Soni
IBM Linux Technology Center
maneesh@in.ibm.com   http://www.ibm.com/linux

Abstract

This paper presents work that we have done to improve scalability of the directory entry cache (dcache). We investigated scalability problems resulting from many cache lookups, global lock contention, a possibly non-optimal eviction policy, and cacheline bouncing due to global reference counters. This paper provides an overview of solutions we tried, such as fast path walking, utilizing the read-copy update mutual exclusion mechanism [McKenney], and lazy updating of the LRU list of dentries. We conclude with performance results showing scalability improvements.

1 Introduction

Every file and directory has a path. The path must be followed to do a lookup in the dcache to get the correct inode number of the file. A path such as /etc/passwd contains three dentries: '/', 'etc', and 'passwd'. Each dentry in a lookup path has a reference counter called d_count, which is atomically incremented and decremented as the dcache is being checked. This keeps the dentry from being put on the least recently used (LRU) list.

Currently, the dcache is protected by a single global lock, dcache_lock. This lock is held during lookup of dentries (d_lookup) as well as all manipulations of the dentry cache and the assorted lists that maintain hierarchies, aliases and LRU entries. The global dcache_lock seems to be an issue as the number of CPUs increase. We experimented with various ways to improve scaling the dentry cache.

2 Workload and Measures

We have used three main workloads for measuring scaling of the dentry cache: dbench [Pool] (with settings to avoid I/O), httperf [Mosberger], profiles [Hawkes] of Linux(R) kernel compiles, and lockmeter [Hawkes2]. The system used is an 8-way Pentium(R)-III Xeon(TM) with 1MB L2 cache and 2 GB of RAM (unless otherwise noted).

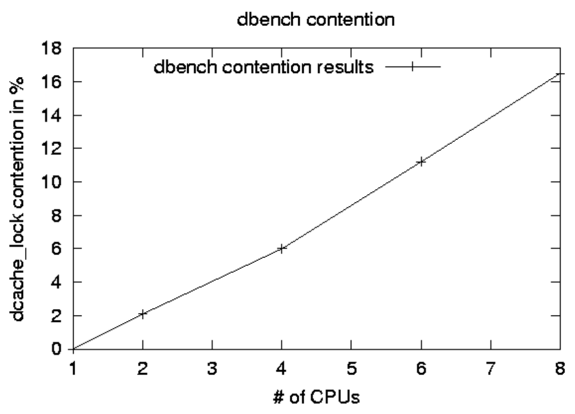


Figure 1: Baseline contention with dbench

2.1 Summary of Baseline Measurements

The baseline measurements show that dcache_lock suffers from an increasing level of contention for some benchmarks. Although other locks such as the Big Kernel Lock (kernel_flag) and lru_list_lock are much higher in the total contention numbers, once those are dealt with, dcache_lock will move up the list.

The following work focuses on ways to increase scalability of the dcache. While looking at the distribution of lock acquisitions for these workloads, it becomes obvious that d_lookup() is the routine to optimize since it is the routine where the global lock is acquired most often.

2.2 Dbench Results of Baseline

The dbench results from our initial investigations [Sarma] show that lock utilization and contention grow steadily with an increasing number of CPUs. On an 8-way system running 2.4.16 kernel, dbench results show 5.3% utilization with 16.5% contention on this lock (see Figure 1). One significant observation with the lockmeter output is that for this workload d_lookup() is the most common operation.

This snippet of lockmeter output for an 8-way in Table 1 shows that 84% of the time dcache_lock was acquired by d_lookup(). Out of about fifteen million holds of the dcache_lock, d_lookup() comprised twelve million of them. The simple explanation for this is that d_lookup is the main point into the dcache. It does the looping search to find the child of the given parent dentry in the hash, then atomically increments the d_count reference of the dentry before returning it, all while the dcache_lock is held.

Apart from contention, a large number of acquisitions of a global lock result in excessive bouncing of the lock cacheline in SMP machines as the number of CPU's increase. It is important to reduce contention as well as utilization of the global lock to achieve better performance.

2.3 Httperf Results of Baseline

The httperf results from our initial investigation show a moderate utilization of 6.2% with 4.3% contention in an 8 CPU environment.

A snippet of lockmeter output showing the distribution of acquisition of dcache_lock appears in Table 2.

This shows that 74% of the time the global lock is acquired from d_lookup(). Again, out of about twenty million acquisitions of the dcache_lock, d_lookup took fifteen million of them.

3 Avoiding Global Lock in d_lookup()

In the paper by Paul E. McKenney, Dipankar Sarma, and Orran Krieger [McKenney] they described the Read Copy Update mutual exclusion mechanism (RCU). To summarize, RCU provides support for reading an item without holding a lock and a special callback method

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
5.3	16.5	0.6	2787	5.0	3094	0.89	15069563	16.5	dcache_lock
0.01	10.9	0.2	7.5	5.3	116	0.00	119448	10.9	d_alloc+0x128
0.04	14.2	0.3	42	6.3	925	0.02	233290	14.2	d_delete+0x10
0.00	3.5	0.2	3.1	5.6	41	0.00	5050	3.5	d_delete+0x94
0.04	10.9	0.2	8.2	5.3	1269	0.01	352739	10.9	d_instantiate+0x1c
4.8	17.2	0.7	1362	4.8	2692	0.76	12725262	17.2	d_lookup+0x5c
0.02	11.0	0.9	22	5.4	1310	0.00	46800	11.0	d_move+0x38
0.01	5.1	0.2	37	4.2	84	0.00	119438	5.1	d_rehash+0x40
0.00	2.5	0.2	3.1	5.6	45	0.00	1680	2.5	d_unhash+0x34
0.31	15.0	0.4	64	6.2	3094	0.09	1384623	15.0	dput+0x30
0.00	0.82	0.4	4.2	6.4	6.4	0.00	122	0.82	link_path_walk+0x2a8
0.00	0	1.7	1.8	0			2	0	link_path_walk+0x618
0.00	6.4	1.9	832	5.0	49	0.00	3630	6.4	prune_dcache+0x14
0.04	9.4	1.0	1382	4.7	148	0.00	70974	9.4	prune_dcache+0x138
0.04	4.2	11	2787	3.8	24	0.00	6505	4.2	select_parent+0x20

Table 1: Lockmeter output for 8-way

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
6.2	4.3	0.8	390	2.7	579	0.12	20243025	4.3	dcache_lock
0.02	6.5	0.5	45	2.7	281	0.00	100031	6.5	d_alloc+0x128
0.01	4.9	0.2	4.6	2.9	58	0.00	100032	4.9	d_instantiate+0x1c
5.0	4.5	0.8	387	2.8	579	0.09	15009129	4.5	d_lookup+0x5c
0.02	5.8	0.6	34	3.1	45	0.00	100031	5.8	d_rehash+0x40
0.19	8.8	0.5	296	2.8	315	0.01	933218	8.8	dput+0x30
0.89	2.3	0.6	390	2.5	309	0.01	4000584	2.3	link_path_walk+0x2a8

Table 2: Lockmeter output, distribution of acquisition of dcache_lock

to update all references to the data when it is written.

The dcache_lock is held while traversing the d_hash list and while updating the Least Recently Used (LRU) list if the dentry found by d_lookup has a zero reference count. By using RCU we can avoid dcache_lock while reading d_hash list [1].

In this, we were able to do a d_hash lookup lock free but had to take the dcache_lock while updating the LRU list. The patch does provide some decrease in lock hold time and contention level. Table 3 shows lockmeter statistics on a 4-way SMP running the 2.4.16 kernel without any patches while running dbench.

Table 4 is the same dbench run with this first RCU patch applied.

Spinning on the dcache_lock via d_lookup went from 12.7% to 10.6%. This demonstrated that simply doing the lock-free lookup of the d_hash was not enough because d_lookup() also acquired the dcache_lock to update the LRU list if the newly found dentry previously had a zero reference count. This often was the case with the dbench workload, hence we ended up acquiring the lock after almost every lock-free lookup of the hash table in d_lookup().

From there we decided we needed to avoid acquiring dcache_lock so often. Therefore, we tried different algorithms to get rid of this lock from d_lookup(), such as a separate lock for the LRU list.

4 Per Bucket Lock for d_hash and d_lru Lists

The goal was to enable parallel d_lookup. We had to abandon this approach due to race conditions and complicated code. The problem was due to dcache having several additional lists apart from d_hash and d_lru that span across buckts. They are d_alias, d_subdir, and d_child, in order to modify or access any of these lists we would need to take multiple bucket locks. This resulted in a serious lock ordering problem which turned out to be unworkable [2].

5 Separate Lock for the LRU List

The motivation behind having a separate lock for the d_lru list was that as d_lookup() only updates the LRU list, we could relax contention on the dcache_lock by introducing a separate lock for LRU lists. This resulted in most of the load being transferred to the LRU list lock. Many routines held the dcache_lock as well, such as prune_dcache, select_parent, d_prune_aliases, because they read or write other lists apart from the LRU list [3]. Results appear in Table 5.

6 Lazy Updating of the LRU List

Given that lock-free traversal of hash chains did not significantly decrease dcache_lock acquisitions, we looked at the possibility of removing dcache_lock acquisitions completely from d_lookup(). After using RCU based lock-free hash lookup, the only remaining use of the dcache_lock in d_lookup() was to update the LRU list.

Our next approach was to relax the rules of an LRU list by allowing dentries with non-zero reference counts to remain in the list for a short delay before being removed in the update [4]. The beneficial side-effect was that multiple dentries could be processed during the update. Previously, the global dcache_lock was held then dropped for every single entry as each dentry was removed from the list during

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
6.3	9.2	0.4	1659	3.4	1648	1.3	23182304	9.2	dcache_lock
0.01	10.1	0.2	7.6	2.9	45	0.01	96649	10.1	d_alloc+0x124
0.03	11.0	0.2	70	2.9	316	0.01	184690	11.0	d_delete+0x10
0.04	8.8	0.2	95	2.7	175	0.01	281340	8.8	d_instantiate+0x1c
3.8	12.7	0.5	123	3.4	1648	0.80	10074944	12.7	d_lookup+0x58
0.02	9.9	0.8	24	2.8	56	0.00	37050	9.9	d_move+0x34
0.01	3.6	0.2	32	3.4	58	0.00	96639	3.6	d_rehash+0x3c
0.00	4.2	0.2	1.5	2.7	9.4	0.00	1330	4.2	d_unhash+0x34
2.3	6.4	0.3	120	3.3	1379	0.48	12336769	6.4	dput+0x18
0.00	5.2	2.0	882	3.9	50	0.00	3006	5.2	prune_dcache+0x10
0.02	4.8	6.1	836	3.2	23	0.00	5280	4.8	select_parent+0x18

Table 3: Lockmeter statistics, kernel 2.4.16 (unpatched)

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
4.3	7.5	0.3	1436	3.0	1222	0.88	23103201	7.5	dcache_lock
0.01	5.6	0.2	18	2.3	54	0.00	104404	5.6	d_alloc+0x128
0.03	8.1	0.2	20	2.4	322	0.01	184690	8.1	d_delete+0x10
0.04	6.9	0.2	30	2.2	79	0.01	289095	6.9	d_instantiate+0x1c
2.1	10.6	0.3	491	3.0	1222	0.54	9961665	10.6	d_lookup+0xd8
0.02	7.4	0.7	4.8	2.3	209	0.00	37050	7.4	d_move+0x34
0.01	3.4	0.2	4.8	3.0	43	0.00	104394	3.4	d_rehash+0x3c
0.00	2.5	0.2	1.3	2.9	8.6	0.00	1330	2.5	d_unhash+0x34
2.0	5.1	0.2	108	3.0	1080	0.32	12342240	5.1	dput+0x18
0.04	3.2	0.9	1436	3.1	74	0.00	65770	3.2	prune_dcache+0x140
0.02	4.1	6.6	926	2.7	8.3	0.00	5275	4.1	select_parent+0x18

Table 4: Lockmeter statistics, first RCU patch

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
3.7	5.7	0.3	1475	3.0	1551	0.63	22434872	5.7	d_lru_lock
1.7	7.9	0.3	90	3.1	1489	0.39	9956382	7.9	d_lookup+0xc8
2.0	3.9	0.2	144	3.0	1551	0.23	12346145	3.9	dput+0x18
0.04	2.8	0.5	22	3.5	79	0.00	127045	2.8	prune_dcache+0x150
0.03	3.6	9.2	1475	3.1	112	0.00	5300	3.6	select_parent+0x18
0.26	0.14	0.2	1474	1.7	204	0.00	1915750	0.14	dcache_lock
0.01	0.51	0.1	1.9	1.8	204	0.00	109702	0.51	d_alloc+0x124
0.02	0.15	0.2	9.3	2.6	169	0.00	184690	0.15	d_delete+0x10
0.03	0.16	0.1	11	1.6	57	0.00	294393	0.16	d_instantiate+0x1c
0.02	0.12	0.7	27	1.3	5.5	0.00	37050	0.12	d_move+0x34
0.01	0.12	0.1	61	1.7	3.5	0.00	109692	0.12	d_rehash+0x3c
0.00	0.23	0.1	1.6	1.1	1.7	0.00	1330	0.23	d_unhash+0x34
0.14	0.09	0.2	38	1.5	141	0.00	1099648	0.09	dput+0x4c
0.01	0.26	0.2	18	1.4	5.5	0.00	69655	0.26	prune_dcache+0x7c
0.03	0.26	8.7	1474	1.2	2.6	0.00	5300	0.26	select_parent+0x24

Table 5: Lockmeter statistics with separate lock for LRU List

the update.

To implement this new functionality, we introduced another flag (DCACHE_UNLINK) to mark the dentry for deferred freeing and a per-dentry lock (d_lock) in struct dentry to maintain consistency between the flag and the reference counter (d_count). For all other lists in struct dentry, the reference counter continued to provide mutual exclusion.

Allowing additional dentries to remain in the lru_list could lead to an unusually large number of dentries, causing a lengthy deletion process during updates. We proposed two different approaches to circumvent this problem:

1. Use a timer to kick off periodic updates.
2. Periodically update the d_lru list while already traversing it.

6.1 Timer Based Lazy Updating

A timer was used to remove the referenced dentries from the d_lru list so that it would be kept manageable. To take the dcache_lock from the timer handler we had to use spin_lock_bh() and spin_unlock_bh() for dcache_lock. This created problems with cyclic dependencies in dcache.h.

This approach did not prove to be any better than the non-timer approach. However, the patch is worth looking at as proper tuning of timer frequency may give better results [5].

6.2 Periodic Updates During Traversal

The d_lru list is made up to date through select_parent, prune_dcache and dput. While traversing the d_lru list in these routines, the dentries with non-zero reference counts are removed. This is the solution we chose to include in the lazy LRU patches due to its simplicity.

6.3 Notes on Lazy LRU Implementation

Per dentry lock(d_lock) is needed to protect the d_vfs_flags and d_count in d_lookup. There is very little contention on the per dentry lock, so this will not lead to a bottleneck. With this patch the DCACHE_REFERENCED flag does more work. It is being used to indicate the dentries which are not supposed to be on the d_lru list. Right now apart from d_lookup, the per dentry lock (d_lock) is used whereever d_count or d_vfs_flags are read or modified. It is probably possible to tune the code more and relax the locking in some cases.

We have created a new function include/linux/dcache.h: dentry_unhash() to delete a dentry from the d_hash list. It sets the DCACHE_UNLINK bit in d_vfs_flags, which marks the dentry for deferred freeing.

As we do lockless lookup, rmb() is used in d_lookup to avoid out of order reads for d_nexthash and wmb() is used in d_unhash to make sure that d_vfs_flags and d_nexthash() are updated before unlinking the dentry from the d_hash chain.

Every dget() marks the dentry as referenced by setting DCACHE_UNLINK bit in d_vfs_flags. This forced us to hold the per dentry lock in dget. Therefore, dget_locked is not needed.

6.4 Lazy LRU Patch Results

Contention for the dcache_lock reduced in all routines. However, the routines: prune_dcache and select_parent take more time because the d_lru list is longer. This is acceptable as both routines are not in the critical path.

We ran dbench and httperf to measure the effect of lazy dcache and the results were very good. By doing a lock-free d_lookup(), we were able to substantially cut down on the number of dcache_lock acquisitions. This re-

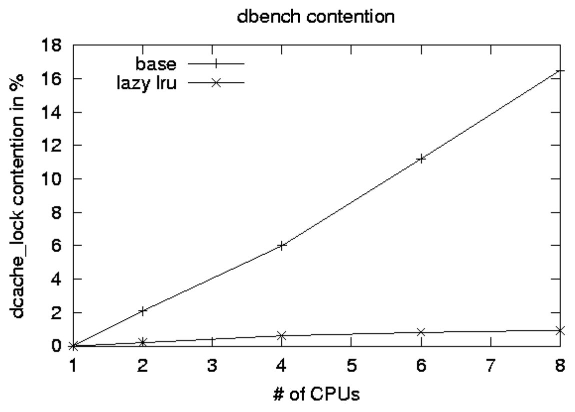


Figure 2: Lazy LRU contention from dbench

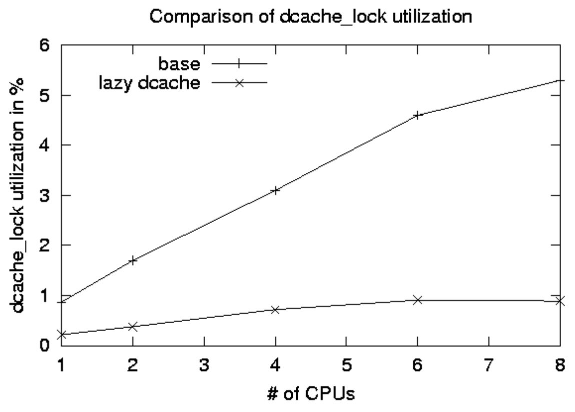


Figure 3: Lazy LRU dcache_lock utilization from dbench

sulted in substantially decreased contention as well as lock utilizations. Results appear in Table 6.

6.5 Dbench Results of Lazy LRU

dbench results showed that lock utilization and contention levels remain flat with lazy dcache as opposed to steadily increasing with the baseline kernel. So for 8 processors, contention level is 0.95% as opposed to 16.5% for the baseline (2.4.16) kernel.

One significant observation is that maximum lock hold time for prune_dcache() and select_parent() are high for this algorithm. However, these are not frequent operations for this workload. Although, this latency could be an issue with real time applications.

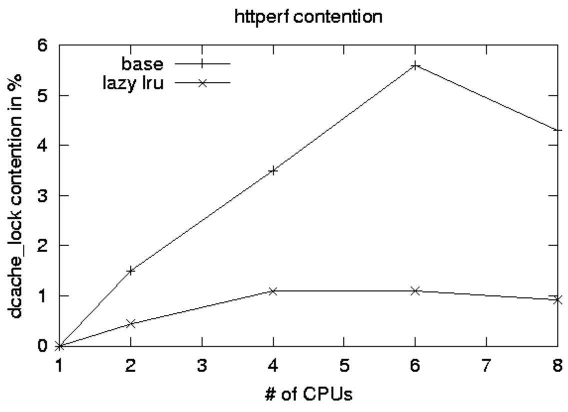


Figure 4: Lazy LRU contention from httperf

A comparison of baseline (2.4.16) kernel and lazy dcache contention and utilization while running dbench can be seen in Figures 2 and 3.

The throughput results show marginal differences (statistically insignificant) for up to 4 CPUs, of 1% (statistically significant) on 8 CPUs. There is no performance regression in the lower end and the gains are small in the higher end.

6.6 Httperf Results of Lazy LRU

The httperf results showed a similar decrease in lock contention and lock utilization. With 8 CPUs, it showed significantly less contention. See Table 7.

A comparison of the baseline (2.4.16) kernel and lazy dcache contention and utilization while running dbench can be seen in Figures 4 and 5.

The results of httperf (replies/sec for fixed connection rate) showed statisticially insignificant differences between base 2.4.16 and lazy dcache kernels.

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
0.89	0.95	0.6	6516	19	6411	0.03	2330127	0.95	dcache_lock
0.02	1.7	0.2	20	17	2019	0.00	116150	1.7	d_alloc+0x144
0.03	0.42	0.2	49	35	6033	0.00	233290	0.42	d_delete+0x10
0.00	0.14	0.8	12	3.4	8.5	0.00	5050	0.14	d_delete+0x98
0.03	0.40	0.1	32	34	5251	0.00	349441	0.40	d_instantiate+0x1c
0.05	0.30	1.7	44	22	1770	0.00	46800	0.30	d_move+0x38
0.01	0.16	0.1	21	4.5	334	0.00	116140	0.16	d_rehash+0x40
0.00	0.65	0.7	3.7	8.4	57	0.00	1680	0.65	d_vfs_unhash+0x44
0.56	1.1	0.7	84	18	6411	0.02	1383859	1.1	dput+0x30
0.00	0.88	0.4	2.3	1.3	1.3	0.00	114	0.88	link_path_walk+0x2d8
0.01	4.4	4.3	6516	4.8	32	0.00	3566	4.4	prune_dcache+0x14
0.07	2.3	1.8	6289	4.4	718	0.00	67591	2.3	prune_dcache+0x150
0.11	0.79	29	4992	28	1116	0.00	6444	0.79	select_parent+0x24

Table 6: The effect of lazy dcache

SPINLOCKS		HOLD		WAIT					
UTIL (%)	CON (%)	MEAN (μs)	MAX (μs)	MEAN (μs)	MAX (μs)	CPU (%)	TOTAL	SPIN (%)	NAME
1.4	0.92	0.7	577	2.2	617	0.00	4821866	0.92	dcache_lock
0.02	2.2	0.6	30	1.9	7.8	0.00	100031	2.2	d_alloc+0x144
0.01	1.7	0.2	12	2.2	9.2	0.00	100032	1.7	d_instantiate+0x1c
0.03	1.5	0.7	9.2	2.3	10	0.00	100031	1.5	d_rehash+0x40
0.24	2.1	1.2	577	1.9	283	0.00	521329	2.1	dput+0x30
1.1	0.70	0.7	366	2.4	617	0.00	4000443	0.70	link_path_walk+0x2d8

Table 7: Results with 8 CPUs

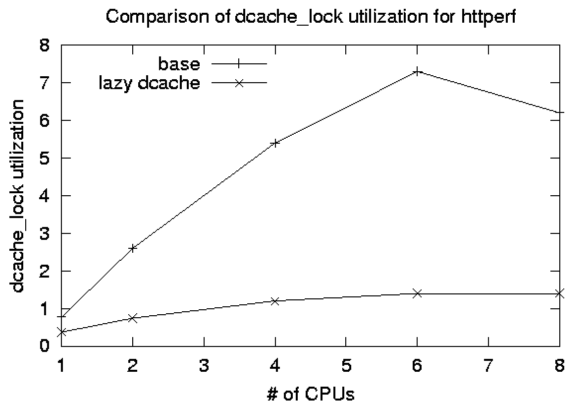


Figure 5: Lazy LRU dcache_lock utilization from httperf

7 Avoiding Cacheline Bouncing of d_count

7.1 fast_walk()

On SMP systems and even moreso on some NUMA architectures, repeated operations on the same global variable can cause excessive cacheline bouncing. This is due to the entire cacheline being read into each CPU's hardware cache while it is being used. For some common directories found in many paths such as '/' or 'usr', this exessive cacheline bouncing will be triggered.

Alexander Viro recommended a possible solution that we implemented. He proposed not incrementing and decrementing the reference counter for dentries that are already in the dentry cache. Instead, hold the dcache_lock to keep them from being deleted.

We used the path_lookup function to implement this change [6]:

```
Before:
    read_lock(&current->fs->lock);
    nd->mnt =
          mntget(current->fs->pwdmnt);
    nd->dentry =
          dget(current->fs->pwd);
    read_unlock(&current->fs->lock);
 }
 return (path_walk(name, nd));

After:
    read_lock(&current->fs->lock);
    spin_lock(&dcache_lock);
    nd->mnt = current->fs->pwdmnt;
    nd->dentry = current->fs->pwd;
    read_unlock(&current->fs->lock);
 }
 nd->flags |= LOOKUP_LOCKED;
 return (path_walk(name, nd));
```

The atomic increment of d_count is all that dget and mntget do.

The rest of the changes were in path_walk (implemented by link_path_walk). While the dentry is found in the cache, just keep walking the path. If a dentry is not in the cache, then increment the d_count to keep it synchronized and drop the dcache_lock, and then simply continue. For coding simplicity, the dcache_lock is always dropped in the path_walk code instead of returned to path_lookup to be dropped.

This patch has been accepted by Linus Torvalds starting with the 2.5.11 kernel.

7.2 path_lookup()

We started with a simple cleanup of replicated code involving path_init, path_walk, and __user_walk [7]. There were sixteen occurrences of the following:

```
    if(path_init(x))
    error = path_walk(x)
Which changed to one call:
    error = path_lookup(x)
In addition there were six
occurrences of the following:
    a = getname(b)
    if(error)
            return
    path_lookup(a)
```

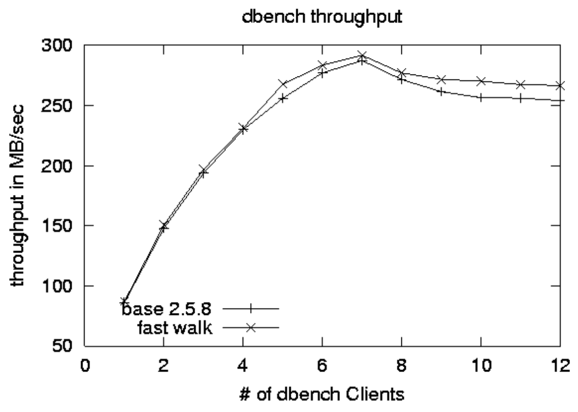


Figure 6: FastWalk increases dbench throughput

```
    putname(a)
which changed to an existing call:
    error = __user_walk(b)
```

This patch has been accepted by Alan Cox starting in 2.4.19-pre5-ac2. Marcelo has not merged this patch into mainline 2.4 as of this writing.

7.3 Fast Path Walking Results

7.4 16-way NUMA Results of Fast Walk

Previously, we mentioned d_lookup was the main user of dcache_lock. This is especially noticeable on a 16-way NUMA system. Martin Bligh, in attempting to get the fastest kernel compile, applied this patch on top of a few others [Bligh]. Not only did it reduce time spent spinning on the dcache_lock, it decreased total kernel compile time by 2.5%.

Following is a profile of kernel during `make -j32 bzImage` on a 16-way NUMA system. This shows an almost 50% reduction in time spinning on the dcache_lock.

```
    Kernel compile time is now
    23.6 seconds.

    Here are the top 10 elements
    of profile before and after
    your patch (left hand column
    is the number of ticks spent
    in each function).

Before:

22086 total                      0.0236
 9953 default_idle             191.4038
 2874 _text_lock_swap           53.2222
 1616 _text_lock_dcache          4.6304
  748 lru_cache_add              8.1304
  605 d_lookup                   2.1920
  576 do_anonymous_page          1.7349
  511 do_generic_file_read       0.4595
  484 lru_cache_del             22.0000
  449 __free_pages_ok            0.8569
  307 atomic_dec_and_lock        4.2639

After:

21439 total                      0.0228
 9112 default_idle             175.2308
 3364 _text_lock_swap           62.2963
  790 lru_cache_add              8.5870
  750 _text_lock_namei           0.7184
  587 do_anonymous_page          1.7681
  572 lru_cache_del             26.0000
  569 do_generic_file_read       0.5117
  510 __free_pages_ok            0.9733
  421 _text_lock_dec_and_lock   17.5417
  318 _text_lock_read_write      2.6949

  ...

  129 _text_lock_dcache          0.3696
```

8 Conclusions

This paper has demonstrated performance improvements of the dcache via the fast path walking patches and the lazy updating of the LRU patches. We are working with the VFS and kernel maintainers to get these patches accepted.

Although the dcache continues to scale, there is more work to be done, much of it happening as this is being written.

9 Availability of Referenced Patches

As of now, all patches have been tested on ext2, ext3, JFS, and /proc filesystem. Our goal was to experiment with dcache, extending it for use with other filesystems, this is in the pipleline.

dcache patches can be found on SourceForge.net under the Linux Scalability Effort project page.

[1] Lockfree read of d_hash
`http://prdownloads.sf.net/lse/dcache_rcu-2.4.10-01.patch`

[2] Per Bucket Lock for d_hash and d_lru
`http://prdownloads.sf.net/lse/dcache_rcu-bucket-2.4.16-05.patch`

[3] Separate lock for the LRU list
`http://prdownloads.sf.net/lse/dcache_rcu-lru_lock-2.4.16-02.patch`

[4] Lazy LRU
`http://prdownloads.sf.net/lse/dcache_rcu-lazy_lru-2.4.17-06.patch`

[5] Lazy LRU updating via timer
`http://prdownloads.sf.net/lse/dcache_rcu-lazy_lru-timer-2.4.16-04.patch`

[6] Fast Path Walking
`http://prdownloads.sf.net/lse/fast_walkA1-2.5.10.patch`

[7] Path walking code cleanup
`http://prdownloads.sf.net/lse/path_lookupA1-2.4.17.patch`

10 Acknowledgements

Alexander Viro has been a tremendous help to us and we thank him for his input and all his hard work. SourceForge.net for supporting Open Source development. Paul Menage for helping to debug. Martin Bligh for running the NUMA tests. Hans-Joachim Tannenberger, our manager. International Business Machines Corporation and its Linux Technology Center. This work represents the view of the authors and does not necessarily represent the view of IBM.

References

[Sarma] Dipankar Sarma, Maneesh Soni *Scaling the dentry cache* `http://lse.sf.net/locking/dcache/dcache.html`

[McKenney] Paul E. McKenney, Dipankar Sarma, and Orran Krieger, *Read-Copy Update*

[Mosberger] David Mosberger, Tai Lin, *httperf: A tool for measuring web server performance.* Hewlett-Packard Inc. Research Labs. `http://www.hpl.hp.com/personal/David_Mosberger/httperf.html`

[Hawkes] John Hawkes *kernprof* Silicon Graphics Inc. `http://oss.sgi.com/projects/kernprof`

[Hawkes2] John Hawkes *lockmeter* Silicon Graphics Inc. `http://oss.sgi.com/projects/lockmeter`

[Pool] Martin Pool *dbench* Samba.org

[Bligh] Martin J. Bligh's *23 second kernel compile (aka which patches help scalibility on NUMA),* linux-kernel@vger.kernel.org, March 8, 2002. `http://marc.theaimsgroup.com/?l=linux-kernel&m=101565828617899&w=2`.

11 Trademarks

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Linux is a registered trademark of Linus Torvalds.

BKL: One Lock to Bind Them All

Rick Lindsley
IBM Linux Technology Center
ricklind@us.ibm.com

Dave Hansen
IBM Linux Technology Center
haveblue@us.ibm.com

Abstract

One ring to rule them all
One ring to find them
One ring to bring them all
And in the darkness bind them.

— *The Fellowship of the Ring,*
J.R.R. Tolkien

When the topic turns to the Big Kernel Lock (BKL), the comparison to Tolkien's one Ring comes naturally. The BKL was among the first locks to be created for the Linux(R) kernel, and many other locks were developed either to complement or replace instances of it. Despite this, coders are reluctant to reduce or eliminate the usage of the BKL so while it may not rule them all, it continues, in a performance sense, to "bind them all."

Once the varied uses of the BKL are understood, the BKL can safely be replaced by other lock mechanisms, which are more appropriate for each instance. The difficulty lies in identifying these distinct instances, determining what protection is provided by the BKL in each, and carefully replacing the BKL without perturbing the rest of the system. In this paper, we examine the history of the BKL, review recent efforts to replace and remove it, and outline the work remaining. The One Lock need not rule nor bind the others any longer.

1 Introduction

Careless locking throughout the Linux kernel adds unneeded complexity and decreases performance. With the introduction of Robert Love's changes[1] to implement a preemptive kernel in 2.5.4, the effects of poor locking now can affect SMP and uniprocessor machines alike. Locks such as the Big Kernel Lock (BKL) have multiple uses and can be confusing to use correctly.

As a result of its overuse, many instances of the BKL subtly intertwine, causing a single `lock_kernel()` call to have several protective and often unrecognized effects. Until recently, for example, the BKL protected list operations on a webcam list in the CPiA driver and would lock out the NFS kernel daemon thread while these operations were being performed. These two activities always executed exclusively, when absolutely no exclusion was necessary. In determining how best to release the BKL's hold over the rest of the kernel, it is useful to examine not only how it is currently used, but how it came to be used that way.

2 History of the BKL

The BKL originated with Linux's first attempts to support SMP. The patch for 1.3.26 shows

[1]Patches available at
`http://www.kernel.org/pub/linux /kernel/people/rml/killbkl/llseek/`

the first signs of the BKL's declaration, but it was not actually used until 1.3.31. The lock was simply a bit which was set whenever a CPU was in kernel context. If another CPU attempted to enter the kernel at the same time, it spun. The net effect was to allow only one process in the kernel at a time. This was a time well before the `spin_lock()` functions, so the authors implemented this spinning behavior themselves in the `ENTER_KERNEL` assembly macro. At this point, the lock was acquired exclusively in `ENTER_KERNEL` and released in `EXIT_KERNEL`; no device drivers or kernel subsystems explicitly interacted with it.

1.3.54 brought with it the now-familiar `lock_kernel()` and `unlock_kernel()` functions defined in C. This opened up the way for code other than the kernel entry code to use the BKL. For all of 1.3 and 2.0, this code was limited to kernel daemons: `bdflush`, `kswapd`, and `nfsd`. With the new C definitions, 1.3.54 also introduced one of the BKL's most striking features: the ability to be held recursively. In that code, the lock's spin loop will terminate if the current processor already holds the lock:

```
while (set_bit(0,
  (void *)&kernel_flag)) {
    if (proc==active_kernel_processor)
      break;
      <snip...>
}
active_kernel_processor = proc;
kernel_counter++;
```

This feature made the BKL more obviously a processor lock rather than a process lock. A single process was not prevented from grabbing it multiple times, but other processors were blocked. It also greatly simplified the programmer's task: there was no worry about deadlocks with yourself. In cases where a function's caller holds the lock, the second `lock_kernel()` will never spin waiting for release.

```
lock()      // spin until acquired
func() {
   lock()   // kernel_counter++;
   unlock() // kernel_counter--;
}
unlock()    // kernel_counter--;
            // (and release if
            // kernel_counter == 0)
```

However, this convenient feature invites abuse. With the threat of deadlocks removed, programmers can take the lock "just to be safe," and there is no penalty for not diligently checking or commenting code. The penalty falls on the inheritors of this code, when they ask, "What is this guarding?" and try to remove the BKL.

The 1.3/2.0 development period saw only very limited spreading of the BKL. As 2.0 development continued, the BKL was added in only one more place, and that was for another kernel daemon.

The BKL as we know it today (a spinlock) was introduced in 2.1.23. The old `ENTER_KERNEL` and `EXIT_KERNEL` semantics were replaced by `lock_kernel()` and `unlock_kernel()` calls around critical regions. At this point, only Sparc and i386 had generic spinlock mechanisms, which meant that, besides semaphores, the BKL offered the only SMP mutex mechanism. The scope of this change in Linux's SMP support is evident from counting how many times `lock_kernel()` is called; 2.1.22 had 9 calls of `lock_kernel()` and `unlock_kernel()` while 2.1.23 had 761!

It might appear that the 2.1.23 patch is the root of all evil. But it did add a very important feature: kernel concurrency. Before this point, no two tasks could be running in the kernel at once. The modern BKL is perhaps not as one-sided as it first looks, as it was the price to pay for kernel concurrency at the time. The current BKL removal process is just a continuation of this effort to allow more kernel concurrency.

3 Current state of the BKL

In 2.4.18, `lock_kernel()` is invoked over 500 times in about 290 files. Determining why it is invoked in those files is a little tricky, since comments are rarely present. Sometimes it's not clear that even the authors understood why it was needed; they appear to have invoked it either because the code they were copying from invoked it, or simply because they feared angering the ancient gods of coding by omitting it.

Semantically, we find that in the 2.4.18 kernel the BKL is used primarily in the following areas:

- `release` (or `close`) routines
- `open` routines
- `mmap` routines
- `ptrace` system call
- file system code
- module protection

Functionally, however, the intended use in each case is far less clear. Most of the uses can only be inferred from code inspection, because usually the users of the BKL did not create comments describing their changes. After some research into the 2.4.18 code and old change logs, however, it is possible to hazard an educated guess at the uses.

`release` (or `close`) routines. These routines are called when a file descriptor is closed for the last time. At one time, the BKL was held across the release call, and when that call was removed and responsibility for acquiring the lock was pushed down into those routines, many authors did not have the time, knowledge, or inclination to determine whether it was truly needed. Many of them remain today, even though most are unnecessary, as we'll see later.

`open` routines. These routines are called when a file descriptor is first created (opened). As with the release routines, the BKL was originally held across the generic open call and thus was held for all devices upon entry to their open code. When the code acquiring the BKL was removed, it became the responsibility of the open routines to acquire it. To avoid breaking any code relying on the ability to acquire the BKL, the patch modified each and every open routine to grab the BKL itself, and left it to the driver owners to take it out if it was unnecessary. Unfortunately, many driver owners chose not to spend time determining whether it was necessary.

`mmap` routines. These routines are called when a `mmap` call is made against the file descriptor to map some portion of the underlying data into memory. While the details of what is being mapped vary with the device, it seems to have been generally accepted that the BKL needed to be held in order to accomplish it. This remains one of the more mysterious uses of the BKL, and will be high on the list for future work.

`ptrace` system call. The BKL appears to be used to lock down the important fields of a process while `ptrace()` (which is architecture-specific) manipulates them.

file system code. In the file system code, it's harder to discern a general pattern of usage. Frequently, the BKL seemed to protect various file-system-specific data structures, as well as some VFS structures. As noted above, file system code exploded with BKL usage in 2.1.23, and since then authors have slowly been weeding it out. What's left is, in general, the code hardest to fathom or the most sensitive to changes, and extricating the BKL from this code requires thorough knowledge of the file system being operated on. Some very recent work has made ext2 much less dependent on the BKL.[2]

module protection. The BKL is used to protect the module list in kernel/module.c. This use of the BKL has the unusual distinction of at least being consistent and well-defined. If it were not for the remote possibility of an unexpectedly positive interaction with other BKL usages elsewhere in the kernel, this mechanism could be replaced with a simple spin lock. As it is, it needs to be inspected as closely and understood as well as any other BKL usage before taking any action.

4 Why does it matter?

It could be argued that "if it ain't broke, don't fix it" and that efforts to reduce or eliminate the BKL are not only difficult in many cases, but pointless. The reference to the module code, above, would be a prime example.

The BKL is not viewed as an obstacle for many benchmarks or workloads. Certainly on uniprocessors, many other concerns are of higher precedence. But when the One Lock does obstruct some task or benchmark, it is a daunting task to remove that roadblock. As mentioned earlier, there are few comments or other documentation to explain why the BKL is used in any particular spot, let alone how it might be excised. Further, because the BKL is used in so many places, the problem may not lie in the region in which it is contended for.

Imagine this scenario. Function `foo()` grabs the BKL 500 times during a particular workload and holds it for 100ms each. Function `bar()`, on the other hand, attempts to grab it 100000 times, but seems constantly thwarted, waiting an average of 35ms 70% of the time. When it does finally get it, it holds it for an average of 10ms before releasing it. One could mistakenly conclude that reducing or eliminating the BKL in `bar()` would make the contention problem go away - and indeed it would. But the real problem probably lies in holding the lock an incredibly long time in `foo()`, thus holding up the many instances of `bar()`.

Does one fix `foo()` to hold the lock a shorter period of time, or does one fix `bar()` to acquire it less? It's bad practice to hold a spinlock a long time in `foo()`, but then it seems optimization may be needed in `bar()` to reduce the number of times locking is required or even the number of times the function is called. Determining the correct answer really requires that both functions be well understood in purpose and scope. In a given instance, the answer may be one or the other, or even both or neither. (Both functions may be completely unnecessary upon closer inspection. Equally possible would be that neither may be able to change their behavior. For example, if `foo()` must hold the BKL while calling a proprietary function or performing some hardware operation it has no control over, it may not be able to reduce its hold time. Similarly, the high number of calls to `bar()` may be a necessary evil that keeps an interface definition clean and more easily supported at the expense of a seemingly high number of function calls.)

Now recall that the BKL is used in hundreds of routines, interacting in thousands of ways, and you've answered the question of why its

[2]Patches (for 2.5 only) are available at
`http://linus.bkbits.net:8080/linux-2.5/cset@1.290`

widespread use and misuse does matter.

5 Recent work

There has already been a great deal of work in 2.5 to remove the BKL where it isn't necessary. Happily, most of the fixes to remove the BKL are dirt-simple (once the arduous investigation to prove their simplicity is completed!) and provide significant, measurable benefits.

5.1 do_exit()

Perhaps the best example of this is `do_exit()`. Lockmeter data from that most-trusted of all benchmarks, the kernel build, showed `do_exit()` holding the BKL for an average of 8ms, with a maximum hold time of 55ms. That is an eternity for a cpu whose time is simply wasted spinning. On first examination of this code, removing the BKL appears difficult because so many complex data structure manipulations are done here. However, upon closer examination, it is evident that many of the functions called under the BKL in `do_exit()` already have their own locking implemented.

So what is the BKL really guarding? Linus Torvalds himself mentioned on the linux-kernel mailing list[3] (LKML) that, even in 2.4, few of these functions actually still need the BKL. The strategy for removal was simple: only hold the BKL around the functions where it is really necessary. In this case, `sem_exit()` and `disassociate_ctty()` appeared to be the most likely candidates for still needing the BKL. After a suggestion from Linus, the BKL was moved into those two functions, and out of `do_exit()` itself. The amount of time the lock was held went from 8ms on average to only *5.5us* in the worst case!

However, this fix was not without its price. Shortly after the initial `do_exit()` patch went into 2.5, posts on LKML reported OOPSes during boot whenever preemption was enabled. Replacing the `lock_kernel()` in `do_exit()` fixed the problem, as did a `preempt_disable` (which, in a preemptive kernel, all `spin_lock()` calls do implicitly). While the task is exiting, `exit_notify()` sets `current->state` to `TASK_ZOMBIE`. However, if a preemption point occurs after the state is set, the return from preemption code `sets current->state` to `TASK_RUNNING`. This makes the previously "zombied" process eligible to run again, instead of being cleaned up. The `schedule()` at the end of `do_exit()`, which is never meant to return, ends up returning.

The fix is to note the task's state when it was preempted and make sure not to make it runnable again if it was exiting when it was preempted. `do_exit()` is a prime example of why it is very dangerous to derive protection from a lock without realizing why, especially from the BKL.

5.2 `release()` removal

Many device drivers' `release` and `open` functions try to guarantee that only one open can be done on the device at a time:

```
static int opened = 0;
open()
{
    if( opened )
       return -EBUSY;
    ... do opening stuff
    opened = 1;
}
```

[3] Mailing List Archive: `http://marc.theaimsgroup.com /?l=linux-kernel&m=101484620020622&w=2`

```
release()
{
    opened = 0;
}
```

This works fine on a uniprocessor machine. However, on SMP systems a race can allow two processes to open the device simultaneously, as demonstrated in the C code in Figure 5.2.

Currently, this is not a problem for character or block devices. The VFS code holds the BKL over the calls to all char and block open functions:

```
int chrdev_open(struct inode *inode,
                struct file * filp)
{
  ...
  if (filp->f_op->open != NULL) {
     lock_kernel();
     ret =
      filp->f_op->open(inode,filp);
     unlock_kernel();
  }
  ...
  return ret;
}
```

There is similar code for block devices, but misc devices are not afforded this protection. Also, as good practice, devices should never depend on the layers above them to protect against races in their own code especially when they depend on protection provided by the BKL. They should be even more careful to avoid depending on the BKL's protection without realizing it (see `do_exit()` example).

Before the `release()` removal patches, many drivers' open/release combinations looked like this:

```
static int opened = 0;

// implicit from VFS code
// for char/block
lock_kernel();
open()
{
   if( opened )
      return -EBUSY;
   ... do opening stuff
   opened = 1;
}
unlock_kernel();

release()
{
   lock_kernel();
   opened = 0;
   unlock_kernel();
}
```

In almost all of these cases, the fix is simple: just use atomic bit operations. No spinlocks or semaphores are needed; just a simple bit operation. Now the functions are safe to use in block, char, or misc devices because they don't rely on VFS to do any locking for them.

```
static int opened;
open()
{
   if( test_and_set_bit(0,&opened) )
      return -EBUSY;
   // I'm the only opener
   ... do opening stuff
   // success
}

release()
{
   clear_bit(0,&opened);
}
```

6 Notable work, and kudos

Valiant, ongoing efforts by many code warriors to reduce or eliminate the BKL are worth mentioning. This is not a complete list, of course, but just some recent efforts that have had a significant impact.

```
open                          open
{                             {
  if( opened )
     return -EBUSY;             if( opened )
                                   return -EBUSY;
  // I'm the only opener        // Uh-oh, we got in before opened
  ... do opening stuff          // was set and there are two of us!
  opened = 1;   // success      opened = 1;
}                             }
```

Figure 1: Open Race

VFS has had a difficult struggle with the BKL. The UNIX(R) “everything is a file” approach forces perfection from filesystem code. VFS is also a place where concurrency is important, so locking must be done carefully and kept finely grained so as not to adversely affect performance. The BKL’s presence in so many other pieces of code has made its presence in VFS code troublesome.

Al Viro has done a noteworthy job of freeing VFS from dependency on the BKL and shifting the responsibility of locking into the underlying code.[4] This underlying code is always closer to the task at hand and can make more well-informed, finely-grained decisions than VFS can. He has also worked on documenting filesystem locking. The BKL still plays a big part in VFS and Al has done a good job of documenting that role in Documentation/filesystems/Locking. He has also documented the locking changes in Documentation/filesystems/porting.

Richard Gooch has been an exemplary advocate for pushing the BKL out of devfs.[5] On one occasion, he responded within minutes of a patch being posted which pushed the BKL into some devfs code, down from the VFS layer. A couple days later, he posted a patch purging the BKL from his subsystem because it derives absolutely no protection from the BKL.

7 Future Work

There are some specific areas that need to be addressed, and some were mentioned above: the `mmap` code, the `ptrace` code, and the file system code, for example. Stalwart, knowledgable code warriors are always sought for this sort of effort.

However, you can join the fellowship and be a BKL Eliminator even without knowledge in these areas. Even if you have no time to eliminate the BKL from existing code, you, as a coder, can help prevent it from proliferating by adopting simple standards for yourself:

- Never submit code that adds the BKL, anywhere.
- If you need exclusion in your driver, provide it yourself with your own lock.
- If you need to sleep while holding a lock, use a semaphore.
- If you still think you need the BKL, ask somebody else first.

If you *are* responsible for code that still uses the BKL, make an effort not to expand where

[4]Patches are available (for 2.5) at `http://linux.bkbits.net:8080 /linux-2.5/user=viro`

[5]Patches are available (for 2.5) at `http://marc.theaimsgroup.com /?l=linux-kernel&m=101787867929523`

it is used. The next time you go to rewrite a big chunk of your code, think about the BKL and start imagining ways to remove it.

A running scorecard for each release can be found at `http://lse.sourceforge.net/lockhier/index.html`. Listed there are versions of a locking reference document for recent releases of both 2.4 and 2.5—in particular, outlining where the BKL is still found. Here is a summary for 2.5.8-pre3:

Top users of BKL in 2.5.8-pre3
(excluding filesystems)

1689	`.`
254	`drivers`
177	`arch`
113	`sound`
107	`sound/oss`
71	`include`
71	`drivers/usb`
47	`drivers/char`
39	`drivers/isdn`
32	`include/linux`
29	`arch/sparc64`
28	`arch/sparc/kernel`
28	`arch/sparc`
23	`drivers/usb/core`
22	`kernel`
19	`drivers/usb/media`
19	`arch/alpha/kernel`
19	`arch/alpha`
17	`arch/sparc64/solaris`
16	`arch/ppc64/kernel`

Top users of BKL in 2.5.8-pre3
(filesystems only)

1026	`fs`
76	`fs/reiserfs`
67	`fs/coda`
61	`fs/ext3`
58	`fs/intermezzo`
58	`fs/hfs`
54	`fs/nfs`
51	`fs/hpfs`
44	`fs/affs`
43	`fs/udf`
35	`fs/autofs`
32	`fs/ufs`
30	`fs/smbfs`
28	`fs/jffs`
24	`fs/ntfs`
22	`fs/bfs`
21	`fs/vfat`
19	`fs/ncpfs`
19	`fs/autofs4`
18	`fs/qnx4`

All maintainers with a subsystem listed above should take a hard look at their code. In most cases, the code which uses the BKL does not actually need it. Understandably, developers are reluctant to change code which they do not have great knowledge about. As the maintainer, you are the authority , you do have intimate knowledge of the code, and you can intelligently and safely remove the BKL.

Perhaps the only generic use for the BKL which is spreading is the use around calls to `daemonize()`. Many of them, like this use from `ibmphp_hpc.c`, hold the BKL for short periods of time.

```
lock_kernel ();
daemonize ();
reparent_to_init ();
...
unlock_kernel ();
```

However, there are many other cases where the same operations are performed without the

BKL. The authors would be very interested to receive any information that readers can add about this use with `daemonize()`.

In general, the kernel would be a better place if Linus would never accept another patch with `lock_kernel()` in it. Truth is, this is not likely to happen anytime soon. But an increased awareness among the Linux community can be almost as useful as eradication in achieving this goal. Until the BKL is reduced to one or zero usages, there will remain fear and uncertainty when it is encountered, and the One Lock will continue its hold over all of us.

8 Acknowledgements

IBM is a registered trademark of International Business Machines Corporation in The United States and other countries.

Linux is a registered trademark of Linus Torvalds.

UNIX is a registered trademark of the Open Source Group.

The opinions expressed are those of the authors, and do not necessarily reflect the opinions of the IBM Corporation.

HPC Federated Cluster Administration with C3 v3.0

Brian Luethke and Stephen Scott

Abstract[1]

KEYWORDS: Cluster, Administration, Federated-cluster, multi-cluster

While administrating TORC, HighTORC, and the various other computation clusters at Oak Ridge National Laboratory (ORNL), it quickly became apparent that a solution for the administration of federated clusters, or "clusters of clusters," was needed. The few cluster tools available when this work began could barely manage a single cluster let alone a number of clusters. They also required that the user be directly logged onto a cluster machine. This meant to administer ten clusters required that the administrator login and repeat a task on each of the clusters. This solution does not scale and therefore is unacceptable for our environment. Thus, a solution was desperately needed whereby an administrator could perform duplicate operations across multiple clusters and portions thereof in a scalable and secure fashion from a single location that may not be directly logged onto the cluster being administered. Thus the development of version 3.0 of the Cluster Command and Control (C3) tool suite began.

C3 prior to version three required, as most tools do, that one is physically logged into a cluster in order to perform administration operations. The few existing tools that permit remote administration of clusters were all web based, therefore they suffered security problems and set up hassles associated with installing and maintaining a web server. What we tried to design is an easy to use command line interface that is powerful enough to do most system administrating jobs and secure. These tools also needed to be useful to regular users in building and maintaining their distributed applications. C3 version 2.x already met those requirements so we decided to emulate their functionality while adding the ability to do this with multiple clusters. This paper describes the use of the C3 3.0 tool suite.

[1]Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

1 Brief Description of Commands

Ten general use tools have been developed in the effort thus far: cexec, cget, ckill, cpush, cpushimage, crm, cname, cnum, clist, and cshutdown. The cpushimage and cshutdown are both system administrator tools that may only be used by the root user. The other eight tools may be employed by any cluster user for both system and application level use.

The cexec command is the general utility tool of the C3 suite in that it enables the execution of any command on each cluster node. As such, cexec may be considered the clusterized version of rsh/ssh[1]. A command string passed to cexec is executed "as is" on each node. This provides a great deal of flexibility in both displaying the command output and arguments passed in to each instruction.

The cget command will retrieve the given files from each cluster node and deposit them in a specified directory location on the local machine. Since all files will originally have the same name, only from different nodes, an underscore and the node's IP or hostname and cluster name are appended to each file name. Whether the IP or hostname is appended depends on which is specified in the cluster specification file. Note that cget operates only on files and ignores subdirectories and symbolic links

The ckill tool runs the standard Linux 'kill' command on each of the cluster nodes for a specified process name. Unlike 'kill', ckill must use the process name as the process ID (PID) will most likely be different on the various cluster nodes. The root user has the ability to further indicate a specific user in addition to process name. This enables root to kill a specific user's process by name and not affect other processes with the same name but owned by other users. Root may also use signals to effectively do a broad based kill command.

The cpushimage enables a system administrator logged in as root to push a cluster node image across a specified set of cluster nodes and optionally reboot those systems. This tool is built upon and leverages the capabilities of SystemImager[2]. While SystemImager provides much of the functionality in this area, it fell short in that it did not enable a cluster-wide push for image transfer. cpushimage essentially pushes a request to each participating cluster node to pull an image from the image server. Each node then invokes the pull of the image from the cluster image server. Of course, this description assumes that SystemImager has already been employed to capture and store a cluster node image on the cluster image server machine.

While cpushimage has the ability to push an entire disk image to a cluster node, as an application support tool, it is too cumbersome when one simply desires to push files or directories across the cluster. Furthermore, cpushimage is only available to system administrators with root level access. From these restrictions grew the desire for a simplified cluster push tool, cpush, providing the ability for any user to push files and entire directories across cluster nodes. cpush uses rsync[3] to push files from server to cluster node.

crm is a clusterized version of the standard 'rm' delete file/directory command. The command will go out across the cluster and attempt to delete the file(s) or directory target in a given location across all specified cluster nodes. By default, no error is returned in the case of not finding the target. The interactive mode of 'rm' is not supplied in crm due to the potential problems associated with numerous nodes asking for delete confirmation.

cshutdown is a cluster wide shutdown operation. cshutdown also has the ability to boot an alternate lilo label for a single boot. This allows you to test a new kernel without making permanent changes or to boot into another operating system temporarily.

The cname command returns the node number based on the cluster and node name supplied at the command line. Both this command and the cnum command are useful when the node names of your cluster and their positions in the configuration file are not easily paired.

The cnum command returns the node name based on the node number and cluster supplied at the command line.

The clist command returns a list of clusters defined in the cluster configuration file and the type of cluster.

2 Installation and Configuration

C3 version 2.x used a list of nodes, one per line, to define a cluster. While it is possible to have several clusters in this list, each node had to be visible to the machine that the C3 command was run from. Many clusters only have the head node exposed with the individual compute nodes on a private network. A list of nodes also does not allow the granularity need to specify which cluster to execute the command from. In version three you have the concept of a cluster configuration block. Each block defines a single cluster – its external entry point, an optional internal entry point (if the nodes are on a private network) and then the list of nodes. This type of cluster is called a direct cluster – the configuration of the cluster is known by the C3 command before runtime. It is possible to have both a local cluster (the machine that the C3 command is run from is the head node) and a remote cluster (the machine that the command is run form is not the head node) use a direct method of definition. One of the advantages of this scheme is that it is possible to build both subsets and supersets of a given cluster. The major drawback to using a direct cluster block on a remote machine is that the user must keep track of which machines are offline and which are online – this can be a real headache. C3 solves this problem with an indirect remote cluster. In this type of cluster block the only thing the C3 command knows is the external interface of a remote cluster. When a C3 command is run it will execute that command on the remote cluster using the default cluster configuration block (the first cluster in the configuration file) on that cluster. In this way a user using his or her desktop need not know how many machines are currently on each cluster they use, only that the head node they have specified exists and has a working copy of C3 version 3 on it. Below is an example configuration file:

```
cluster home { #the cluster
# named home.
# The default cluster as it is the
# first in the configuration file
node0 #the head node,
#      external name only
node[1-15] #the compute nodes
}

cluster TORC { #the cluster
#                named TORC
heimdal:node0 #the head node,
# heimdal is
# the external interface name
# and node0 is
# the internal interface name
node[1-64] #compute nodes
}

cluster htorc { #the cluster
#                named htorc
:htorc-00 #this is a indirect remote
#cluster, htorc-00 is the external
#interface name
}
```

3 Usage

In early versions of C3 we were tied to the implementation of a PERL[4] package to parse the command line. In version 3 we parse the command line ourselves giving us much more flexibility. When designing our API we tried to stay as close to the respective Linux tool as possible so that a user would have a minimum amount of learning to do. We also now have the ability to specify node ranges on the command line. This is very useful for system administrators for doing rolling upgrades. An example of the new API that would execute hostname on the default cluster would be as follows:

```
cexec hostname
```

Extending the paradigm to federated clusters is just as simple, simply specify the clusters you wish to execute on. To execute on the default cluster and on nodes four through six and node eight on the cluster named TORC would be as follows:

```
    cexec :  TORC:4-6,8 ls -l
```

In the above example the command is cexec (a general purpose exec, similar to a cluster wide rsh) ":" signifies the default cluster, TORC: signifies the cluster named TORC in the configuration file, 4-6,8 is a node range, and ls –l is the command to be run. There are several ways the C3 tools were designed to be used. The most basic way is from the command line, one command at a time. Next is writing scripts using the C3 tools. And the third way is extending the C3 tools themselves for site-specific functionality.

A good example of using the tools directly on the command line is effecting rolling upgrades. Using cpushimage and SystemImager it is very simple to test out a cluster configuration. A system administrator could build a small test cluster using SystemImager to clone the current cluster. After installing the software and testing the new image you would find it acceptable for roll-out. You would again use SystemImager to retrieve the image from the test cluster. Next, make sure you have a backup image from the production cluster and use cpushimage to test it on a small number of machines. Assuming the image name is new_image the sample command would look like:

```
 cpushimage -reboot :0-3 new_image
```

This pushes the new image to only the first four nodes in the cluster and reboots the machine. This allows you easily test the new image. Assuming it works, just type the same command as before removing the :0-3 from the command. That would push the image to every node in the cluster. One of the nice features of this is if you later find a problem with the new image it is easy to roll back to an earlier image that is known to work.

Using C3 from the command line is also useful for a general user of a cluster. Using the indirect remote cluster a user can develop an application on their desktop and easily distribute the binary to either a single cluster or multiple clusters (via cpush). Using C3 in this way makes a cluster a "black box" – that is the user only has an indiscriminate resource out there called a cluster. They do not need to keep up with the addition of new nodes nor if a few nodes have been taken offline. One of the features of writing the C3 power tools in a platform independent language is that the tools only must run homogeneous within there self. For example, take the above user who wishes to push a binary to several clusters. One of the clusters is an Intel PC cluster and the other is an alpha cluster, both running Linux. Using the GNU gcc cross compiler the user has compiled a binary for an Intel machine and a binary for an alpha cluster (it is possible that their desktop be a power macintosh). They are using text data files so the data can be used by all systems. Assuming that the head nodes home directory is NFS mounted the following commands would distribute the application, its data, and run it:

```
cpush --head Intel: app.Intel app
cpush --head alpha: app.alpha app
cpush --head Intel: alpha: data.txt
cexec --head Intel: alpha: app
```

Notice that once the binaries are renamed when pushed out to the cluster so that a single cexec can start the application. This demonstrates that the level of homogeneity required by each command can be different. In the first two lines each cluster must be separated but in the last two command their actual architecture is irrelevant as the command being run is identical on each cluster. This is a power paradigm for both users and system-administrator.

The next way the C3 Power Tools can be used is with scripting. Using scripting and image management with C3 is useful for effecting changes for a single user. With C3, once the image is built, it is quite easy to temporarily install a new image with differences ranging from a slightly different communications package, to a different flavor of Linux one the fly. For example we have a user who requires kerberos[5] to be installed in order to run their code, we do not wish to support or maintain this. It only a matter of an hour or so of time (because of the size of the image being transferred, all the interaction required is the initial command run and after it is done checking to make sure the machines rebooted) to switch to a completely different image complete with their data and special configuration fully operational. With scripting this can even be done within a PBS script to change the image before the run and to restore it to the default one afterwards.

The next way the C3 can be used is in scripting. While administrating our clusters one of the largest problems we encountered was generating and managing ssh keys when creating a new user. Unfortunately it is very difficult to write a tool that is a general purpose ssh manager as the policies differ from site to site. While this script is included with C3 it is not part of C3 proper – it is in the examples directory due to the above problems. See figure 1 for the example code. This script works by getting the user-name and group of a new user from the command and then calling the standard Linux adduser binary. next it sets the password for that user with the standard Linux password facility. Then, using C3 all the files that were touched are sent to the cluster nodes, and the any needed directories are created (/home is NFS mounted so the directory only needs to be created on the head node). Lastly the ssh-keys are generated and the authorized_keys2 file is created (to allow users to ssh to one of the compute nodes without the use of a password). Where this script and C3 really show the power available is in combining this method with a command line. Assuming this script is located in /usr/sbin a command as follows:

```
cexec -all /sbin/add_user zbml1 users
```

would add the user zbml1 with the group users to every cluster that the machine this is executed on has access to. Thus it is just as easy to add a user to one cluster, as it is five. The only redundant typing would be when the password is generated but it is trivial to write an expect script that handles this for you.

We also use the C3 tools to take the place of some of the daemons we would probably run. We do not run to run NTP[6] to keep our cluster's date in sync so we wrote our own bash script that gets the current date on the head node and then issues a cexec to set the cluster nodes to the current date. The script is ran once a day in a cron job to keep the cluster in sync.

The third and most powerful way that the C3 Tools can be used is in extending them. When we wrote C3 one of the focuses was to make it modular. We chose Python[7] as a language both because it is well known and common and it is also very easy to write packages for. The two main parts we separated out of the code into packages are the command line parser and the configuration file parser. This allows you to add functionality such as hardware monitoring, BIOS maintenance, or any functionality you would choose. Splitting the file parser into its own package also allows a system administrator to both read the c3.conf file but to also use it as a base. A nice example of this would be setting a cron job that once a night reads the c3.conf file and generates an up to date configuration file for PVM. The command line package lets a system administrator to create new tools that have the look and feel of the C3 tools making it easier on their users learning a new command line API.

Included in C3 Version 3.1 is a "contrib" directory where the script mentioned here and other are included. The scripts in this directory are offered for use if your site is configured such that they are applicable (such as the add_user script assumes NFS mounted home directories and use of ssh). These scripts are also intended to be concrete examples of extending and using C3. Also included are full package documentation on the command line parsing object and the c3.conf parser.

4 The Future

Versions 3.x of the C3 Power Tools offer both a system-administrator and a general user great power for managing both a single cluster and multiple clusters. One of the areas that the current versions of C3 are short in is scalability. We are currently working on a version 4 of the tools that take into account homogeneity in clusters and their topography in order to scale the commands into clusters with thousands of nodes.

5 Conclusion

Version 3.0 of the C3 tools suite is a major advance in the tools. The ability to administrate multiple clusters simultaneously from anywhere you can access each head node is very useful. In the same number of command in C3 v2.7 it would take to add a user to a cluster you can now add a user to any number of clusters. Users who write a distributed application that will run on several clusters now have an easy way to distribute their application to the clusters, even form their own desktop.

6 References

1. http:/www.openssh.org/

2. http://www.systemimager.org/

3. http://samba.anu.edu.au/rsync/

4. http://www.perl.com/

5. http://web.mit.edu/kerberos/www/

6. http://www.eecis.udel.edu/sim/

7. http://www.python.org/

8. http://www.csm.ornl.gov/torc/

Figure 1: add_user script

```
#!/usr/bin/env python2
#########################################################
#this script adds a user to the local cluster.  It assumes that
#the home directory is nfs mounted and no others are.  Put this in
#a well known location (/root/bin in our case) so it can be called
#with cexec.  This is an example of using the C3 tools in a script
#to automate tasks on a cluster.
#########################################################
import os, sys
try:  #get user name from command line
        user_name = sys.argv[1]
except IndexError:
        print "must supply a user name"
        sys.exit()
try:  #get group name from command line
        user_group = sys.argv[2]
except IndexError:
        print "must supply a group name"
        sys.exit()
#adduser to local machine
os.system(  "adduser -g " + user_group + " -m " + user_name )
#set password for local user
os.system(  "/usr/bin/passwd " + user_name )
#distribute the password files and group files to compute nodes
os.system(  "/opt/c3-3/cpush /etc/passwd" )
os.system(  "/opt/c3-3/cpush /etc/shadow" )
os.system(  "/opt/c3-3/cpush /etc/group" )
os.system(  "/opt/c3-3/cpush /etc/gshadow" )
#create ssh directory
os.system(  "mkdir /home/" + user_name + "/.ssh")
#since the script is run by root change ownership of users file to that user
os.system(  "/opt/c3-3/cexec chown -R " + user_name + ":" + user_group + "
/home/" + user_name )
#create users ssh-keys
os.system(  "/bin/su " + user_name + " -c \'/usr/bin/ssh-keygen -b 512 -t dsa
-N \"\" -f " + os.path.expanduser( "~" + user_name ) + "/.ssh/id_dsa\'" )
#set up keys such that they can loginto nodes without a password
os.system(  "cp /home/" + user_name + "/.ssh/id_dsa.pub /home/" + user_name +
"/.ssh/authorized_keys2" )
#make sure everything in their directory is owned by them
os.system(  "/opt/c3-3/cexec chown -R " + user_name + ":" + user_group + "
/home/" + user_name )
```

The Open Clustering Framework

Lars Marowsky-Brée
Research & Development
SuSE Linux AG
lmb@suse.de

Abstract

The Open Clustering Framework is an effort to unify the current projects underway with regard to clustering on the Linux platform with a common component model, common APIs and at least one Open Source implementation. The initial focus is on High Availability Clustering and Linux, but this is not a long-term or inherent limitation.

1 Introduction

1.1 Current status of clustering on Linux

Many, many High-Availability clustering projects have appeared on Linux in the last few years; not only Open Source ones, but also almost every major vendor has ported their solution from their proprietary operating system to Linux.

Today, we have at least ten Open Source clustering solutions for Linux, and in excess of *twenty-five* commercial ones. A solution exists for almost every niche. In fact, it is safe to say that at least *N+1* solutions exist for every niche. For an overview, see the very useful *Linux Clustering Information Center*[1] by Joe Greenseid.

And this number is still constantly growing.

[1]`http://www.lcic.org/`

Linux enjoys an abundance of riches. Users, application or operating system programmers and system-integrators have many choices available for building the perfect solution from these many pieces.

This sounds like the topic can be safely considered solved. . .

1.2 The problem

Unfortunately, the pieces do not fit together **at all**.

They share neither a common API, nor a common concept of what (High-Availability) clustering is. This may seem surprising at first, because at first glance most of them appear to solve the same general problems and provide mostly the same functionality; largely, they are even structured roughly in the same way.

This is actually true; and the reasons why they do not fit together are largely historic. As already said, many of them were written independent of each other, be it due to commercial interests, the *I just want to solve this part*-phenomenon, *My thesis has to be academically perfect*, plain ignorance or any other number of reasons.

On the proprietary platforms, there is the nine-hundred pound gorilla[2] called *The Vendor* defining the standard; and maybe one or

[2]Metaphor courtesy of Alan

two competing products, which would either do something fundamentally different or complement the vendor's solution.

On Linux, such a single vendor does not exist[3] for obvious reasons, which is commonly considered a good thing. It also implies that no-one has a higher right to claim they are the standard than the rest, unless they would be both vastly superior and captured a substantial share of the market.

However, the Linux clustering *market* is sufficiently evenly fragmented that this is true of no-one; everybody can claim roughly the same weight as the rest, so no API, no conceptual model has established itself as a reference.

1.3 Consequences

All solutions are incompatible with the rest; they will not inter-operate. And what is worse, even if they do *appear* to inter-operate successfully, it is by pure chance; not the best foundation for a highly-available mission-critical system and bound by Murphy to fail at the most inconvenient time.

They have different APIs, different concepts of what a *cluster* is, slightly different approaches of how to cope with specific failure situations and so duplicate large parts of the picture between them. This can potentially lead to all sorts of nasty deadlocks, data corruption and loss of service.

1.4 Affected parties

Everyone has a different perspective on the problem; but the short summary is: **Everybody loses**.

[3]Despite every Linux vendor claiming it is them.

1.4.1 The end-user

The end-user usually could not care less about how his problem is solved; he will just care whether it is solved, and what it costs. He is not particularly amused that yes, almost all his problems can be solved, but that a coherent solution is not possible; data has to be replicated, independent systems running different parts of the solution have to be purchased, installed and maintained, and that he has to spend an amazingly large sum on making it all work.

A special case of the end-user is the system administrator[4]; he has to maintain the whole mess of different solutions with different interfaces, and not even a common *SNMP MIB* for monitoring exists for the most basic aspects of the whole.

1.4.2 Independent software vendors

Independent software vendors want to sell their application to the largest audience possible; preferably, everyone. Having a very heterogeneous environment means to either abandon a large part of the market, cope with many different solutions or implement the entire stack themselves and be independent of the mess[5].

Each of these approaches has its own problems; be it a smaller potential market, huge compatibility matrices or lots of code which has nothing to do with the core competency of the company.

[4]Put away the LART, will you.

[5]Which is what Oracle 9i Real Application Cluster does.

1.4.3 Commercial providers of clustering solutions

A provider of a clustering solution has largely the same problems as the independent software vendors; they do not have much of a choice with regard to the scope of their solution. Even if they just want to solve part of the problem—for example *just* a cluster-aware filesystem—they are either stuck with building the full package or tieing themselves to one or more other vendors.

Even vendors of full solution stacks aren't better off by far; they have to compete fiercely for the attention of the end-users and the independent software vendors to support *their* version of the wheel. If they are just slightly better for a particular niche which would theoretically be enough for them to make do or even be off pretty well, it may not be large enough for an ISV to add compatibility for their solution.

1.4.4 Open Source projects

Not only does everything said before apply here; but the Open Source community faces a rather severe additional problem which is often overlooked: Split of rare resources.

While it can be generally considered a good thing to have two or even more solutions for a particular area to keep competition up, being split between so many different projects prevents many of them from reaching critical mass and becoming really successful.

This is especially true of an area which is inherently complex, has a limited audience or simply requires more resources to work on than the common programmer has at home, like multiple nodes for a cluster.

1.4.5 System integrators

System integrators are in a tricky position; all choices what to include are wrong. They are unable to provide a coherent clustering solution without annoying the other ninety-five percent of the market.

1.4.6 Consulting companies

If the solution is build by a consulting company, one might assume they are in heaven – after all, lots of complexity means longer projects, which means more money. Or, as *despair.com* puts it: *If you are not part of the solution, there is good money to be made in prolonging the problem*.

This is a false guess; consulting companies prefer to be paid for not doing any *real work*, and this is largely inevitable if you want to deliver a working clustering solution on Linux because of the above factors.[6]

2 The solution

After hopefully having demonstrated that everyone will benefit from having this problem solved, let's look at the two obvious solutions in more detail:

2.1 Copy-cat standard

The first thoughts were to look at other platforms; was there a standard which could easily be adopted – POSIX, *best current practices* or even a de facto industry standard.

The search was unsuccessful; neither a coherent full standard exists nor a subsets of the

[6]You did notice the tongue-in-cheek, didn't you.

whole[7]. All other platforms have said nine hundred pound gorilla setting the standard by vendor decree. POSIX has not yet bothered to work on this topic, nor has any other larger standards body.

Mimicking one of the vendor standards from another platforms also has intellectual-property issues and the problem that you cannot get any other commercial vendor to agree to it.

2.2 Standard by appointment

The next obvious solution to the problem is to do as the other platforms: just pick one solution and declare it the standard; many affected parties will not care how the standard appeared, as long as it makes their pain go away.

This has been tried by every vendor and even many Open Source projects; the issue is that no-one so far had sufficient weight for it, and those who do had political reasons not to.

Another issue comes with the fact that none of the solutions solves all aspects; a complete, coherent solution does not exist, despite all marketing claims.

Having more than two or three solutions be *the standard* does not solve the problem; so unless a solution which is agreeable for everyone or at least a large enough part appears, this does not work on Linux; compare the lack of a nine-hundred pound gorilla described before.

3 The framework

3.1 Mission statement

And this is where the *Open Clustering Framework* enters the picture; the easier routes have been considered, but found to be unusable. Everybody sighed, shrugged and figured that real, not-fun work might be necessary.

Recalling that almost all solutions are conceptually similar, it is assumed that a common model and associated APIs can be defined which are generic enough to cover the required functionality.

The mission statement therefore is:

- To define a common model for clustering on Linux.
- To define and implement a standard set of APIs for these on the Linux platform.

3.2 Scope

While the primary focus is Linux, the standards aspire to be platform-agnostic; as stated above, our search for inpedendent standards on other platforms was negative, so they might benefit as well.

We try to keep the standard open for input from the High Performance Computing community too, but the bias is towards High Availability. Fortunately, the overlap in the requirements is large enough that we could accommodate both sides so far.

3.3 Requirements

3.3.1 Overall project

The project as a whole has the following requirements:

- Bandwagon effect; capture enough mindshare that a substantial piece of the market adopts the standard; otherwise the work is mood.

[7]As the focus is on HA clustering, MPI does not count.

- The work must be royality and IP free. We do want to support both Open Source and proprietary implementations.
- Timely results. It was felt that early release of something usable and iteratively improving upon this was important in preserving interest and building momentum.

3.3.2 APIs

For the APIs[8] to be generally accepted, they must meet the following requirements:

- Wherever possible, the APIs shall adopt best practices of current implementations or be abstract enough to support as many of them as possible.
- The APIs, though primarily targeted at Linux, should not be applicable to Linux only.
  - Linux implementation is the primary target. Supporting another operating system should not negatively impact the Linux implementation.
  - Nevertheless we do want to define the APIs so as to make it as easy to implement in other operating systems as is consistent with the first point; it was mentioned that the APIs need to be OS-independent to be useful in HPC work, where applications are often written by people who don't own the clusters they will run on, and are often run on more than one cluster.
- API specs should aspire to POSIX compliance; in general, they should extend existing specifications with cluster functionality if sensible.
- APIs should in no way dictate an *in-kernel* or *user-space* implementation exclusively.
- Consistency.
- While every project is free to implement these APIs directly, it should also be possible to provide a compatibility layer on top of legacy code.

3.3.3 Reference implementation

The following requirements for the reference implementation have been put forward:

- The build system should be chosen to build on various platforms[9].
- Components should be portable wherever possible.
- Component interfaces should be agnostic with regard to OS and to kernel vs. non-kernel implementation.

3.4 History

While the problem has lurked in the mind of people for a long time already[10], the issue has not received enough attention until recently, when the pain became noticeably worse.

The success of the *heartbeat-stonith* library, now used by at least three clustering packages, also suggested that the time might be right now.

At the *Ottawa Linux Symposium 2001* an introductionary meeting took place, where many people got together and discussed the problem; as a follow-up, a three day workshop was held directly prior to the *Linux Kongress 2001*

[8]In this document, the terms *API providers* and *consumers* are used respectively.

[9]i.e., autoconf

[10]See [HM97], section thirteen, or [GP97], section 15.3.

in Enschede, where the general direction was agreed upon and the first cut at the proposed component model was outlined – see [GL01].

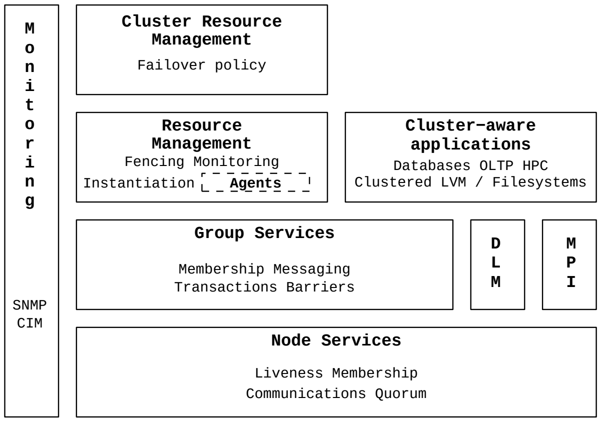
Since then, there has been a lively discussion on the mailing list, a website has been setup and more and more people, projects and companies are getting involved every week.

In 6, the author tries to give a prediction of where OCF is going.

4 The current component model

4.1 Overview

The current proposal for the functional components of the Open Clustering Framework is shown in the following diagram; it is based on our experience with how common clustering software is structured. It reflects the current status of our discussion and is subject to change.



The components are broken up according to the *objects* they deal with; *nodes*, *process groups*, *locks* et cetera. The APIs we are preparing will map to the external interfaces of these components.

A given implementation may chose to implement only one of these components or a subcomponent, using the other components at will. However, the implementation might also provide the functionality of more than one component in one brick, and use whatever internal structure it desires. As long as it exposes these APIs, a cluster-aware application or an other component will be able to use it nonetheless.

This allows the programmer to focus on his main area of interest and expertise. Within certain limits, an system architect will be able to build a system from the best-fitting components.

As components can be exchanged, the need to build the one-size-fits-all solution also diminishes; for example, a two-node cluster requires sufficiently less complex node services than a cluster of arbitrary size and may be configured more easily. The architect can chose the proper component for the intended use.

4.2 Node Services

Node services encompass all services relating to nodes.

One of the open questions here is how to identify a specific node; the main conceptual difference is whether nodes are identified by *host names*, *an integer sequence number* or even their primary *IP address*. All of these have precedents and good reasons in their favour, and specific optimizations the higher layers or applications can implement if they know about it.

This demonstrates one of the issues we are facing; namely, level of abstraction in the API and the model. We have to cope with all of the possibilities reasonably well, so the *node identifier* will be either treated as an opaque blob in the API and the API consumers will have to deal with it, or different kinds of clusters might require a recompile of the consumers.

4.2.1 Node Liveness

The component responsible for monitoring node health; mostly implemented as a binary *alive* or *dead*, it could also yield finer-grained data.

4.2.2 Node Communication Services

The lowest level of the cluster communication services; the instances of the cluster software on each node need to talk to those on the other nodes.

This is potentially used not only by the node membership services but also the higher-level group communication services.

The challenge will be to define a lowest-level denominator which can abstract the different models used for authentication, encryption, messaging and addressing semantics.

4.2.3 Node Membership Services

The membership – the list of currently active or eligible nodes – in a cluster is always in flux, because outages are only detected asynchronously. However, it is vital that the members have a coherent picture of the cluster; this component uses the *Node Liveness* data to compute the current consensus membership.

4.2.4 Node Quorum

In a high-availability cluster, one of the most important questions is who is allowed to modify – in any way – the shared resources and the data. In the case of a so-called cluster partition, where part of the cluster loses contact to the rest, the cluster has to arbitrate between the fragments, ensuring that a maximum of one partition continues to operate on the data.

A quorum is *the number (as a majority) of officers or members of a body that when duly assembled is legally competent to transact business*, and this – majority of eligible cluster nodes survives – is exactly how this is often achieved, but implementations where the *tie-breaker algorithms* take additional inputs such as access to a specific device are also common and in fact required in case of evenly split clusters where a majority does not exist.

Most clusters implement the notion that only one group of nodes has *quorum*; this is a globally exclusive. However, this breaks down for large and potentially hierarchial clusters, where multiple levels of quorum exist; the discussion is not yet closed on this topic.

4.3 Group Services

A cluster-aware application knows that it is being run in a distributed environment, spanning multiple nodes. A common model for this is that the application forms a *process group* across the cluster nodes, hence the name of this component.

4.3.1 Group Membership Services

This is very similar to the node membership services, just a layer above – the group also needs to know the list of processes joined and be informed of leaves and died processes.

4.3.2 Group Quorum Services

Discussions have also begun on deciding whether quorum is only a property of the node layer, or whether different process groups can have different ideas of whether they have quorum or not; good points have been put forward for both cases. While current clusters mostly only provide node quorum, the framework should be flexible enough to be extended to allow for group quorum, too.

4.3.3 Group Messaging Services

The group messaging service provides communication services for the process group.

Facing the same challenges as the node communication services, this is also further complicated by the fact that group messaging often has additional guarantees with regard to ordering of messages – *virtual synchrony*, where all messages in a group are globally ordered. Not all messaging services have this property, though.

While the syntax can be the same in both cases, the semantics and guarantees are different; it has not been decided yet how to cope with this.

4.3.4 Group Voting Services

A group very often has the need to vote; for example, to agree on a primary coordinator for performing a particular operation.

4.3.5 Group Barrier Services

A distributed process group also needs to ensure that certain operations are only started if all members of the group have reached a specific state; this state is called the *barrier*.

The barrier services will provide an easy means to implement such a synchronization mechanism.

4.3.6 Group Transaction Services

Transactions are a common approach to providing fault resilience; a transaction either commits as an atomic, correct transformation of the system state or is rolled back completely, the system is always in a valid state.

For clusters, distributed transactions with two-phase commit are especially useful. A *transaction monitor* initiates the transaction and leads the clients through, ending with either a *commit* or *rollback*. By utilizing the transactional framework, the distributed application inherits transactional properties. Journaling, logging and recovery are simplified compared to home-grown solutions.

The transaction framework itself is very generic; only the *resource managers*[11] vary. At the same time, it is also very complex – it has to deal with a multitude of failure modes and guarantee the *ACID*[12] properties for all of them – and this implies that a well-tested component which can be shared between different implementations is very desirable.

For a more detailed treatment of transactions, see [Gray93].

4.4 Resource Management

High-availability clustering, especially the so-called fail-over or switch-over systems, mostly centers around managing groups of resources.

A resource is a single physical or virtual entity that provides a service to clients or other resources. For example, a resource can be a single disk volume, a particular network address, or an application such as a web server. A resource is generally available for use over time

[11]Not to be confused with the resource concept used in fail-over systems!

[12]Atomicity, Consistency, Isolation and Durability

on two or more nodes in a cluster, although it usually can be allocated to only one node at any given point in time.

These basic resources are combined into resource groups, which are treated as atomic units by the fail-over system; moving only the IP address but not the database itself would be devastating.

4.4.1 Resource Instantiation Facility

If the cluster decides to online or offline a resource on a specific node, a generic mechanism for doing so must exist; in the most simple case, this could be an abstraction to provide *Secure Shell*-like functionality in a portable fashion.

4.4.2 Resource Monitoring (RWATCH)

Resources also have to be monitored for health; fail-over solutions which only deal with health at the node level are less useful, as approximately eighty percent of all failures are due to software.

This sub-component should provides a generic interface for polling the health status of a resource or notifying the cluster that a resource has failed.

4.4.3 Resource Fencing Services

As explained above, most resources that are not designed to be cluster-aware only support being active once at a time; havoc and data corruption will result otherwise.

The fencing sub-component ensures this by allowing the cluster to enforce policy with regard to access to shared resources. Various levels of granularity exists; some clusters will radically fence a failed or partitioned node from all shared resources by flipping its power switch[13], while others might support fine-grained control – see [Brower00] for a proposal.

Work has already begun on this sub-component; the *heartbeat* package by Alan Robertson includes a *STONITH* library for driving various power switches, which is a very effective, albeit brutal, way of ensuring that a node stops accessing shared resources immediately.

This sub-component is also related to node or group quorum, as some systems treat quorum as just another resource which a cluster partition either has or does not; if one side fully fenced the other, it can be sure that only itself has quorum and can proceed.

4.4.4 Resource Agents

Resource Agents are the glue layer between the switch-over software and the actual resources being managed. They aim to integrate the resource with the switch-over software without any modifications to the actual resource provider itself, by encapsulating it carefully and thus making it movable between real nodes in a cluster.

They are obviously very specific to the resource type they are encapsulating, however there is no reason why they should be specific to a particular switch-over solution.

All resources have a common set of methods which they must expose to the fail-over software; starting, stopping, a status query; this mostly maps one-to-one to Linux Standard Base init script functionality, with the excep-

[13]*Shot the other node in the head*, often abbreviated to *STONITH*

tion that multiple instances of the same type can be active on a given node simultaneously, uniquely identified by the resource instance parameters and that they are usually much more paranoid than init scripts.

These *resource agents* are a common concept among all switch-over solutions; they directly map to the range of applications supported by them.

However, once more, the actual interfaces vary in details, which means that the ISVs cannot provide a common plug-in with their application for all switch-over clustering solutions as they should, because hopefully nobody knows better how to control their application than the ISV, but instead the vendors of the clustering solutions usually have to provide this code themselves.

The current, already rather complete, draft for the interface between the *Resource Agent* and the switch-over software can be found on *on the OCF website*[14].

4.4.5 Cluster Resource Manager

If the node membership changes, a resource monitor reports failure or the administrator requests it, the *cluster resource manager* coordinates the recovery of the resource group; either locally or by transition to another node.

It is also responsible for ensuring exclusivity as explained under *Resource Fencing*.

4.5 Distributed Lock Manager

Many cluster-aware applications coordinate access to shared resources by locking the objects in question; this is a common approach to ensuring coherency and synchronization. Recovering locks in case of failures is a very complex topic and probably these are the hairiest programs in existence.

Due to historic reasons, almost every lock manager provides an interface both syntactical and semantically very similar to the VAX cluster lock manager, so we have a good precedent for the API here. It is assumed that just minor adjustments will be made to ensure a coherent design in the API.

One example of an Open Source distributed lock manager can be found *on IBM's website*[15]; it is already using *heartbeat* for the cluster infrastructure.

4.6 Cluster Monitoring

After a coherent API for a component has been defined, it will also be possible to have a common interface to *Network Management Systems* for all clusters on Linux.

An excellent case is the definition of a SNMP MIB for monitoring the cluster; basically functionality – exporting the current membership view, sending traps in case of membership changes et cetera map one-to-one to the APIs being discussed.

4.7 Cluster Configuration

Another complex issue is the configuration of a cluster; many approaches from a system administrators point of view have already been tried. This is not currently being discussed as part of the Open Clustering Framework for now.

[14]`http://www.opencf.org/standards/`

[15]`http://oss.software.ibm.com/dlm/`

4.8 The glue

4.8.1 PILS

Many modern Linux systems make extensive use of dynamically loadable object modules (plug-ins); this framework architecture is the perfect example.

However, most of these systems implement their plug-in and interface management in a way that satisfies their own immediate needs only, and is not generally directly usable by other projects.

PILS is an generalized and portable open source plug-in and interface loading system. PILS is being developed as part of the Open Cluster Framework reference implementation; please visit Alan's talk for details on it.

4.8.2 Kernel to user-space and back

Component providers can live both in the kernel and in user-space; nevertheless, consumers from both environments need access to the functionality.

If a common API for both exists, and a component only offers one aspect, a generic layer should be able to translate between kernel and user-space calls and vice versa.

4.8.3 Generic event mechanism

Almost all of the components have the need to deliver events to and to react to events triggered by their consumers.

An initial, very well prepared draft by Ram Pai and Joe DiMartino was posted to the OCF mailing list in April this year and triggered a lot of discussion; it is available via the mailing list archives referenced from our website.

5 Affiliation with other groups

5.1 Free Standards Group

Our goal is to become a working group under the umbrella of the FSG; we have already begun work on this. Our initial draft for the answers to their *questionnaire for new working groups*[16] can be found *on the OCF website*[17].

We hope to benefit from their organizational experience and legal framework, and being able to concentrate on the technical work instead.

5.2 IEEE Task force on Cluster Computing

The IEEE does have a task force dedicated to cluster computing; however, their focus is primarily on High Performance Computing. The *subgroup working on High Availability*[18] *TFCC-HA* takes a passive stance and mostly monitors the development; they have shown interest in working together with the Open Clustering Framework group.

5.3 MPI Forum

The High Performance Computing community has their own well-accepted message passing interface standard. As it has different goals, it is not immediately adaptable to the needs of the High Availability community. However, the Open Clustering Framework should allow a MPI layer on top of it, so that HA and HPC applications can run in the same cluster.

[16] `http://www.freestandards.org/policy/fsg102-newworkgroup-draft.txt`
[17] `http://www.opencf.org/OCF-fsg102-1.html`
[18] `http://www.csse.monash.edu.au/~rajkumar/tfcc/high-availability.html`

5.4 Service-Availability Forum

The *SAForum*[19] is a group of companies aiming to provide *Open Standards for on-demand, uninterrupted communication services*, which is marketing-speak for saying that they are aiming to provide much the same standards as OCF, just with an initial focus on the telco industry.

When OCF was created, we did not know about the SAForum; they appear to have formed roughly around the same time. However, we have tentatively begun to talk and found no major obstacles to a cooperation yet.

The main difference is that the SAForum is more closed, requiring a substantial entrance fee, even though the resulting standards are also supposed to be royality-free and should also allow an Open Source implementation.

The future relationship of the two efforts is unknown as of this time; the possibilities range from two totally disjunct efforts – which would be a waste, but even two standards is better than twenty-five – to a very close cooperation and potential joint working groups.

5.5 Members of the OCF group

It is kind of hard to answer this question at this point in time; as the Open Clustering Framework does not have a fixed organizational structure yet, no formal membership has to be requested or granted; as a consequence, while the list of members subscribed to our mailing list includes all the major players in the field, they cannot be listed here.

For an abbreviated list, please see *the OCF website*[20], or even better, visit the presentation itself, where you can meet the supporters in person.

[19]`http://www.saforum.org/`

[20]`http://www.opencf.org/`

5.5.1 The Linux HA Project

The *Linux HA project*[21] was founded by Alan Robertson; it provides *heartbeat*, the most well-known two-node switch-over solutions for Linux.

It is listed here in particular because its *heartbeat-stonith* library was the first component specifically targeted at clustering shared by multiple HA solutions – *heartbeat* itself, *Linux FailSafe* and *Kimberlite* – and because Alan has begun work on evolving *heartbeat* to a reference implementation of the OCF work.

6 Crystal ball gazing

The Open Clustering Group is beginning to capture vendor attention; as such, a more formal organization is likely required in the near future. We hope that this can be achieved in cooperation with the Free Standards Group.

Of course, at the same time, increasing mind- and market-share is very important; the cooperation with the FSG and the SAForum is an important issue here.

This is showing good progress; if a clustering vendors does not understand the benefits of standarization, their competition will gladly point them out to the customers. So everybody has an incentive to not be left out. As ISVs strongly benefit from this effort, their are putting their weight on it, too.

So far, we have not met someone who has not been supportive of the effort; everybody agrees it is required for Linux to succeed in the enterprise market, and that all parties involved benefit from it.

[21]`http://www.linux-ha.org`

Work has begun on the standards; two drafts have already been produced. The next step is to provide Open Source implementations of these and convince vendors to also implement them. The first API which will see deployment is very likely the Resource Agent specification.

There is still a lot of work to be done both on the standards, the models, and on developing a common choice of words. Participation is actively invited[22].

Join and contribute now, while admission is still free!

7 Acknowledgments

Besides thanks to all participants of the Open Clustering Framework – you are too many to list one by one – the following documents were particularly helpful in the preparation of this paper:

References

[AlanR01] `Alan's paper` where he outlined the idea behind the Open Clustering Framework for the first time.

[HM97] *Linux High Availability HOWTO*[23] by Harald Milz.

[GP97] *In search of clusters* by Gregory Pfister.

[GL01] *Linux Kongress 2001 Workshop Summary*[24] by Greg Louis.

[Gray93] *Transaction Processing: Concepts and techniques* by Jim Gray, Andreas Reuter; published by Morgan Kaufmann.

[Brower00] *Resource fencing framework*[25] by David Brower; the initial draft posted to the *linux-ha-dev* mailing list in 2000; lots of discussions followed.

[22] As a blunt advertisement, we are in serious need of a webmaster for the OCF website.

[23] `http://www.ibiblio.org/pub/Linux/ALPHA/linux-ha/High-Availability-HOWTO.html`

[24] `http://www.opencf.org/enschede2001/Enschede.summary.txt`

[25] `http://lists.community.tummy.com/pipermail/linux-ha-dev/2000-March/000394.html`

POSIX Threads and the Linux Kernel

Dave McCracken
IBM® Linux® Technology Center
Austin, TX
dmccr@us.ibm.com

Abstract

POSIX® threading (commonly called pthreads) has long been an issue on Linux. There are significant differences in the multi-thread archictecture pthreads expects and the architecture provided by Linux clone().

This paper describes the environment expected by pthreads, how it differs from what Linux provides, and explores ways to add pthread compatibility to the Linux kernel without interfering with Linux's current multithread model.

1 Introduction

POSIX threads has become a widely used way of adding concurrency to an application. However, it doesn't map well onto Linux because of significant differences in how each of them defines a process, and the effects those differences have on the runtime environment.

In this paper we will describe the two models, how they differ, and offer some suggestions for changes to Linux that will allow it to emulate the POSIX model for those applications that use POSIX threads while preserving the current behavior for all other applications.

1.1 Definitions

In our discussion of POSIX threads, first we need to define some terms. Much confusion arises in thread discussions because of disagreement over what various terms mean. For the purposes of this paper we'll use the following definitions:

process Traditionally a UNIX® process corresponded to an instance of a running program. More precisely it was an address space and a group of resources all dedicated to running that program. This definition is formalized in POSIX. For this paper we will use the term 'process' to mean this POSIX definition.

thread The term thread comes from the concept of a single thread of execution, ie a linear path through the code. POSIX defines a thread to be the resources necessary to represent that single thread of execution. A process contains one or more threads. We will use the term thread in this paper when referring to the resources necessary to define a single execution path as seen by the application.

task In Linux, the basic unit is a task. In a program that only calls fork() and/or exec(), a Linux task is identical to a POSIX process. The difference arises when a task uses the clone() system call to implement multithreading. The program then becomes a cooperating set of tasks which share some resources. We will use the term task to mean a Linux task.

2 History of POSIX Threads

Historically, the UNIX operating system has always had the concept of a process, which roughly equates to a running instance of a program. Each process has a set of resources associated with it, including an address space, a set of CPU registers, a process ID, a set of open file descriptors, a user ID, a stack, etc. While this is a powerful model, it only allows a single linear execution path. To gain any kind of concurrency with this model it is necessary to create multiple processes, often requiring some kind of inter-process communication, which is often expensive and unwieldy.

In the late 1980s, the concept of multiple threads of control became popular in the UNIX community. The fundamental idea was to take a limited set of a process's resources and make multiple instances of them, thus allowing concurrency within a single process. The resources selected were the minimum necessary to represent a single execution state. This primarily consisted of CPU registers and stack. Each instance was then called a 'thread'. This allowed concurrency within an application without the necessity of an inter-process communication mechanism. It is important to note that this model preserved the concept of a single process, and extended the definition to include multiple threads within that process.

At that time there were only a few experimental implementations of threads, and none in production. It was clear, however, that it addressed a growing need, especially with the prospect of multi-processor UNIX machines on the horizon.

At around this time the POSIX standardization effort was also underway. There was a strong push to create a common set of APIs that all UNIX implementations could be guaranteed to have. Several people put together an API that encapsulated the multi-thread model and proposed it for inclusion in POSIX. It was accepted in draft form under the real time extensions. While there was little real-world experience with threads at the time, the intent was to provide a common framework before multiple competing implementations appeared.

In the years since then the POSIX thread API (commonly known as pthreads) went through many revisions and was incorporated into the POSIX standard in 1996. Most if not all UNIX implementations include a pthread library, and there are many applications that use it.

3 POSIX Thread Model vs Linux Task Model

As we've stated before, the multithreading model used by POSIX is that of a single process that contains one or more threads. In contrast, the Linux multithread model is that of separate tasks that may share one or more resources. While this may sound like a small difference, the effects of this difference are far reaching.

3.1 Resources

The POSIX model is that all resources are global to the process except for the minimum set of resources that are necessary to represent a single thread of execution. This means that modifications to a resource will be seen by all other threads in the process.

In contrast, the Linux model has an independent set of tasks that have separate instances of all resources except for a few selected resources that may be shared. This sharing is selectable on a per-resource basis via flags passed to the clone() system call. All other resources have a separate instance for each task. A change to the resource in one task may not affect the equivalent resource in any other task.

The following resources are specific to a thread in POSIX, while all other resources are global to a process:

CPU registers
User stack
Blocked signal mask

The following resources may be shared between tasks via clone() in Linux, while all other resources are local to each task:

Address space
Signal handlers
Open files
Working directory

There are a number of resources that are process-wide in POSIX, but only task-specific in Linux, that cause compatibility problems. A partial list of the ones that cause the most problems includes process ID, parent process ID, credentials (user ID, group ID, etc), and pending signal mask.

3.2 Process-wide Actions

The fundamental difference between the models is that in POSIX a process can be addressed as a single entity, while in Linux it is a collection of independent tasks. There are several actions that can be done both from outside the process and within the process that will affect the whole process. In Linux, however, each of these actions will only affect one task, leaving the other tasks to continue without knowledge of the event.

The actions that are of special concern are:

Signals POSIX states that all signals sent to a process will be collected into a process-wide set of pending signals, then delivered to any thread that is not blocking that signal. Linux only supports signals that are sent to a specific task. If that task has blocked that particular signal, it may remain pending indefinitely.

Exit In POSIX, there are several actions that can request the death of the entire process. All threads are killed and the process exits with a status indicating the cause of the death. All of these actions in Linux will only kill the specific task, leaving all other tasks unaffected.

Suspend/Resume Certain signals have the default action of doing a suspend or resume. In POSIX, this action is defined to take effect on the entire process, which is translated to include all threads in that process. In Linux, the action only takes effect on the task the signal was delivered to.

Exec In POSIX, the effect of the execve() system call is to terminate all threads in the process, throw away the address space, and instantiate a new address space with a single thread. In Linux, if there is more than one task that shares the address space, the task that calls execve() is detached from the address space and has a new one created. All other tasks sharing that address space will continue to run.

4 Implementations of Multithread Libraries

4.1 Threading Styles

Multithread libraries typically come in one of three basic styles. Each has its advantages and disadvantages.

4.1.1 M:1

The first style, M:1, implements all threads in user space and appears to the kernel as s single-threaded process. This style is the most portable, in that it does not require any special features from the underlying kernel. One drawback is that it requires that all blocking system calls be emulated in the library via non-blocking calls to the kernel. This emulation adds significant overhead to system calls, in particular most IO. There are also some blocking system calls that can not be emulated via non-blocking calls. When these calls are used the entire process blocks. This style also does not allow the application to take advantage of any multiprocessor scheduling, since to the kernel scheduler it is still a single-threaded process.

This style is primarily of historical interest. Most current operating systems provide some support for multithreading at the kernel level, which provides improved performance.

4.1.2 1:1

The second style, 1:1, creates a kernel thread or task for each application thread. This has the advantage of being the simplest to implement at the library level, but each thread created becomes more expensive of kernel resources. This style is also the most dependent on the multithreading model of the underlying kernel.

There are some types of applications where this style is desirable, primarily when the application wants to create a small number of threads that each act independently and spend much of their time in runnable state.

4.1.3 M:N

The third style, M:N, provides the most flexibility. It is like M:1 threading in that it does not create a kernel thread for each application thread. The library creates multiple kernel threads, then schedules application threads on top of them. Most M:N thread libraries will dynamically allocate as many kernel threads as it needs to service the application threads that are actually runnable. This style is in some ways more heavyweight in that scheduling is occurring both in the kernel among the kernel threads for the process and in the library for the application threads, but it has the advantage of not consuming kernel resources for the application threads that are not actually runnable. This style also provides significantly better performance when threads in an application are synchronizing with each other, ie taking local mutexes. The library-level scheduler can switch between threads much faster because it doesn't have to enter the kernel.

Most multithreaded application perform better with this style, particularly applications that create large numbers of threads that only run sporadically.

4.2 Current Linux Thread Libraries

There have been multiple efforts to provide a pthread-compliant library for Linux. Early on in Linux's history only M:1 thread libraries were created, but were mostly abandoned as Linux developed better multithreading support at the kernel level.

The default library shipped with all the distributions is currently LinuxThreads. It is supported by the same group that provides glibc. The LinuxThreads library provides a pthread API, but internally it is primarily a wrapper for the Linux task model. It uses the 1:1 style, creating a task for each application thread using clone() and sharing the address space, the signal handlers, and the open files. This approach generally performs well, but the underlying differences from the POSIX thread model are exposed to the application. Applications that were coded to work with pthreads as specified by the standard may not work, and must be ported.

There is a new pthread library under development called NGPT. This library is based on the GNU Pth library, which is an M:1 library. NGPT extends Pth by using multiple Linux tasks, thus creating an M:N library. It attempts to preserve Pth's pthread compatibility while also using multiple Linux tasks for concurrency, but this effort is hampered by the underlying differences in the Linux threading model. The NGPT library at present uses non-blocking wrappers around blocking system calls to avoid blocking in the kernel.

5 Linux Kernel Changes for POSIX Compatibility

While it would be possible to emulate POSIX compatibility in a library, it would be extremely painful in many areas. A much simpler solution would be to add compatibility code to the Linux kernel, either to provide compatible behavior or provide hooks that would make it easier for a library to provide it. In this section we will describe some changes that make POSIX compatibility feasible. Some have already been included, some have patches available, and some have not yet been addressed. All the changes are intended to be optional, only enabled by request from the application or library. This would most likely be via additional flags to the clone() system call.

5.1 Thread Groups

One of the fundamental barriers to adding POSIX compatibility to Linux has been that Linux had no easy way to group all the tasks together that are part of what POSIX would call a process, and iterate through them. It was possible to find all tasks with the same address space, but only by looking at all tasks in the system. This limited what could be added at the kernel level.

This was addressed during the 2.4 development cycle with the addition of a concept called a ’thread group’. There is a linked list of all tasks that are part of the thread group, and there is an ID that represents the group, called the tgid. This ID is actually the pid of the first task in the group (pid is the task ID assigned with a Linux task), similar to the way sessions and process groups work. This feature is enabled via a flag to clone().

The task whose ID becomes the tgid is known as the ’thread group leader’. This task takes on special properties, since in most library implementations it will be the initial task running after exec(), and its ID is the one known to the parent who originally invoked the application.

As part of the thread group change, the getpid() system call was changed to return tgid instead of pid. This means that all tasks in a thread group will see the same pid. While this is correct for applications, pthread libraries will still need to be able to get the actual pid of the task, so the gettid() system call was added for them.

A corollary to the getpid() system call is getppid(). At present it returns the pid of the task that cloned() the task making the system call. For POSIX compatibility it should return the parent ID of the thread group leader.

While thread groups by itself only adds limited functionality, it provides the grouping necessary for other changes that will improve compatibility.

5.2 Signals

Signals have long been a difficult issue, beginning with early versions of the UNIX system. The question of how to handle signals in a multithreaded process has been debated since the early days of POSIX threading, and went through extensive changes in various drafts of the standard.

The kernel state maintained for a given signal consists of three pieces of information, the signal handler, the blocked flag, and the pending flag. The signal handler is an address of a user-level function to run when the signal is received. Special values of the signal handler allow the application to specify default behavior for that signal or to ignore it completely. The blocked flag is a flag that can be set by the application to temporarily prevent the signal from being delivered. The pending flag is set whenever that signal is sent to the application, and reset when the signal is actually delivered, ie the handler is run or other action is taken.

Signal handlers in Linux can be either per-process or per-task, controlled by a flag to clone(). This allows POSIX compatibility for handlers. POSIX specifies that the blocked flag should be per-thread, so the existing Linux behavior of having blocked flags for each task is compatible with POSIX.

The compatibility issue arises with the pending signal flag. POSIX states that signals are sent to the entire process, which means a thread context must be selected to run the handler. POSIX specifies that the delivery code must search the threads in the process and find one that does not have that signal blocked. If all threads are blocking that signal, it remains pending until one thread unblocks it, at which time that thread will run the handler. If more than one thread is not blocking the signal, POSIX does not specify which one will run the handler.

In Linux, all signals are sent to a specific task. Each task has its own pending signal flags, and the flag for that signal will be set. If that task has that signal blocked, it will remain pending until the task unblocks it, even though there may be other tasks in the process that do not have it blocked.

It is possible to partially emulate POSIX behavior in a pthread library by providing a complete signal layer, complete with its own handler array, per-thread blocked masks, and pending signal mask. This requires that the library register its own signal handler in the kernel for all signals, and to not block signals at the kernel level. The biggest problem with this approach is the significant added complexity and performance cost of duplicating the functionality. There are also circumstances where the application will still see interrupted system calls when all threads are blocking a signal or the signal is supposed to be ignored.

In support of the NGPT project I wrote a patch that allows libraries to provide POSIX signal emulation. The patch works in conjunction with thread groups. When a signal arrives for any task in a thread group, that signal is redirected to the thread group leader. This allows a pthread library to leave signals unblocked in the thread group leader task, and receive all signals directed at any task in the process. It doest not directly support POSIX compatibility, but gives the library the tool it needs to provide its own compatibility.

The thread group leader patch has some drawbacks of its own, however. It creates a bottleneck in an application with large numbers of signals. It also still requires significant code in the library to handle blocking signals for each thread, ie if all threads block a signal, it still needs to be blocked at the kernel level.

Another issue with this approach is that it would make it more difficult to do a thin 1:1 pthread library, since it would still have to provide significant signal code in the library. A better solution for this would be to actually add a shareable structure to the kernel for pending signals, with the attendant code to check all tasks in a thread group to see whether any of them can receive the signal. This solution would also address the bottleneck issue.

5.3 Credentials

Credentials are the collective identity associated with a process or task, ie the user ID, the group ID, the list of groups, and the capabilities. POSIX states that the credentials are per-process, ie when one thread within the process changes some part of the credentials, all threads see the change. In Linux, the credentials are per-task, so it's possible to have two tasks in a process running under different user IDs, for example.

The simple solution to this is to change credentials to be a shareable structure. This would preserve existing behavior, but allow processes that wish POSIX behavior to share credentials.

5.4 Semaphore Undo

Another resource that under POSIX is process-wide is System V semaphores. This primarily becomes an issue when an application uses the undo feature. This feature will reset semaphores on process exit. In Linux, the semaphore state is per-task, so when each task exits it will undo the semaphore. POSIX processes assume that the semaphore will continue to maintain its state until the entire process exits.

This problem is another one that can be solved by sharing state between tasks when a flag is passed to clone(). A patch for this exists, but has not yet been accepted.

5.5 Process-wide Actions

There are some actions that POSIX defines to be process-wide which under Linux are per-task. Some of these actions are initiated from inside the kernel and can not be detected and emulated inside a library.

Exit A difficult compatibility issue is that of exit. POSIX defines several actions that can result in the entire process exiting, including the exit() system call and default actions for many signals. This process exit should produce an exit status that can be passed to a waiting parent. This means that any thread in the process can cause the entire process to exit and produce a status back to a waiting parent.

The Linux behavior is dramatically different. Each of these exit actions results in the termination of a single task, leaving all other tasks in the process running. If the task is the initial one created by fork(), the parent will receive its exit status and may assume the process has exited when in fact it is still running in other tasks.

Exec Under POSIX, an execve() system call from any thread in a multithreaded process will cause all other threads in that process to terminate and the calling thread will complete the exec. The entire address space associated with that process will be discarded, and a new one created.

In Linux, when a task calls execve(), it is detached from the address space, then a new address is created to complete the exec. If any other task is using the old address space it will continue to run.

Suspend/Resume Some signals have the default action of initiating a suspend or a resume. POSIX states that this will occur on the entire process by suspending or resuming all threads in that process. Linux only applies the suspend or resume to the task receiving the signal and does not affect any other task.

A possible solution for these would be a kernel function that iterates through an entire thread group and applies the requested action to each task in that group. Special care would have to be taken to preserve the proper exit status to any waiting parent. Synchronizing all the tasks in a thread group is expected to be a difficult problem.

6 Conclusion

We have shown how POSIX threading uses a different model than the Linux task model, and how that affects pthread libraries on Linux. We have also discussed some things that have been and could be done to the Linux kernel to better allow pthread libraries to emulate the POSIX behavior. These changes could be added without disrupting the current Linux task behavior, allowing Linux to support both the POSIX multithread model and its own cooperating task model.

Lawyer Foo

Read Copy Update

Paul E. McKenney
Linux Technology Center
IBM Beaverton
pmckenne@us.ibm.com
http://www.rdrop.com/users/paulmck

Dipankar Sarma
Linux Technology Center
IBM India Software Lab
dipankar@in.ibm.com

Andrea Arcangeli
SuSE Labs
andrea@suse.de

Andi Kleen
SuSE Labs
ak@suse.de

Orran Krieger
IBM T. J. Watson Research Center
okrieg@us.ibm.com
http://www.eecg.toronto.edu/˜okrieg

Rusty Russell
Linux Technology Center
IBM Canberra
rusty@au.ibm.com

Abstract

Read-copy update is a mechanism for constructing highly scalable algorithms for accessing and modifying read-mostly data structures, while avoiding cacheline bouncing, memory contention, and deadlocks that plague highly scalable operating system implementations. In particular, code that performs read-only accesses may be written without any locks, atomic instructions, or writes to shared cachelines, even in the face of concurrent updates. We reported on the basic mechanism last year, and have produced a number of Linux™ patches implementing and exploiting read copy update.

This paper evaluates performance of a number of read copy update implementations for non-preemptive Linux kernels, and outlines a new implementation targeted to preemptive Linux kernels.

1 Introduction

The past year has seen much discussion of read-copy update and the design and coding of a number of read-copy-update implementations. These implementations make a number of different tradeoffs, and this paper takes a first step towards evaluating them.

Comparison of read-copy update to other concurrent update mechanisms has been done elsewhere [McK01b, Linder02a]. These comparisons have shown that read-copy update can greatly simplify and inprove performance of code accessing read-mostly linked-list data structures (such as FD management tables and dcache data structures). Evaluation of read-

The views expressed in this paper are the authors' only, and should not be attributed to SuSE or IBM.

copy update in other environments has shown that the read-copy update can also improve performance of code modifying linked-list data structures when there is a high system-wide aggregate update rate across all such data structures [McK98a].

Section 2 fills in some background on read-copy update. Section 3 gives an overview of the design choices of the Linux read-copy update non-preemptive implementations. Section 4 compares performance and complexity of these implementations, with emphasis on the grace-period latency that determines the incremental memory overhead compared to non-read-copy-update locking algorithms. Section 5 overviews the implementations, focusing on `call_rcu()`, scheduler instrumentation, and timer processing. Section 5 also describes how the *rcu* algorithm may be adapted to a preemptible kernel. Section 6 describes future plans, Appendix A provides implementation details, and Appendix B discusses memory ordering issues encountered when inserting into a read-copy-protected data structure.

2 Background

This section gives a brief overview of read-copy update, more details are available elsewhere [McK98a, McK01a, McK01b]. Section 2.1 contains a glossary of read-copy-update-related terms, Section 2.2 presents concepts, Section 2.3 presents the read-copy-update API, Section 2.4 describes the IP route cache patch that uses read-copy update, Section 2.5 describes the module race reduction patch that uses read-copy update, and Section 2.6 gives an overview of how read-copy update may be used in a preemptive kernel.

2.1 Glossary

Live Variable: A variable that might be accessed before it is modified, so that its current value has some possibility of influencing future execution state.

Dead Variable: A variable that will be modified before it is next accessed, so that its current value cannot possibly have any influence over future execution state.

Temporary Variable: A variable that is only live inside a critical section. One example is a auto variable used as a pointer while traversing a linked list.

Permanent Variable: A variable that is live outside of critical sections. One example would be the header for a linked list.[1]

Quiescent State: A point in the code where all of the current entity's temporary variables that were in use before a specified time are dead. In a non-preemptive Linux kernel, a context switch is a quiescent state for CPUs. In a preemptive Linux kernel, a voluntary context switch is a quiescent state, but for threads. In this paper, quiescent states are global events, as opposed to being associated with a specific data structure.

Grace Period: Time interval during which all entities (CPUs or tasks, as appropriate) pass through at least one quiescent state. Note that any time interval containing a grace period is itself a grace period.

The key point underlying read-copy update is that if you remove all permanent-variable references to a given item, then wait for a grace

[1]Yes, it is possible for the same variable to be temporary sometimes and permanent at other times. However, this can lead to confusion, so is not generally recommended.

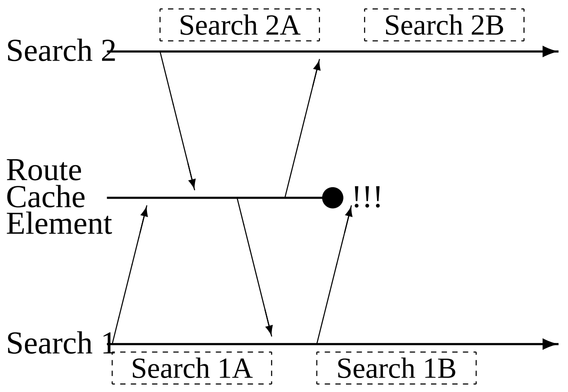


Figure 1: Race Between Deletion and Search

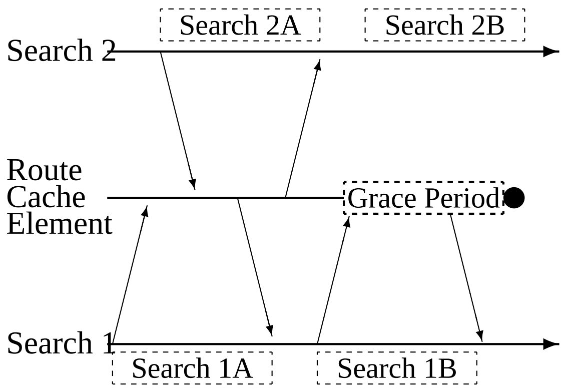


Figure 2: Read-Copy Update Handling Race

period to expire, there can be no remaining references to that item. The item can then be safely freed up. This process is described in more detail in the next section.

2.2 Concepts

Read-copy update allows lock-free read-only access to data structures that are being concurrently modified. The accessing code needs neither locks nor atomic instructions, and can often be written as if the data structure were unchanging, in a “CS 101” style. Read-copy update is typically applied to linked data structures where the read side code traverses links through the data structure in a single direction.

Without special action on the update side, the read side would be prone to races with deletions, as illustrated in Figure 1, which shows two tasks searching a list that contains an element that is concurrently deleted by a third task (signified by the line labelled "Route Cache Element"). To handle such race conditions, the update side uses a two-phase update discipline:

1. Remove permanent-variable pointers to the item being deleted.
2. After a grace period has elapsed, free up the item’s memory.

The grace period is not a fixed time duration, but is instead inferred by checking for per-CPU quiescent states, such as context switches. Since kernel threads are prohibited from holding locks across a context switch, they also prohibited from holding pointers to data structures protected by those locks across context switches–after all, the entire data structure could well be deleted by some other CPU at any time the lock is not held.

Therefore, a simple implementation of read-copy update might declare the grace period over once it observed each CPU performing a context switch. Now, the first phase removed all global pointers to the item being deleted, and kernel threads are not permitted to hold references to the item across a context switch. Therefore, CPUs that have performed a context switch after the completion of the first phase have no way to gain a reference to the item being deleted. Thus, once all CPUs have performed a context switch, it is safe to free up the item being deleted from the list.

With this approach, searches already in progress when the first phase executes might (or might not) see the item being deleted. However, searches that start after the first phase completes are guaranteed to never reference this item. Therefore, the item may be safely

```
void synchronize_kernel(void);
struct rcu_head {
    struct list_head list;
    void (*func)(void *obj);
    void *arg;
};
void call_rcu(struct rcu_head
    *head,
    void (*func)(void *arg),
    void *arg);
```

Figure 3: Read-Copy Update API

freed once all searches in progress at the end of the first phase have completed, as shown in Figure 2.

Efficient mechanisms for determining the duration of the grace period are key to read-copy update.

2.3 Read-Copy Update API

Figure 3 shows the external API for read-copy update. The `synchronize_kernel()` function blocks for a full grace period. This is a simple, easy-to-use function, but imposes expensive context-switch overhead on its caller. It may not be called with locks held or from BH/IRQ context.

Another approach, taken by `call_rcu()` is to schedule a function to be called after the end of a full grace period. Since `call_rcu()` never sleeps, it may be called with locks held or from BH (and perhaps also IRQ) context. The `call_rcu()` function uses its `struct rcu_head` argument to store the specified callback function and argument, and the read-copy-update subsystem then uses this struct to schedule the callback invocation. An `rcu_head` is often placed within a structure being protected by read-copy update.

A typical use of `call_rcu` is shown in Figure 4, where an element is deleted from a circular doubly linked list with a header element. Here `my_free()` is a wrapper around `kfree()`, and the lock is used only to serialize concurrent calls to `delete()`. Since the element's `next` and `prev` pointers are unaffected, and since `my_free()` is not called until a grace period has elapsed, non-sleeping reading tasks may traverse the list concurrently with the deletion of the element without danger of a NULL pointer or a pointer to the freelist. This is a common read-copy-update idiom: `kfree()` is replaced by a `call_rcu()` to a function that is a wrapper around `kfree()`.

```
void delete(struct el *p)
{
   spin_lock(&list_lock);
   p->next->prev = p->prev;
   p->prev->next = p->next;
   spin_unlock(&list_lock);
   call_rcu(&p->my_rcu_head,
   my_free, p);
}
```

Figure 4: Read-Copy Dequeue From Doubly-Linked List

2.4 Read-Copy Update and IP Route Cache

Read-copy update has been used in a number of OSes, including several patches to Linux [McK01b, Linder02a]. This section describes how read-copy update may be used in the Linux IP route cache. This modification was done to validate the RCU implementations, rather than in response to a known performance problem in the IP route cache.

The Linux IP route cache uses a reader-writer lock, so multiple searches may proceed in parallel. However, the multiple readers' lock acquisitions result in the cacheline bouncing. Read-copy update may be used to eliminate this read side cacheline bouncing:

```
@@ -314,13 +314,13 @@
 static inline void rt_free(
                struct rtable *rt)
 {
-   dst_free(&rt->u.dst);
+   call_rcu(&rt->u.dst.rcu_head,
      (void (*)(void *))dst_free,
               &rt->u.dst);
 }

 static inline void rt_drop(
                struct rtable *rt)
 {
    ip_rt_put(rt);
-   dst_free(&rt->u.dst);
+   call_rcu(&rt->u.dst.rcu_head,
+     (void (*)(void *))dst_free,
+              &rt->u.dst);
 }
```

Figure 5: dst_free() Modifications

1. Delete all calls to `read_lock()`, `read_unlock()`, `read_lock_bh()`, and `read_unlock_bh()`.
2. Replace all calls to `write_lock()`, `write_unlock()`, `write_lock_bh()`, and `write_unlock_bh()` with the corresponding member of the `spin_lock()` family of primitives.
3. Add `rmb()` primitives on the read side between the fetch of the pointer and its dereferencing. These should be replaced by `read_barrier_depends()` when it becomes available.
4. Replace all calls to `dst_free()` with a call to `call_rcu()` which causes `dst_free()` to be invoked after the end of a following grace period, as shown in Figure 5.

This results in a significant decrease in `ip_route_output_key()` overhead during a workload that transmits a fixed number of random-sized IP packets to a single destination, as shown in Figure 6. This workload was run on an 8-CPU 700MHz Pentium^{TM} III Xeon^{TM} with 1MB L2 cache and 6GB of memory.

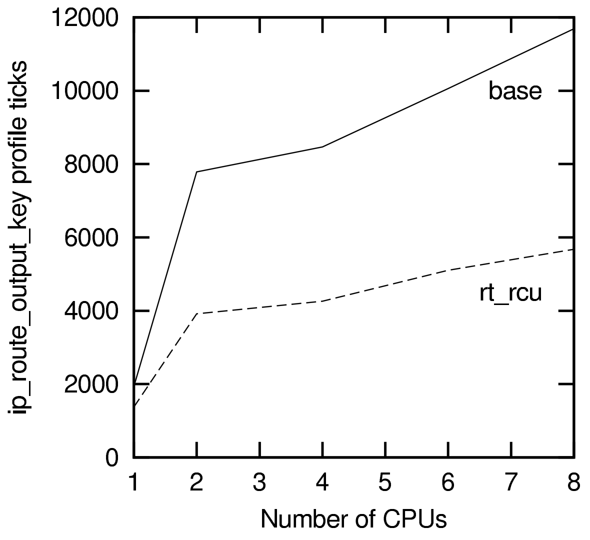


Figure 6: IP Route Cache Speedup Using rcu

Figure 7 shows the total non-idle kernel profile ticks for this same workload. This data shows the IP route cache speedup is real; it is not happening at the expense of other processing in the system. The overall speedup is quite small, as expected, given that the change was not motivated by a known performance problem.[2] More compelling Linux-based read-copy-update results include a 30% improvement for FD management [McK01b] and a 25% improvement for dcache management [Blanchard02a, Linder02a]

2.5 Read-Copy Update and Module Race Reduction

Linux 2.4 is subject to races between module unloading and use of that module. These races

[2]However, we will be measuring this patch on various workloads as Linux's scaling continues to improve.

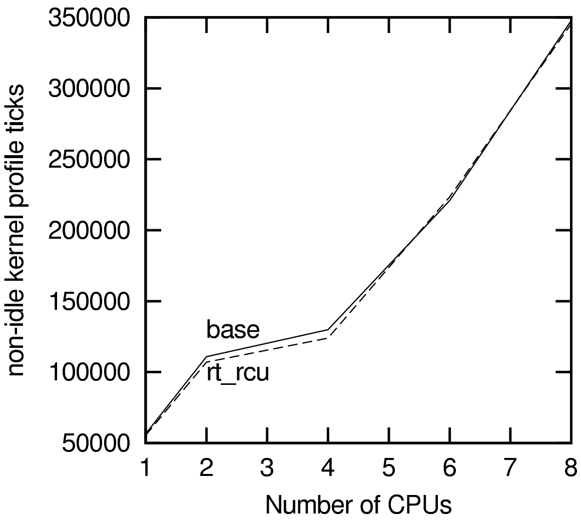


Figure 7: IP Route Cache System Performance Using rcu

can result in the racing code that is attempting to use the module holding a reference to newly freed memory, most likely resulting in an “oops.”

One way to reduce the likelihood of these races occurring is to wait for a grace period after removing the module structure from the `module_list` before `kfree()`ing it in `free_module()` [Kleen02a]. Races can still occur, but the race’s window has been decreased substantially. The change is a one-liner (not counting comments), as shown in Figure 8.

As noted earlier, this change does not address all the module-unloading problems. However, we hope that it can be a basis for a full solution. This approach is now being used in production in SuSE Linux.

2.6 Read-Copy Update and Preemption

Preemption has recently been added to Linux in 2.5.4. The addition of preemption means that read side kernel code is subject to involuntary context switches. If not taken into account, this leads to premature flagging of the ends of grace periods. There are two ways to handle preemption: (1) explicitly disabling preemption over read side code segments, and (2) considering only *voluntary* context switches to be quiescent states.

```
@@ -1065,6 +1066,12 @@
      p->next = mod->next;
  }
  spin_unlock_irqrestore(&modlist_lock,
                         flags);

+ /* Wait for all other cpus to go
+  * through a context switch. This
+  * doesn’t plug all module unload
+  * races, but at least some of
+  * them and makes the window much
+  * smaller.
+  */
+ synchronize_kernel();

      /* And free the memory.  */
```

Figure 8: Module Unloading

Explicitly disabling preemption over read side code segments adds unwanted overhead to reading processes, and removes some of the latency benefits provided by preemption. In contrast, considering only voluntary context switches to be quiescent states allows the kernel to reap the full benefit of reduced latency. This scheme for tracking only voluntary context switches is inspired by the K42 implementation [Gamsa99].[3] The main drawback is increased length of grace periods. This paper focuses on the voluntary context switches option and its effects.

[3]K42’s extensive use of blocking locks and short-lived threads results in use of thread termination rather than voluntary context switch as the K42 quiescent state. In addition, Linux migrates preempted tasks to other CPUs, which requires special tracking of tasks that have been preempted since their last voluntary context switch.

3 Read-Copy Update Implementations

As noted earlier, the key to read-copy update is a CPU-efficient mechanism for determining the required duration of the grace period. This mechanism is permitted to overestimate the grace-period duration, but the greater the overestimation, the greater the amount of memory that will be consumed by waiting callbacks. There are a number of simple and efficient algorithms to determine grace-period duration, and this paper reviews a number of them.

There are a number of design parameters for a read-copy update implementation:

1. Batching. Many implementations batch requests, so that a single grace-period identification can satisfy multiple requests. Batching is particularly important for implementations with heavyweight grace period identification mechanisms. Although there have been implementations without batching [McK01a], all implementations described in this paper do batching.

2. Deducing the length of the grace period. The simplest mechanisms force a grace period by a reschedule on all CPUs in non-preemptive kernels. However, this approach is relatively expensive, particularly if extended to cope with preemptible kernels. More efficient implementations use something like per-CPU quiescent-state counters to deduce when the natural course of events has resulted in the expiration of a grace period.

3. Polling mechanism. Implementations that deduce when a grace period has ended must use some mechanism to be informed of this event:

   (a) Adding explicit checks to code corresponding to quiescent states, for example, *rcu-sched*'s hooks in the Linux scheduler shown in Figure 29. Explicit checks allow fast response to quiescent states, but add overhead when there are no read-copy callbacks in flight.

   (b) Adding counters to code corresponding to quiescent states, and using kernel daemons to check the counters, as shown in Figure 13. This approach adds some complexity, but greatly reduces the overhead when there are no read-copy callbacks in flight.

   (c) As above, but use tasklets instead of kernel daemons to do the checking. This further reduces the overhead, but uses more exotic features of Linux.

   (d) As above, but use a per-CPU timer handler [Sarma02a] instead of tasklets to do the checking. It is not yet clear which of tasklets and timer handlers are preferable.

   If the implementation forces the end of the grace period, it must similarly use a mechanism for doing so:

   (a) Scheduling a thread on each CPU in turn. This has the advantage of immediacy, but cannot be used from BH or IRQ, and gains no performance benefit from batching.

   (b) Reserving a kernel daemon that, upon request, schedules itself on each CPU in turn. This permits batching and use from BH and IRQ, but is more complex.

4. Request queuing. Requests may be queued globally or on a per-CPU basis.

Grace periods must of course always be detected globally, but per-CPU queuing can reduce the CPU overhead incurred by `call_rcu()`. This is a classic performance/complexity tradeoff. The correct choice depends on the workload.

5. Quiescent state definition. For non-preemptive kernels, context switch is a popular choice. For preemptive Linux kernels (such as Linux 2.5), voluntary context switch may instead be used.

6. Environments. If `call_rcu()` use is prohibited in the BH or IRQ contexts, then more kernel functionality is available to the implementor of `call_rcu()`, and less overhead is incurred.

Section 5 describes a number of Linux implementations of read-copy update, summarized in Table 1.

All the implementations in Table 1 except *rcu-preempt* assume a run-to-block kernel. Section 5.7 describes *rcu-preempt*, which operates efficiently in a preemptive kernel.

The "QS" column lists the quiescent states that each algorithm tracks, "I" for idle-loop execution, "C" for context switch, and "U" for user-mode execution.

The "BH/IRQ Safe" column indicates whether code running in BH/IRQ context may safely delete elements of a read-copy-update-protected data structure that is accessed by base-level code with interrupts enabled. The *rcu-poll* implementation is BH safe, but is IRQ unsafe by choice, in order to eliminate the overhead of interrupt disabling and enabling that would otherwise be incurred on each call to `call_rcu()`. If a strong need arises for use of `call_rcu()` from IRQ context, trivial changes to *rcu-poll* will render it IRQ safe.

The read-copy-update implementations discussed in this paper choose different points in this design space. These implementations are freely available [LSE]. The *X-rcu*, *rcu*, and *rcu-ltimer* implementations are similar to the ptx^{TM} implementation, using per-CPU timers, kernel daemons, and architecture-dependent timer support, respectively. The *rcu-taskq* implementation is an extremely compact implementation in which a kernel task forces per-CPU kernel daemons to run on their respective CPUs. The *rcu-sched* implementation uses ring counters within the Linux scheduler, and boasts an extremely low overhead `call_rcu()` implementation. It is also the only known read-copy-update implementation that uses absolutely *no* locks, interrupt masking, memory barriers, or atomic instructions. The *rcu-poll* implementation is designed for minimal overhead when there are no outstanding read-copy callbacks, and boasts very low `call_rcu()` latencies. Finally, the *rcu-preempt* implementation adapts the *rcu* implementation to work correctly in preemptible kernels. We will adapt some of the other implementations for preemptible use, as well. These implementations are described in more detail in Section 5 and Appendix A.

4 Performance and Complexity Comparisons

Table 2 shows the amount of overhead incurred by each implementation when there is no read-copy update activity in the system. The *rcu-taskq* implementation does best by this measure, with absolutely no overhead. The *rcu-poll* and *rcu-preempt* are next, with but a single local non-atomic increment in the scheduler. The *rcu-preempt* also incurs overhead on each preemption, as *rcu-poll* likely will once it is adapted to run in a preemptive kernel. The other implementations incur timer overhead under idle conditions.

An important figure of merit for a read-copy-update implementation is the grace period latency. The greater the latency, the more memory is waiting on the internal lists for the current grace period to end. On the other hand, longer latency results in higher efficiency, since the per-callback-batch processing is done less frequently, spreading the overhead over more `call_rcu()` requests. The best tradeoff depends on the workload: systems with very infrequent `call_rcu()` invocations would prefer small latency in order to conserve memory, while systems with very frequent `call_rcu()` invocations would prefer larger latencies in order to amortize the overhead of detecting a grace period over more `call_rcu()` invocations.

Name	Batch?	Deduce	Poll	Queuing	QS	BH/IRQ Safe
X-rcu	Yes	counters	timers	per-CPU	IC	Yes
rcu	Yes	counters	daemons	per-CPU	C	Yes
rcu-poll	Yes	counters	tasklet	global	C	BH Only
rcu-ltimer	Yes	counters	tasklets	per-CPU	IUC	Yes
rcu-taskq	Yes	No	daemons	global	C	Yes
rcu-sched	Yes	counter ring	N/A	per-CPU-rrupt	IC	Yes
rcu-preempt	Yes	counters	timers	per-CPU	IC	Yes

Table 1: Read-Copy Implementations

	RCU Idle Memory Refs			
Name	Switch	Preempt	Timer	Timer Type
X-rcu	1 local		8 local + 1 global + 1 timer	50ms per CPU
rcu	1 local		2 local + 1 global read + 1 global write + 1 timer + #CPU * up()	50ms global
rcu-poll	1 local			
rcu-ltimer	1 local		7 local + 1 global + 1 tasklet	per CPU
rcu-taskq				
rcu-sched	1 global read			
rcu-preempt	1 local	6 local		

Table 2: Read-Copy Idle Overhead

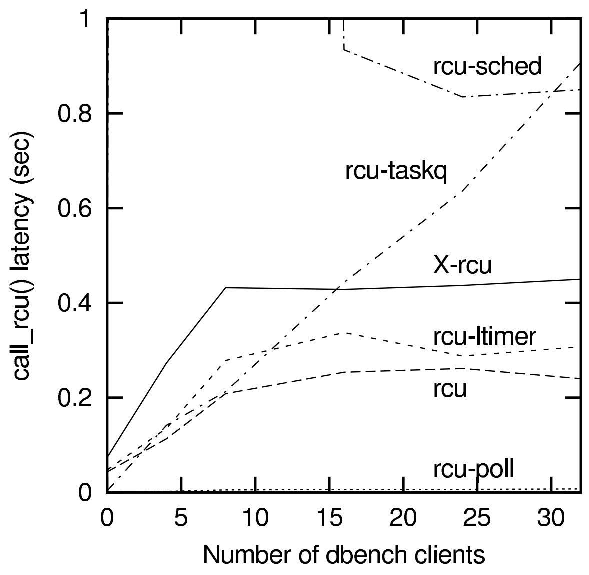


Figure 9: call_rcu() Latency Under dbench Load

This latency depends on worst-case kernel codepath length, the workload, and the details of the read-copy-update implementation. Figure 9 shows the `call_rcu()` latency for the different read-copy update algorithms as a function of offered load to the dbench benchmark. It was run on an 8-CPU 700MHz Xeon system with 1MB L2 caches and 6GB of memory using the dcache-rcu patch [LSE]. The winner by far is *rcu-poll*, which keeps latencies below 10 milliseconds (and below 250 *microseconds* on an idle system) by allowing quiescent states to be detected in parallel and by its aggressive forcing of scheduling when a grace period is required (see Figure 10, which shows the same data on a semilog plot). Therefore, *rcu-poll* is preferable on systems that invoke `call_rcu()` infrequently. The *X-rcu*, *rcu-*

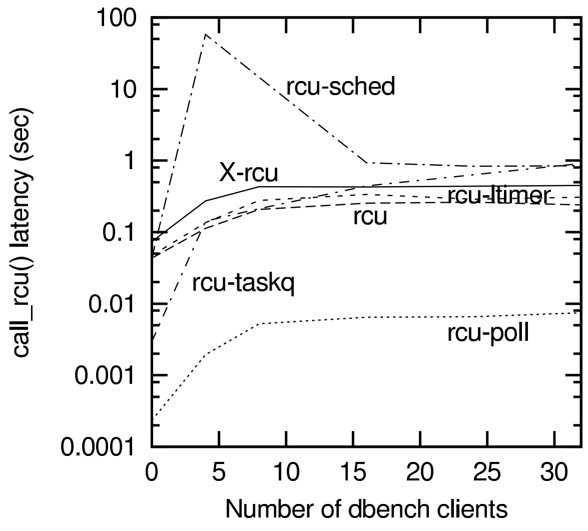


Figure 10: call_rcu() Latency Under dbench Load (logscale)

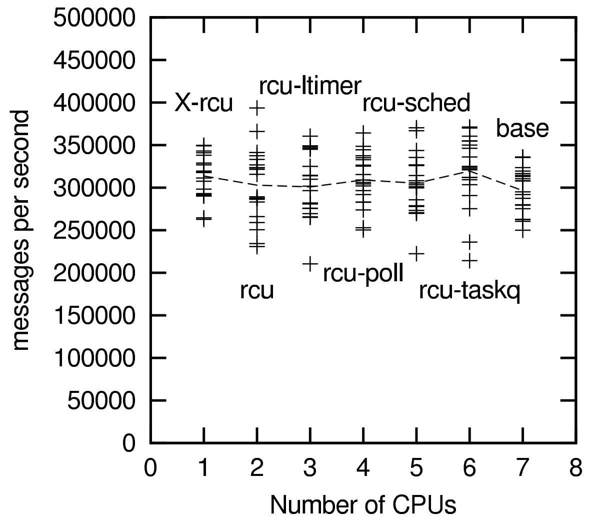


Figure 11: RCU Performance on Chat Benchmark

ltimer, and *rcu* implementations have larger latencies that are well bounded as the number of clients increase. These algorithms are thus preferable on systems that have very high rates of `call_rcu()` invocation.

The *rcu-sched* algorithm exhibited very large latencies (14.5 seconds at 8 clients and 57.7 seconds at 4 clients), which we are investigating. The *rcu-taskq* algorithm's latencies increases with increasing numbers of clients, because this algorithm requires the CPUs to pass through quiescent states sequentially, and because keventd (which runs the taskq's) runs at low priority.

Read-copy update can pose a tradeoff between latency and overhead, since increased latency increases the number of callbacks that are serviced by a single grace period. To evaluate this tradeoff, Figure 11 compares the performance of the chat benchmark with 20 rooms and 500 messages on a 4-CPU 700MHz Pentium III Xeon system with 1MB L2 caches and 1GB memory. This benchmark was run using the read-copy-update-based IP-route-cache and FD management patches [LSE]. These results show little sensitivity to the read-copy-update algorithm. We are collecting more data on other workloads.

Table 3 shows the number of lines in each algorithm's patch. The "All Archs" column gives the size of the patch applied to all architectures currently in the kernel, while the "One Arch" column gives the size of each patch applied to only one architecture. Architecture-independent patches will have the same number in both columns. The *rcu-taskq* implementation is the simplest, and so might be a good place to start looking at read-copy-update implementations.

The *rcu-ltimer* patch works only on the i386 architecture, so the figure for "All Archs" is an estimate based on the i386-specific portion of the patch, which simply invokes `RCU_PROCESS_CALLBACKS()` from the `smp_local_timer_interrupt()` function. The *rcu-sched* patch contains code to guard against architectures that shut down their CPUs when idle.

Name	Size of Unified Diffs	
	All Archs	One Arch
rcu-taskq-2.5.3-1.patch	237	237
rcu-poll-2.5.3-1.patch	378	378
X-rcu-2.5.3-4.patch	424	424
rcu-sched-2.5.3-1.patch	575	333
rcu-2.5.3-2.patch	603	603
rcu-preempt-2.5.8-3.patch	682	682
rcu-ltimer-2.5.3-1.patch	*742*	514

Table 3: Read-Copy Implementation Complexity

5 Read-Copy Update Implementation Overviews

The following sections summarize the `call_rcu()` implementation, the quiescent-state instrumentation (usually in the scheduler), and the high-level timer processing. More details on the more-complex implementations may be found in Appendix A, and patches for each may be found on the Linux Scalability Effort website [LSE].

5.1 X-rcu

X-rcu is loosely based on the ptx read-copy-update implementation. It uses a per-CPU context switch counter to instrument this quiescent state, uses per-CPU queues to track callbacks, and per-CPU timers to track quiescent states as needed to find the end of grace periods. The timers further check for running from idle, which is a second quiescent state. Dipankar Sarma implemented this variant to evaluate the use of timers rather than the kernel daemons or architecture-dependent timer hooks used by the *rcu* and *rcu-ltimer* implementations.

This implementation depends on patches that have not yet appeared in 2.4, 2.5, or both. The required patches include:

1. Rusty Russell's per-CPU data area patch [Russell02a] permits more natural maintenance of per-CPU data. It permits the context switch counter to be maintained separately from the rest of the per-CPU state, which avoids some nasty header file cyclic dependencies between *interrupt.h*, *fs.h*, and *sched.h*. This separation means that *rcupdate.h* need not include *interrupt.h*, which makes it easier to include *rcupdate.h* in lower-level kernel subsystems, such as dcache. This patch recently was accepted into the Linux 2.5 kernel.

2. Per-CPU timer support [Sarma02a]. This patch enhances Ingo Molnar's smptimers patch to guarantee that timers queued in a CPU always get executed on the same CPU where they were enqueued. This guarantee allows per-CPU quiescent state checking to be performed in a clean and architecture independent way. In addition, timers have significantly lower overhead than kernel daemons.

The `call_rcu()` function constructs the callback and enqueues it onto the current CPU's `rcu_nextlist`, as shown in Figure 12.

Figure 13 shows how the scheduler is instrumented. The added line 5 compiles to a local increment, with no locking, atomic operations, or cacheline bouncing.

Figure 14 shows the processing done by the per-CPU timer handler, currently set up to execute every 5 jiffies on each CPU. This code detects idle-loop execution and counts this as a quiescent state. It then invokes `rcu_process_callbacks()` to advance callbacks as ends of grace periods are detected.

```
void call_rcu(struct rcu_head
    *head,
    void (*func)(void *arg),
    void *arg)
{
    unsigned long flags;

    head->func = func;
    head->arg = arg;
    local_irq_save(flags);
    list_add_tail(&head->list,
        &this_cpu(rcu_nextlist));
    local_irq_restore(flags);
}
```

Figure 12: *X-rcu* call_rcu() Implementation

```
@@ -685,6 +686,7 @@
 switch_tasks:
  prefetch(next);
  prev->work.need_resched = 0;
+ per_cpu(rcu_qsctr, prev->cpu)++;

  if (likely(prev != next)) {
          rq->nr_switches++;
```

Figure 13: *X-rcu* Scheduler Instrumentation

```
static void rcu_percpu_tick(void)
{
    /* Check for idle loop */
    if (task_idle(current))
        this_cpu(rcu_qsctr)++;
    rcu_process_callbacks();
}
```

Figure 14: *X-rcu* Timer Processing

This callback advancement is described in Appendix A.1.

5.2 rcu

The *rcu* patch is also based on the ptx algorithm. Unlike the *X-rcu* patch described in Section 5.1, *rcu* has minimal dependencies on other patches. It is otherwise quite similar, using per-CPU queues of callbacks and context-switch counters instrumenting the quiescent states. However, it uses per-CPU kernel daemons to periodically check for the end of grace periods, which means that it cannot easily check for the CPU having been idle. These daemons are awakened by a timer that is scheduled only when there is at least one callback in the system. Dipankar Sarma implemented this variant to evaluate use of kernel daemons rather than architecture-dependent timer hooks.

The `call_rcu()` function simply constructs the callback, enqueues it onto the current CPU's `RCU_nxtlist`, then schedules the current CPU's tasklet, as shown in Figure 15.

The scheduler is instrumented as shown in Figure 16. As with *X-rcu*, this is a local increment without locking, atomic instructions, or cacheline bouncing, but, due to the lack of a per-CPU data area, array-indexing instructions are required.

The code that performs periodic RCU processing is shown in Figure 17. UP kernels invoke it

```
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg)
{
    int cpu = cpu_number_map(
                smp_processor_id());
    unsigned long flags;

    head->func = func;
    head->arg = arg;
    local_irq_save(flags);
    list_add_tail(&head->list,
                  &RCU_nxtlist(cpu));
    local_irq_restore(flags);
    tasklet_schedule(&RCU_tasklet(cpu));
}
```

Figure 15: *rcu* call_rcu() Implementation

```
@@ -685,6 +686,7 @@
 switch_tasks:
     prefetch(next);
     prev->work.need_resched = 0;
+    RCU_qsctr(prev->cpu)++;

     if (likely(prev != next)) {
             rq->nr_switches++;
```

Figure 16: *rcu* Scheduler Instrumentation

```
static void
rcu_percpu_tick_common(void)
{
    rcu_process_callbacks(0);
}
```

Figure 17: *rcu* Timer Processing

directly from the timeout handler, while SMP kernels invoke it from *krcud* daemons that are awakened by the timeout handler.

Details of *rcu*'s callback processing are discussed in Appendix A.2.

5.3 rcu-poll

The *rcu-poll* algorithm was written by Andrea Arcangeli and Dipankar Sarma. It appears in the "-aa" series of kernels and in recent SuSE releases. Unlike the *X-rcu* and *rcu* algorithms, *rcu-poll* uses a single set of lists to process read-copy-update callbacks, which are processed by a single tasklet. This results in more cacheline bouncing than do the other algorithms, but is considerably shorter and simpler, and, as noted earlier, boasts extremely short average grace-period latencies and low incremental overheads when there are no read-copy update callbacks in flight.

The `call_rcu()` function constructs the callback, enqueues it onto a global `rcu_nxtlist`, then schedules the tasklet, as shown in Figure 18.

The scheduler is instrumented in much the same way as for the previous algorithms, as shown in Figure 19.

Periodic RCU processing is handled by a single tasklet, whose body is shown in Figure 20. This tasklet invokes `rcu_prepare_polling()` to snapshot each CPU's quiescent state counters if polling is not yet in progress and if there are

```
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg)
{
    head->func = func;
    head->arg = arg;

    spin_lock_bh(&rcu_lock);
    list_add(&head->list, &rcu_nxtlist);
    spin_unlock_bh(&rcu_lock);

    tasklet_hi_schedule(&rcu_tasklet);
}
```

Figure 18: *rcu-poll* call_rcu() Implementation

```
@@ -685,6 +686,7 @@
 switch_tasks:
     prefetch(next);
     prev->work.need_resched = 0;
+    RCU_quiescent(prev->cpu)++;

     if (likely(prev != next)) {
         rq->nr_switches++;
```

Figure 19: *rcu-poll* Scheduler Instrumentation

pending callbacks. If polling has already been started, it instead invokes `rcu_polling()` to check to see if the grace period has ended. This ensures all CPUs have passed through their quiescent states via the context switch.

Details of *rcu-poll*'s callback processing are discussed in Appendix A.3.

```
static void rcu_process_callbacks(
                  unsigned long data)
{
    int stop;

    spin_lock(&rcu_lock);
    if (!rcu_polling_in_progress)
        stop = rcu_prepare_polling();
    else
        stop = rcu_polling();
    spin_unlock(&rcu_lock);

    if (!stop)
        tasklet_hi_schedule(&rcu_tasklet);
}
```

Figure 20: *rcu-poll* Tasklet Body

```
void call_rcu(struct rcu_head *head,
        void (*func)(void *arg),
        void *arg)
{
    int cpu = cpu_number_map(
            smp_processor_id());

    head->func = func;
    head->arg = arg;
    local_bh_disable();
    list_add_tail(&head->list,
            &RCU_nxtlist(cpu));
    local_bh_enable();
}
```

Figure 21: *rcu-ltimer* call_rcu() Implementation

5.4 rcu-ltimer

The *rcu-ltimer* implementation is similar to *X-rcu* and *rcu*, but it inserts calls to `RCU_PROCESS_CALLBACKS()` into `do_timer()` and into the architecture-specific `smp_local_timer_interrupt()` functions, instead of using timers or a kernel daemon to check for the ends of grace periods. This allows *rcu-ltimer* to count user-mode execution as a quiescent state, in addition to the idle loop and context switch. The current patch is fully implemented only on the i386 architecture. Dipankar Sarma implemented this variant to obtain the closest analog to the ptx implementation.

The `call_rcu()` function constructs the callback and enqueues it onto a per-CPU `RCU_nxtlist`, as shown in Figure 21.

The scheduler is instrumented in much the same way as for the previous algorithms, as shown in Figure 22.

Periodic RCU processing is handled by per-CPU tasklets, which are invoked as shown in Figure 23. Lines 3–4 note a quiescent state if the CPU was interrupted from user

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3     prefetch(next);
4     prev->work.need_resched = 0;
5 +   RCU_qsctr(prev->cpu)++;
6
7     if (likely(prev != next)) {
8         rq->nr_switches++;
```

Figure 22: *rcu-ltimer* Scheduler Instrumentation

```
 1 #define RCU_PROCESS_CALLBACKS(cpu,regs) \
 2   do { \
 3      if (user_mode(regs) || idle_cpu(cpu)) \
 4          RCU_qsctr(cpu)++; \
 5          if ((RCU_tasklet(cpu).state & \
 6              ((1 << TASKLET_STATE_SCHED) | \
 7                (1 << TASKLET_STATE_RUN))) \
 8                    == 0) \
 9              tasklet_schedule(
10                  &RCU_tasklet(cpu)); \
11      } while(0)
```

Figure 23: *rcu-ltimer* Timer Processing

mode or the idle loop. Lines 5–10 schedule this CPU's tasklet if it is not already either scheduled or running. This tasklet invokes `rcu_process_callbacks()`, which is described in more detail in Appendix A.4.

5.5 rcu-taskq

Dipankar Sarma implemented the *rcu-taskq* algorithm to obtain a minimal efficient implementation. And this implementation does in fact have the smallest patch, using a single task and a global set of callback queues. The task forces each of a set of per-CPU kernel daemons to schedule itself; when each done so, the grace period has expired. This implementation thus directly forces quiescent states, unlike the other implementations, which instead measure naturally occurring quiescent states. Its grace-period latency increases with increasing load on the system, as noted earlier, but is the only implementation with absolutely zero load on the system when there are no read-copy callbacks in flight.

```
 1 void call_rcu(struct rcu_head * head,
 2             void (*func)(void * arg),
 3             void * arg)
 4 {
 5   unsigned long flags;
 6   int start = 0;
 7
 8   head->func = func;
 9   head->arg = arg;
10
11   spin_lock_irqsave(&rcu_lock, flags);
12   if (list_empty(&rcu_wait_list))
13       start = 1;
14   list_add(&head->list, &rcu_wait_list);
15   spin_unlock_irqrestore(&rcu_lock, flags);
16
17   if (start)
18       schedule_task(&rcu_task);
19 }
```

Figure 24: *rcu-taskq* call_rcu() Implementation

Figure 24 shows the `call_rcu()` implementation. Lines 8–9 initialize the callback, lines 11 and 15 handle locking, lines 12–13 record the initial list state, and line 14 adds the callback to the `rcu_wait_list`. Lines 17–18 start the task if lines 12–13 found the list initially empty.

The task started by `call_rcu()` invokes the function `process_pending_rcus()`, shown in Figure 25. Lines 8–10 snapshot `rcu_wait_list` into a local list. Line 13 then invokes `wait_for_rcu()` to wait for a full grace period to elapse. Finally, lines 15–23 invoke the callbacks from the local list.

Figure 26 shows `wait_for_rcu()`. Lines 6–10 awaken the *krcud* daemons for the other CPUs, and lines 11–13 wait for these daemons to respond.

Figure 27 shows the code for the *krcud* daemons. Lines 6–20 initialize the daemon, set its priority high, blocking signals, binding to the corresponding CPU, setting the task name, initializing the task name, and informing the `spawn_krcud()` task that the dae-

```
static void process_pending_rcus(
                   void *arg)
{
    LIST_HEAD(rcu_current_list);
    struct list_head * entry;

    spin_lock_irq(&rcu_lock);
    list_splice(&rcu_wait_list,
                rcu_current_list.prev);
    INIT_LIST_HEAD(&rcu_wait_list);
    spin_unlock_irq(&rcu_lock);

    wait_for_rcu();

    while ((entry = rcu_current_list.prev)
              != &rcu_current_list) {
        struct rcu_head * head;

        list_del(entry);
        head = list_entry(entry,
                  struct rcu_head, list);
        head->func(head->arg);
    }
}
```

Figure 25: *rcu-taskq* process_pending_rcus() Implementation

```
static void wait_for_rcu(void)
{
 int cpu;
 int count;

 for (cpu = 0; cpu < smp_num_cpus; cpu++) {
     if (cpu == smp_processor_id())
         continue;
     up(&krcud_sema(cpu));
 }
 count = 0;
 while (count++ < smp_num_cpus - 1)
     down(&rcu_sema);
}
```

Figure 26: *rcu-taskq* wait_for_rcu() Implementation

```
 1 static int krcud(void * __bind_cpu)
 2 {
 3   int bind_cpu = *(int *) __bind_cpu;
 4   int cpu = cpu_logical_map(bind_cpu);
 5
 6   daemonize();
 7   current->policy = SCHED_FIFO;
 8   current->rt_priority = 1001 +
 9     sys_sched_get_priority_max(SCHED_FIFO);
10
11   sigfillset(&current->blocked);
12
13   /* Migrate to the right CPU */
14   set_cpus_allowed(current, 1UL << cpu);
15
16   sprintf(current->comm,
17           "krcud_CPU%d", bind_cpu);
18   sema_init(&krcud_sema(cpu), 0);
19
20   krcud_task(cpu) = current;
21
22   for (;;) {
23       while (down_interruptible(
24                &krcud_sema(cpu)));
25       up(&rcu_sema);
26   }
27 }
```

Figure 27: *rcu-taskq* krcud() Implementation

mon is ready to process requests. Lines 22–26 process each request, alternately sleeping on the `krcud_sema` and waking up the `process_pending_rcus()` task.

5.6 rcu-sched

The *rcu-sched* implementation was developed by Rusty Russell [Russell01d], with a goal of minimizing `call_rcu()` overhead. It uses a ring of per-CPU counters, and each CPU sets its counter to one greater than that of its neighbor on each pass through the scheduler when read-copy-update callbacks are pending. Thus, when a given CPU sees its neighbor's counter change, it is guaranteed that each CPU has passed through the scheduler (a quiescent state) since the given CPU last incremented its own counter.

This implementation also maintains not just per-CPU callback queues, but two sets of per-CPU-per-IRQ callback queues. This allows the queues to be accesses without the need for either locks (per-CPU) or for interrupt masking (per-IRQ). One set of these queues accumulates new callbacks from `call_rcu()`, while the other set holds callbacks waiting for the end of the current grace period.

Finally, this implementation places checks in the idle loop in order to ensure that idle CPUs do not indefinitely delay the end of the grace period. This has the beneficial side effect of causing idle-loop execution to be a quiescent state without using the active entities (tasklets, timers, kernel daemons) used by the other implementations.

Figure 28 shows the `call_rcu()` function. Lines 9–10 initialize the `rcu_head` callback. Lines 11–14 determine the interrupt state, which is used later as an index to the array of lists of callbacks. Lines 17–18 find the right queue for the callback. The `rcu_batch[cpu].queueing` is a bit that contains the index of the half of the array that is accumulating new callbacks. The sense of this bit is reversed in `rcu_batch_done()` at the end of each grace period. Line 20 increments the number of pending callbacks, which signals the scheduler to start looking for a grace period, and lines 23–24 enqueues the callback.

Figure 29 shows the first patch to the scheduler. Lines 12–13 check to see if there are read-copy-update callbacks pending, and, if so, branch to the `rcu_process` label in the second patch shown in Figure 30

Lines 8–10 of Figure 30 set local variable `c` to one greater than the previous CPU's ring counter. If `c` is different than this CPU's ring count, a grace period has ended, and is handled by lines 16–23. Line 11 checks for scheduler reentry, and if this has not occurred, lines 19–23 invoke `rcu_batch_done()`, protecting against scheduler re-entry by manipulating this CPU's `finished_count`. Line 25 updates

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *data),
 3               void *data)
 4 {
 5     unsigned cpu = smp_processor_id();
 6     unsigned state;
 7     struct rcu_head **headp;
 8
 9     head->func = func;
10     head->data = data;
11     if (in_interrupt()) {
12         if (in_irq()) state = 2;
13         else state = 1;
14     } else state = 0;
15
16     /* Figure out which queue we're on. */
17     headp = &rcu_batch[cpu].head[
18            rcu_batch[cpu].queueing][state];
19
20     atomic_inc(&rcu_pending);
21     /* Prepend to this CPU's list:
22        no locks needed. */
23     head->next = *headp;
24     *headp = head;
25 }
```

Figure 28: *rcu-sched* call_rcu() Implementation

```
 1 @@ -634,10 +639,16 @@
 2      prio_array_t *array;
 3      list_t *queue;
 4      int idx;
 5 +    int c, this_cpu;
 6
 7      if (unlikely(in_interrupt()))
 8          BUG();
 9      release_kernel_lock(prev,
10              smp_processor_id());
11 +
12 +    if (unlikely(is_rcu_pending()))
13 +        goto rcu_process;
14 +
15 +rcu_process_back:
16      spin_lock_irq(&rq->lock);
17
18          switch (prev->state) {
```

Figure 29: *rcu-sched* Scheduler Instrumentation, Part 1

```
 1 @@ -700,6 +711,23 @@
 2      }
 3      spin_unlock_irq(&rq->lock);
 4
 5 +rcu_process:
 6 +    /* Avoid cache line effects
 7 +       if value hasn't changed */
 8 +    this_cpu = smp_processor_id();
 9 +    c = ring_count((this_cpu + 1) %
10 +           smp_num_cpus) + 1;
11 +    if (c != ring_count(this_cpu)) {
12 +        /* Do subtraction to
13 +           avoid int wrap corner case */
14 +        if (c - finished_count(this_cpu)
15 +                                 >= 0) {
16 +            /* Avoid reentry: temporarily
17 +               set finish_count
18 +               far in the future */
19 +            finished_count(this_cpu) =
20 +                           c + INT_MAX;
21 +            rcu_batch_done();
22 +            finished_count(this_cpu) =
23 +                        c + smp_num_cpus;
24 +        }
25 +        ring_count(this_cpu) = c;
26 +    }
27 +    goto rcu_process_back;
28 +
29      reacquire_kernel_lock(current);
30      return;
31  }
```

Figure 30: *rcu-sched* Scheduler Instrumentation, Part 2

this CPU's ring count, which will result in the next CPU seeing the end of a grace period. Line 27 returns control to the mainline scheduler.

Figure 31 shows how the idle loop is instrumented to prevent architectures that shut down CPUs on idle from indefinitely extending the grace period. The other implementations get this effect through use of timers or forced context switches.

Figure 32 shows `rcu_batch_done()`, which is invoked from the scheduler at the end of a grace period. Line 7–8 pick up a pointer to this CPU's set of read-copy-update callback queues. Lines 11–22 invoke all the callbacks in each of this CPU's callback queues (one for each possible IRQ level) that was waiting for the current grace period to expire (selected by `!mybatch->queueing`), and empty each list. Line 25 swaps the sets of queues, so that the callbacks previously waiting for a new grace period to begin are now waiting for the now-current grace period, and the newly emptied queues will now accept new callbacks registered by future calls to `call_rcu()`.

```
@@ -84,7 +85,8 @@
  get into the scheduler unnecessarily. */
  long oldval = xchg(
        &current->work.need_resched, -1UL);
  if (!oldval)
-   while (current->work.need_resched < 0);
+   while (current->work.need_resched < 0
+          && !is_rcu_pending());
  schedule();
  check_pgt_cache();
}
```

Figure 31: *rcu-sched* Idle Loop Instrumentation

```
 1 void rcu_batch_done(void)
 2 {
 3     struct rcu_head *i, *next;
 4     struct rcu_batch *mybatch;
 5     unsigned int q;
 6
 7     mybatch =
 8         &rcu_batch[smp_processor_id()];
 9     /* Call callbacks: probably delete
10        themselves, may schedule. */
11     for (q = 0; q < 3; q++) {
12         for (i = mybatch->head[
13                   !mybatch->queueing][q];
14              i;
15              i = next) {
16             next = i->next;
17             i->func(i->data);
18             atomic_dec(&rcu_pending);
19         }
20         mybatch->head[
21            !mybatch->queueing][q] = NULL;
22     }
23
24     /* Start queueing on this batch. */
25     mybatch->queueing = !mybatch->queueing;
26 }
```

Figure 32: *rcu-sched* rcu_batch_done()

5.7 Preemptible Algorithm

With the addition of preemption to the Linux kernel, read-copy update must also handle preemption. Rusty Russell [Russell01b] produced such a patch, but it requires scanning all tasks on the runqueue, a job made more complex by the addition of the multi-queue scheduler.

Dipankar Sarma created a prototype preemptible algorithm that is similar to *rcu*,[4] but adds per-CPU counts of preempted tasks, which operate in a manner in some ways similar to the generation mechanism in K42 [Gamsa99]. The key concept is that a preemptible kernel must track tasks rather than CPUs. However, to avoid potentially expensive scans of the task list or the runqueues, the tasks are tracked on a per-CPU basis. When a task returns from a voluntary context switch (or is created), it is implicitly associated with the CPU that it starts running on. No matter how many times the task is preempted, from a read-copy-update perspective, it remains affiliated with that CPU, even if it is migrated to other CPUs. Once it performs a voluntary context switch, it gives up its affiliation.

However, no additional work is done (over that done by a non-preemptible kernel running a non-preemptible implementation of read-copy update) until that task is preempted. The task then increments a per-CPU counter, which remains incremented until the task executes a voluntary context switch, possibly by exiting. The task then decrements that same per-CPU counter, even if the task is running on some other CPU at the time.

Of course, if there is a lot of preemption, it might be that a particular CPU *always* has at least one preempted task affiliated with it. However, the end of a grace period is marked not by the absence of tasks, but by each of the tasks that was either running or preempted at the start of the grace period having either exited or voluntarily switched context.

This distinction is maintained by providing each CPU with a pair of counters, a "next" counter that is incremented by tasks returning from their voluntary context switch onto the corresponding CPU, and a "current" counter that is only decremented. Note that the "next" counter will be also decremented whenever a task resumes execution quickly enough after being preempted. The end of the grace period occurs when all CPUs' "current" counters reach zero.[5] The roles of the counters in each pair are now reversed in order to start the next grace period, just after the base *rcu* portion of the algorithm moves the callbacks in the `rcu_nextlist` to `rcu_currlist`.

Each CPU's pair of counters is as shown in Figure 33, along with the pair of pointers that handle the reversing of their roles. The `next_preempt_cntr` pointer points to the element of `rcu_preempt_cntr[]` that is atomically incremented (by a new `rcu_preempt_get()` function) when task affiliated with this CPU is preempted for the first time since its preceding voluntary context switch. The task records this pointer in a new `cpu_preempt_cntr` pointer in its task structure, which is initially NULL. After the task resumes and voluntarily relinquishes the CPU[6], it atomically decrements the counter pointed to by its `cpu_preempt_cntr`, using a new `rcu_preempt_put()` function, then NULLs its `cpu_preempt_cntr` pointer.

[4]However, as noted earlier, this preemptible version of *rcu* has greatly reduced CPU overhead when there are no read-copy callbacks in the system.

[5]Unless one of the CPUs has been running a task continuously since before the start of the grace period, but this case is handled by the base *rcu* portion of the implementation.

[6]Possibly after having been preempted several more times along the way. This is why the counter cannot be decremented immediately when the task is resumed, but must instead wait for the task to voluntarily relinquish the CPU.

```
extern atomic_t
   rcu_preempt_cntr[2] __per_cpu_data;
extern atomic_t
   *curr_preempt_cntr __per_cpu_data;
extern atomic_t
   *next_preempt_cntr __per_cpu_data;
```

Figure 33: rcu_preempt Per-CPU Counters

The `curr_preempt_cntr` pointer points to the element of `rcu_preempt_cntr[]` that `next_preempt_cntr` does not point to. This element of the array contains the number of tasks affiliated with this CPU that were first preempted before the beginning of the current grace period, and that must resume and voluntarily relinquish a CPU before the current grace period can expire. When this CPU becomes aware of the end of the current grace period, it exchanges the values of `next_preempt_cntr` and `curr_preempt_cntr`, so that the elements of the `rcu_preempt_cntr[]` array exchange roles.

The rest of the callback processing is very similar to that of the *rcu* algorithm. The major difference is that `rcu_check_quiescent_state()` must check that all tasks preempted on this CPU prior to the current grace period have voluntarily relinquished the CPU.

6 Conclusions and Future Plans

Andrea Arcangeli's *rcu-poll* implementation exhibits the best `call_rcu()` latency, and is therefore a good implementation for workloads that do not have high aggregate `call_rcu()` invocation rates. The longer (but well-bounded) `call_rcu()` latencies of the *X-rcu*, *rcu-ltimer*, and *rcu* implementations may make them preferable for systems with higher `call_rcu()` invocation rates.

We are continuing our work on preemptible read-copy-update implementations, in order to obtain the best implementation compatible with the 2.5 kernel. Finally, we are continuing our measurements with various workloads, which we expect will evolve as the 2.5 kernel evolves. In particular, we will measure performance under heavy *call_rcu()* load.

7 Acknowledgments

We owe thanks to Martin Bligh and Hanna Linder for their able assistance with the machines we used to gather the data shown in this paper, and to Hans Tannenberger and Gerrit Huizenga for arranging access to these machines. We are especially grateful to Maneesh Soni and Hanna Linder for their efforts with read-copy update, and to Jonathan Appavoo for many enlightening discussions. We are indebted to Dan Frye, Randy Kalmeta, Vijay Sukthankar, Hugh Blemings, and Manish Gupta for their support of this effort.

References

[Blanchard02a] A. Blanchard *some RCU dcache and ratcache results*, Linux-Kernel Mailing List, March 2002. `http://marc.theaimsgroup.com/?l=linux-kernel&m=101637107412972&w=2`.

[Compaq01] Compaq Computer Corporation *Shared Memory, Threads, Interprocess Communication*, Ask The Wizard, August 2001. `http://www.openvms.compaq.com/wizard/wiz_2637.html`.

[Gamsa99] B. Gamsa, O. Kreiger, J. Appavoo, and M. Stumm. *Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system*, Proceedings of the 3rd

Symposium on Operating System Design and Implementation, New Orleans, LA, February, 1999.

[Linder02a] H. Linder, D. Sarma, and Maneesh Soni. *Scalability of the Directory Entry Cache*, To appear in Ottawa Linux Symposium, June 2002.

[Kleen02a] A. Kleen *Reduce Module Races*, kernel.org, January 2002. `ftp://ftp.us.kernel.org/pub/linux/kernel/people/andrea/kernels/v2.4/v2.4.19pre7aa2/00_reduce-module-races-1`.

[LSE] D. Sarma et al. *Linux Scaling Effort (LSE)*, SourceForge Project, April 2002. `http://prdownloads.sourceforge.net/lse/`.

[McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998. (revised version available at `http://www.rdrop.com/users/paulmck/rclockpdcsproof.pdf`).

[McK01a] P. E. McKenney and D. Sarma. *Read-Copy Mutual Exclusion in Linux*, `http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html`, February 2001.

[McK01b] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni. *Read-Copy Update*, Ottawa Linux Symposium, July 2001. (revised version available at `http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf`).

[McK01c] P. E. McKenney, et al. *RFC: patch to allow lock-free traversal of lists with insertion*, LKML, October 2001. `http://www.ussg.iu.edu/hypermail/linux/kernel/0110.1/0239.html`.

[McK01d] P. E. McKenney, et al. *Data Dependencies and wmb()*, LSE, October 2001. `http://lse.sourceforge.net/locking/wmbdd.html`.

[Russell01b] R. Russell. *Re: [PATCH for 2.5] preemptible kernel*, `http://www.uwsg.indiana.edu/hypermail/linux/kernel/0103.3/1070.html`, March 2001.

[Russell01d] R. Russell *Re: 2.4.10pre7aa1*, Linux-Kernel Mailing List, September 2001. `http://www.ussg.iu.edu/hypermail/linux/kernel/0109.2/0021.html`.

[Russell02a] R. Russell *Re: [PATCH] per-cpu areas for 2.5.3-pre6*, Linux-Kernel Mailing List, February 2002. `http://marc.theaimsgroup.com/?l=linux-kernel&m=101255391528359&w=2`.

[Sarma02a] D. Sarma *[RFC][PATCH] Ingo's smptimers patch experiment*, Linux-Kernel Mailing List, February 2002. `http://marc.theaimsgroup.com/?l=linux-kernel&m=101301053225522&w=2`.

[Sarma02b] D. Sarma *[PATCH] memory barriers*, Linux-Kernel Mailing List, March 2002. `http://www.ussg.iu.edu/hypermail/linux/kernel/0203.2/1604.html`.

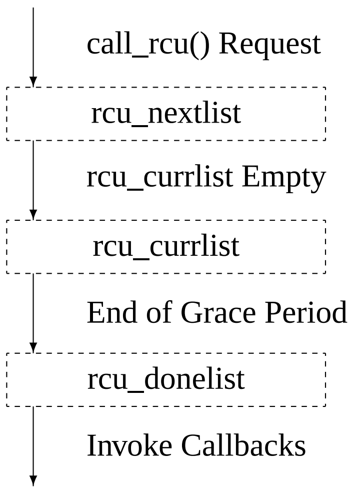


Figure 34: RCU Callback Flow

```
 1 static void rcu_process_callbacks(void)
 2 {
 3     if (!list_empty(
 4             &this_cpu(rcu_currlist)) &&
 5         RCU_BATCH_GT(rcu_currbatch,
 6                    this_cpu(rcu_batch))) {
 7         list_splice(
 8             &this_cpu(rcu_currlist),
 9             &this_cpu(rcu_donelist));
10         INIT_LIST_HEAD(
11             &this_cpu(rcu_currlist));
12     }
13
14     rcu_move_next_batch();
15
16     rcu_check_quiescent_state();
17
18     if (!list_empty(
19             &this_cpu(rcu_donelist))) {
20         rcu_invoke_callbacks(
21             &this_cpu(rcu_donelist));
22     }
23 }
```

Figure 35: *X-rcu* rcu_process_callbacks()

8 Trademarks

Linux is a trademark of Linus Torvalds.
Pentium and Xeon are trademarks of Intel Corporation.
IBM and ptx are trademarks of International Business Machines Corporation.

Appendix

A Implementation Details

These appendices contain more implementation details of the various algorithms.

A.1 *X-rcu* Callback Processing

This section describes the *X-rcu* callback processing. The processing proceeds as shown in Figure 34.

The `rcu_process_callbacks()` function shown in Figure 35 handles the overall flow. Lines 3–12 move callbacks from `rcu_currlist` to `rcu_donelist` after the end of a grace period. Line 14 invokes `rcu_move_next_batch()` (shown in Figure 36), which moves callbacks from `rcu_nextlist` to `rcu_currlist`, initiating grace-period detection if needed. Line 16 calls `rcu_check_quiescent_state()`, which checks to see if the current CPU has passed through a quiescent state since the beginning of the current grace period. Lines 18–22 call `rcu_invoke_callbacks()` to invoke any callbacks in `rcu_donelist`.

The `rcu_move_next_batch()` function shown in Figure 36 disables local interrupts (line 3), and then checks to see if `rcu_currlist` is empty and `rcu_nextlist` is not (lines 4–7). If so, it moves the contents of `rcu_nextlist` to `rcu_currlist` (lines 8 and 9), then re-enables interrupts (line 12). It then obtains a new RCU batch number (lines 18–19) and registers it using `rcu_reg_batch()` (line 20, see Figure 39 for this function's definition) under the `rcu_lock`.

If lines 4–5 find `rcu_currlist` to be nonempty, `rcu_move_next_batch()` simply re-enables interrupts and returns (line

```
 1 static void rcu_move_next_batch(void)
 2 {
 3   local_irq_disable();
 4   if (!list_empty(
 5           &this_cpu(rcu_nextlist)) &&
 6       list_empty(
 7           &this_cpu(rcu_currlist))) {
 8       list_splice(&this_cpu(rcu_nextlist),
 9                 &this_cpu(rcu_currlist));
10       INIT_LIST_HEAD(
11                 &this_cpu(rcu_nextlist));
12       local_irq_enable();
13
14       /*
15        * start the next batch of callbacks
16        */
17       spin_lock(&rcu_lock);
18       this_cpu(rcu_batch) =
19                  rcu_currbatch + 1;
20       rcu_reg_batch(this_cpu(rcu_batch));
21       spin_unlock(&rcu_lock);
22   } else {
23       local_irq_enable();
24   }
25 }
```

Figure 36: *X-rcu* rcu_move_next_batch()

23).

The `rcu_check_quiescent_state()` function shown in Figure 37 checks to see if the current CPU has gone through a quiescent state, and, if so, publicizes it.

Lines 6–8 check to see if this CPU has already passed through a quiescent state during the current grace period, and, if so, line 6 simply returns. Lines 17–22 check to see if this is the first that this CPU has heard of the current grace period, and, if so, lines 19–20 take a snapshot of this CPU's context-switch counter in `rcu_last_qsctr` and returns. Lines 23–26 check to see if this CPU has passed through a quiescent state since the snapshot, and, if not, line 25 simply returns.

Execution reaches line 29 when this CPU first determines that it has passed through a quiescent state in the current grace period. Lines 28–44 publish this fact under the global `rcu_lock`, which possibly marks the end of the current grace period. Line 33 clears this CPU's bit in `rcu_cpumask`, which publicizes the fact that this CPU has passed through a quiescent state during the current grace period. Lines 34–35 set `rcu_last_qsctr` to an invalid quantity, which will indicate that this CPU is not yet aware of the next grace period. If there are other CPUs that have not yet passed through their quiescent states, then lines 36–41 release the `rcu_lock` and return. Execution reaches line 42 if this CPU is the last one to detect that it has passed through a quiescent state during the current grace period, which marks the end of the grace period. Line 42 increments `rcu_currbatch`, which signals the end of the grace period. Line 43 invokes `rcu_reg_batch()` to initiate a new grace period if needed, and line 36 releases the `rcu_lock`.

Figure 38 shows `rcu_invoke_callbacks()`, which simply loops through the list of callbacks, invoking each in turn.

Figure 39 shows `rcu_reg_batch()`, which publicizes the beginning of a new grace period, if needed. Lines 4–7 check to see if the batch number of the requested grace period is larger than that of the largest-numbered grace period that has been requested thus far (the `RCU_BATCH_LT()` macro handles wraparound). If so, line 6 publicizes the new maximum batch number. If the largest-numbered grace period requested thus far has already completed or if a grace period is currently in progress, lines 8–12 simply return. Otherwise, line 13 sets `rcu_cpumask` to indicate that all CPUs need to pass through a quiescent state, which publicizes the start of a new grace period.

A.2 *rcu* Callback Processing

The *rcu* algorithm's callback processing is very similar to that of the *X-rcu* algorithm, shown in

```
static void rcu_check_quiescent_state(void)
{
    int cpu = cpu_number_map(
                 smp_processor_id());

    if (!test_bit(cpu, &rcu_cpumask)) {
        return;
    }

    /*
     * May race with rcu per-cpu tick -
     * in the worst case
     * we may miss one quiescent state
     * of that CPU. That is tolerable.
     * So no need to disable interrupts.
     */
    if (this_cpu(rcu_last_qsctr) ==
                    RCU_QSCTR_INVALID) {
        this_cpu(rcu_last_qsctr) =
                    this_cpu(rcu_qsctr);
        return;
    }
    if (this_cpu(rcu_qsctr) ==
            this_cpu(rcu_last_qsctr)) {
        return;
    }

    spin_lock(&rcu_lock);
    if (!test_bit(cpu, &rcu_cpumask)) {
        spin_unlock(&rcu_lock);
        return;
    }
    clear_bit(cpu, &rcu_cpumask);
    this_cpu(rcu_last_qsctr) =
                    RCU_QSCTR_INVALID;
    if (rcu_cpumask != 0) {
        /* All CPUs haven't gone
           through a quiescent state */
        spin_unlock(&rcu_lock);
        return;
    }
    rcu_currbatch++;
    rcu_reg_batch(rcu_maxbatch);
    spin_unlock(&rcu_lock);
}
```

Figure 37: *X-rcu* rcu_check_quiescent_state()

```
static inline void rcu_invoke_callbacks(
                  struct list_head *list)
{
    struct list_head *entry;
    struct rcu_head *head;

    while (!list_empty(list)) {
        entry = list->next;
        list_del(entry);
        head = list_entry(entry,
                  struct rcu_head, list);
        head->func(head->arg);
    }
}
```

Figure 38: *X-rcu* rcu_invoke_callbacks()

```
static inline void rcu_reg_batch(
            rcu_batch_t newbatch)
{
  if (RCU_BATCH_LT(rcu_maxbatch,
                     newbatch)) {
      rcu_maxbatch = newbatch;
  }
  if (RCU_BATCH_LT(rcu_maxbatch,
          rcu_currbatch) ||
      (rcu_cpumask != 0)) {
      return;
  }
  rcu_cpumask = cpu_online_map;
}
```

Figure 39: *X-rcu* rcu_reg_batch()

Appendix A.1. Differences include:

1. *rcu* must explicitly index into arrays containing per-CPU elements, while *X-rcu* directly accesses the per-CPU data area.

2. *rcu*'s `rcu_process_callbacks()` contains code that clears the current CPU's bit from `rcu_active_cpumask`.

3. *rcu*'s `rcu_move_next_batch()` contains code that sets the current CPU's bit in `rcu_active_cpumask` and schedules the timer if there are RCU callbacks active and the timer is not already scheduled.

A.3 *rcu-poll* Callback Processing

Figure 40 shows the `rcu_prepare_polling()` function. This function relies on `rcu_process_callbacks()` (see Figure 20) acquiring the `rcu_lock`. Lines 12–27 check to see if there are callbacks waiting in `rcu_nxtlist`, and, if so, starts a grace period. Lines 13–14 move the list from `rcu_nxtlist` to `rcu_curlist`. Line 16 records the fact that a grace period is now in progress. Lines 18–25 mark each CPU

```
 1 static int rcu_prepare_polling(void)
 2 {
 3   int stop;
 4   int i;
 5
 6 #ifdef DEBUG
 7   if (!list_empty(&rcu_curlist))
 8       BUG();
 9 #endif
10
11   stop = 1;
12   if (!list_empty(&rcu_nxtlist)) {
13    list_splice(&rcu_nxtlist, &rcu_curlist);
14    INIT_LIST_HEAD(&rcu_nxtlist);
15
16    rcu_polling_in_progress = 1;
17
18    for (i = 0; i < smp_num_cpus; i++) {
19        int cpu = cpu_logical_map(i);
20
21        rcu_qsmask |= 1UL << cpu;
22        rcu_quiescent_checkpoint[cpu] =
23                     RCU_quiescent(cpu);
24        force_cpu_reschedule(cpu);
25    }
26    stop = 0;
27   }
28
29   return stop;
30 }
```

Figure 40: *rcu-poll* rcu_prepare_polling()

(other than the current one) as needing to go through a quiescent state, take a snapshot of each CPU's context-switch counter, and expedite a context switch. Line 26 indicates that grace-period polling needs to continue – if `rcu_nxtlist` had been empty, polling would cease until the next `call_rcu()` invocation.

Figure 41 shows the `rcu_polling()` function. Lines 6–13 check each CPU that has not yet been observed passing through a quiescent state (as indicated by the `rcu_qsmask` check at line 9) to see if that CPU's `RCU_quiescent` counter has advanced since the `rcu_prepare_polling()` started the current grace period. If it has, then that CPU has recently passed through a quiescent state, so line 12 clears its bit from `rcu_qsmask`. Line 16 then checks to see if all CPUs have now passed through

```
 1 static int rcu_polling(void)
 2 {
 3   int i;
 4   int stop;
 5
 6   for (i = 0; i < smp_num_cpus; i++) {
 7       int cpu = cpu_logical_map(i);
 8
 9       if (rcu_qsmask & (1UL << cpu))
10           if (rcu_quiescent_checkpoint[cpu]
11                 != RCU_quiescent(cpu))
12               rcu_qsmask &= ~(1UL << cpu);
13   }
14
15   stop = 0;
16   if (!rcu_qsmask)
17       stop = rcu_completion();
18
19   return stop;
20 }
```

Figure 41: *rcu-poll* rcu_polling()

```
 1 static int rcu_completion(void)
 2 {
 3     int stop;
 4
 5     rcu_polling_in_progress = 0;
 6     rcu_invoke_callbacks();
 7
 8     stop = rcu_prepare_polling();
 9
10     return stop;
11 }
```

Figure 42: *rcu-poll* rcu_completion()

their quiescent states. If so, line 17 invokes `rcu_completion()` to mark the end of the grace period. If another grace period is required, `rcu_completion` will have started it, and will then return zero to signal that grace-period polling should continue.

Figure 42 shows the `rcu_completion()` function that is invoked at the end of a grace period. Line 5 records the fact that a grace period is no longer in progress, line 6 invokes `rcu_invoke_callbacks()` to invoke the callbacks, and line 8 starts a new grace period, if required.

Figure 43 shows the

```
static void rcu_invoke_callbacks(void)
{
    struct list_head *entry;
    struct rcu_head *head;

#ifdef DEBUG
    if (list_empty(&rcu_curlist))
        BUG();
#endif

    entry = rcu_curlist.prev;
    do {
        head = list_entry(entry,
                 struct rcu_head, list);
        entry = entry->prev;

        head->func(head->arg);
    } while (entry != &rcu_curlist);

    INIT_LIST_HEAD(&rcu_curlist);
}
```

Figure 43: *rcu-poll* rcu_invoke_callbacks()

`rcu_invoke_callbacks()` function. This is similar to that shown for *X-rcu* in Figure 38, but processes a single global list rather than a per-CPU list, and removes elements from the list in a slightly different manner.

A.4 *rcu-ltimer* Callback Processing

This implementation is closest to that in ptx, and is thus driven from timer handlers, as noted in Section 5.4. The `rcu_process_callbacks()` function, shown in Figure 44 is invoked on every timer tick to process the per-CPU callback lists. This function invokes `rcu_check_callbacks()` if any of the following are true:

1. There are callbacks in `RCU_curlist` and the corresponding grace period has expired (lines 7–9).
2. There are no callbacks in `RCU_curlist`, but there are some in `RCU_nxtlist` waiting to start a grace period (lines 10–11).
3. This CPU has not yet passed through a quiescent state for the current grace period (line 12–13).

```
 1 static void rcu_process_callbacks(
 2               unsigned long data)
 3 {
 4   int cpu = cpu_number_map(
 5               smp_processor_id());
 6
 7   if ((!list_empty(&RCU_curlist(cpu)) &&
 8       RCU_BATCH_LT(RCU_batch(cpu),
 9             rcu_ctrlblk.curbatch)) ||
10      (list_empty(&RCU_curlist(cpu)) &&
11          !list_empty(&RCU_nxtlist(cpu))) ||
12      test_bit(cpu,
13             &rcu_ctrlblk.rcu_cpu_mask))
14          rcu_check_callbacks();
15 }
```

Figure 44: *rcu-ltimer* rcu_process_callbacks()

Figure 45 shows `rcu_check_callbacks()` advances callbacks for the current CPU through the lists. Lines 7–13 check to see if the grace period corresponding to callbacks in this CPU's `RCU_curlist` has expired, and, if so, moves the contents of this list to the local variable `list`. Lines 15–29 check to see if this CPU's `RCU_curlist` is empty and if there are callbacks in this CPU's `RCU_nxtlist` waiting to start a grace period, and, if so, moves them from `RCU_nxtlist` to `RCU_curlist` on lines 17–19 and requests a new grace period in lines 24–28. Line 30 then checks to see if this CPU has passed through a quiescent state. Lines 31–32 invoke any callbacks on local variable `list`.

Figure 46 shows `rcu_reg_batch()`, which schedules a new grace period if required. Lines 4–7 check to see if the new batch number is larger than the largest seen thus far, and, if so, records the new maximum batch number on line 6. Lines 8–10 check to see if the grace period corresponding to the largest batch number has already expired (lines 8–9), or if a grace period is already in progress (line 10), and, in either case, simply returns. Otherwise, lines 13–14 record the fact that all CPUs need to go through a quiescent state for the new grace period. As before, the `RCU_BATCH_LT()` macros check for batch-number wraparound.

```
static void rcu_check_callbacks(void)
{
    int cpu = cpu_number_map(
                 smp_processor_id());
    LIST_HEAD(list);

    if (!list_empty(&RCU_curlist(cpu)) &&
        RCU_BATCH_GT(rcu_ctrlblk.curbatch,
                     RCU_batch(cpu))) {
        list_splice(&RCU_curlist(cpu),
                    &list);
        INIT_LIST_HEAD(&RCU_curlist(cpu));
    }

    if (!list_empty(&RCU_nxtlist(cpu)) &&
        list_empty(&RCU_curlist(cpu))) {
        list_splice(&RCU_nxtlist(cpu),
                    &RCU_curlist(cpu));
        INIT_LIST_HEAD(&RCU_nxtlist(cpu));

       /*
        * start the next batch of callbacks
        */
        spin_lock(&rcu_ctrlblk.mutex);
        RCU_batch(cpu) =
            rcu_ctrlblk.curbatch + 1;
        rcu_reg_batch(RCU_batch(cpu));
        spin_unlock(&rcu_ctrlblk.mutex);
    }
    rcu_check_quiescent_state();
    if (!list_empty(&list))
        rcu_invoke_callbacks(&list);
}
```

Figure 45: *rcu-ltimer* rcu_check_callbacks()

```
static void rcu_reg_batch(
             rcu_batch_t newbatch)
{
    if (RCU_BATCH_LT(rcu_ctrlblk.maxbatch,
                              newbatch)) {
        rcu_ctrlblk.maxbatch = newbatch;
    }
    if (RCU_BATCH_LT(rcu_ctrlblk.maxbatch,
                    rcu_ctrlblk.curbatch) ||
        (rcu_ctrlblk.rcu_cpu_mask != 0)) {
        return;
    }
    rcu_ctrlblk.rcu_cpu_mask =
                       cpu_online_map;
}
```

Figure 46: *rcu-ltimer* rcu_reg_batch()

Figure 47 shows how `rcu_check_quiescent_state()` checks that the current CPU has passed through a quiescent state since the beginning of the current grace period. Lines 6–9 check to see if this CPU has already passed through a quiescent state, and, if so, simply returns. Lines 19–20 checks to see if this CPU is unaware of the current grace period, and, if so, snapshots the current quiescent-state counter on lines 21–22 and then returns. Lines 25–28 checks to see if this CPU has passed through a quiescent state since it became aware of the current grace period, and, if not, simply returns. Execution reaches line 30 the first time that this CPU realizes that it has passed through a quiescent state since it became aware of the current grace period. Lines 36 and 37 publish the fact that this CPU has passed through a quiescent state. Lines 38–41 check to see if this is the last CPU to pass through a quiescent state, thus ending the grace period, and returning if not. Line 42 publicizes the end of the grace period, and line 43 invokes `rcu_reg_batch()` to start a new grace period, if one is needed.

B Memory Ordering Issues

This paper has heretofore focused on lock-free search on lists subject to concurrent deletion. Insertion poses additional problems on systems with very weak memory ordering, as noted in recent discussions on LKML [McK01c]. This appendix focuses on these problems and some solutions.

Some of these problems may be addressed by using the `wmb()` primitive as shown on line 9 of Figure 48. This `wmb()` guarantees that the element initialization in lines 6–8 is not executed before the element is added to the list on line 10. On many (*but not all*) CPUs, this is sufficient, and the lock-free search on lines 14–26 will then operate correctly.

```
static void rcu_check_quiescent_state(void)
{
 int cpu = cpu_number_map(
                smp_processor_id());

 if (!test_bit(cpu,
         &rcu_ctrlblk.rcu_cpu_mask)) {
     return;
 }

 /*
  * Races with local timer interrupt -
  * in the worst case
  * we may miss one quiescent state
  * of that CPU. That is
  * tolerable. So no need
  * to disable interrupts.
  */
 if (RCU_last_qsctr(cpu) ==
             RCU_QSCTR_INVALID) {
     RCU_last_qsctr(cpu) =
                  RCU_qsctr(cpu);
     return;
 }
 if (RCU_qsctr(cpu) ==
          RCU_last_qsctr(cpu)) {
     return;
 }

 spin_lock(&rcu_ctrlblk.mutex);
 if (!test_bit(cpu,
     &rcu_ctrlblk.rcu_cpu_mask)) {
     spin_unlock(&rcu_ctrlblk.mutex);
     return;
 }
 clear_bit(cpu, &rcu_ctrlblk.rcu_cpu_mask);
 RCU_last_qsctr(cpu) = RCU_QSCTR_INVALID;
 if (rcu_ctrlblk.rcu_cpu_mask != 0) {
     spin_unlock(&rcu_ctrlblk.mutex);
     return;
 }
 rcu_ctrlblk.curbatch++;
 rcu_reg_batch(rcu_ctrlblk.maxbatch);
 spin_unlock(&rcu_ctrlblk.mutex);
}
```

Figure 47: *rcu-ltimer* rcu_check_quiescent_state()

```
 1 struct el *insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     spin_lock(&mutex);
 6     p->next = head.next;
 7     p->key = key;
 8     p->data = data;
 9     wmb();
10     head.next = p;
11     spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         /* BUG ON ALPHA!!! */
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

Figure 48: Insert and Lock-Free Search

However, some CPUs, such as Alpha, have extremely weak memory ordering such that the code on line 20 of Figure 48 could see the old garbage values that were present before the initialization on lines 6–8.

Figure 49 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating caches lines are processed by the different partitions of the caches. Assume that the list header `head` will be processed by cache bank 0 and that the new element will be processed by cache bank 1. On Alpha, the `wmb()` will guarantee that the cache invalidates performed by lines 6–8 of Figure 48 will reach the interconnect before that of line 10 does, but makes absolutely no guarantee about the order in which the new values will reach the reading CPU's core. For example, it is possible that the reading CPU's cache bank 1 is very busy, but cache bank 0 is idle. This could result in the cache invalidates for the new element being delayed, so that the reading CPU gets the new value for the

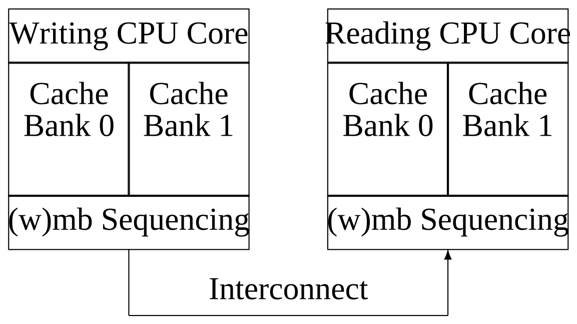


Figure 49: Why rmb() is Required

pointer, but sees the old cached values for the new element. See Compaq's Alpha documentation [Compaq01] for more information, or if you think we are just making all this up.

This can be fixed in an implementation-independent manner by inserting an `rmb()` between the pointer fetch and dereference, as shown on line 19 of Figure 50. However, this imposes unneeded overhead on systems (such as i386, IA64, PPC, and SPARC) that respect data dependencies on the read side. A `read_barrier_depends()` primitive has been proposed to eliminate overhead no these systems [Sarma02b]. It is also possible to implement a software barrier that could be used in place of `wmb()`, which would force all reading CPUs to see the writing CPU's writes in order[McK01d]. However, this approach is deemed to impose excessive overhead on extremely weakly ordered CPUs such as Alpha.[7]

For the moment, `rmb()` should be used on lock-free code paths traversing lists subject to concurrent insertion.

```
 1 struct el *insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     spin_lock(&mutex);
 6     p->next = head.next;
 7     p->key = key;
 8     p->data = data;
 9     wmb();
10     head.next = p;
11     spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         rmb();
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

Figure 50: Safe Insert and Lock-Free Search

[7]CPUs that respect data dependencies would define such a barrier to simply be `wmb()`.

The Linux Kernel Device Model

Patrick Mochel
Open Source Development Lab
mochel@osdl.org

Abstract

Linux kernel development follows a simple guideline that code should be only as complex as absolutely necessary. This design philosophy has made it easy for thousands of people to contribute code, especially in the realm of device drivers: the kernel supports hundreds of devices on over a dozen peripheral buses.

This bottom-up approach to development has provided a great deal of benefit for users of typical systems in the last decade. However, as Linux progresses into new niches and more requirements are imposed on operating systems of modern hardware, lack of unification among device subsystems poses some serious roadblocks.

The new Linux Device Model (LDM) is an effort to provide a set of common interfaces for device subsystems to use. This foundation is intended to enhance the kernel's support for modern platforms and devices, which require a more unified approach to devices.

This paper discusses the attributes of the LDM and the issues they are designed to resolve. It describes the interfaces in a bottom-up approach; in the same manner in which they were devloped. It also discusses the current progress of the effort, and some potential uses of it in the future.

1 Introduction

The LDM was initially motivated by a single goal: to provide a global device tree that could be used to suspend and resume all devices in a computer during system sleep transitions.

Figure 1 show how all devices in a computer connected. Like devices are grouped on a bus. Buses are linked together via bridge devices. All physical devices can be represented via a single tree structure. This tree structure can be walked to provide proper suspend and resume sequences.

Kernel device subsystems have been developed to concisely represent devices of a particular physical type. Because of this, and because of the vast number of physical configurations possible, there is little data or code shared between subsystems. Figure 2 shows how the PCI device hierarchy is represented internally. Though the PCI tree is physically connected to other devices, this hierarchy is autonomous with regard to other internal device representations.

2 The Linux Device Model Core

In order to construct a global device tree, a common device structure was created to represent each physical device in the system. Listing 1 includes the definition of `struct device`, which is the minimum set of data necessary to describe each device in the sys-

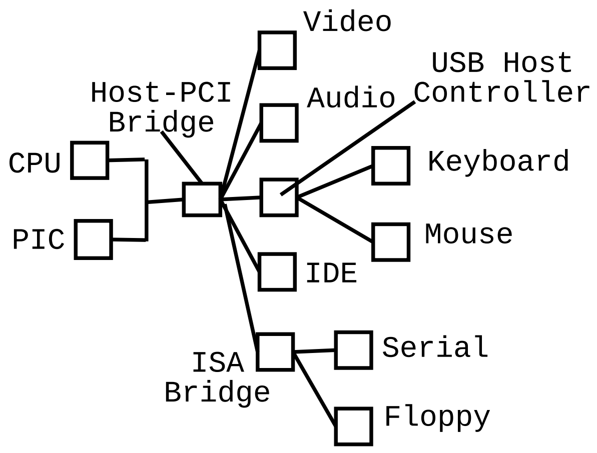


Figure 1: Physical Device Topology

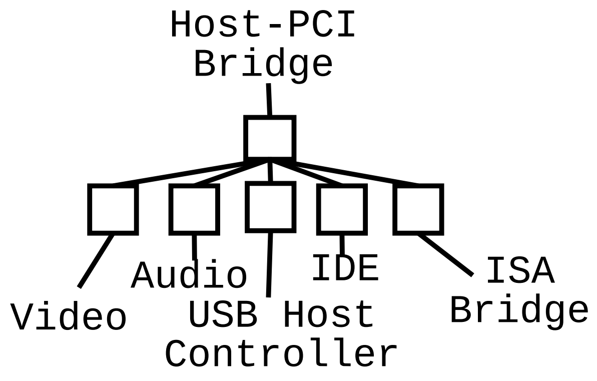


Figure 2: Kernel Repesentation of PCI Topology

```
struct device_driver {
  char   * name;
  list_t   node;
  int (*probe) (struct device * dev);
  int (*remove) (struct device * dev,
                u32 flags);
  int (*suspend)(struct device * dev,
              u32 state, u32 level);
  int (*resume) (struct device * dev,
                u32 level);
};

struct device {
  list_t g_list;
  list_t node;
  list_t bus_node;
  list_t children;
  struct device   * parent;

  char    name[DEVICE_NAME_SIZE];
  char    bus_id[BUS_ID_SIZE];

  spinlock_t      lock;
  atomic_t        refcount;

  struct device_driver * driver;
  void                 * driver_data;
};

int device_register(struct device
                    *dev);

/* device reference counting */
void get_device(struct device *dev);
void put_device(struct device *dev);

/* device-level locking */
void lock_device(struct device *dev);
void unlock_device(struct device
                   *dev);
```

Listing 1: The Device Model Core

tem. It contains little detail about the physical attributes of the device, but provides proper linkage information and support for device-level locking and reference counting.

System bus drivers allocate a device structure for each physical device discovered when probing. The bus driver is responsible for initializing the `bus_id` and `parent` fields of the device and registering the device with the LDM core. The LDM core will then initialize the other fields of the device and add it to the device hierarchy.

Device Reference Counting

The LDM core exports device reference counting primitives

`get_device`, which increments the reference count, and `put_device`, which decrements it. When the reference count reaches 0, it is removed from the device hierarchy and the remove callback of its driver is called to free resources.

The LDM core does not export an interface to explicitly unregister the device. Instead, it relies on reference counting to handle proper garbage collection and removal from the global hierarchy.

The device reference count is initialized to 2 in

`device_register`. It is decremented to 1 when the function exits, leaving the device structure pinned in memory.

Device Locking

The LDM core exports simple primitives to provide device-level locking. The current implementation is a simple spinlock, though this is abstracted from the caller should the type of lock change (e.g. to a semaphore or R/W lock).

Device Drivers

A global device hierarchy allows each device in the system to be represented in a common way. This allows the core to easily walk the device tree to do such things as properly ordered power management transitions. `struct device_driver` in Listing 1 defines a simple set of operations for the core to perform these actions on each device.

The `suspend` and `resume` callbacks provide power management functionality. The `remove` callback is called to logically remove the device from the system. It is called when the device reference count reaches 0, or during system reboot to quiesce all the devices in the system.

`probe` is called when attemptingto bind a driver to a device. This callback is currently unused since driver binding currently happens solely at the bus driver level.

Currently, many bus drivers define a driver similar to this. Instead of converting every device driver to use this common structure, bus drivers implement only one instance of this common structure and bind it to each device discovered. This generic driver then forwards calls to the bus-specific driver. This solution is an interim one only; eventually each driver will use this common structure and register itself with the LDM core instead of a bus.

3 Completing the Device Tree

The Device Model core was designed to explicitly support the semantics of modern peripheral buses and their drivers, such as PCI and USB. These bus drivers have well-defined and mature methods for discovering devices and representing them locally in a tree-like manner. Because the LDM was based on the existing data and behavior of these bus drivers, convert-

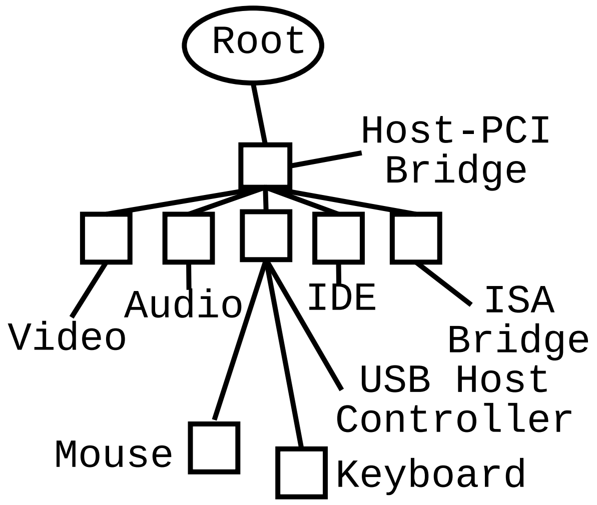


Figure 3: Device Hierarchy with Logical Root Device

ing them to the generic interface typically only involves modifying references to bus-specific structures to generic structures.

There is no common peripheral bus for many of the devices in the system. These devices are referred to as either "platform" devices, including Host-Peripheral Bus bridges and legacy devices; or "system" devices, including CPUs and interrupt controllers. The Linux drivers for these devices represent this logical autonomy.

To complete a global hierarchial representation, these devices must be also be represented. The global hierarchy thus needs some common, top-level entry point.

Device Root

Referring to the figure of device topology, it is apparent that devices are arranged in an acyclical graph, though not necessarily a tree. The kernel bus drivers map subsets of this graph into local tree structures with an explicit root node: the bridge device to the bus. The global hierarchy binds the local trees into one global tree.

Root buses (e.g. root PCI buses) do not have upstream bridges to other peripheral buses. As such, they do not have an explicit parent, and create a forest of devices, instead of one unified tree.

To bind all the devices together, the LDM core creates a logical root device that is the ancestor of all devices in the hierarchy. It is statically allocated and initialized when the LDM core is initialized. Buses that have no obvious parent are registered as children of this device. Figure 3 shows the logical device root and the its relation to the hierarchies of peripheral buses.

Platform Devices

Platform devices are all devices that are physically located on the system board. This includes all legacy devices and host bridges to peripheral buses. host-peripheral bridges are typically not represented in the kernel as devices on a bus; only as parent devices to buses.

These devices appear as autonomous devices in the system responding to I/O requests on hard-coded ports. Drivers for these devices perform device discovery and immediately bind to the devices. These differ from modern bus drivers which perform device discovery in a separate stage than driver binding.

In many modern systems, the system firmware provides information about the devices in the system, often enumerating all of the platform devices. The OS can use this information in lieu of probing legacy I/O ports on platforms that do not support them.To support this firmware enumeration, drivers for platform devices must be taught to use the firmware data for discovery rather than their legacy methods.

Instead of creating special cases in the platform drivers for every firmware discovery mechanism, the method of device discovery is decoupled from the driver binding; legacy probing becomes only one method of device discovery.

```
struct platform_device {
   list_t         node;
   char           name[BUS_ID_SIZE];
   u32            instance;
   struct device  device;
};

int platform_add_device(
   struct device * parent,
   char * busid,
   u32 instance);

struct platform_driver {
   char    * name;
   list_t  node;
};

int platform_register_driver(
      struct platform_driver * drv);
```

Listing 2: The platform bus interface

To implement this, a "platform" bus driver is created to manage platform devices and drivers. As platform devices are discovered, via legacy probing or via a firmware driver, it is added to the bus's list of devices. As drivers are loaded, they register with the bus, and it attempts to bind them to specific devices. Listing 2 lists the interfaces to the platform bus.

Firmware enumeration usually knows the proper ancestral ordering of the devices, so the device is added in the proper location in the hierarchy. Legacy probing usually does not, though it is not necessary to add any special cases for those devices.

Platform devices are of two types: host-peripheral bridges and legacy devices. Bridges do not have parent devices, so it is valid to pass a NULL parent to `platform_add_device`. Figure 3 displays the logical relationship between the device root and the Host-PCI bridge; `platform_add_device` is the means for representing that relationship in the kernel.

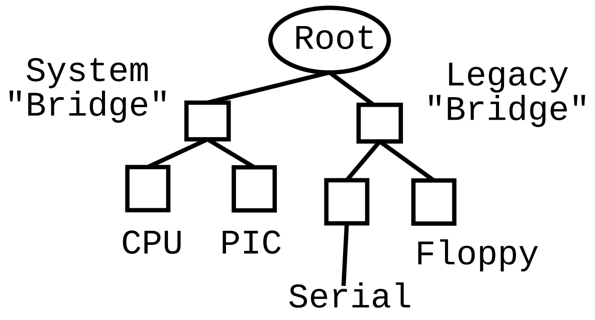


Figure 4: Logical Legacy and System Buses

Legacy Devices

Legacy devices usually do have a parent, though it is difficult to infer exactly who it is when legacy probing is used for discovery. Rather than attempt to guess, a logical "legacy bridge" is created to act as surrogate parent for all legacy devices. To register as a legacy device, a driver uses `legacy_add_device`, which internally calls `platform_add_device`, the legacy bridge as the parent.

```
int legacy_add_device(char * busid,
                      u32 instance);
```

Listing 3: Legacy device interface

System Devices

System devices are devices integral to the function of the computer, such as CPUs, APICs, and memory banks. These devices do not follow traditional Unix *read/write* semantics. They do have attributes though, and most have drivers exporting sort of interface to the rest of the kernel and userspace. However, there are no common bus-level semantics for communicating with the set of system devices as a whole.

It is desirable to group these devices for logical organization. To do this, a logical bus represents the bus that the system devices reside on. Similar to legacy devices, a logical bridge device is created to parent system devices. Devices are added to the system bus using `system_add_device`. Figure 4 displays the hierarchy of the logical buses, and their relationship to the device root.

```
int system_add_device(char * busid,
                      u32 instance);
```

Listing 4: System device interface

4 The User Interface: *driverfs*

During the early development stages of the Device Model, a debugging aid was desired to test various aspects of the code. A device tree, it turns out, maps nicely to a filesystem directory structure.

The device tree was initially exported to userspace using the *procfs* filesystem. A new filesystem, *driverfs*, was soon created to specifically represent the devices. *driverfs* is a simple filesystem based on *ramfs*. It is initialized when the LDM core is initialized, and can be mounted anywhere in filesystem hierarchy.

When registered, every devicehas a *driverfs* directory created on its behalf. It is created in its parent's directory, representing the physical topology. The name of the directory is the `struct device::bus_id field`.

Exporting Device Attributes

The device directories can be populated with files to export device and driver attributes to userspace. These attributes can be accessed using standard the *read* and *write* system calls.

Attributes can be added at any level. The LDM core adds three default files: *name*, *power*, and *status*. Upon device discovery, the bus drivers may add files to export bus-specific attributes. When a driver is bound to a device, it may add files to export device-specific attributes.

```
struct driver_file_entry {
   struct driver_dir_entry * parent;
   struct list_head          node;
   char                    * name;
   mode_t                    mode;
   struct dentry           * dentry;

   ssize_t (*show)(struct device
             * dev,
             char * buf,
             size_t count,
             loff_t off);
   ssize_t (*store)(struct device
             * dev,
             const char * buf,
             size_t count,
             loff_t off);
};

int
device_create_file(struct device
  *device,
  struct driver_file_entry * entry);
```

Listing 5: driverfs interface

Listing 5 shows the `driver_file_entry` object, which is how *driverfs* files are represented. The `show` callback is called when a user reads from a file. The `store` callback is called when a user writes to a file.

To create a file, a caller statically declares a `driver_file_entry` object and initialize the `name`, `mode`, `show` and `store` fields of it. `device_create_file` is used to add the file to the device's directory.

The LDM core clones the `driver_file_entry` structure by allocating a new structure of that type and copying the object passed in. This allows the caller to reuse the same file description to create files for multiple devices without having to manually allocate and initialize each instance.

Operation

The *driverfs* core stores a pointer to the `driver_file_entry` structure in the private data fields of the VFS objects representing the file. From this pointer, the *driverfs* core can obtain the pointer to the device structure. This pointer is then referenced on read and write operations.

During a read operation, *driverfs* allocates a page-sized buffer and passes the buffer pointer to it to the `show` callback. The driver fills the buffer and returns. It is then copied to userspace.

When the file is written to, *driverfs* allocates another page-sized buffer and fills it with data copied from userspace. It passes the buffer pointer to the `store` callback of the driver, which consumes the data.

File Format

The preferred contents of *driverfs* files is one ASCII-encoded value per file. Although these preferences are not enforced, maintaining this standard has several usability advantages:

- A user can read from and write to the file using **cat** and **echo**; tools found on any Linux distribution.
- Coordination between kernel drivers and user space consumers becomes easier; there is no proprietary format for each file.
- A file's contents are obvious to a command line user.
- A file's contents are obvious to a programmer looking at the driver source code.
- It eases creation of automated tools to export device attributes in a more user-friendly manner (i.e. via a GUI).

5 Bus Drivers

```
struct bus_type {
       char        * name;
       list_t      node;
       spinlock_t  lock;
       list_t      devices;
       list_t      drivers;
};

int bus_register(
    struct bus_type * bus);
void bus_unregister(
    struct bus_type * bus);

int driver_register(
        struct device_driver * drv);
void driver_unregister(
        struct device_driver * drv);
```

Listing 6: Bus driver interface

At the time of this writing, most LDM development is concentrated on creating a generic bus type object and set of operations to operate on this type. Listing 6 defines a structure to wrap attributes common across all buses. Consolidating bus data affords the creation of generic routines to manipulate that data.

Bus drivers typically maintain a list of all devices on all buses of their type. This allows for easy searches of devices when binding drivers. Insertion into this list can happen when the bus driver calls `device_register()` for a device.

Device drivers register with their bus, which insert the driver into an internal list and attempt to bind it with every present device. Drivers can instead be taught to use only the generic `struct device_driver` and register with the LDM core.This would insert the driver in the bus's list of drivers, then attempt to bind the driver to the devices on that particular bus (by calling the driver's probe callback for each device).

Device insertion at runtime requires registering a device with the bus and attempting to bind a known driver to it. A userspace agent (**/sbin/hotplug**) is executed to finish configuring the device.

Hotplug insertion events are nearly identical to device discovery when a bus is initially probed, though no buses attempt to bind drivers to devices when they are initially discovered. Driver binding can be coupled to device discovery when the bus is probed, making all device discoveries appear as hotplug events. With centralized lists to manage devices and drivers, this binding can take place when the device is registered with the LDM core.

Many buses do not do locking on their internal lists or reference counting on their devices and drivers. By centralizing the list manipulation routines, proper locking and reference counting can be guaranteed for all buses.

driverfs exports an accurate physical representation of the device hierarchy. It is difficult to navigate, though, since devices can be buried under several obscure directories. By centrally managing bus lists, devices and drivers can easily be added to a driverfs directory owned by the bus.

Conclusion

The Linux Device Model is an effort to consolidate data and interfaces from the many disparate device and driver models in the Linux kernel today. It allows the kernel to do things never possible before, like proper power management and shutdown sequences. It provides common infrastructure to guarantee proper locking, reference counting, and handling of hotplug events for all bus types.

LART Lessons Learned: cpufreq

J.A.K. (Erik) Mouw, Koen Langendoen, Johan Pouwelse
UbiCom program
Delft University of Technology
PO BOX 5031, 2600 GA Delft, The Netherlands
{erik,koen,pouwelse}@ubicom.tudelft.nl

Abstract

In order to run as long as possible on a single battery, battery-powered computers need to be efficient. A large part of that efficiency can be gained by using low-power hardware, but software can also help to reduce power consumption. One way to do that is to let the OS control the CPU frequency and core voltage. This paper will explain the backgrounds of power consumption in CPUs and how clock and voltage scaling can help to decrease the power consumption. It will show the current Linux implementation (cpufreq) and compare it with other implementations.

1 Introduction

The Mobile Multimedia Communications project (MMC, 1996 – 2000)[MMC] and the Ubiquitous Communications program (UbiCom, 1998 – 2002)[UbiCom] at the Delft University of Technology are two related projects that research high data rate cellular networks. The MMC project focused on multimedia communication protocols and applications (text, audio, video) for mobile use, while the UbiCom program extended this to augmented reality and wearable computer systems. Both projects needed a mobile computer platform to test their theories. This platform had to be small, low power, powerful, affordable, and extendible. To solve the tension between these requirements, the emphasis was put on best computing power per watt. Unfortunately there was not such a computer platform available on the market around 1997, so MMC project members decided to design such a system themselves: the Linux Advanced Radio Terminal (LART)[LART].

2 LART

The LART is build around the Intel StrongARM SA-1100 CPU, an embedded processor with an excellent power/MIPS ratio and a large set of built-in peripherals[SA-1100]. The CPU normally runs at 221 MHz, at which speed it delivers a performance comparable to an Intel Pentium 200. The SA-11x0 CPU family is well supported by the Linux operating system, and the mature userland utilities (gcc, etc.) and openness of Linux allows for easy integration of special purpose device drivers.

Figure 1 shows the LART processor board (7.5×10 cm), holding the CPU, 32 MB of EDO DRAM, 4 MB of Flash boot ROM, a connector for two (simple) serial ports, a JTAG debug interface connector, a high-speed extension connector and a low-speed extension connector (at the back of the board). The complete LART processor board weighs only 50 g.

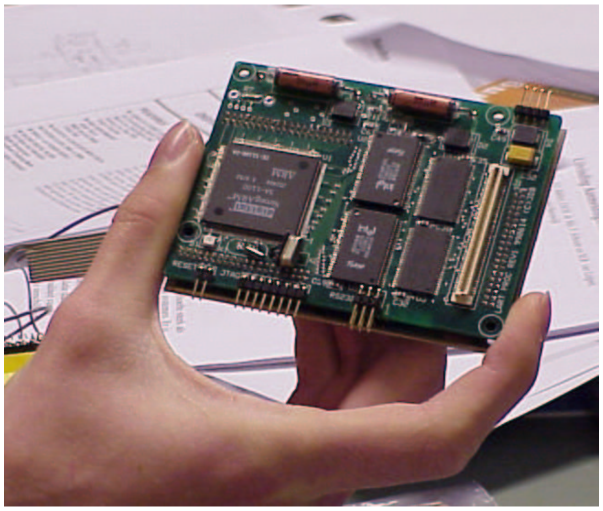


Figure 1: LART processor board

An extension board known as the Kitchen Sink Board (KSB) can be connected and provides a PS/2 interface ($2\times$) for keyboard and mouse, USB client interface, IrDA infra-red link, IDE disk interface, stereo 16 bit 48 kHz audio output, mono 12 bit 26 kHz audio I/O (speakers and microphone), telephony interface, touch panel interface, and an LCD interface. Both the LART and the KSB design files are available under an open license allowing everybody to build boards for themselves or even improve the designs.

At full CPU utilization the processor board consumes about 1 W, which allows it to run for several hours from a single 4.5 V battery. However, the LART design was flexible enough that frequency and voltage scaling could be added at a later stage. This allows the CPU to run at lower frequencies and voltages thereby saving energy. The amount of energy saved depends on the type of application: applications with different CPU load patterns yield different amounts of energy savings. This paper will focus on the Linux implementation of frequency and voltage scaling, Pouwelse et. al. discuss the power saving techniques for different kinds of workloads[Pouwelse].

3 Frequency and voltage scaling

To understand the advantages of frequency and voltage scaling, we will first discuss the theory behind it. Digital CMOS (Complementary Metal-Oxide Semiconductor) circuits as used in the majority of modern microprocessors have both static and dynamic power consumption[Pouwelse][Burd][Ishira]. The static power consumption is caused by bias and leakage currents, and can usually be ignored for designs that consume more than 1 mW of power.

The dynamic power consumption is caused by the logic transactions of the gates in the digital circuit: every charge and subsequent discharge of the gate capacitance drains power. The dynamic power consumption can be modeled by

$$P_{dynamic} = \sum_{i=1}^{N} C_i \cdot f_i \cdot V_{DD}^2 \qquad (1)$$

where N is the total number of gates in the circuit, C_i the load capacitance of gate g_i, f_i the switching frequency of gate g_i, and V_{DD} the supply voltage. Equation 1 clearly shows that lowering V_{DD} yields the largest reduction in power. However, reducing V_{DD} will increase the circuit delay, which can be described by

$$\tau \propto \frac{V_{DD}}{(V_G - V_T)^2} \qquad (2)$$

where τ is the propagation delay of the CMOS transistor, V_T the threshold voltage, and V_G the input gate voltage. The propagation delay restricts the maximum clock frequency for any clock driven digital CMOS circuit. Equations 1 and 2 show there is a trade-off between switching frequency and supply voltage: digital CMOS circuits (and hence microprocessors) can only operate at a lower supply voltage when the clock frequency is lowered to compensate for the increased propagation delay.

Equation 1 can be simplified by assuming that all gates g_i create a collective switching capacitance C operating at a common switching frequency f:

$$P = \alpha \cdot C \cdot f \cdot V_{DD}^2 \quad (3)$$

This equation shows that lowering the clock frequency linearly decreases power, but that voltage reduction results in a squared power reduction. Figure 2 illustrates this conclusion for a LART running a CPU intensive workload at various clock frequencies.

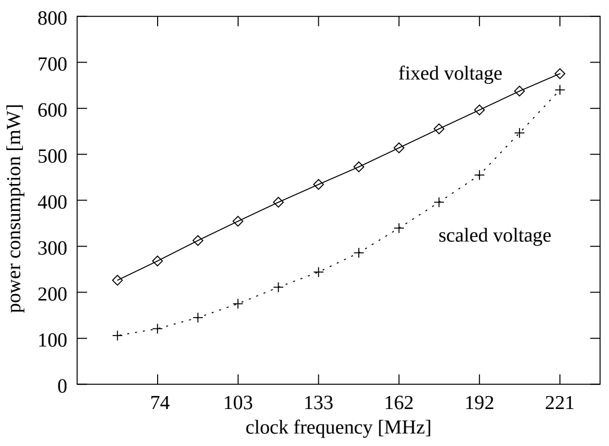


Figure 2: Total power consumption for a LART running a CPU intensive workload

An important observation is that frequency scaling alone only saves *power*, but not *energy*. Running a task at a decreased clock frequency makes that it takes longer to complete that particular task. The task completion time is proportional to $1/f$, and hence the total energy consumed remains the same. Combining frequency scaling with voltage scaling will save power *and* energy because V_{DD} can be scaled with respect to f.

4 Implementation

To exploit the potentials of frequency and voltage scaling, we implemented it on our LART computer platform. The LART frequency and voltage scaling consists of a hardware and software part. The SA-1100 natively supports frequency scaling: the clock frequency can be set in 15 MHz steps from 58 to 221 MHz. It does not, however, support voltage scaling. Therefore the LART design includes additional circuitry to control the core voltage supply.

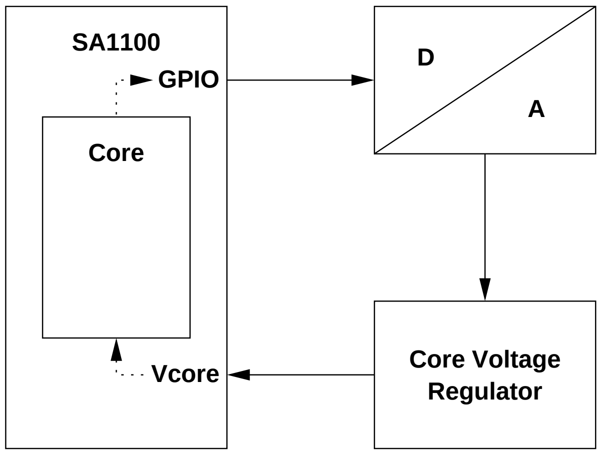


Figure 3: LART voltage scaling hardware

Figure 3 shows how the CPU controls the core voltage: eight General Purpose I/O (GPIO) pins are used to set the output voltage of an 8 bit digital to analog converter (DAC), which on its turn controls the core voltage regulator. The core voltage is thus completely software controlled, and there are a couple of hardware safety measures to prevent the CPU from exposing itself to excess voltages.

The SA-1100 is an embedded CPU and among its built-in interfaces is a memory controller, which should be programmed to generate the necessary waveforms for the memory connected to the system (e.g. SRAM and DRAM). This memory controller is directly driven by the core frequency oscillator, so it has to be reprogrammed at each clock speed change. The SA-1100 is special in that it needs software to reprogram the core voltage and memory settings: most other CPUs have external memory controllers independent of the CPU frequency and hardware controlled core voltage regulators. Figure 4 shows the order of events that have to happen when increasing the clock speed. Decreasing the clock speed reverses the order: decrease clock speed, decrease core voltage, tighten memory settings. When switching to a higher clock speed, the generated memory waveforms are too wide for the current frequency speed and hence decrease the available memory bandwidth. However, this situation only exists for such a small amount of time that it does not decrease the system performance.

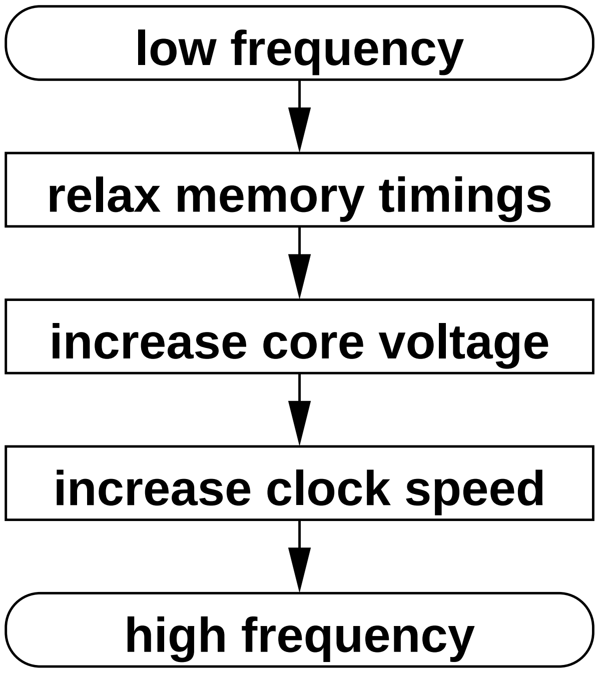


Figure 4: Order of execution for switching to a higher clock speed

The initial Linux driver for the LART clock and voltage scaling hardware exactly followed the procedure depicted in Figure 4. The switching was controlled from a file in the `/proc` file system: in this way the mechanism was implemented in the kernel, while the policy of *when* to change clock speed could be implemented in userland. The initial implementation worked well for a simple system with only the LART processing board, but it did not have enough flexibility to support a LART system with more hardware (like hard disk, PCMCIA interface, etc.), or a system build around a different kind of CPU.

5 Cpufreq

Quite some kernel drivers depend on the `udelay()` function for timed access to hardware. For the ARM family, this function is implemented as a busy wait that uses the `loops_per_jiffy` variable to check if the requested number of micro seconds already passed. The value of the `loops_per_jiffy` variable is derived during the famous *Calibrating delay loop* event when the kernel boots. Because `loops_per_jiffy` depends on the CPU frequency, it needs to be adjusted after a speed change. Fortunately the variable does not need to be recalibrated: it is directly proportional to the CPU frequency so it can be calculated.

When frequency and voltage scaling support for several 80x86 CPUs was added, it became clear that those CPUs use a timer independent from the CPU core frequency to calculate the amount of time to be spend in `udelay()`. Also, these CPUs did not need to reprogram their memory controller. Therefore, Russell King designed a flexible framework for clock and voltage scaling: cpufreq [Cpufreq].

Cpufreq separates the act of changing the CPU speed from the other measures that have to be taken upon a speed change. At kernel initialization, the CPU dependent driver needs to register its `validatespeed()` and `setspeed()` functions with cpufreq. All other hardware drivers that depend on the CPU frequency also need to register themselves with cpufreq so they can be notified for speed changes. A 80x86 cpufreq driver only need to register its `validatespeed()` and `setspeed()` functions, while the SA-1100 driver also has to register the functions that change the memory timings. The value of `loops_per_jiffy` is automatically changed by cpufreq; it is not necessary for 80x86 CPUs, but it is nice that `/proc/cpuinfo` gives an indication of the current CPU speed, even though it is a bogus one.

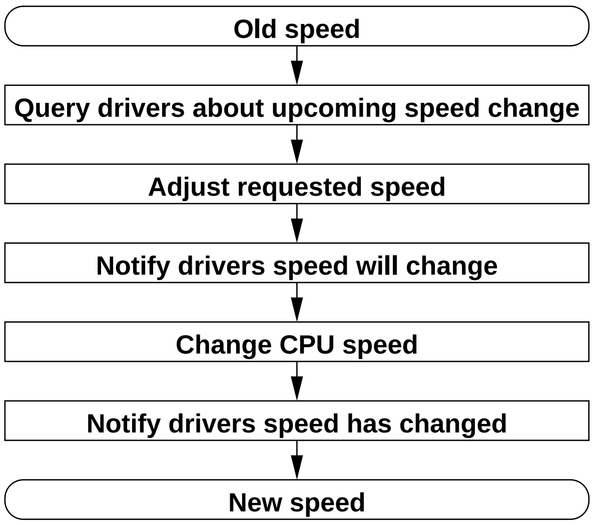

Old speed
Query drivers about upcoming speed change
Adjust requested speed
Notify drivers speed will change
Change CPU speed
Notify drivers speed has changed
New speed

Figure 5: Cpufreq order of execution

Figure 5 shows the cpufreq order of execution at a CPU speed-change request. First of all, all registered drivers are queried about the speed range they can tolerate. A driver that for some reason (like the SA-1100 LCD controller that needs a certain amount of bandwidth) currently can't accept a speed range can limit the requested range to the range it is able to handle. If the new CPU frequency is out of the range the drivers can currently tolerate, it is adjusted to fall within the range. The drivers are then notified about the upcoming CPU speed change, so they can decide to adjust certain parameters. For example: when going to a faster speed, the SA-1100 memory driver will relax the memory timings. Next, the CPU speed will be changed to the requested value using the CPU `setspeed()` function. After that, all drivers will be notified that the CPU speed has changed, so they can adjust their parameters. For example: when going to a slower speed, the SA-1100 memory driver will tighten the memory timings. This completes the speed change and the system can continue to do whatever it was doing before the speed change. Again, the kernel only implements the switching mechanism; the policy can be controlled through a sysctl interface by a userland process.

6 Discussion

The flexible cpufreq framework supports StrongARM SA-1100, StrongARM SA-1110, ARM Integrator, VIA Longhaul, AMD Elan, AMD PowerNow K6, and Intel SpeedStep, while work is underway to add support for AMD PowerNow K7. The current cpufreq implementation is stable and scheduled to be included in Linux-2.5. Following the Unix design rules, cpufreq only implements the *mechanism* to change the CPU speed; the *policy* of when to change speed is left to userland.

A simple userland scheduler that changes the CPU speed by observing the CPU idle time works nice for most workloads, but it breaks down at bursty real-time tasks like real-time video decoding. The CPU speed scheduler will select a low clock frequency when the video decoder decodes low-complexity frames, but it will be too late to select a high clock frequency when the video decoder encounters a high-complexity frame. As a result, the frame will be decoded too late which will be visible to the user. The CPU speed scheduler can also decide to select a high clock frequency so all frames will be completed in time, but in this case the CPU will waste energy. Pouwelse et. al. show that a power aware video decoder is able to combine close-to-optimal energy savings with real-time decoding performance[Pouwelse2][Pouwelse3].

7 Related work

There are two software frameworks for CPU power management. Advanced Power Management (APM)[APM] is an older standard still widely in use that allows the CPU to enter a low power state when executing the idle loop. APM only implements an on/off power savings approach: intermediate power saving levels are not available, even when the CPU is able to switch to multiple performance levels.

The Advanced Configuration and Power Interface (ACPI)[APCI] is the successor of APM. ACPI has a fine-grained CPU power management interface that can be controlled by the OS. Unfortunately, the standard also allows the ACPI BIOS to control the CPU speed without notifying the OS thereby removing the ability for userland scheduling tools to control the CPU speed policy. Another disadvantage of ACPI is that it depends on the BIOS implementation. In many cases, frequency and voltage scaling is not implemented in the BIOS, thereby missing an opportunity to save energy. Fortunately, work is being carried out to fit cpufreq within the Linux ACPI subsystem.

A hardware approach to CPU power management is implemented in the Transmeta Crusoe TM5400 CPU[Crusoe] which implements frequency and voltage scaling in its microcode (“LongRun”). This means that the policy is implemented in the CPU and operates without knowledge about the applications. The scheduler works the same as the simple scheduler described in the previous section, and thus has the same limitations.

8 Conclusions

A well designed experimental computer platform can lead to interesting results: the flexible LART platform allowed to exploit the theoretical power and energy savings of frequency and voltage scaling. The resulting software framework was used, together with other implementations, to get at the generic cpufreq frequency and voltage scaling driver which allows the OS to control the CPU power consumption. Other approaches to control the CPU power either lack the fine grained control cpufreq offers, or try implement the power saving policy at the wrong place.

Cpufreq only implements the mechanism of frequency and voltage scaling. The policy of *when* to change CPU speed is still an active area of research. It is clear that the simple speed scheduler as described in Section 5 does not yield optimal power savings and fails for bursty real-time tasks, but the ideal scheduler still has to be written[Pouwelse3]. As Linus Torvalds remarked: “The really interesting things happen in userland.”

9 Acknowledgements

This work was carried out within the MMC project and the UbiCom program and funded by the Dutch Foundation of Applied Sciences (STW) and the TU Delft DIOC research program. We would like to thank Jan-Derk Bakker for designing an excellent low-power platform and Russell King for the cpufreq framework and the many discussions we had.

References

[APCI] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation, *Advanced Configuration and Power Interface, Revision 2.0*, July 2000.

[APM] Intel Corporation, Microsoft Corporation, *Advanced Power Management (APM) BIOS Interface Specification, Revision 1.2*, February 1996.

[Burd] T. Burd, R. Brodersen, *Processor design for portable systems*, Journal of VLSI Signal Processing, Aug/Sept 1996.

[Cpufreq] D. Jones, R.M. King, J.A.K. Mouw, J.A. Pouwelse, A. van der Ven, *Cpufreq homepage*, `http://www.lart.tudelft.nl/projects/scaling/`

[Crusoe] Transmeta Corporation, *The technology behind the Crusoe processor*, `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`

[Ishira] T. Ishihara, H. Yasuura, *Voltage scheduling problem for dynamically variable voltage processors*, ISLPED, Aug. 1998.

[LART] J.-D. Bakker, M.A.H.G. Joosen, J.A.K. Mouw, *Linux Advanced Radio Terminal*, `http://www.lart.tudelft.nl/`

[MMC] *Mobile Multimedia Communications Project*, `http://www.mmc.tudelft.nl/`

[Pouwelse] J.A. Pouwelse, K. Langendoen, H. Sips, *Voltage scaling on a low-power microprocessor*, Mobile Computing Conference (MOBICOM), Jul. 2001.

[Pouwelse2] J.A. Pouwelse, K. Langendoen, R.L. Lagendijk, H. Sips, *Power-aware video decoding*, Picture Coding Symposium (PCS), 2001.

[Pouwelse3] J.A. Pouwelse, K. Langendoen, H. Sips, *Energy priority scheduling for variable voltage processors*, International Symposium on Low-Power Electronics and Design (ISLPED), Aug 2001.

[SA-1100] *Intel StrongARM SA-1100 microprocessor developer's manual*, available at `http://www.lart.tudelft.nl/doc.php3`

[UbiCom] *Ubiquitous Communications Program*, `http://www.ubicom.tudelft.nl/`

User Interfaces for Clustering Tools

John L. Mugler and Thomas Naughton and Stephen L. Scott[*]
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN
{muglerj, naughtont, scottsl}@ornl.gov

Abstract

This paper discusses ongoing research at Oak Ridge National Laboratory (ORNL) to make computing clusters easier to use. Cluster administration, setup, and use is an active research area with many different components. Two systems for cluster control and administration, that have been experimented with at ORNL, are Managing Multiple Clusters (M3C) and Cluster Control GUI (C2G). M3C uses a Java Servlet in conjunction with a Java application server to handle communications between a remote user and the head node. C2G takes an alternate approach and uses the sshd to handle these messages. Another important issue to consider is the mechanism that is used to handle communications between compute nodes.

A new system that is under construction at ORNL is designed to allow a user to easily keep track of software that is loaded on a node. This system has two components, a node manager daemon and a package services back-end that is basically a database. Multiple software configurations for a compute node can be stored and loaded on a node with this system.

[*]Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

1 Introduction

This paper generally addresses software that is being used on High Performance Computing (HPC) clusters. The goal of such a cluster is running computational code. However, this does not preclude the software from being used to manage or monitor server farms or even groups of desktop workstations.

The control of clusters is a large research area with a wealth of problems to be addressed. The goal of this work is to make clusters easier to install, administrate, and use. There are several inherent problems with designing tools that attempt to meet these goals.

Installation tools that install cluster software have to be simple enough for a beginner to use, and also flexible enough for the expert. This can create the problem of having a tool at the end of the day that pleases neither category of user. Administration tools suffer similar design difficulties, and the problem of differing user skill levels remains. Most administration tools that are available today are command line tools with GUI interfaces. Not many have been designed from the ground up as GUI only tools. User level tool sets have some of the same issues, as users have wide margins of skill when it comes to basic UNIX/clustering knowledge. Also, its difficult to provide a generic system that can handle the wide range of tasks that cluster users perform.

Additionally, representing a cluster with a GUI is not a trivial task. So far, representing 64 machines or even 128 is possible with conventional techniques. When the number of machines starts exceeding this margin, representing cluster nodes with individual icons starts to fail. This is the issue of scalability, and it is a real problem in GUI design for large clusters. Yet scalability must be addressed in order to have modern GUI tool sets for clusters.

The intent of this paper is to summarize ongoing efforts at ORNL at designing and implementing tools to make clusters easier to use. Additionally, the next section surveys some systems that have been developed in other places. Like most software, each tool set has both advantages and disadvantages.

2 Related Work

In order to design new interfaces for clusters, it is important to understand what has been built in the past, and what is in use today. Several tool sets have been developed to help run commands across clusters. These tools are typically command line oriented and are general purpose in nature.

These tool sets basically give a cluster user or administrator the power to easily run commands across an entire cluster. Several different approaches have been taken, and both new software and revisions to existing software are appearing rapidly. The following subsections survey three different tools that are available and in use today.

2.1 Cluster Command and Control (C3)

The C3 set of command line tools from Oak Ridge National Laboratory (ORNL) is up to version 3.x. C3 started life as a collection of Perl scripts and has been re-written in Python. C3 allows a user to run commands either sequentially or in parallel across a cluster. The basic set of commands that C3 provides and their general functionality is listed below [1, p-5]:

- cexec: This is the command that C3 offers, and can be thought of as the basis for most of the other commands. This command enables the execution of any command across an entire cluster.
- cget: Enables file movement from the compute nodes of a cluster to the head node.
- ckill: Kills or terminates a process across clusters.
- cps: Can do a ps command across an entire cluster and produce output for each node. The results are usually stored in a text file.
- cpush: Utilizes rsync to push files or whole directories from the head node of a cluster to all of the compute nodes.
- cpushimage: Uses Systemimager to push an operating system image to a cluster node, or to all of the nodes at once.
- crm: Deletes a file or directory across an entire cluster.
- cshutdown: Can shutdown an entire cluster with one command.

C3 leverages quite a few existing applications to accomplish its work. It uses either SSH or RSH for communication. Additionally, it uses rsync to help speed cpushimage, cget, and cpush, as only the difference between the old and the new image must be transferred.

Another key feature of C3 is its ability to handle multiple clusters from a remote host. C3 uses a c3.conf file to specify both clusters and nodes within clusters. This is currently a unique feature among execution environments for clusters, which is the ability to run the same command across multiple clusters. Additionally a user can use a personal configuration file instead of the default on the host, or even specify a different configuration file when using the tools.

2.2 Scalable Unix Tools (SUT)

This collection of utilities by Argonne National Laboratory, leverages the MPI communication environment to achieve scalability. This set of programs is basically a reimplementation of common UNIX tools to be useful in a parallel environment. The commands are named the same as most UNIX utilities, but are prefaced by a pt[8, p-2,3], such as ptcp, the replacement for cp. Additionally, all of the commands produce output in text, so this extends the UNIX command line to a parallel environment. All of the output can be piped to other common UNIX utilities.

Four new utilities have been produced that have no traditional UNIX counterpart, and it is worth listing them here [8, p-3]:

- ptfps: A parallel implementation of the classic UNIX find command with the same syntax.
- ptdistrib: This command is used to basically run a complex task over a set of files on a remote node. It can also retrieve the results of its operation.
- ptexec: Executes a command on all the nodes in a parallel fashion.
- ptpred: Runs a test to see if a file is present on compute nodes, and returns a one if it is there, a zero otherwise.

The tools also make use of MPD, or the multipurpose daemon, that can quickly start up jobs across an entire cluster, although MPI must be installed to run with MPD to make use of this feature. MPD was created specifically for fast command startup across clusters of computers.

2.3 Ganglia Execution Environment GEXEC

A new system that has been recently been released (April 23, 20002) is the Ganglia Execution Environment, or GEXEC. This system is really a building block for other tools. It is comprised of both a daemon/server and a client that has access to the daemons. Additionally, a library is offered as part of the package, so that new applications can be written and directly use the system. The daemons arrange themselves into a n-ary tree for scalabilityGEXEC

An authd must be run with the system, that verifies who a user is that wants to run a command. This forces security by making a job authenticate on the host on which it is trying to run. The authd system makes use of RSA based encryption via OpenSSL[2].

The client half of this system, uses the gexec command on the command line. Real time node information can be provided by the Ganglia monitoring core, which can prevent trying to run jobs on nodes that are not responsive [2].

3 GUI interface tools

This section of the paper surveys some past work at Oak Ridge National Laboratory, and summarizes current work and development. The last section of the paper dealt with some toolkits that can expand a users control of one cluster, with the notable exception of C3. This is an important task, but control of multiple clusters is becoming increasingly important. It is common to leverage more than one cluster in todays computing world. Splitting up large clusters into multiple systems for better control or just simply segregation for various users is also increasingly common. The notion of using multiple clusters within one domain gives rise to the term federated cluster.

At ORNL, we are defining a federated cluster to be one or more groups of clusters. Command line tool sets are probably not capable of handling situations like this, when the number of clusters can rise to the hundreds and the number of nodes to the thousands. Some type of graphical user interface system will have to developed to handle a system of this nature.

3.1 Managing Multiple Clusters

Managing Multiple Clusters or M3C, was a system that was originally conceived to handle multiple clusters. It was designed to be a distributed application having several distinct pieces. This consisted of a client application in the form of a java applet, a server application in the form of a cgi program running on a web server, and a proxy [6, p2].

To use the system, a user would initiate an action on the client, and pass a message via HTTP to either the proxy or directly to the cgi script. The action would be a request for a cluster to do something. To perform an action on a cluster, the cgi script, running on the head node of a cluster, would write to a file or possibly send a message to a back end process. The back end process would fulfill the request, and provide some form of output back to the cgi script. The cgi script would then send a message back to the client and the client would update the applet in the browser. The proxy was the means by which a client could communicate with more than one cgi script, thus the multiple cluster aspect of M3C [6, pp 3-4] [5].

M3C was designed as a framework first and a package of services second, although six applications were designed in the initial package. A significant advantage of this system was the ability to accept plug-ins. The intent being that many different applications could be written to make use of the system. A plug-in would be provided to the applet, the cgi script, and also to the proxy to make this work.[6, p-2]

3.1.1 M3C: A Different Approach

M3C went through several design changes and modifications. The last prototype of M3C revolved around a major design change. The applet was bypassed in favor of a standalone java client application. The proxy was dropped completely, and the client was designed to be enhanced to take over its responsibility in communicating with multiple clusters.

This simplified the system quite a bit. All communication was still performed via HTTP. The CGI script evolved into a Java Servlet, running on Tomcat which is an open source server designed for running Java Servlets. A simple GUI was constructed to be the client and hard coded with the necessary instructions to run C3 commands on the back end. After implementing and testing this simpler prototype of M3C, it was decided that most of the goals of the system could be met by a standalone client.

3.2 Cluster Control GUI (C2G)

The C2G system has been designed and is being implemented using most of the key ideas of M3C, but with much less infrastructure. The idea of extensibility and using plug-ins to extend a basic system has been kept. In fact, the current design of C2G has been vastly simplified to a standalone client. Since there is such a problem representing large clusters, this new approach tries to avoid the notion of a strict GUI framework altogether. The design approach has evolved to these basic goals:

1. Provide a framework for loading programs that are in the form of python scripts, and provide some form of general GUI interface that ties the system together.
2. Provide secure communication services, or access to such services, that allow these scripts to have access to cluster head nodes. Provide some mechanism so that information about available clusters can be readily determined.
3. Give a script or plug-in writer the ability to provide for their own display of results. Avoid forcing a developer to use the default style of GUI or inherit provided classes.
4. Provide a simple default GUI and API for displaying the output of simple programs.
5. Provide an API for communicating with common cluster execution environments.

The guiding principal behind this system is simplicity. While it might be possible to anticipate some needs of some small subset of cluster users and administrators, it is completely impossible to predict how to support very many cases within a rigid GUI framework.

A GUI does not have to be fast or scalable, it must be responsive to a user. It is reasonable to believe that a C2G client can be run by itself on a user's desktop, and thus computational overhead is not so important. It is up to the GUI to rely on scalable back ends and communication packages to achieve the overall desired goal of increased cluster throughput.

Initially, SSH is being used to pass the messages from the C2G client to the headnode. A configuration file that supports the notion of federated clusters is being used, so that clusters can be defined in a uniform fashion. This file is an XML format and a schema has been produced to describe the file. A basic GUI system has been implemented using Python/tk, and a prototype API is currently being constructed and evaluated. The initial back end execution environment is SSH or RSH, as C2G needs to be able to run some basic commands without reliance on any back end package. The system is still a prototype, but initial results are encouraging.

4 Node Manager/Package Services

As part of the SciDAC:SSS [4] initiative, new software is being designed and written with the purpose of creating scalable clustering software. The goal is to design and implement a complete system that has interoperability between all the pieces. A message passing API has been agreed upon by the participating organizations, based upon XML over sockets. This ensures that the independent pieces can communicate with each other. At this juncture the SciDAC:SSS working groups are principally concerned with identifying and publishing these component interfaces. At Oak Ridge National Laboratory, two components are currently under development Node Manager and Package Services.

4.1 Node Manager

The *Node Manager* (NodeMgr) is a generalized administration component that oversees most static characteristics of the cluster, (e.g. OS, installed software). The current design has NodeMgr providing a select set of functions which can be requested through the prescribed XML/socket interface. These functions include *reboot, halt, power (cycle), getimage, setimage, rebuild, setstate* and *getstate*. The initial prototype leverages C3 [1], OSCAR [7] and the current prototype of Package Services.

NodeMgr uses services provided by other components like Package Services to determine what software is available on a given node. The state information is also currently maintained by the Package Services prototype but may be transferred elsewhere as the system evolves. NodeMgr delegates dynamic aspects such as CPU load, available memory, etc. to the monitoring components. When considering the example of rebuilding a compute node there are obvious interactions among all these components, which is performed through the published component XML/socket interface.

4.2 Package Services

Package Services (PS)is the back end database to NodeMgr. PS is currently a PostgresSQL database which is intended to run on the head node of a cluster. PS has been designed to be as general as possible, and merely stores information. In the current implementation of PS, tables are in place to store information about the nodes and the software that runs on a cluster. A few highlights of PS include:

- The ability to have an image associated with a host or group of hosts. An image is defined as a collection of software packages. Currently an rpm is considered to be a package, but tarballs and other types of packages will also be supported.
- The ability to further tune your software bookkeeping by having software groups. A software group is defined as a collection of compatible packages.
- An image may be defined to be made up of both software groups and images.
- The notion of a hardware group is also available. A hardware group may be associated with an image or collection of images.

PS is still in the design and prototype phase. There are several issues that remain with PS. The first is that it must be able to support large numbers of nodes, and scale well. In its current form this may not be possible, at least to the extent that SciDAC:SSS envisions. It may be necessary to build a front end to the database that is capable of communicating with other PS's as necessary. This is somewhat dependent on the functionality of NodeMgr and the topology of a supported cluster.

The second modification/addition that may be necessary is to provide PS with message passing ability. It has not been decided if NodeMgr will provide all the necessary communications with PS, if another SciDAC:SSS component wants to talk with it.

5 Summary

This paper has summarized three general purpose parallel execution environments that are appropriate for High Performance Computing. These environments are suited to many tasks that administrators and users perform on clusters everyday. A general Graphical User Interface that allows general access to tools of this type is a desired item, and an active area of research.

ORNL is working on a tool named M3C/C2G to satisfy this need, and it is thought that this system can become an interface to many back end clustering tools. The earlier efforts at designing and implementing M3C, helped bring about many of the current design decisions. Like many pieces of widely used software, C2G tries to solve the problem of a general

GUI interface by implementing a fairly small utility, that is good at one thing, and interfaces well with other software such as communications software and parallel execution environments.

Additionally, a system consisting of a pair of services for controlling the software that is loaded on compute nodes was discussed. The design of this system is a direct result of working with the Scidac:SSS project. Cluster distributions like NPACI Rocks [9] and OSCAR, have produced software that can do an initial cluster installation, but the need for modifying and managing compute nodes after the initial installation is still apparent.

References

[1] M. Brim, R. Flanery, A. Geist, B. Luethke, and S. Scott. Cluster Command & Control (C3) tools suite. In *To be published in Parallel and Distributed Computing Practices, DAPSYS Special Edition*, 2002.

[2] Brent Chun. Ganglia cluster toolkit. http://www.cs.berkeley.edu/˜bnc/gexec/.

[3] Scyld Computing Corporation. Scyld beowulf clustering for high performance computing. Technical report, Scyld Corporation, April 2001.

[4] Al Geist et al. Scalable Systems Software Enabling Technology Center, March 7, 2001. http://www.csm.ornl.gov/scidac /ScalableSystems/.

[5] R. Flannery, A. Geist, B. Luethke, J. Schwidder, and S. Scott. The scalable system administrator: via c3 and m3c tools. In *The Second International Workshop on Cluster-Based Computing*, 2000.

[6] Al Geist and Jens Schwidder. Managing multiple pc clusters. Technical report, Oak Ridge National Laboratory, 2000. http://www.csm.ornl.gov/˜geist.

[7] Thomas Naughton, Stephen L. Scott, Brian Barrett, Jeff Squyres, Andrew Lumsdaine, and Yung-Chin Fang. The Penguin in the Pail – OSCAR Cluster Installation Tool. In *The 6th World Multi Conference on Systemics, Cybernetics and Informatics (SCI 2002)*, Invited Session of SCI'02, Commodity, High Performance Cluster Computing Technologies and Applications, Orlando, FL, USA, 2002.

[8] Emil Ong, Ewing Lusk, and William Gropp. Scaleable unix commands for parallel processors: A high-performance implementation. Technical report, Argonne National Laborotory, 2002. http://www-fp.mcs.anl.gov/sut.

[9] Philip M. Papadopoulis, Mason J. Katz, and Greg Bruno. Npaci rocks: Tools and techniques for easily deploying manageable linux clusters. In *Cluster*, 2001. http://www.cacr.caltech.edu.

Improving Linux Block I/O for Enterprise Workloads

Peter Wai Yee Wong, Badari Pulavarty, Shailabh Nagar, Janet Morgan, Jonathan Lahr, Bill Hartner, Hubertus Franke, Suparna Bhattacharya
IBM Linux Technology Center
{wpeter,pbadari,nagar,janetinc,lahr,bhartner,frankeh}@us.ibm.com, bsuparna@in.ibm.com
http://lse.sourceforge.net/

Abstract

The block I/O subsystem of the Linux kernel is one of the critical components affecting the performance of server workloads. Servers typically scale their I/O bandwidth by increasing the number of attached disks and controllers. Hence, the scalability of the block I/O layer is also an important concern.

In this paper, we examine the performance of the 2.4 Linux kernel's block I/O subsystem on enterprise workloads. We identify some of the major bottlenecks in the block layer and propose kernel modifications to alleviate these problems in the context of the 2.4 kernel. The performance impact of the proposed patches is shown using a decision-support workload, a microbenchmark, and profiling tools. We also examine the newly rewritten block layer of the 2.5 kernel to see if it addresses the performance bottlenecks discovered earlier.

1 Introduction

Over the past few years, Linux has made remarkable progress in becoming a server operating system. The release of Version 2.4 of the Linux kernel has been heralded as helping Linux break the enterprise barrier [5]. Since then, the kernel developer community has redoubled its efforts in improving the scalability of Linux on a variety of server platforms. All major server vendors such as IBM, HP, SGI, Compaq, Dell and Sun not only support Linux on their platforms, but are investing a considerable effort in improving Linux's enterprise capabilities. The Linux Technology Center (LTC) of IBM, in particular, has been a major contributor in improving Linux kernel performance and scalability. This paper highlights the efforts of the LTC in improving the performance and scalability of the block I/O subsystem of the Linux kernel.

Traditionally, the kernel block I/O subsystem has been one of the critical components affecting server workload performance. While I/O hardware development has made impressive gains in increasing disk capacity and reducing disk size, there is an increasing gap between disk latencies and processor speeds or memory access times. Disk accesses are slower than memory accesses by two orders of magnitude. Consequently, servers running I/O intensive workloads need to use large numbers of disks and controllers to provide sufficient I/O bandwidth to enterprise applications. In such environments, the kernel's block I/O layer faces a twofold challenge: it must scale well with a large number of I/O devices and it must minimize the kernel overhead for each I/O transfer.

This paper examines how the Linux kernel's block I/O subsystem handles these twin goals of scalability and performance. Using version 2.4.17 of the kernel as a baseline, we systematically identify I/O performance bottlenecks using kernel profiling tools. We propose solutions in the form of kernel patches, all but one of which has been developed by the authors. The performance improvements resulting from these patches are presented using a decision-support workload, a disk I/O microbenchmark and profiling data. In brief, the I/O performance bottlenecks addressed are as follows:

- **Avoiding the use of bounce buffers**: The kernel can directly map only the first gigabyte of physical memory. I/O to high memory (beyond 1 GB) is done through buffers defined in low memory and involves an extra copy of the data being transferred. Capitalizing on the ability of PCI devices to directly address all 4GB, the block-highmem patch written by Jens Axboe can circumvent the need to use bounce buffers.

- **Splitting the I/O request lock**: Each I/O device in the system has an associated request queue which provides ordering and memory resources for managing I/O requests to the device. In the 2.4 kernel, all I/O request queues are protected by a single `io_request_lock` which can be highly contended on SMP machines with multiple disks and a heavy I/O load. We propose a solution that effectively replaces the io_request_lock with per queue locks.

- **Page-sized raw I/O transfers**: Raw I/O, which refers to unbuffered I/O done through the `/dev/raw` interface, breaks I/O requests into 512-byte units (even if the device hardware and associated driver is capable of handling larger requests). The 512-byte requests end up being recombined within the request queue before being processed by the device driver. We present an alternative that permits raw I/O to be done at a page-size granularity.

- **Efficient support for vector I/O**: I/O intensive applications often need to perform vector (scatter/gather) raw I/O operations which transfer a contiguous region on disk to discontiguous memory regions in the application's address space. The Linux kernel currently handles vectored raw I/O by doing a succession of blocking I/O operations on each individual element of the I/O vector. We implement efficient support for vector I/O by allowing the vector elements to be processed together as far as possible.

- **Lightweight kiobufs**: The main data structure used in raw I/O operations is the kiobuf. As defined in 2.4.17, the kiobuf data structure is very large. When raw I/O is performed on a large number of devices, the memory consumed by kiobufs is prohibitive. We demonstrate a simple way to reduce the size of the kiobuf structure and allow more I/O devices to be used for a given amount of system memory.

Most of the kernel performance bottlenecks listed above stem from the basic design of the 2.4 block I/O subsystem which relies on buffer heads and kiobufs. The need to maintain compatibility with a large number of device drivers has limited the scope for kernel developers to fix the subsystem as a whole. In the 2.5 development kernel, however, the challenging task of overhauling the block I/O layer has been taken up. One of the goals of the rewrite has been addressing the scalability problems of earlier designs [2]. This paper discusses the new design in light of the performance bottlenecks described earlier.

The rest of the paper is organized as follows. Section 2 presents an overview of the 2.4 kernel block I/O subsystem. The benchmark environment and workloads used are described in Section 3. Sections 4 through 8 describe the performance and resource scalability bottlenecks, proposed solutions and results. The newly written 2.5 kernel block layer is addressed in Section 9. Section 10 concludes with directions for future work.

2 Linux 2.4 Block I/O

For the purpose of this paper, our review of the 2.4 kernel block I/O subsystem will be limited in scope. Specifically, it will focus on the "raw" device interface, which was added by Stephen Tweedie during the Linux 2.3 development series.

Unix has traditionally provided a raw interface to some devices, block devices in particular, which allows data to be transferred between a user buffer and a device without copying the data through the kernel's buffer cache. This mechanism can boost performance if the data is unlikely to be used again in the short term (during a disk backup, for example), or for applications such as large database management systems that perform their own caching.

To use the raw interface, a device binding must be estabished via the raw command; for example, `raw /dev/raw/raw1 /dev/sda1`. Once bound to a block device, a raw device can be opened just like any other device.

A sampling of the kernel code path for a raw open is as follows:

```
sys_open
. raw_open
. . alloc_kiovec
```

Notice the call to `alloc_kiovec` to allocate a kernel I/O buffer, also known as a kiobuf. The kiobuf is the primary I/O abstraction used by the Linux kernel to support raw I/O. The kiobuf structure describes the array of pages that make up an I/O operation.

The fields of a kiobuf structure include:

```
// number of pages in the kiobuf
int nr_pages;

// number of bytes in the data buffer
int length;

// offset to first valid byte
// of the buffer
int offset;

// list of device block numbers
// for the I/O
ulong blocks[KIO_MAX_SECTORS];

// array of pointers to
// 1024 pre-allocated
// buffer heads
struct buffer_head
   *bh[KIO_MAX_SECTORS];

// array of up to 129 page
// structures, one for each
// page of data in the kiobuf
struct page
  **maplist[KIO_STATIC_PAGES];
```

The `maplist` array is key to the kiobuf interface, since functions that operate on pages stored in a kiobuf deal directly with page structures. This approach helps hide the complexities of the virtual memory system from device drivers – a primary goal of the kiobuf interface.

Once the raw device is opened, it can be read and written just like the block device to which it is bound. However, raw I/O to a block device must always be sector aligned, and its length must be a multiple of the sector size. The sector size for most devices is 512 bytes.

Let us examine the code path for a raw device read:

```
sys_read
. raw_read
. . rw_raw_dev
. . . map_user_kiobuf(READ,
                        &mykiobuf,
                        vaddr, len)
```

The result of the call to `map_user_kiobuf()` is that the buffer at virtual address `vaddr` of length `len` is mapped into the kiobuf, and each entry of the kiobuf `maplist[ ]` is set to the page structure for the associated page of data. Note that some or all of the user buffer may first need to be paged into memory:

```
. . . map_user_kiobuf
. . . . find_vma
. . . . handle_mm_fault
```

Once all of the pages of the data buffer are locked in memory, read processing continues with a call to `brw_kiovec()`, where for each sector-size chunk of the data buffer, a pre-allocated buffer head associated with the kiobuf is initialized and passed down to `__make_request`. `__make_request()` calls `create_bounce()` to create a bounce buffer as needed, acquires the `io_request_lock`, and uses buffer head information to merge/enqueue the request onto the device-specific request queue.

```
. brw_kiovec(READ, num_kiobufs=1,
            &mykiobuf,dev,
            mykiobuf->blocks,
            sector_size=512)
. . submit_bh
. . . generic_make_request
. . . make_request(&request_queue,
                   &buff_head)
. . . . . create_bounce
. . . . . generic_plug_device
. . . . . <elevator processing>
. . . . . add_request (enqueue)
. kiobuf_wait_for_io
```

Requests are dequeued when the scheduled `tq_disk` task calls `run_task_queue()` which invokes `generic_unplug_device()`. In the case of SCSI, `generic_unplug_device()` invokes `scsi_request_fn()` which dequeues requests and sends them to the driver associated with the request_queue/device.

```
run_task_queue
. generic_unplug_device
. . q->request_fn(scsi_request_fn)
. . . blkdev_dequeue_request(dequeue)
. . . scsi_dispatch_cmd
```

The `read()` system call returns once the I/O has completed; that is, after all buffer heads associated with the kiobuf have been processed for completion.

3 Workload and experimental setup

We have been using a decision support benchmark and a disk I/O microbenchmark to study the performance of block I/O. The decision support workload (henceforth called DSW) consists of a suite of highly complex queries accessing a 30GB database. We use IBM DB2 UDB 7.2 as the database management system.

The disk I/O microbenchmark (henceforth called DM) is a multi-threaded disk test. There are a total of 32 raw devices which are mapped to 32 physical disks. DM creates 32 processes. For the read test, each process issues 4096 reads of 64KB each to a raw device. The readv test issues the same number of reads, but uses 16 iovecs of 4KB each.

For both benchmarks, the system was rebooted before each set of runs. For DSW, each set consisted of a sequence of queries run back to back three times. For DM, each set consisted of the read/readv runs performed back to back three times. We took the average of three runs for the score and CPU utilization.

The benchmarks were run on an 8-way 700MHz Pentium III machine with 4 GB of main memory. The system used for DSW had a 2 MB L2 cache and 6 RAID controllers. The system used for DM had a 1 MB L2 cache and 4 RAID controllers. Each controller was connected to two storage enclosures with each enclosure containing 10 9.1 GB, 10000 RPM drives. The large number of attached disks allowed a high degree of parallel data access and is typical of the environments in which decision-support workloads are run.

Our baseline (henceforth called Baseline) was Linux 2.4.17 with Ingo Molnar's SMP timer patch applied, plus a number of resource-related changes. In addition, readv was used by the database management system for I/O prefetching. The four main patches discussed in subsequent sections are block-highmem, io_request_lock, rawvary and readv/writev. To measure their performance impact incrementally, we used 4 kernels: SB for Baseline+block-highmem, SBI for SB+io_request_lock, SBIR for SBI+rawvary and SBIRV for SBIR+readv/writev.

As a first step towards identifying I/O bottlenecks, the Baseline kernel was profiled using the Kernprof tool [4]. Table 1 shows the percentage of time spent in the most time-consuming kernel functions running a DSW query on the Baseline kernel. We see that `bounce_end_io_read()` is the most expensive function of non-idle time. This function is used when the kernel performs I/O using bounce buffers. The problem caused by bounce buffers and its resolution is described in the next section.

Kernel Function	% Total Time
default_idle	52
bounce_end_io_read	8
do_softirq	7
tasklet_hi_action	6
__make_request	3

Table 1: Profiling data showing percentage of time spent in different kernel functions while running a DSW query on the Baseline kernel.

4 Avoiding the use of bounce buffers

To explain the bounce buffer problem we first take a look at how the Linux 2.4 kernel addresses physical memory. The discussion assumes an x86 architecture though most of the concepts apply to all 32-bit systems. The 4 GB address space defined by 32 bits is divided into two parts: a user virtual address space (0-3GB) and a kernel virtual address space (3-4GB). The physical memory of a system (which is not limited to 4 GB) is divided into three zones:

- DMA Zone (0-16 MB): ISA cards with only 24-bit DMA space use this zone.
- Normal Zone (16 MB-896 MB): Memory in this range is directly mapped into the kernel's 1 GB of virtual address space starting at PAGE_OFFSET (normally 0xC0000000).
- High Memory Zone (896 MB-64 GB): Page frames in this zone need an explicit mapping into kernel virtual address space (via the `kmap()` system call) before they can be used by the kernel.

DMA operations on memory by I/O devices use physical addresses. Since the kernel cannot address high-memory DMA buffers directly while setting up a buffer for DMA, it

Kernel	Increase in MOI (%)	CPU Utilization (%)		
		user	kernel	idle
Baseline	—	16	43	41
SB	37	22	71	7
SBI	78	41	37	22
SBIR	16	47	34	19
SBIRV	18	55	9	36

Table 2: Performance impact of various patches on the metric of interest (MOI) and CPU utilization for the decisions support workload (DSW). Increases are reported w.r.t the kernel on the previous line.

Kernel	I/O transfer rate		CPU Idle Time	
	Value (MB/s)	Increase (%)	Value (%)	Increase (%)
Using read				
Baseline	54	—	64	—
SB	133	147	21	-68
SBI	235	77	61	192
SBIR	240	2	94	55
2.5.17 kernel	243	—	97	—
Using readv				
SBIR	104	—	41	—
SBIRV	241	132	94	130
2.5.17 kernel	150	—	61	—

Table 3: Performance impact of various patches on the I/O transfer rate and CPU utilization for the disk I/O microbenchmark (DM). Increases are reported w.r.t the kernel on the previous line. Results are also shown for the 2.5.17 kernel.

allocates an area in low memory called the bounce buffer. It then supplies the buffers physical address to the I/O device. Consequently, data transfer between the device and the high-memory target buffer necessitates an extra copy through the bounce buffer. This degrades system performance by using up low memory (for the bounce buffer) and adding the overhead of a memory copy for each I/O transfer.

The bounce buffer is unnecessary for 32-bit PCI devices, which can normally address 4 GB of physical memory directly. Such devices can access high memory directly even though the kernel cannot. The block-highmem patch from Jens Axboe utilizes this property to permit high-memory DMA to occur without the use of bounce buffers.

To make use of the block-highmem patch, most device drivers require a few changes which are documented in the I/O Performance HOWTO [9].

The elimination of bounce buffers is illustrated by Table 4 which again shows the most time-consuming kernel functions while running DSW using the SB kernel. Comparing the entries to those shown in Table 1, we find that

Kernel Function	% Total Time
__make_request	35
default_idle	17
scsi_dispatch_cmd	4
do_ipsintr	4
scsi_request_fn	4

Table 4: Profiling data showing percentage of time spent in different kernel functions while running a DSW query on the `SB` kernel

bounce buffers are no longer being used.

The second row of Table 2 indicates the performance improvement seen by DSW using the block-highmem patch. The metric of interest (MOI) increases by 37%. Similar trends are seen in the performance of DM in Table 3 with the I/O throughput of the read test increasing from 54 MB/s to 133 MB/s (corresponding to a 147% improvement).

Eliminating bounce buffer usage causes another I/O bottleneck to appear. Comparing Tables 4 and 1 we find that `__make_request` is now the most expensive kernel function and the idle time has been reduced from 64% to around 21% under DM, 41% to 7% under DSW. Both these changes are due to the I/O request lock which is the next bottleneck discussed.

5 Splitting the I/O request lock

As mentioned in the last part of the previous section, Tables 1 and 4 indicate a large fraction of time spent in __make_request and a large drop in idle time when DSW is run on `SB`. Using the Lockmeter [3] profiling tool allows us to investigate whether there are any highly contended locks (spinlocks or reader/writer locks). Table 5 shows the lockmeter statistics for the io_request_lock when DSW is run on `SB`. It shows that 66.2% of 8 CPUs are consumed by spinning on the global `io_request_lock` and the function in which the lock sees high contention also corresponds to the most expensive kernel function in Table 1.

The `io_request_lock`, which is a global serialization device, imposes system-wide serialization on enqueuing block I/O requests. The request enqueuing functions use the lock to protect all request queues collectively which means that only one request can be queued at a time.

During normal I/O operations, request queues are accessed and modified by enqueuing and dequeuing functions. Since multiple threads execute these functions, queue integrity must be protected. Code analysis shows that queuing operations on a given queue involve access to queue-specific data, request list anchor (queue_head), request free list (rq), plug state (plugged), but do not require access to data used by queuing operations on other queues. This means that maintaining queue data integrity does not require serialization of queuing to different queues. Queuing operations on different queues are logically independent and can execute concurrently. Of course, multiple queuing operations to the same queue must still be serialized.

To implement concurrent enqueuing, we replaced `io_request_lock` in enqueuing functions with per queue locks (`request_queue.queue_lock`). This serializes enqueuing to the same queue while allowing concurrent enqueuing to different queues. With this change dequeuing functions can no longer rely on `io_request_lock` to serialize with enqueuing functions. To restore this serialization, dequeuing functions were modified to acquire `queue_lock` in addition to `io_request_lock` when accessing queues.

Kernel function holding lock	Lock Utilization (%)	Mean Lock Hold Time (μs)	Lock Spin Time Mean (μs)	Lock Spin Time % CPU	Number of lock acquisitions
`All spinlocks`		3.7	62.0	66.8	68774051
io_request_lock	50.2	5.2	65.0	66.2	15640659
. __make_request	23.5	3.8	64.0	42.8	9973270
. do_ipsintr	8.3	20.0	66.0	3.1	660212
. scsi_dispatch_cmd	6.8	13.0	66.0	3.9	877838
. generic_unplug_device	4.5	8.8	65.0	3.2	835530

Table 5: Lockmeter data for io_request_lock with DSW on the SB kernel.

To minimize interlocking between dequeueing and enqueueing functions, we added another level of locks inside dequeueing functions. This allows us to maintain our focus on enqueuing and avoid the impact of further reducing the scope of the `io_request_lock`.

When the above modifications to the generic block I/O code were published for comment, the Linux development community expressed concern about making such major changes to the mature 2.4 kernel. Since the patch modified the locking structure in code which affected all block I/O devices, many viewed the code impact as undesirably pervasive. Unforeseen impacts to other code such as IDE and some device drivers were also pointed out. Since SCSI configurations represent a significant part of our scalability goal and concurrent queuing can be implemented for SCSI without affecting generic i/o code, we decided to isolate SCSI code for our development purposes. Fortunately, the block I/O subsystem provides for such isolation through dynamically assigned I/O queuing functions stored in the request queue and indirectly invoked as function pointers.

To contain code impact within the SCSI subsystem, generic enqueuing and dequeuing functions were copied, renamed, and modified for concurrent queuing. The following generic block I/O (ll_rw_blk.c) functions provided baselines for SCSI functions:

```
__make_request => scsi_make_request
generic_plug_device =>
  scsi_plug_device
generic_unplug_device =>
  scsi_unplug_device
get_request => scsi_get_request
get_request_wait =>
  scsi_get_request_wait
blk_init_queue => scsi_init_queue
```

Concurrent queuing is activated for all devices under an adapter driver by setting the new `concurrent_queue` field of the `Scsi_Host_Template` structure used for driver registration. This allows control over which drivers use concurrent queuing and preserves original request queuing behavior by default. Drivers which enable concurrent queuing must protect any request queue access with queue locks.

With the application of the `io_request_lock` patch (IORL), the MOI of DSW improves by 78% over the baseline `SB`, as is seen in row three of Table 2. The transfer rate of DM also increases significantly from 133 MB/sec to 235 MB/sec (Table 3). Note that there is a significant increase of idle time in both cases due to the reduction of the spin time. Table 6 verifies that the lock contention seen by the io_request_lock has been reduced. `scsi_make_request()` is shown using a per-queue lock and the aggregate contention on the per-queue locks is reduced as well.

Table 7 lists the most expensive kernel func-

Kernel function holding lock	Lock Contention (%)	Mean Lock Hold Time (μs)	Lock Spin Time Mean (μs)	Lock Spin Time % CPU	Number of lock acquisitions
`All spinlocks`		2.1	15.0	13.9	63777886
io_request_lock	39.6	8.7	32.0	7.6	2490263
. do_ipsintr	16.3	26.0	32.0	1.4	339486
. scsi_unplug_device	11.7	18.0	32.0	1.2	357540
. scsi_dispatch_cmd	8.4	13.0	31.0	1.4	363421
scsi_make_request	15.3	0.9	13.0	0.3	9520872

Table 6: Lockmeter data showing benefits of the IORL patch for DSW on the SBI kernel.

Kernel Function	% Total Time
default_idle	41
schedule	4
ips_make_passthru	4
tasklet_hi_action	3
do_softirq	3
brw_kiovec	3
scsi_back_merge_fn_dc	3
scsi_release_buffers	3
scsi_back_merge_fn_	2
scsi_dispatch_cmd	2
end_buffer_io_kiobuf	2

Table 7: Kernprof data for DSW on the SBI kernel.

tions for DSW running on `SBI`. A significant fraction of kernel time is spent in `brw_kiovec()` and many SCSI mid-layer functions. One reason for that is the use of 512-byte blocks for raw I/O as explained in the next section.

6 Raw I/O optimization patch

This section provides information on the optimization patch that we developed to increase the block size used for raw I/O. The patch can significantly improve CPU utilization by reducing the number of buffer heads needed for such operations.

As explained in Section 2, `rw_raw_dev` calls `map_user_kiobuf` to map the user buffer into a kiobuf, and then invokes `brw_kiovec` to submit the I/O. `brw_kiovec` breaks up each mapped page into sector-size pieces (normally 512 bytes) and passes them one at a time to `make_request`. Each sector-size piece is represented using one of the 1024 pre-allocated buffer heads associated with the kiobuf. Assuming a sector-size of 512 bytes, `brw_kiovec` would use 512 buffer heads and invoke `make_request` 512 times to process a 256K raw read or write.

`make_request` uses the buffer head information to enqueue the request on the device-specific request queue and returns to `brw_kiovec`. When the lesser of all mapped pages or `KIO_STATIC_PAGES` of the kiobuf have been processed in this way, `brw_kiovec` calls `kiobuf_wait_for_io`. `kiobuf_wait_for_io` returns after the I/O completion routine has been called for all of the mapped buffer heads of the kiobuf.

While the block I/O subsystem will normally merge buffer heads into larger requests, there is still overhead incurred with each buffer head. For example, the interrupt handler for the block device must invoke the `b_end_io` method for each buffer head at I/O completion. The second column of Table 8 shows function call frequencies in a call graph trace for 128 reads of 128KB each using a 512-byte block size. The

Kernel function	Frequency	
	Baseline	Baseline+rawvary
sys_read	138	138
. raw_read	128	128
. . rw_raw_dev	128	128
. . . brw_kiovec	128	128
. . . . submit_bh	32768	4096
. . . . . generic_make_request	32789	4160
. . . . . . _make_request	32789	4160
. . . . elevator_linus_merge	32659	4029
. . . . scsi_back_merge_fn_c	32641	4013

Table 8: Reduction in frequencies of function calls using the rawvary patch for 128 reads of 128KB each.

large number of calls to `submit_bh()` indicates the severity of the problem.

The patch we developed can reduce 8-fold the number of buffer heads required for a raw I/O operation. This was accomplished by changing `brw_kiovec` to break up the user buffer into sector-size pieces only until the buffer address is aligned on a page boundary. Once properly aligned, the remainder of the mapped pages are submitted to `make_request` with a block size (`b_size`) of 4 KB instead of sector-size. Note that the last buffer head may have a `b_size` which is neither sector-size nor 4 KB depending on the total length of the I/O request.

Since we could not practically determine whether a given device driver can support buffer heads of variable-block sizes in a merged request, the patch enables the optimization for the Adaptec, Qlogic SCSI and IBM ServeRAID drivers only. Other drivers can make use of the patch by setting the `can_do_varyio` bit in the `Scsi_Host_Template` structure before calling `scsi_register`.

The third column of Table 8 highlights the reduction in kernel overhead as a result of using the patch. The number of calls to `submit_bh` are reduced by a factor of 8. The MOI of DSW improved by 16% over SBI, as seen in the fourth row of Table 2. The transfer rate of DM also increased slightly from 235 MB/sec to 240 MB/sec (Table 3). However, there was an improvement of 55% in the idle time.

The raw I/O optimization patch, also known as the rawvary patch, has been integrated into Andrea Arcangeli's 2.4.18pre7aa2 kernel and Alan Cox's 2.4.18pre9-ac2 kernel.

7 Efficient support for vector I/O

Scatter-gather I/O is needed by an application when it needs to transfer data between a contiguous portion of a disk file and non-contiguous memory buffers in its address space. Typically this is done by invoking the `readv()/writev()` system calls and passing an array of `struct iovec` entries. Each `iovec` entry represents a contiguous memory buffer of length `iov_len` located at `iov_base`. This entry is henceforth called an *iochunk* since the kernel does not define a distinct name for it and the term iovec suggests an array rather than an individual element. To simplify the discussion, we refer only to the readv operation. For raw I/O operations, writev differs mainly in the direction of data transfer.

In the 2.4 kernel, the readv system call using a file descriptor is implemented by calling the corresponding file's readv function. When there is no readv function exported, as is the case for raw I/O, the kernel defaults to using repeated invocations of the file's read function which is always defined. Each iochunk of the iovec leads to a separate blocking read being performed. This imposes a dual penalty on the application. It imposes the overhead of multiple calls to various functions in the entire I/O processing path from the top level `sys_readv()` down to the SCSI layer elevator and merging functions. Worse, it serializes the I/O requests seen by the low-level device driver. Since a separate read/write is performed for each iochunk and these calls block until I/O completes, the kernel's ability to take advantage of large DMA operations is severely limited. The elevator code invoked by the `make_request()` function cannot merge requests from different iochunks and hence the SCSI device driver cannot create large scatter-gather lists for the controller.

To reduce this inefficiency, we created a patch defining readv and writev functions for raw devices. The functions operate in two phases while processing an iovec. In the first phase, they map the pages of several iochunk buffers into a single kiobuf. The number of pages mapped to a single kiobuf is limited by the KIO_STATIC_PAGES limit (which is 65 when the system page size is 4 KB). Once this limit is reached (or if the entire iovec has been mapped), `brw_kiovec()` is invoked to submit the I/O represented by the kiobuf. As explained in Section 2, `brw_kiovec()` is a blocking function that returns only when the corresponding I/O is complete or if there is an error. The two phases are repeated until all iochunks of the iovec are processed.

The patch relies upon one important modification to `struct kiobuf`. As explained in Section 2, `struct kiobuf` has only one offset and length field. The offset field represents the offset into the (virtual) memory buffer. When the pages of multiple memory buffers are mapped in to the same kiobuf, we need a per-page offset and length information. We modified `struct kiobuf` to add this information using the following structure:

```
struct pinfo
{
  int poffset[KIO_STATIC_PAGES];
  int plen[KIO_STATIC_PAGES];
};

struct kiobuf
{
  :
  :
  struct pinfo *pinfo;
}
```

There are other approaches to providing readv/writev support. In an earlier attempt, we tried to map an iovec onto a `kiovec` consisting of multiple kiobufs. However, that approach increased memory consumption since `struct kiobuf` is quite heavyweight and also because the `brw_kiovec()` function only submits I/O for `KIO_STATIC_PAGES` one at a time. Mapping one iochunk onto one kiobuf would have resulted in wasted pointers in the `map_array` without increasing the granularity at which I/O was submitted to the lower layers. Our current approach fits in well with the 2.4 kernel's practice of using only one kiobuf per file. The issue of the heavyweight `struct kiobuf` is discussed in Section 8 though the changes shown there do not warrant reexamining our choice to map multiple iochunks into a single kiobuf.

Using the readv/writev patch improves the MOI of DSW by 18% (Table 2) and the I/O transfer rate of DM from 104 MB/s to 241 MB/s (Table 3). CPU utilization also decreases significantly for both cases.

8 Lightweight kiobufs

In the 2.4.17 kernel, a kiobuf is allocated for each raw device open. The allocated kiobuf is saved in the `f_iobuf` field of the `file` object for the device special file and is used for doing reads/writes on the raw device. Each kiobuf is 8792 bytes in size and is allocated from `vmalloc()` space which is generally 128 MB. Middleware such as database managers often keep a large number of files open. For raw I/O, the number of open calls generally scales with the number of devices (which are accessed through device special files). In such cases, a heavyweight kiobuf is a drain on the kernel's low memory in general and `vmalloc` space in particular.

To enable a large number of raw devices to be opened simultaneously, we modified the kiobuf structure to reduce its memory footprint. Much of the memory consumed by a kiobuf is due to the two arrays:

```
struct buffer_head
   *bh[KIO_MAX_SECTORS];
unsigned long
   blocks[KIO_MAX_SECTORS];
```

With `KIO_MAX_SECTORS` being 1024, these arrays consume 8192 bytes.

We changed the kiobuf structure as follows:

1. The buffer head array `bh` was replaced by a linked list. To link the various buffer heads of a kiobuf together, we used the `b_next_free` field of struct buffer_head. This field is not used in buffer-head processing in the raw I/O path.

2. The `blocks` array was replaced by a single number. Normally, the `blocks` array contains the physical disk block numbers corresponding to the logical blocks of a file. For accesses which don't go through a filesystem, the logical and physical disk blocks are the same. Hence, for raw I/O, the blocks array contains sequential numbers. We replaced the blocks array by a single number indicating the starting disk block and modified the code doing raw I/O to generate the remaining sequence of disk block numbers.

Together these modifications reduced the size of the kiobuf to 608 bytes and allowed them to be allocated using `kmalloc()` instead of `vmalloc()`.

A further reduction in the memory footprint of the kiobuf was enabled by the use of the raw-vary patch described in Section 6. Since I/O is done 4KB at a time, a kiobuf needs only `KIO_STATIC_PAGES` (65) buffer heads instead of `KIO_MAX_SECTORS` (1024) to represent the maximum I/O that can be done using a single kiobuf.

9 2.5 changes – tackling the root of the problem?

In part, the block layer rewrite in 2.5 was motivated by some of the well known shortcomings of the 2.4 block layer that we came across in the earlier sections. Of major concern was the suboptimal performance and resource overhead in the case of large I/O requests, I/O on high memory addresses, and I/O operations that do not originate directly from the buffer cache like raw/direct I/O and page I/O.

Most of these problems stemmed from the use of the buffer head as the unit of I/O at the generic block layer, and the basic limitations on the size and nature of I/O buffers that could be represented by a single buffer head. It could only be a contiguous chunk at a virtually mapped address, of size one blocksize unit, which could not exceed a page and had to be aligned at a block boundary (as per the block size used). This led to the described inefficiencies in handling large I/O requests and readv/writev style operations, as it forced such requests to be broken up into small chunks so that they could be mapped to buffer heads before being passed on one by one to the generic block layer, only to be merged back by the I/O scheduler when the underlying device is capable of handling the I/O in one shot. Also, using the buffer head as an I/O structure for I/Os that didn't originate from the buffer cache unnecessarily added to the weight of the descriptors which were generated for each such chunk.

At the same time, one of the good things about the original design was that splitting and merging of requests was a simple matter of breaking or chaining pointers, without requiring any memory allocation or move.

In the context of raw or direct I/O, a second aspect of concern was the weighty nature of the higher level kiobuf data structure as discussed in earlier sections. One of the shortcomings of the kiobuf is that a single kiobuf can represent only a contiguous user address range, which makes it unsuitable for user space memory vectors of the form supplied by readv/writev. While arrays of kiobufs, namely kiovecs, are defined, they are too heavyweight for use in readv/writev.

Another crucial issue addressed in the rewrite was the matter of the single global I/O request lock bottleneck, especially in the case of independent/parallel I/Os to multiple disks.

9.1 The origin of BIO

The solution implemented in 2.5 by Jens Axboe [2] addresses these inefficiencies at a fundamental level by defining some new data structures. A flexible structure called BIO has been created for the block layer instead of using the buffer head structure directly, thus eliminating any associated baggage and restrictions. The abstraction is sector oriented and is unaware of filesystem block sizes.

The BIO structure uses a generic vector representation pointing to an array of tuples of `<page, offset, len>` to describe the I/O buffer and has various other fields describing I/O parameters and state that needs to be maintained for performing the I/O. The core memory vector representation is capable of describing a set of non-page aligned fragments in a uniform manner across various layers including zero copy network I/O, and kernel asynchronous I/O [1]. This makes it possible for the same descriptor to be passed across subsystems and be useful for things like streaming I/O from network to disk and vice-versa. Such a descriptor can directly refer to user space buffers in a process context independent way, and forms an I/O currency similar to that proposed in [7].

The new scheme enables large, as well as vectored I/Os, to be described as a single unit within the limits of the device capabilities and is adequate for specifying high memory buffers as well since it doesn't require a virtual address mapping. The underlying DMA mapping functions have been modified to work with this representation. Bounce buffers become necessary only where the device does not support I/O into high memory buffers. In situations where the driver needs to access the buffer by virtual address, it performs a temporary kmap (e.g. if falling back to PIO in IDE).

A low level request structure may consist of a chain of BIOs (potentially arising from multiple sources or callers) for a contiguous area on disk, a concept which retains some of the goodness of the original design in terms of ease of request merging, and treatment of individual completion units. The BIO structure maintains an index into the vector to help keep track of which fragments have been transferred so far, in case the transfer or a subsequent copy happens in stages. Notice also, that potentially, a single entry in the vector could describe a fragment greater than a page size, i.e. across contiguous physical (or perhaps more accurately, logical) pages. Splitting an I/O request involves cloning the BIO structure and adjusting the indices to cover the desired portions of the original vector.

Using a separate structure introduces a level of allocation and setup in some cases as a BIO has to be constructed for each I/O (e.g. rather than directly utilizing a bh in the case of buffered I/O). Typically BIOs are allocated from a designated BIO mempool, where mempool refers to Ingo Molnar's new memory pool infrastructure in 2.5. The allocation scheme is designed to avoid deadlocks as in a scenario when the I/O in question is a writeout issued under memory pressure. A caller avoids possibilities of holding on to a BIO without initiating any action (like starting low level I/O) that would eventually recycle it back to the pool. The situation gets tricky if further BIO allocations become necessary in order to proceed with the request (e.g. a bounce BIO in situations where the device doesn't support highmem I/O, or BIO allocations required for splitting the I/O in the case of lvm/md/evms). To avoid any possibility of a deadlock, multiple allocations held at a time from the same pool by a thread ought to be atomic or pipelined. Alternatively, the allocations could be spread across multiple pools in an established order.

9.2 Elimination of IORL

Another major improvement in 2.5 is the removal of the global I/O request lock present in 2.4. Instead, every queue is associated with a pointer to a lock, which is held during queuing. This enables per-queue locks or shared locks across queues depending on the level of concurrency supported by the underlying mid/driver layers. The SCSI mid-layer, for example, sets the lock pointer to the same per adapter value for all request queues associated with the devices connected to a given host adapter. Unlike our patched 2.4 SCSI mid-layer which serializes enqueuing per device, this locking scheme serializes at a coarser per adapter granularity.

A notion of command pre-building outside of the queue lock and ahead of request processing by the device has been considered for its potential to improve throughput and interrupt responses, but it has not been explored entirely. Choosing the right moment to prebuild is not trivial—done too early, it would require rebuilding on every subsequent merge; done too late, e.g. at the time of actually scheduling a request, it takes up cycles in request processing context which dilutes the desired effect.

9.3 Better per-queue tuning

Improved modularization at the generic block level now enables better per-queue level tuning and consideration of higher level attributes for I/O scheduler performance under specific configurations and workloads. There is support for efficient I/O barriers in cases where corresponding hardware support exists, which could be useful for transaction oriented I/O.

9.4 A job to do – utilizing the framework

At this point, work remains to be done in terms of modifying higher levels in the OS to make optimum use of this new infrastructure. Preliminary experiments running DM show that the 2.5.17 kernel outperforms SBIR for reads but does worse than SBIRV when readv is used (Table 3. This is consistent with the current state of implementation of the new block layer where the readv path has not seen the benefits of the bio structure. In fact, we can even expect a slight degradation for small I/Os because the memory vector structure is inherently a little more complex than the simple virtually mapped buffer in 2.4. For small single segment I/O the drivers end up with an added check for the end of the array, and many of the BIO fields become almost redundant.

Therefore, intelligent pre-merging at higher levels makes sense in this context. A 1:1 mapping between buffer heads and BIOs is not quite efficient. There is ongoing work to rewrite some of the filesystem interfaces to move in this direction. Andrew Morton's multi-page read and writeout patches [8] assemble large BIOs for pagecache pages (for as many corresponding blocks that are contiguous on disk) and submit them directly to the request layer, bypassing buffer heads altogether.

From the perspective of raw/direct I/O, which are the main areas of consideration in the current paper, the relatively heavyweight kiobuf infrastructure would have to be replaced by something like the lighter kvec data structures in Ben LaHaise's asynchronous I/O patches [6], which can support readv/writev operations efficiently.

A `kvec` is pretty close to a bare abstraction of a memory vector array of the form used in a BIO, each tuple of the vector being referred to as a kveclet. It is usually more useful to pass around a `kvec_cb` structure which refers to a `kvec` and its associated callback data for I/O completion purposes.

```
struct kveclet {
   struct page *page;
   unsigned     offset;
   unsigned     length;
}

struct kvec {
   unsigned        max_nr;
   unsigned        nr;
   struct kveclet veclet[0];
}

struct kvec_cb {
   struct kvec *vec;
   void        (*fn)(...);
   void        *data;
}
```

A `kvec` can be mapped to BIO structures for block I/O and similarly to equivalent `skb fragment` structures in the case of network I/O. A single kvec may be split across multiple BIO structures (each pointing to the corresponding section of the `kvec`), each of which acts as a distinct completion unit when more than one low level device requests are involved in serving the I/O. A large user space buffer (especially in the case of vectored I/O), might even be mapped to a big `kvec` a section at a time, and appropriately pipelined for I/O through multiple BIO requests to potentially enhance throughput and latencies for partial completions.

In the case of direct I/O, extents of non-contiguous blocks would have to be mapped to separate BIO units.

There also has been some discussion on the maximum size of BIOs that may be pushed down to the block layer, from the perspective of avoiding chopping up an I/O unless it violates the underlying device limits. Because this decision is more complex than just a matter of absolute size, and may even depend on request queue state, Linus Torvalds has suggested that drivers could supply a `grow_bio` helper function to handle this. Further complications arise in the case of layered drivers like lvm/md/evms. Andrew Morton has proposed a dynamic `get_max_bytes` interface exported by drivers (cascaded down layered drivers if required), to help build up appropriately sized BIOs to avoid splitting by the lower layers.

Observe that in 2.4 with fixed size (small) buffer heads, the approach was to never split a buffer, but include it as part of the request or create a new request depending on whether it could be fitted within the limits allowable for the device in question. In 2.5, the BIO represents larger variable sizes, having variable number of segments. Such a simplistic approach could result in underutilization of request slots when merging I/Os from different sources. If a buffer exceeds the request size which the device can handle, it breaks up the request. However, splitting up a BIO for a correct fit requires an additional memory allocation. Some points of caution with regard to such allocations at the block layer level have been discussed in an earlier subsection. This is why the question of constructing BIOs of right size arises.

A suitable solution would have to take into account that splitting is expected to be relatively infrequent. Since the general direction is to move towards merging early, `get_max_bytes()` could turn out to be a useful hint even for the corresponding clustering decisions. At the same time, it may not always be feasible or efficient in practice to absolutely guarantee elimination of the need to split I/Os. Thus, a provision for splitting may be required with due caution possibly with a structured use of multiple (layered) mempools and pipelined piecewise submissions to avoid deadlocks.

10 Conclusion and Future Work

In this paper we have highlighted some of the scalability and performance limitations of the 2.4 Linux kernel's block I/O subsystem. Using a decision-support benchmark that is representative of real-world enterprise workloads, we have shown that the 2.4.17 kernel sees I/O related performance bottlenecks when large I/O's are done on raw devices. We systematically investigated these bottlenecks and proposed solutions (as kernel patches) to alleviate them. As a result of using these patches, the decision-support workload sees an 233% improvement in its metric of interest. The benefits of these patches, all but one of which were written by the authors, are further demonstrated through a disk I/O microbenchmark and profiling data.

Most of the problems that we demonstrated are seen because of the use of the buffer head and kiobuf data structures. The new block I/O layer being written for the 2.5 kernel looks very promising as it addresses almost all the problems outlined here. Much work remains to be done to efficiently utilize the new data structures introduced in 2.5. We will continue to actively participate in the kernel community's efforts to improve the performance of both the 2.4 and 2.5 kernels for enterprise workloads.

11 Acknowledgments

We would like to thank the many people on the `lse-tech@lists.sourceforge.net` mailing list who provided us with valuable comments and suggestions during the development of these patches. In particular, we would like to thank Ruth Forester for helping resolve numerous issues with the decision-support workload and Helen Pang for collecting data on the disk I/O microbenchmark. We also appreciate the excellent DB2 performance analysis provided by John Tran, Karen Sullivan and James Cho.

This work was developed as part of the Linux Scalability Effort (LSE) on SourceForge (`sourceforge.net/projects/lse`). All the patches mentioned in this paper can be found in the "I/O Scalability Package" at the LSE site.

This work represents the view of the authors, and does not necessarily represent the view of IBM.

References

[1] Suparna Bhattacharya. Design Notes on Asynchronous I/O (aio) for Linux. http://lse.sourceforge.net/io/aionotes.txt.

[2] Suparna Bhattacharya. Notes on 2.5 Block I/O Layer Changes. http://lse.sourceforge.net/io/bionotes.txt.

[3] R. Bryant and J. Hawkes. Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, Oct 2000.

[4] John Hawkes et. al (Silicon Graphics Inc.). Kernprof. Available at http://oss.sgi.com/projects/kernprof /index.html.

[5] InfoWorld Test Center K. Railsback. Linux 2.4 breaks the enterprise barrier. http://www.infoworld.com/articles/tc/xml /01/01/15/010115tclinux.xml.

[6] Benjamin LaHaise. Kernel Asynchronous I/O Patches. http://www.kvack.org/˜blah/aio.

[7] Larry McVoy. The Splice I/O Model.

[8] Andrew Morton. Multi-page writeout and readahead patch. http://www.zip.com.au /˜akpm/linux/patches/2.5/2.5.8.

[9] Sharon Snider. I/O Performance HOWTO. http://www.tldp.org/HOWTO/IO-Perf-HOWTO/index.html.

Trademarks

A Comparative Study of Device Driver APIs Towards a Uniform Linux Approach

Wadih Zaatar and Iyad Ouaiss
Lebanese American University
Byblos, Lebanon
iyad.ouaiss@lau.edu.lb

Abstract

Linux Application Program Interfaces (APIs) lack stability and standardization. There is a need for a standard API for Linux device drivers that allow backward compatibility while easing the development of new drivers. The advantage of standardizing the API is to make the kernel core more robust and the development of new drivers easier; however the main challenge is performance-based. This work starts by carefully studying the available APIs for Linux as well as for other platforms. Current solutions studied include the Uniform Driver Interface (UDI), the Intelligent I/O architecture (I2O), WinDriver, and APIs implemented in Solaris, and Windows XP. By listing the strengths and weaknesses of available APIs, a proposal for a new Linux API is constructed that defines a standard interface, provides backward compatibility, ensures kernel security, and handles errors, uniform block sizes, buffering, etc.

1 Introduction

Device driver implementations have always been an important field of study in the world of operating systems from proprietary models of companies such as Sun and Microsoft to providers of open source technologies such as RedHat, Mandrake, Suse, essentially targeted towards the Linux platform or NewBus toward the Unix platform [OpSys] [LinDriv] [NTDriv] [DrivDes]. Other solutions proposed by some companies such as WinDriver, would allow you to get a shareware Graphical User Interface (GUI) to build your set of drivers with prewritten code to get you started. Recently, an interesting approach with a mixed hardware and software solution named I2O comes with another innovative concept that would be explained later in Section 3.1. Therefore, we can see that there exists a multitude of different solutions for the problem, each with its own set of advantages and disadvantages.

This paper does not pretend to give a final solution to the problem; it simply tries to classify all the currently available models into categories and proposes a draft of a work matrix for an improved device driver interface. But proposing a solution requires a small introduction on how device drivers operate in general and Linux in particular:

A device driver is essentially a kernel component, but is developed independently form the rest of the kernel. Therefore, there should be some kind of interfacing service between the driver and the host operating system. A standard implementation would have two interfaces: The first one would communicate with the hardware itself, namely the Driver-Hardware interface and the second one would take care of communicating with the operating system and of course the user, called the Driver-Kernel interface. This is where the problem really resides: Due to several causes (architecture, OS and hardware differences), it is really rare to find a piece of software to be binary portable between two different machines, and thus nearly impossible to have some device driver (which is after all just a small specialized program) to run on different computers, running different operating systems. This is where layering and abstraction come handy: by analyzing all common components of devices, one could come up with a standard, platform-independent API that would group all repetitive system calls. This solution has two main advantages: unify device driver implementation, which will lead to faster development and better code reuse, but also would guarantee on the long run platform independence; and, in the case of API upgrade would allow old device drivers to benefit from the new API.

The following sections describe the currently available API implementations, starting with proprietary API with Microsoft and Sun, independent implementations with WinDriver and UDI, and finally, a new approach towards solving the problem with the I2O architecture that encompasses both hardware and software components. The final section will recapitulate all advantages and disadvantages of every API implementation, giving a skeleton for a work matrix and proposing a primary set of steps that will guide the development of a Linux device driver API that meets the requirements set forth in this paper.

2 Review of Software Approaches

The following section describes the different API implementations for the most widely used operating systems, namely Microsoft Windows and Linux/Unix main implementations alongside some third-party commercial and public license developers.

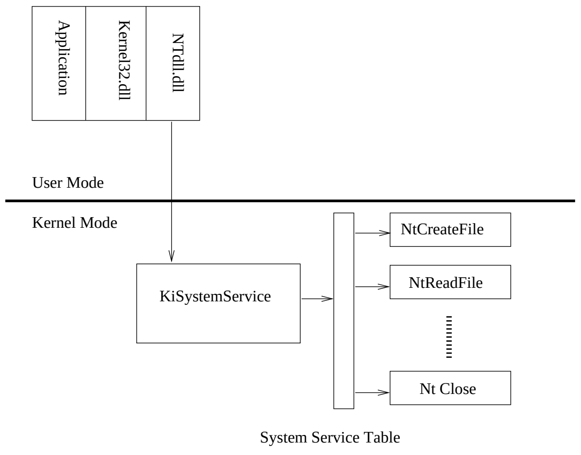


Figure 1: Windows API Model

2.1 Microsoft Windows API

Microsoft's device driver API [MSModel1] [MSModel2] were gradually enhanced with every new OS release, reaching a good level of stability due to two important factors: the first being software maturity starting from Windows 3.0 until the latest release of Windows XP based on NT technology, and second due to Microsoft's device driver compliance program that would take every device driver for any new hardware, run it and make sure that it is stable enough to be released with a "Certificate of Compliance" for it to be properly integrated with the latest OS release. This significantly reduced erratic OS crashes and restarts that made Windows platforms untrustworthy amongst the IT community. This model is depicted in Figure 1.

To be Microsoft-certified, a device driver has to have the following features:

- Handle I/O requests in standard format.

- Be object-based following the Windows model.
- Allows plug and play devices to be dynamically added or removed.
- Allow power management.
- Be configurable in terms of resources.
- Be multiprocessor code reentrant.
- Be portable across all Windows platforms.

From these major points, we can draw the following advantages:

- Windows drivers are portable between all platforms, as part of their requirements.
- Customizable as they are object based.
- Support new technology such as PnP and Power Management.

Are these really applied in reality? According to Microsoft themselves, their device model suffers the following:

- System instability: since they are run in kernel mode, and thus not isolated from one another or from the operating system, a failure in any device driver would result in system instability or a blue-screen.
- Little abstraction: the device driver interface is very low level and as such, there is little abstraction of the inner workings of the exported functions. This means that the device driver developer has to understand more about the workings to the interface than probably necessary.
- Plug and Play implementation: the PnP implementation is entirely set on the programmer's shoulders, requiring additional synchronizations and thus extra overhead.

2.2 Sun Solaris API

In System V Release 4 (SVR4), the interface between device drivers and the rest of the UNIX kernel has been standardized and completely documented [SunModel1] [SunModel2]. These interfaces are divided into the following subdivisions:

- The Device Driver Interface/Driver Kernel Interface (DDI/DKI) that includes architecture-independent interfaces supported on all implementations of System V Release 4 (SVR4).
- The Solaris DDI that includes architecture-independent interfaces specific to Solaris.
- The Solaris SPARC DDI that includes SPARC Instruction Set Architecture (ISA) interfaces specific to Solaris.
- The Solaris x86 DDI that includes x86 Instruction Set Architecture (ISA) interfaces specific to Solaris.
- The Device Kernel Interface (DKI) that includes DKI-only architecture-independent interfaces specific to SVR4. These interfaces may not be supported in future releases of System V.

The Solaris 2.x DDI/DKI allows platform-independent device drivers to be written for SunOS 5.x based machines. These drivers would allow third-party hardware and software to be more easily integrated into the OS. Furthermore, it is designed to be architecture independent and allow the same driver to work across a diverse set of machine architectures. The following main areas are addressed:

- Interrupt handling.

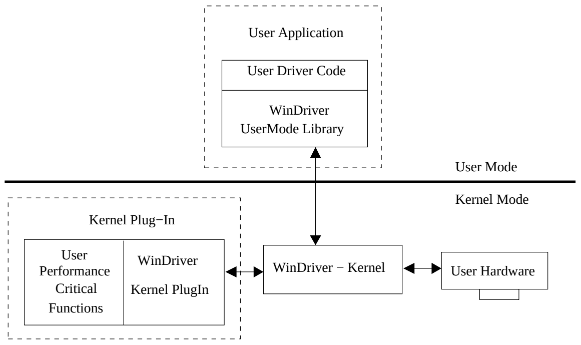


Figure 2: WinDriver Model

- Accessing the device space from the kernel or a user process (register mapping and memory mapping).
- Accessing kernel or user process space from the device (DMA services).
- Managing device properties.

2.3 WinDriver

The WinDriver API is a commercial, OS independent approach toward a common device driver API [WDModel]. WinDriver has the following important features:

- Source code compatibility between all supported operating systems: Windows, Linux, Solaris and VxWorks.
- Binary code compatibility between all Windows flavors (95, 98, Me, NT4, 2000, XP).
- Supports numerous architectures (PCI, E/ISA, and USB).
- Rapid device driver programming through the availability of many wizards. These wizards allow the device driver programmer, through a series of GUI-oriented steps to build the skeleton of the driver source code with initial procedure definition, global variables and program entry-points and basic calls.

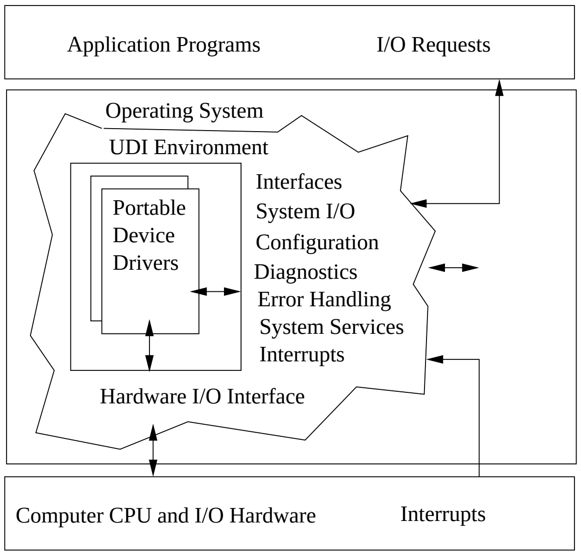


Figure 3: Uniform Device Driver Interface Model

From these features, we can deduce the following list of advantages:

- **C**ross platform compatibility and code reuse: No need for re-writing new drivers for the same hardware when porting to different architectures.
- **M**inimal performance hit: WinDriver offers a plug-in that would run performance critical parts right into the kernel, thereby achieving kernel mode performance.
- **E**asy drivers programming: WinDriver supports generic code generation in several programming languages, namely C/C++ and Delphi.

2.4 Uniform Device Driver Interface

The Uniform Device Driver Interface (UDI) is another initiative to create an architecture, platform, and OS independent solution for device drivers [UDIModel]; it is depicted in Figure 3. Similar to WinDriver, UDI has the following features:

- Platform neutrality, it abstracts all PIOs, DMAs, and interrupt handling through a set of interfaces that hide all architectures' variations.
- Drivers are written in ISO standard C and do not use any compiler specific extensions.
- UDI imposes shared memory restrictions, allowing system isolation of the driver code from the remainder of the OS, improving reliability and debuggability.
- Strict versioning allows evolution of the interfaces while preserving full binary compatibility of existing drivers.

The UDI device driver implementation is very much similar to WinDriver; therefore one could expect the following advantages:

- Cross platform compatibility, noting that UDI supports a narrower range of operating systems.
- Performance is comparable to native I/O drivers due to the fact that the code executes in kernel space, allowing minimal performance degradation.

3 Review of Mixed Approaches

The following section presents a new approach towards solving the device drivers' incompatibilities: By proposing a software and hardware components instead of a unique software approach.

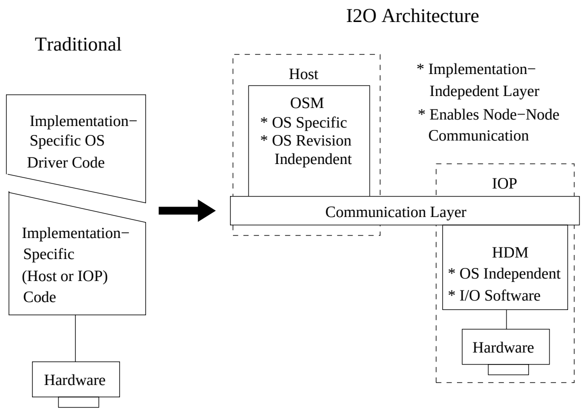

I2O Architecture
Traditional
Implementation−Specific OS Driver Code
Implementation−Specific (Host or IOP) Code
Hardware
Host
OSM
* OS Specific
* OS Revision Independent
* Implementation−Indepedent Layer
* Enables Node−Node Communication
IOP
Communication Layer
HDM
* OS Independent
* I/O Software
Hardware

Figure 4: I2O Architecture Model

3.1 The I2O Architecture

The I2O Architecture model is depicted in Figure 4.

Contrary to the standard implementation that relies on the host CPU to process all interrupt requests, the I2O eliminates processing bottlenecks by alleviating these tasks from the processor and taking care of them directly [I2OModel].

The I2O defines three software layers:

- The OS Services Module (OSM), that handles the communication between the host CPU operating system and device class.
- The I2O Messaging Layer that handles communication between the OSM and HDM in as standard way (see Figure). The standard I2O messaging layer does away the current requirement for OEMs to develop multiple drivers for the same device.
- The Hardware Device Module (HDM) handles the communication between the peripheral device and the I2O messaging

layer. The HDM is unique for each device. However, unlike a traditional device driver, only one HDM is required because it is independent of the host CPU operating system.

The I2O implementation offers the following improvements over traditional I/O processing:

- Standardized extensible architecture that is OS independent.
- Increased reliability and fault isolation to improve stability.
- Provides optimized I/O and system performance because the I/O process can be managed directly.

4 Analysis

Based on the previous sections, we can draw comparative measures that encompass all currently available solutions. The work matrix is shown in Table 1.

The following criteria were selected to distinguish between all available solutions:

- **Performance:** It is an essential component in any comparative study; however the measurement factor was much debated during the analysis of every solution. The number of interrupts per second that could be processed was finally selected. But due to the difficult nature of the task, this work is still taking an important amount of time and still not completed. One important note: The I2O group ceased operation and it was not possible to get access to any I2O based chips for testing.
- **Security:** This parameter describes the various security-related issues in every solution, memory-protection, and recoverability after a driver crash, etc. All four software solutions specify in their implementation the need for memory protection.
- **Compatibility:** Despite the Microsoft requirement that drivers should be upward compatible. Some tests performed on Microsoft Windows 2000 based drivers would erratically crash the system if installed on Windows XP, which would prove that these drivers still have compatibility issues. WinDriver and UDI drivers are still under test. The I2O alternative seems excellent on paper but for the same reason as described above, no tests were performed.
- **Portability:** WinDriver is the most versatile as it would run under most operating systems and is fully source compatible, a simple recompilation being enough. This would be ideal for multi-OS based solutions. UDI follows the lead with fewer but still promising implementations. The Windows and Solaris API models are proprietary and as such are not portable.

5 Conclusion

This paper represents the results of a data collection operation performed in order to analyze available device driver implementations for several operating systems. Many parameters still have to be exploited and carefully investigated. Most importantly, the performance factor that would gear towards choosing one solution over another is still being debated. With a better understanding of device driver APIs, the qualities and importance of each feature in the API and its role in the operating system can be analyzed. The next step in this research effort is to select a set of features that can be bundled together in order to form a compact, robust, and efficient Linux device driver API.

	Security	Compatibility	Portability	Notes
Windows API	Microsoft	Problems	Windows, Binary	Generic
Solaris API	Sun	Yes	SunOS Binary	Generic
Win Driver	To be tested	Yes	All, Source	Commercial
UDI	To be tested	Yes	Most, Source	Freeware
I2O	Hardware	Hardware	All, Independent	Discontinued

Table 1: Comparison Matrix

6 Acknowledgments

The authors would like to thank the students who participated in this work: Jamal Maalouf, Fady Matar, Naji Charbel, Francois Nader, and Elie Hajj.

References

[OpSys] Andrew S. Tanenbaum, *Modern Operating Systems*, 2nd Edition, Prentice Hall. (2001).

[LinDriv] Alessandro Rubim and Jonathan Corbet, *Linux Device Drivers*, 2nd Edition, OReilly. (2001).

[NTDriv] Edward N. Dekker and Joseph M. Newcomer, *Developing Windows NT 4.0 Device Drivers*, Addison Wesley Longman. (1998).

[DrivDes] *Introduction to Device Driver Design*. `http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=9103&scid=264`

[MSModel1] *Windows Driver Model: Compatible Drivers for Microsoft Windows Operating Systems*. `http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=20024&scid=273`

[MSModel2] *Windows Device Drivers Architecture*. `http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=13052&scid=273`

[SunModel1] *Porting to Solaris 2.X, Sun Whitepaper*. `http://sunsite.nstu.nsk.su/sun/inform/whitepapers.html`

[SunModel2] *An Engineering Tutorial on Porting your Device Driver to Solaris 2.0, Sun Whitepaper*. `http://sunsite.nstu.nsk.su/sun/inform/whitepapers.html`

[WDModel] *Jungo Ltd.* `http://www.jungo.com/windriver.html`

[UDIModel] *Uniform Driver Interface, Official Specification Documents* `http://www.project-UDI.org/specs.html`

[I2OModel] Pauline Shulman, *Overview of the I2O Architecture*. PC Developer Conference. (1998).

GConf: Manageable User Preferences

Havoc Pennington
Red Hat, Inc.
hp@redhat.com, http://pobox.com/˜hp

Abstract

GConf is a system for storing user preferences, being deployed as part of the GNOME 2.0 desktop. This paper discusses the benefits of the GConf system, the strengths and weaknesses of the current implementation, and plans for future enhancements.

1 Introduction

GConf started out very simply in late 1999, as my first project at Red Hat. It was a straightforward response to concrete problems encountered while creating the GNOME desktop, and designed to be implementable in a few months by a programmer with limited experience (i.e., by me). Much of the current implementation (let's call it Phase One) was created at that time.

GConf saw some limited use as part of the GNOME 1.4 platform—specifically in Nautilus and Galeon—but it wasn't widely adopted until recently, as part of GNOME 2.0. It has been quite successful as a GNOME 2.0 component, both as a labor-saving device for programmers and as a foundation for some of GNOME's UI enhancements.

This paper first presents an overview of Phase One from a conceptual standpoint; then discusses the Phase One implementation. Then it goes on to briefly describe some other systems for managing user preferences, such as IntelliMirror and ACAP. Finally, it presents some initial ideas for a Phase Two version of GConf. Phase Two is very much a work-in-progress.

2 Motivation

Several years ago it was becoming obvious that GNOME's initial approach to storing preferences (a simple API for writing key-value pairs to files) had a lot of limitations:

- It did not work for preferences that affected or were manipulated by multiple applications, because there was no way for applications to know a setting had changed. GNOME 1.x contains numerous hacks to work around this, and apps have to be restarted before certain preferences take effect.
- Administration features were missing, such as applying different defaults to different groups of users, or storing defaults for many machines on one network server.
- From a programmer standpoint, applications were usually designed with `load_prefs()` and `save_prefs()` routines, and had a number of bad hacks to update parts of the application when preferences were modified. To avoid this mess, a model-view design would be ideal.
- We wanted features such as "Undo" and

"Revert To Defaults" that were complicated to implement manually.

- We wanted a way to document each configuration option, and present the documentation in generic tools for managing preferences.
- To change the default setting for a preference, it was necessary to edit each code path that loaded the setting, and modify its fallback in the case that the setting was not present.

Some of these issues can be avoided by simple elaborations to the trivial file-based API, but others require a more elaborate solution. GConf set out to decide on the right thing to do and then do it.

3 Phase One: Design

The first principle of the GConf design was to keep it simple; it had to be implemented in only a short time, and had to be comprehensible to application developers.

The second principle of the design was to make GConf a system for storing end user preferences. Support for system configuration (e.g. Apache or networking) was explicitly excluded from the list of requirements. Support for storing arbitrary data was also excluded.

3.1 Key-Value Hierarchy

From an application point of view, GConf looks like a hierarchy of key-value pairs. Keys are named as UNIX-file-like paths. Here are the default settings for the GNOME CD player for example, listed using the `gconftool-2` utility:

```
$ gconftool-2 -R /apps/gnome-cd
theme-name = red-lcd
on-stop = 0
on-start = 0
device = /dev/cdrom
close-on-start = false
```

The key `/apps/gnome-cd/theme-name` has a string value `red-lcd`, the key `/apps/gnome-cd/close-on-start` has the boolean value `false`, and so forth.

Each key may be writable or not writable; applications are expected to disable the GUI for modifying a writable key. That is, if a setting is locked down and not available to a particular user, it should be made insensitive ("grayed out").

The API for interacting with the key hierarchy has a model-view design. That is, applications receive a notification message when a key has a new value; code in the application which *sets* a value need not have any relationship to code which *is affected by* the value.

The key hierarchy is process transparent; that is, a change to the hierarchy made by one process will be seen immediately by any other interested processes as well. This avoids ad hoc hacks for propagating settings changes across the desktop, and is an essential feature when writing single applications made up of multiple processes.

A "symlinks" feature for the GConf hierarchy has been suggested several times, but links (hard or symbolic) were deliberately left out of the design because they create implementation complexity. For example, notifying interested applications when a key changes value would require the ability to locate all symlinks pointing to the key that was modified.

3.2 Values

Values are limited to a fixed set of simple primitive types: integer, double, UTF-8 string, and boolean. Lists of each type are also allowed (list of integer, etc.). Recursive lists (lists of lists) are not allowed. Lists must be homogeneous in type. Phase One also supports pairs of primitive types (as in Lisp pairs, with `car` and `cdr`), but applications have universally ignored the pair type and in retrospect it was not a useful feature.

The limitations on value types caused a lot of controversy in the GNOME community. Some developers wanted the ability to push structs or other complex data structures into the configuration database, as a programming convenience. There are several reasons why GConf does not support this feature.

First, serialized structs are essentially binary blobs, a frequent complaint about the Windows Registry. All GConf data is human-readable. While a complex type system such as CORBA could be used to create a generic serializer, and a generic un-serializer, the process of manipulating an arbitrary, possibly-recursive serialized CORBA data type is much more complex than the process of manipulating strings and numbers. With the current GConf limitations, it's very easy to write scripts and tools that can handle any possible GConf value.

Second, many of the use-cases presented for storing arbitrary structs imagined using the GConf API for storing application data, rather than user preferences. For example, the GNOME 1.x configuration file API was used for ".desktop" files and the session manager save file in addition to user preferences. But in the requirements phase, GConf was limited to preferences only, so this was not a concern. Uses of the API that were storing preferences, rather than data, rarely benefited from anything beyond the primitive types.

Third, there are at least two simple workarounds if you need a "struct" type, which don't have the disadvantages of actually using a serialized struct: do the serialization yourself and store a string, or use a directory containing one key for each field in the struct.

Finally, the elaborate type system needed to handle serialization of arbitrary structs would mean either binding GConf tightly to a particular object/type system, or making up Yet Another Type System, not something anyone would like to see. This would limit our ability to "drain the swamp" (Jim Gettys's words); broadening the adoption of GConf Phase Two outside of the GNOME Project will require the client side library to be dead simple. If we had a widely-adopted object/type system, as with COM on Windows, using that object/type system to store complex types might make sense. But instead we have XPCOM, UNO, CORBA/Bonobo, GObject, QObject, ad infinitum, and no one is undertaking a serious effort to fix this problem.

I believe that the lack of a "struct serialization" API was a good decision on the whole. It places slightly more burden on application developers, but a good kind of burden; it forces them to describe their application's preferences in clear, human-readable terms. And it deters people from storing non-preferences data in GConf.

3.3 Schemas

Each key in the GConf hierarchy is associated with a *schema*. A schema contains metainformation about the key, including the following:

- The expected type of the key's value.
- A short one-sentence description of the key.

- A longer paragraph documenting the key and its possible values.
- The name of the application that provides the schema and uses the key.
- A default value to be used when the key is not set.

The documentation strings and default values can be localized.

The main schema-related design question was: where are they stored, and how are the default values located by applications at runtime? We went with the simplest solution: store the schemas in the configuration hierarchy itself. In addition to the normal primitive types (integer, string), GConf keys can store a schema value, containing the above metainformation.

Schemas are then associated with another key by name. That is, each key in the hierarchy may have the name of the key containing its schema value associated with it. By convention, schemas are stored under a "`/schemas`" toplevel, using names that parallel the main key hierarchy. So `/schemas/foo/bar` might be the schema name associated with `/foo/bar`.

When an application requests the value of `/foo/bar`, the GConf system first checks for a value at `/foo/bar`; if none is found, it checks whether a schema key is associated with `/foo/bar`; finding the schema key `/schemas/foo/bar`, it then looks up the value of `/schemas/foo/bar`. If `/schemas/foo/bar` stores a schema value, the default value is read from the schema value, and returned as the value of `/foo/bar`.

3.4 System Administrator's View

The system administrator sees a somewhat more complex picture of the GConf key/value hierarchy than the application does. To applications, GConf appears to be a single hierarchy of key-value pairs. Moreover, applications have no idea how the data in the hierarchy is stored.

From an administrator standpoint, GConf generates its hierarchy by merging a list of *configuration sources*. A configuration source is a concrete storage location, with three important attributes:

- a *backend*, i.e. which configuration source implementation should be used. For example, you might have a backend that uses XML files, or one that uses ACAP.
- *flags*, most importantly "read-only" or "read-write."
- the *address details*, for a file-based backend this might be the location to write files, for a network backend it might be the server's hostname.

Configuration sources are listed in a configuration file, `/etc/gconf/2/path`. Each time an application asks for the value of a key, GConf searches the configuration sources in order until it finds a value, or runs out of sources. When an application sets a new value, that value is stored in the first writable source. However, the set will fail if a non-writable source earlier in the search path sets the value already. This allows administrators to impose mandatorythree settings.

The default GConf search path has three elements: one read-only systemwide configuration source intended to contain mandatory settings, one read-write configuration source in the user's home directory intended to store changes made by the user, and finally one read-only configuration source at the end of the search path intended to contain both systemwide default values and schema values.

To impose a mandatory value, the administrator sets that value in the read-only source at the front of the search path; to provide a system default, the administrator sets the default in the read-only source at the end of the search path. Factory defaults are kept in the schemas, so are separate from site defaults.

The documentation strings found in schemas are intended to assist administration tools in presenting a reasonable interface for performing these kind of tasks.

4 Phase One: Implementation

Even with a fairly simple design, there were some challenging implementation issues that had to be addressed for Phase One. In summary:

- Process transparency; how to make all applications see the same hierarchy and deliver dynamic notification of changes?
- Caching; the GConf hierarchy contains a fairly large amount of data in total, it would be bad if each application had to maintain its own cache of this data.
- Storing the hierarchy; a filesystem-like data structure is fairly complex to store on disk, balancing efficiency, robustness, and so forth.
- Locking; you can't have two processes trying to modify the same file at the same time.

4.1 Per-Home-Directory CORBA Server

The implementation of Phase One is to have a small per-user daemon communicating with applications via CORBA. CORBA was used as a prebuilt IPC mechanism, for speed of development. The definition of "user" in per-user was "home directory"; the GConf daemon holds a lock in the user's home directory, and if the user logs in to two machines at the same time, the same daemon instance will be shared between those machines (one machine connects to the daemon on the other).

The per-user daemon has three functions:

- it serves as a global shared cache
- it serves as a global lock on the configuration data
- since it processes all changes to the data, it can notify applications of said changes

CORBA was not a great match for the networking needs of the GConf daemon. All communication between client and server needs to be nonblocking; this can only be achieved through the use of ORBit-specific nonstandard CORBA features. CORBA is pretty much overkill for this very simple IPC, and discourages the adoption of GConf outside of the GNOME Project.

Scoping the daemon per-home-directory wasn't the best decision either. One problem is that `fcntl()` file locking doesn't work so well; most Linux distributions are shipping buggy versions of nfs-utils that will leave stuck locks in some situations, and some combinations of NFS client OS and NFS server OS don't work quite correctly. Due to rumors of problems like this, the original GConf implementation tried to use a home-brew locking solution; but while that was portable, it was also non-working, and duplicate copies of the GConf daemon would often be created.

The per-home-directory solution was also unpopular because it involves remote TCP/IP connections (and open ports) between all machines where a user might log in using the same home directory. This turned out to be inappropriate for many sites, especially those using AFS. It also involves enabling TCP/IP for ORBit, meaning that a lot of programs are suddenly listening on open ports.

On a higher level, Owen Taylor pointed out that definining "per-user" as "per-home-directory" isn't correct anyway; a user may have multiple home directories, for example one on their laptop and one at work. You want preferences to be associated with a real world human user, not a specific login on a specific computer.

4.2 XML Backend

The GConf daemon dynamically loads backend modules that know how to read and write configuration data to some persistent storage location. Two backends were implemented for Phase One, an XML backend and a Berkeley DB backend. The DB backend was never enabled by default, and few users have tried it to my knowledge. It stores the GConf hierarchy in a single DB database file.

The XML backend stores a directory hierarchy on the filesystem that corresponds to the GConf hierarchy, and in each filesystem directory stores an XML file containing GConf values. The general idea was to split the hierarchy into multiple files (for robustness, and to avoid having to parse a singe huge file on startup, when relatively little of it might be used).

The XML backend's approach is a little bit messy, and inode-intensive. It's also surprisingly complicated to implement and has been a noticeable source of GConf bugs.

However, in practice the XML backend has worked reasonably well, now that it's been debugged; it is fairly scalable, assuming that a single directory never contains a huge amount of data. The problems of the XML backend seems difficult or impossible to avoid while using human-readable text files—the alternative design is to have one big file, which would require oceans of RAM to parse, and would put all the user's eggs in one basket.

5 Prior Art

GConf is hardly the first system for storing preferences ever invented. It's worth surveying some of the other notable systems alongside the Phase One design, to aid in planning Phase Two.

5.1 Windows 95 Registry

The much-maligned registry really isn't anything complicated; it's pretty much just a big hash table. Though at least one entire book has been written about it ([Petrusha]).

People like to compare GConf to the registry. While GConf also stores key-value pairs, it has little else in common with the registry:

- The registry stores systemwide configuration, GConf contains only user preferences.
- The registry typically contains binary data blobs, GConf goes out of its way to avoid those.
- GConf keys are documented and clearly named.
- The GConf design and application-visible semantics do not expose a specific format or location for the persistent data store.
- GConf provides mechanism for system/workgroup defaults and mandatory settings; the registry does not (but see the next section on IntelliMirror).

- The registry lacks change notification; if one application changes the registry, ad hoc hacks must be used to notify other applications. Or the user has to reboot.

(Some of the above information may no longer be accurate for newer versions of Windows; the above is based on Ron Petrusha's book.)

5.2 Windows 2000 IntelliMirror and Group Policy

While the registry is boring, IntelliMirror at least has good hype. Microsoft's white paper ([Microsoft]) on IntelliMirror cites three problems addressed by the feature:

- "User Data Management"
- "Software Installation and Maintenance"
- "User Settings Management"

"User Data Management" means that IntelliMirror keeps a copy of a user's documents in a central network location. When the user connects a machine to the network, their documents are copied to the local system. If the user edits a document while disconnected, the new document becomes the master copy and is synced to the network later. This means that documents are automatically backed up, and available on any machine the user logs on to.

"Software Installation and Maintenance" means that when a user logs on to a machine, the software appropriate for that user gets installed automatically. So users always have the same applications available in their menus.

"User Settings Management" means that user preferences (registry settings presumably) are automatically synced to/from network storage in the same way that documents and other data are synced. Also, certain settings may be locked down, and defaults may be established for particular groups of user.

These three features are most useful in combination, because providing all three allows users to easily move between machines, or switch to a new machine when their old machine breaks. Support for disconnected operation is a truly useful feature of IntelliMirror that can't be achieved using NFS on UNIX. If you take your laptop home, edit your preferences or a document, then reconnect it to the network at work, your changes will be automatically synced to network storage.

The IntelliMirror process is driven by directory services; Active Directory stores the information about a user, including what software they are supposed to have, where their data lives, and so forth. Policies can be established for particular workgroups and applied to all users in those groups.

GConf needs to be part of an IntelliMirror-style solution for Linux and UNIX operating systems, rather than an impediment to such a solution. Other existing components of the solution might include Red Hat's Kickstart, and filesystems such as InterMezzo. For situations where disconnected machines aren't important, NFS and AFS might also be useful solutions.

5.3 ACAP

RFC 2244 defines ACAP, or Application Configuration Access Protocol. It seems that ACAP never caught on, and is more or less dead; the only server implementation is written in ML, few if any applications support it, and the web page doesn't seem to have been updated in years. ACAP was an attempt to do almost exactly what GConf is supposed to do, however, and is worth looking at. The RFC describes ACAP's design goal thusly: "ACAP's primary purpose is to allow users access to their configuration data from multiple network-connected computers." ACAP is a text-based protocol similar to IMAP.

The ACAP RFC is somewhat vague, and there's not much in the way of implementation code to look at, so some of my descriptions of the system may be inaccurate.

Like GConf, ACAP defines a filesystem-like hierarchical namespace. However, the contents of a "directory" are (a lot) more complicated than they are in GConf. A GConf directory contains entries, each of which is a name and a primitive value. An ACAP "directory" contains a difficult-to-explain object called a "dataset." [Troll] tries to explain datasets, as the ACAP RFC itself isn't very clear. They are roughly speaking Yet Another Hash Table, but with some twists.

A dataset can inherit from another dataset. This feature allows system or workgroup defaults to be set up, much like GConf's configuration source search path. However, it's apparently necessary to configure inheritance on a per-dataset basis, which seems cumbersome.

ACAP datasets can have a "subdataset" associated with them; as far as I can tell, the RFC never explains the purpose or semantics of subdatasets.

ACAP has ACLs for settings, and quotas for individual users. ACLs allow settings to be made read-only, in order to impose mandatory settings. Quotas are useful for obvious reasons.

The concept of multiple users is visible to the ACAP client application. ACAP also exposes the inheritance chain (search path) for looking up values. GConf keeps this information purely in the configuration of the server; the client sees only a single hierarchy.

ACAP provides server-side sorting and searching, allowing clients to search through data stored in ACAP without downloading all the data.

Unlike GConf, the ACAP design goals include storage for "lightweight data," such as an address book, in addition to preferences. GConf is not intended to be used in this way; the assumption is that users will have somewhere to put documents and such anyway. By limiting GConf to preferences only, a GConf configuration source can realistically be locked down entirely (no writable locations), and the GConf design can assume that sending the value of a key over the wire is a fairly fast operation. The GConf implementation can be simple (does not need to scale to large data values), and the design need not support a rich type system.

ACAP does not have a standard concept like GConf's schemas, though arbitrary attributes can be added to a dataset. Keys in ACAP don't come with documentation as GConf keys do.

ACAP supports IMAP-like authentication (SASL), so works nicely with Kerberos and such.

The article *ACAP vs. Other Protocols* ([Wall2]) summarizes ACAP's design as follows:

> The key characteristics of ACAP are:
>
> - ACAP is designed to accommodate disconnected use
> - ACAP is designed to allows server data (and data structures) to be writable by user/clients
> - ACAP is designed to handle potentially (though not necessarily) large sets of data
> - ACAP is designed to allow granularity in access to data through an Access Control List mechanism

- ACAP is designed to allow per-user storage of information (accommodating problems of mobile, disconnected, and "kiosk"-model users)
- ACAP is designed to allow client definition of data fields, allowing user-side flexibility
- ACAP is designed with per-user security and authenticated operation states
- ACAP is structured to enable server-side searching.

GConf needs to be modified to support many of these design features. However, I would not advocate using ACAP as-is; ACAP doesn't seem quite right, it is not compatible with the current GConf client API, and because no one else uses ACAP there's little or no value to following an existing RFC. If a reasonable free ACAP server implementation existed (vs. an old one written in ML), it might be interesting to use as the backend for GConf Phase Two.

6 Phase Two

6.1 Design Changes

The application-visible GConf model has worked reasonably well so far. The model/view architecture, process transparency, filesystem-like hierarchical namespace, and so forth are straightforward yet powerful. The current architecture seems roughly correct in terms of balancing simplicity and feature set; much more complex, and people would not be able to use it correctly, much simpler, and it would not address all the problems that need addressing.

However, the current GConf *implementation* needs a lot of work to scale beyond single-user systems. Like ACAP, it should be client-server oriented. Like IntelliMirror, it should provide for disconnected operation. In general, the Phase One implementation's equation of "user" with "home directory" was wrong; users may have multiple machines, including a laptop.

6.2 Implementation Changes

Phase Two should be mostly invisible to applications, in fact it could conceivably be done without modifying the GConf ABI as shipped with GNOME 2.0. The changes will all be in the implementation.

6.2.1 Client-Server Architecture

In short summary, here is what I envision:

- The GConf daemon will become per-user-login rather than per-home-directory. (Side bonus: it can use a guaranteed-not-to-be-on-NFS directory for locking.) It will listen on a UNIX domain socket and communicate with a simple, fast custom protocol designed for "oneway" mode (avoiding round trips).
- The client side of GConf will become as trivial as possible; it will not use GLib or CORBA, just a tiny custom protocol. The current GConf client library will exist as a GNOME-friendly API but will wrap the more general API.
- More of the daemon's functionality will be moved to the loadable backends; in particular, change notification will come from the backend, instead of from the daemon.
- The default backend will know how to store data in two places; a local filesys-

tem cache, and a remote server. The local cache will be used when disconnected. On connection to the network, the local cache is fully synced with the remote server. While connected, it will sync on a regular basis. For standalone systems, the local cache is simply never synced. (The name "cache" is somewhat misleading, since the cache never expires.)

- The remote server used by the default backend has to be implemented. It will be a system daemon, and will serve requests from multiple users. It should support authentication via SASL (and thus Kerberos, etc.). The remote server is heavily inspired by ACAP.

The general goals are to move to a more secure client-server architecture, support disconnected operation, encourage adoption outside of the GNOME Project, increase robustness, and improve performance.

One outstanding question is how to improve on the current tree of XML files for storing the GConf database. The tree of XML files is robust, but complex to implement and fairly inefficient in terms of both space and speed.

6.2.2 Small Tweaks

In addition to the big-picture rearrangement, there are a number of smaller features to be creeped.

- Hints for interpreting values. This feature adds a simple descriptive string such as "keybinding" to a key, indicating that the key's value is to be interpreted in a particular standardized way. Generic configuration tools can then provide a nicer UI for editing the key.
- Atomic change sets. This feature allows a block of changes to be made as a unit, without exposing the intermediate state to applications.
- Prompting for authentication. This feature means that backends have a way to ask the user for passwords and other authentication information.
- Clean up the C API a bit. The API has various historical artifacts and just plain mistakes, and better ideas have appeared as it has become clearer how real applications use it. Most likely a new API will be introduced alongside the old, and both will be supported for a while.
- Convenience widgets for preferences dialogs. It's nice to have "data-aware widgets" for GConf, such as a text entry box associated with a string value in the GConf hierarchy.
- The server will probably need to enforce a space quota on individual users to avoid denial-of-service attacks.
- It might be nice to allow admins to define a separate configuration source search path for a particular subtree of the GConf hierarchy, instead of defining a single search path for the whole thing.
- Searching through data needs to be implemented on the server side, so that admin tools can efficiently provide a search function.

7 More Information

For more information about GConf, including an online copy of this paper (perhaps an updated version), see `http://www.gnome.org/projects/gconf/`. To report bugs, see `http://bugzilla.gnome.org`. To discuss GConf, join `gconf-list@gnome.org`.

8 Acknowledgments

Many people have contributed to GConf. All the hackers at Red Hat, especially Owen Taylor and Jonathan Blandford, have given valuable feedback and design suggestions. Red Hat deserves credit for supporting the initial development of GConf when I first arrived at "RHAD Labs" several years ago; Labs director Michael Fulbright gave me the freedom to work on GConf. Thanks are also due to Colm Smyth at Sun for his contributions of code and ideas, and to Sun Microsystems in general.

Dave Camp (now at Ximian) and Kjartan Maraas tie for the honor of being the first outside contributors to GConf, according to the ChangeLog. Since then many people have helped out, too many to list here. Thanks to everyone.

References

[ACAP] *RFC 2244* `http://asg.web.cmu.edu/rfc/rfc2244.html` (1997)

[Microsoft] Microsoft, Inc. White Paper, *Introduction to IntelliMirror Management Technologies*

[Petrusha] Ron Petrusha, *Inside the Windows 95 Registry*, O'Reilly and Associates. (1996)

[Troll] Ryan Troll, *ACAP Dataset Model* `http://www.ietf.org/proceedings/99nov/I-D/draft-ietf-acap-dataset-model-01.txt` (1999)

[Wall] Matthew Wall, *The Application Configuration Access Protocol and User Mobility on the Internet* `http://asg.web.cmu.edu/acap/white-papers/acap-white-paper.html` (1996)

[Wall2] Matthew Wall, *ACAP vs. Other Protocols* `http://asg.web.cmu.edu/acap/white-papers/acap-vs-others.html` (1996)

A Directory Index for Ext2

Daniel Phillips

Abstract

The native filesystem of Linux, Ext2, inherits its basic structure from Unix systems that were in widespread use at the time Linus Torvalds founded the Linux project more than ten years ago. Although Ext2 has been improved and extended in many ways over the years, it still shares an undesireable characteristic with those early Unix filesystems: each directory operation (create, open or delete) requires a linear search of an entire directory file. This results in a quadratically increasing cost of operating on all the files of a directory, as the number of files in the directory increases. The HTree directory indexing extension was designed and implemented by the author to address this issue, and has been shown in practice to perform very well. In addition, the HTree indexing method is backward and forward-compatible with existing Ext2 volumes and has a simple, compact implementation. The HTree index is persistent, meaning that not only mass directory operations but single, isolated operations are accelerated. These desirable properties suggest the the HTree index extension is likely to become part of the Ext2 code base during the Linux 2.5 development cycle. hotel & travel press photo archive audio key signing 2002 2001 2000 1999

1 Introduction

The motivation for the work reported in this paper is to provide Linux's native filesystem, Ext2, with directory indexing, one of the standard features of a modern filesystem that it has lacked to date. Without some form of directory indexing, there is a practical limit of a few thousand files per directory before quadratic performance characteristics become visible. To date, an applications such as a mail transfer agent that needs to manipulate large numbers of files has had to resort to some strategy such as structuring the set of files as a directory tree, where each directory contains no more files than Ext2 can handle efficiently. Needless to say, this is clumsy and inefficient.

The design of a directory indexing scheme to be retrofitted onto a filesystem in widespread production use presents some interesting and unique problems. First among these is the need to maintain backward compatibility with existing filesystems. If users are forced to reconstruct their partitions then much of the convenience is lost and they might as well consider putting in a little more work and adopting an entirely new filesystem. Perhaps more importantly, Ext2 has proved to be more reliable than any of new generation of filesystems available on Linux, in part due to its maturity, but also no doubt partly due to its simplicity. Ext2 has proved to be competitive with any of the new filesystems in manipulating the relatively small files and filesets (by today's standards) that are common on typical installations. Ext2's suite of filesystem utilities which includes e2fsck for performing filesystem checking and recovery, and debugfs for performing manual filesystem examination and repair, is particularly mature and capable. Finally, Ext2, being Linux's native filesystem, almost by definition is the filesystem that gives the broadest range of support for Linux's many features.

The simplicity of Ext2's design places a special burden on the designer of a directory indexing extension to strive for a similar degree of simplicity. The idea of adding BTree directory indexing to Linux's Ext2 filesystem has been much discussed but never implemented, more probably due to an aversion to complexity than any laziness on the part of developers. Unfortunately, a survey of implementations of directory indexing strategies in existing "big iron" filesystems shows that the directory indexing code by itself, is comparable in size to all of Ext2. So a simple port of such code would not achieve the desired results, and that would still leave open the question of how to make the new indexing facility backward compatible with existing Ext2 volumes.

In the event, with some luck, determination and expert advice, I was able to come up with a design that can be implemented in a few hundred lines of code and which offers not only superior performance with very large directories, but performance that is at least as good as traditional Ext2 with small directories, and not only backward compatibility with existing Ext2 volumes, but forward compatibility with pre-existing versions of Linux as well. In fact, the new design uses the same directory block format as Ext2 has traditionally used, and to earlier versions of Linux, an indexed directory appears to be a perfectly valid directory in every detail.

2 Background

The earliest versions of Linux used the Minix filesystem.[1] Recognizing its limitations, Remy Card designed and implemented the Extended Filesystem, BSD's UFS. That design was improved and became the Second Extended Filesystem, or Ext2.

Ext2's design is characterized by extreme simplicity, almost to a fault. For example, certain features that were planned for Ext2 were never implemented, such as fragment support modeled on UFS, and BTree directory indexing. Instead of adding features, Ext2 maintainers have tended to concentrate on making the existing features work better. Today Ext2 is well known for its high degree of stability, and ruggedness in the face of abuse. To some extent, the perhaps subconscious philosophy of minimalism can be credited for this. Nonetheless, as Linux evolves it encounters ever-rising expectations from its users, including those who have traditionally worked with "big iron" variants of Unix and tend to measure the worth of Linux by that yardstick. Pressure has increased to address those areas where Ext2 does not scale very well into enterprise-class applications.

Although there is a new crop of enterprise-class filesystems appearing in Linux this year—XFS and JFS, which were ported from SGI's Irix and IBM's OS/2 respectively—there exists a strong sentiment that Ext2 should retain the role of Linux's native filesystem, both for pragmatic reasons such as its feature set—by no accident well matched to Linux's requirements—and its stability. Perhaps there is also an element of pride, since if anything can be said to be the heart of Linux, it would be its filesystem, Ext2. Whatever the reason, motivation is strong to address the remaining weaknesses that separate Ext2 from a true enterprise-class filesystem. One of the most obvious is the lack of any kind of directory indexing: simple linear search is used, as it was in the earliest days of Unix.

For some common operations, a linear directory search imposes an $O(n^2)$ performance penalty where n is the number of files in a directory. With n in the thousands, the observed slowdown is very noticable. Typically, enterprise-class filesystems use some form of balanced tree indexing structure, which gives

$O(Log(n))$ performance for individual directory operations, or $O(nLog(n))$ across all the files of a directory. Ext2 was expected eventually to adopt such a scheme and in fact some provision was made for it in internal structures, but none was ever implemented. That this was never done should be ascribed to an aversion to complexity rather than any laziness on the part of Ext2 maintainers. After all, Ext2 in its entirety consists of about 5,000 lines of code and an implementation of a standard BTree algorithm could easily double that.

At the Linux Storage Management Workshop last year in Miami, Ted Ts'o described to me some design ideas he had for a simplified BTree implementation. At the time, there still seemed to be unresolved design issues that could lead to significant complexity, so implementation work was not begun at that time. Some months later while describing Ted's ideas in a posting to a mailing list, a fundamental simplification occurred to me, namely that we could use logical addressing for the BTree instead of physical, thus avoiding any change to Ext's existing metadata structure. Thus inspired, I went on to discard the BTree approach, and invented a new type of indexing structure whose characteristics lie somewhere between those of a tree and a hash table. Since I was unable to find anything similar mentioned in the literature I took the liberty of giving this structure the name HTree, a hash-keyed uniform-depth index tree.

After a week of intense development, and with the assistance of my coworker Uli Luckas, I managed to produce a prototype implementation with enough functionality to perform initial benchmarks. The measured results, which Uli prepared for me in the form of a chart, were described in a post to the Linux Kernel mailing list that same day.[8] Aided by the lopsided relationship between $O(n^2)$ and $O(nLog(n))$complexity, I was able to show a spectacular 145-times improvement[5] against standard Ext2, for a test case which I had of course chosen carefully. Whatever my bias, the performance improvements in practice were real and measurable. Suddenly Linux's venerable Ext2 no longer seemed to be on the verge of extinction in the face of competition from a new crop of enterprise-class filesystems.

The original prototype achieved its performance gains using a degenerate index tree consisting of only a single block. In the following months I carried on further development, incorporating many suggestions from Andreas Dilger, Ted and others, to finalize the disk format and bring my prototype to the a stage where it could be tested in live production testing. At this time, I, together with members of the Ext3 development team (Stephen Tweedie, Andrew Morton and Ted Ts'o) am preparing to incorporate this feature into Ext3, which task should be completed by the time this paper is published.

3 Data Structures and Algorithms

HTree Data Structure

A flag bit in a directory's inode indicates whether a directory index is indexed or not. If this bit is set then the first directory block is to be interpreted as the root of an HTree index.

Given the design goal that HTree-indexed directories appear to preexisting versions of Linux as normal, unindexed directory files, the structure of an HTree every directory block is dictated by the traditional Ext2 directory block format. If this were not the case, then a volume with directories created with HTree indexes would appear to have garbage in directories if mounted by an older version of Linux. Fortunately, it is possible to place an empty directory entry record in a block which is actually an HTree index constructed so that the entire block appears to be free when interpreted as an Ext2 directory block, yet only the first 8 bytes are actually used. This leaves the remainder of the block free for HTree-specific structures.

The root of an HTree index is the first block of a directory file. The leaves of an HTree are normal Ext2 directory blocks, referenced by the root or indirectly through intermediate HTree index blocks. References within the directory file are by means of logical block offsets within the file. The possibility of using direct physical pointers was considered for reasons of efficiency, but abandoned due to the onerous requirement of incorporating special handling for such pointers in many parts of Ext2 outside the directory handling code. Luckily, it turned out that with Linux's recent change to logical indexing of file data, logical block pointers are no less efficient than physical ones.

An HTree uses hashes of names as keys, rather than the names themselves. Each hash key references not an individual directory entry, but a range of entries that are stored within a single leaf block. An HTree first level index block contains an array of index entries, each consisting of a hash keys and a logical pointer to the indexed block. Each hash key is the lower bound of a all the hash values in a leaf block and the entries are arranged in ascending order or hash key. Both hash keys and logical pointers are 32-bit quantities. The lowest bit of a hash key is used to flag the possibility of a hash collision, leaving 31 bits available for the hash itself.

The HTree root index block also contains an array of index entries in the same format as a first level index block. The pointers refer to index blocks rather than leaf blocks and the hash keys give the lower bounds for the hash keys of the referenced index block. The HTree root index also contains a short header, providing information such as the depth of the HTree and information oriented towards checking the HTree index integrity and such other functions as specifying which of several possible hash functions was used to create the directory.

Each index entry requires 8 bytes, so allowing for a 32 byte header, a single 4K index block can index up to 508 leaf blocks. Assuming that each leaf block can hold about 200 entries and each leaf block is 75% full, a single index block can index about 75,000 names. A second index level is required only for very large directories, which can accommodate somewhat more than 30 million entries. A third level would increase capacity to over 11 billion entries. Such a large number of directory entries is unlikely to be needed in the near future, so the current implementation provides a maximum of two levels in the index trees.

Hash Probe

The first step in any indexed directory operation is to read the index root, the first block of the directory file. Then a number of tests are performed against the header of the index root block in an attempt to eliminate any possibility of a corrupted index causing a program fault in the operating system kernel. It is intended that the detection of any inconsistency would cause the directory operation to revert to a linear search. In this way the user is given the best possible chance to access data on a corrupted volume. This mechanism also allows for certain changes to be made to the index structure in the future; for example, more than two levels might be allowed in the tree, while still allowing earlier versions of the directory index code to access the directory. Should such an inconsistent index be detected an error flag is set in the filesystem superblock so that appropriate warnings can be issued and the directory index can be automatically rebuilt by the fsck utility.

Next the hash value of the target name is computed, and from that a determination is made of which leaf block to search. The desired leaf block is the one whose hash range includes the hash of the target entry name. Since index entries are of fixed size and maintained in sorted order, a binary search is used here. The format of an index entry is the same whether it references a leaf block or an interior index node, so this step is simply repeated if the index tree has more than one level.

As the hash probe descends through index entries an access chain is created, for use by the lookup and creation operations described below. Finally, the target block is read.

Entry Lookup

Once a target leaf block has been obtained lookup proceeds exactly as without an index, i.e., by linearly searching through the entries in the block. If the target name is not found then there is still a possibility that the target could be found in the following leaf block due to a hash collision. In this case, the parent index is examined to see if the hash value of the successor leaf block hash has its low bit set, indicating that a hash collision does exit. If set, then the hash value of the target string is compared for equality to the successor block's hash value, less the collision bit. If it is the same then the successor leaf block is read, the access chain updated, and the search is repeated.

If the leaf block happens to be referenced by the final index entry of an index block then the successor hash value is obtained from the parent index block, which has already been read. If the possibility of a collision with the target exists then the successor index block will be read as a prelude to reading the successor leaf block.

Although it sounds messy, the resolution of hash collisions described here is accomplished in just a few lines of code. Furthermore, it is very efficient to determine whether a hash collision with the target string could exist. Therefore, the common case where no collision exists can be checked for without any significant overhead. With a hash range of 31 bits collisions will occur very rarely even in large directories, and collisions that lie on either side of a block boundary will be rarer yet.

In summary, once the hash probe step has identified a leaf block to search, the target name will always be found in that block if it exists, except in the unlikely event its hash collides with that of an entry in a successor block. Typically, then, the number of blocks that need to be accessed to perform a lookup is two—the index and the leaf —or three, for extremely large directories. Because the index tree consists of a very small number of blocks, even for large directories, it is a practical certainty that they will all be retained in cache across multiple operations on the same directory.

Entry Creation

Except for the leaf splitting operation—described separately below—creation of entries is simpler than lookup. The target leaf block in which the entry will be created is located as for a lookup operation (and with exactly the same code) then, if there is sufficient space, the entry is created in that block as in unindexed Ext2. If the target block has insufficient space then the block is first split, as described below, and the entry is created in either the original block or the new block, according to its hash value.

Entry Deletion

Deletion of a directory entry is accomplished in exactly the same way as with no index: the entry is located via a lookup operation, and marked as free space, merging it with the preceding directory entry record if possible.

It is possible that, after a large number of deletions and creations in a directory, a significant amount of free space could accumulate in the blocks which are indexed by only a narrow range of hash values. In such a case, it is would be desirable to coalesce some adjacent blocks. So far, no form of directory entry coalescing has been implemented in Ext2 and this has caused few problems, if any. However, it is unknown at this time how prone the HTree algorithm is to fragmentation so it may turn out to be necessary to implement coalescing sooner rather than later, as opposed to relying on the fsck utility.

Splitting Leaves and Index Blocks

Splitting a leaf block is the most complex step in the HTree algorithm, accounting for somewhat more than half the lines of code in the implementation. Nonetheless, there is little here that is subtle or difficult.

Splitting a leaf block requires moving approximately half the contents of the original to a newly created block such that the original entries are partitioned into two ranges of hash values. The two ranges are distinct, except that the highest hash value of the lower range may be the same as the lowest hash value of the upper range. This exception is made necessary by the possibility of hash collisions.

Since entries are stored unsorted in leaf nodes, the first step of the partitioning is a sort. First, all the entries in the block are scanned and a hash value is computed for each one. The entry locations and hash values are stored in a map, and this map is sorted rather than the entries themselves. The sort (a combsort) executes in $O(nLog(n))$ time where n is the number of entries. It has been suggested to me that the partitioning could be done in $O(n)$ time, but this is true only if we are willing to pick an *a priori* pivot value for the partition. In any event, the splitting occurs relatively rarely—for 4K blocks, less than once per hundred creates—and the sort is efficient.

In the current implementation, leaf blocks are always split at the halfway point in terms of number of entries, which is not entirely optimal. At the expense of somewhat more complexity the split point could be chosen in terms of total data size and could take account of the knowledge of which of the two blocks a new entry will be inserted into. Currently, the lowest hash value in the upper range is always chosen as the lower bound of that range. An improved strategy would use the hash value which is exactly between the lower bounds of the two adjoining hash ranges whenever possible, dividing up the hash space more evenly. These two improvements would allow the theoretical 75% average fullness of leaf blocks to be more closely approached.

After choosing the split point the entries of the upper hash range are copied to the new block and the remaining entries are compacted in the original block. This is done with the aid of the sort map described above. Some complexity in the step arises from the desire to carry out this operation within the space of the two blocks involved plus a small amount of stack space, so no working storage needs to be allocated.

Having split the entries, a new index entry consisting of a pointer to and lower hash bound of the new leaf block is inserted into the parent index. This may require that the parent index block be split or, if the index block is the root, a new level is added to the tree. Since only two levels of index are supported the recursion does not go further than this in the current implementation.

The lowest bit of the hash value of the new leaf block is used to flag the relatively rare case where the split point has been chosen between two entries with the same hash value. This bit forms part of the new leafs hash value and is carried naturally through any recursive splitting of index blocks that is required. So, as far as entry creation is concerned, just two lines of code are required to handled the messy-sounding problem of hash collisions in entry creation. (A few more lines are required to handle collisions on lookup.) As a side note, I did manage to conceive and implement a much more complex and hard to verify solution to this hash collision problem, which attempted to avoid splitting apart entries with equal hash value. Then I realized that the rarity of the event meant that the simple approach could be used with no significant impact on performance. In general, it is impossible to guarantee that colliding entries will never have to be split between blocks, since we may be so unlucky as to have a large number of strings hash to the same value.

Splitting an index block is trivial compared to splitting a leaf. Half the index entries are copied to a newly allocated index block, the count fields of block blocks are updated, and an index entry is created for the new block in the same way as for a new leaf block. Adding a new tree level is also trivial: the entries contained in the root index are copied to a new block and replaced by an index entry for the new block. In the current implementation, should the root of a two level tree be found to be full then the index is deemed to be full and the create operation will fail. At this point the directory would contain several tens of millions of entries or the index would have become badly fragmented. Though neither possibility is considered likely, both can be addressed by generalizing the implementation to N levels, and the second could be corrected by adding an index-rebuilding capability to the fsck utility, or by implementing a coalesce-on-delete feature as described in the penultimate section of this paper.

After all necessary splitting and index updating has been completed, and appropriate working variables updated, a new directory entry is created in the appropriate leaf block in the same way as for unindexed Ext2.

4 Comparison to Alternatives

In this section I briefly examine three alternative directory index implementation techniques that offer similar functionality to HTrees. All three of these techniques have been used successfully in other filesystems, but each of them has some flaw that makes it less than perfect for Ext2's requirements and design philosophy.

BTrees

The BTree ("balanced tree") algorithm offers good average and worst case search times with reasonably efficient insertion and deletion algorithms. Some variation on the BTree structure is typically the choice for a directory indexing design, and indeed BTree indexing is used in at least a number of Linux's supported filesystems.

Linux's ReiserFS[9] uses B*Trees which offer a 1/3rd reduction in internal fragmentation in return for slightly more complicated insertions and deletion algorithms. Keys in ReiserFS BTrees are fixed-length hashes of the indexed strings, therefore duplicate keys are allowed to accommodate key collisions. and the B*Tree algorithms are modified accordingly. SGI's XFS uses B+Trees with hashed keys. IBM's JFS (now ported to Linux) uses B+Trees with full-length key strings in the leaf nodes and minimal prefixes of the keys in the interior nodes. This variant is called a Prefix BTree.

Though directory performance is seldom specifically tested, all three of these filesystems are known for their good performance with large directories. Two of these three filesystems use hash directory keys for the same reason HTree uses them: the small fixed key size gives a high branching factor and thus a shallow tree.

The main difference between an HTree and a BTree is that the leaves of an HTrees are all at the same depth. Leaf nodes do not have to be specially marked and rebalancing is unnecessary, saving considerable complexity. A second distinction is the an HTree has one index entry for each leaf block whereas a BTree has one index entry for each directory entry. This means that an HTree has far fewer index blocks than a BTree and is therefore roughly one level shallower than a BTree with the same number of entries. In fact, the high branching factor and block granularity together make it improbable that an HTree will ever need to have more than two index levels, which are sufficient to contain several tens of millions of entries.

As names are inserted into and deleted from a BTree it may happen that some of the leaf nodes end up significantly further from the root than others. If such imbalance becomes too extreme then average search times may begin to suffer. To combat this, BTree algorithms incorporate rebalancing steps that detect excessive imbalance resulting from an insert or delete operation and correct it by rearranging nodes of the tree. Such rebalancing algorithms can become complex in implementation, especially when hash key collisions or additional requirements of B+Trees and B*Trees need to be handled.

In summary, the various forms of BTrees have all the functionality required for a directory index, but because of the rebalancing algorithms, are more complex to implement than HTrees. No clear advantage is offered in return.

Hash Tables

A normal hash table is a linear array of buckets, and the hash key directly indexes the bucket which is to be searched. Thus, finding the correct bucket to search is very fast. The chief drawback is that the hash table's size be chosen to be neither too large nor too small. A hash table that is too large will waste space and one that is too small will cause many collisions. Filesystem directories tend to vary in size by many orders of magnitude, so choosing an appropriate size for a hash table is problematic. This problem can be solved by allowing the hash table to grow as the number of strings in the hash table increases. When the hash table passes a certain threshold of fullness its contents are transfered to a larger table, an operation called "rehashing." If an integral factor is used for the expansion then hash values do not need to be recomputed and the process is efficient.

A linear hash table with a rehashing operation is thus seems a promising avenue to explore for a directory index. The rehash operation is mildly unappealing for filesystem use—how to store the hash table—how to represent collisions—can't use pointers in the objects—interaction with collisions and rehashing . . .

Cached Index

The above-mentioned structural problems with hash tables can all circumvented neatly if the hash table is not persistent on disk but is instead constructed each time the directory is opened. This approach has been tried with good results by Ian Dowse[7].

Although the index can be maintained incrementally it must be initially constructed in its entirety so that the file creation operation can provide the necessary guarantee of uniqueness. This gives rise to two problems. First, starting with a cold cache the first access to any directory forces all blocks of the directory be read, even if only a single entry needs to be accessed. Thus, randomly accessing files in a large volume containing a large number of directories will be significantly slower than with a persistent index, until the cache has been fully initialized. This could visibly affect latency for an application such as a web server. Second, there is the requirement to cache hash tables for all directories accessed. It is not difficult to construct a case where a single file is accessed in each directory of a volume, cyclically. With a sufficiently large volume this must cause cache thrashing. Either of these problems could be exploited by a malicious user, and either could cause spikes of performance degradation on certain applications.

An advantage of the cached index approach is that the implementor is relieved of responsibility for maintaining structural compatibility of the index format across future revisions. A persistent index can be added at a later date, buying time to study and perfect design alternatives. The disadvantages of a cached index—cache thrashing and latency problems—do not affect the common cases unduly. Most users will be pleased with the afforded performance increase as compared to linear directory searching. However, if a persistent index design is available which performs well and does not have the disadvantages of a cached index, then it is hard to see why it should not be adopted.

In summary, the cached index approach is considered to be a worthwhile acceleration strategy, the value of which lies in providing performance enhancement for common cases over the short term.

5 Hashing

Hashing, in its specific application to directory indexing, is a subject to which an entire paper should be devoted. Here, I will merely touch on a few of the relevant details.

The most important goal of a hash function is to distribute the output values across the output range. Secondary goals are speed and compactness.

Uniformity of distribution is especially important to the HTree algorithm. A nonuniform hash function could lead to very uneven splitting of the hash key space, which could dramatically increase the danger of directory fragmentation. It should be noted that some directory index designers have sought to exploit nonrandomness in hash functions, with a view to improving cache coherency for operations applied across entire directories. It is my opinion that such a goal is difficult to attain and in any event imposes a needless burden on the user. A better strategy is design the directory operations to be essentially as efficient with a random hash as with a favorably chosen nonrandom hash.

As part of the development process of the HTree indexing code I examined many hash algorithms, with the assistance of a number of others. Surprisingly, I found most hash algorithms in common use to be flawed in fundamental ways. The most common flaw I found is a reliance on randomness in the input string for randomness of the resulting hash value. When tested with nonrandom input strings such as names varying only in their last few characters, such hash function tend to produce very poorly distributed results. Unfortunately, nonrandom strings are all too common in directory index applications.

As an *ad hoc* test of the effectiveness of various hash functions I implemented a small userspace program that creates a large number of directory entries with nonrandom names; specifically, names that are identical in their first N characters, with a linearly increasing counter appended. After creating the entries I computed statistics on the leaf nodes to determine how evenly filled they were. In general, only one statistic matters: average fullness. In theory, perfectly uniformly distributed hash values would result in all leaf nodes being 75% full. In practice, I have seen as high as 71% average fullness and as low as 50%, the worst possible result.

As a result of this testing process I made the following empirical observations:

- If any step in a hash algorithm loses information then a final "mixing" step that attempts to improve randomness cannot repair the damage, and the result will be a poor distribution. The hash algorithm must attempt to use all information in the input string as fairly as possible.
- Where the hash value is built using a byte at a time from the input string, it is desirable that each byte affect the full range of bits of the intermediate result.
- CRC32 produced relatively poorly distributed results, apparently by design: it is not supposed to produce uniformly distributed results, but to detect bursts of bit errors.
- The best performing algorithms where based on theoretically sound pseudorandom generators, where at each step the random value is combined with a portion of the input string and used as the seed value for the next step.

After a number of marginally succesful experiments I hit on the idea of using a linear shift feedback register to generate a pseudo-random sequence to combine progressively with the input string. At each step, a character is taken from the input string, multiplied by a relatively prime constant and *xor*ed with the current value of the pseudorandom sequence, which forms the seed for the next step. Whatever its theorectical basic, or lack thereof, this hash function produced very good and uniform results.

Later I invested some time surveying hash functions from the literature and around the web. None that I tested was able to outperform my early effort, to which I gave the name "dx_hack_hash". Interestingly, the hash function that came nearest in its ability to generate consistently uniform random results was obtained from the source code of Larry McVoy's BitKeeper source code management system. It too, is based on a pseudo-random number generator, although of a slightly different kind. The performance of various hash functions and associated theorectical basis needs to be investigated further.

The HTree index currently relies on the dx_hack_hash hash function. It is necessary to subject this apparently well performing hash function to rigorous testing before the Ext2 HTree directory extension enters production use, because, in a sense, the hash function really forms part of the ondisk data structure. So it must perform well right from the start.

To accomodate the possibility that the initially adopted hash function might prove to be inadequate in the long run, either because weak spots in its performance are discovered or a superior hash function is developed, a simple scheme was devised whereby a new hash function could be added to the HTree code at a later date, and the old one retained to be used with any directories originally created with that hash. The newly incorporated hash function would assigned an ID number, one higher than the hash function before it, and each directory created thereafter would have the ID of the new hash function recorded in its header, so that any subsequent accesses to the same directory would use the same hash function.

Security: Guarding Against Hash Attacks

Modern computer systems must be proof not only against failure in normal course of operation, but when manipulated by a determined attacker with malicious intent. Where a hash algorithm is being used, if the attacker knows the hash algorithm then they might be able to induce a system to create a large number of directory entries all hashing to the same value, thus forcing long linear searches in the directory operations. It is conceivable that an effective denial of service attack could be developed by this means. Or perhaps the attacker would be able to fragment an HTree directory intentionally by creating a series of names all hashing to a given value until the block splits, then deleting the names and repeating with a new series hashing to an immediately adjacent value.

It turned out to be possible to devise a method that prevents an attacker from predicting the hash values of strings, even if the attacker has access to the source code and knows the algorithm. This method relies on the hash algorithm having at least one variable parameter that can be randomly generated at the time the directory is first created. This generated parameter is stored in the root index block and used for every operation on that directory. Since the attacker cannot predict the value of the random parameter, they cannot carry out the attack.

It is open for consideration whether this level of paranoia is justified.

6 Further Work

Further work is planned in number of areas including coalescing of partially empty directory blocks, improvements to cache efficiency for directory traversal operations on very large directories.

Coalescing

Traditionally, Ext2 has never performed any kind of coalescing on partially empty directory blocks. On the other hand, Ext2 directories are seldom very large, in part due to its poor performance on large directories. The larger directories made practical by efficient indexing make the issue of coalescing more important.

Coalescing presents a problem in that it is an inherently nonlocal operation. It is not desirable to impose a requirement of examining several neighboring blocks at each deletion step to see if they can be coalesced. I felt that the operation could be made much more efficient by recording some information about the fullness of each leaf block, directly in the index. It would then be possible to test neighbor blocks for suitability for coalescing without having the leaf blocks themselves in memory. To be effective, just a few bits of descriptive information would be needed. At the same time, it is clear that 32 bits is a far larger range than will ever be required for the logical blocks of a directory. Accordingly, I set aside the top 8 bits to be used as hints to help accelerate directory block coalescing, should this feature be implemented.

For forward compability these high order bits are masked off by the current code. This means that, should a volume with indexes created by a later version of the indexing code be remounted by an earlier version that knows nothing about coalescing, the advisory bits will simply be ignored. The later code will have to accommodate the possibility that the advisory bits may be wrong, but since this is just an optimization that does not present a serious problem.

Coalescing applies to the hash bucket divisions as well as to the data of leaf blocks. In other words, if enough insertions and deletions were performed in a directory the hash space could be cut into many small fragments. In turn, leaf blocks corresponding to the small fragments of hash space would likely be underfilled. This could lead to significant growth in the size of a directory. In practice, this has not been observed. Accordingly, further work on coalescing has been deferred for the time being.

Cache Efficiency

Tests using a directory of one million files on a machine with 128 MB of memory showed that mass file deletion was slower than mass file creation by roughly a factor of four, whereas for smaller directories (below a few hundred thousand entries) deletion was roughly as fast as creation. After some investigation the difference was found to be due to the mismatch between the storage order of directory entries and inodes. Inodes are 128 byte records, packed 32 per 4K block. During creation, inode numbers tend to be allocated sequentially so that after each inode blocks fills completely it is never referenced again. On the other hand, mass deletion is performed in the order in which directory entries are stored in directory leaf blocks. This is random by design. Unfortunately, each delete operation requires not only that the directory entry be cleared but that the inode be marked as deleted as well. Thus, the inode table blocks are touched in random order. This does not present a problem if sufficient cache is available, but if that is not the case then sometimes a block with undeleted inodes will have to be written out to make room for some other block on which an inode is to be deleted. In other words, thrashing will result.

To confirm this theory I wrote a test program that would first read all the directory entries, sort them by inode number, then delete then in the sorted order. The thrashing effects disappeared. While this served to prove the the problem had been correctly identified, it is not a practical solution since we cannot in general control the order in which user programs will carry out mass deletion: it will normally be in the order that directory entries are retrieved via a readdir operation. The source of the problem thus identified, it became apparent that not only deletion, but any directory traversal involving inode operations would be affected.

Next I wished to establish the worst case performance impact of this type of thrashing. This is obtained when every single deletion requires two inode table block disk transfers, resulting in a 32 times increase in IO operations. This worst case result can only be approached if available cache memory is very small in relationship to directory size, which might in turn be due to competition for cache from parallel processes.

Inode table thrashing can be controlled by adding memory. For example, the inode table blocks for one million files will fit comfortably in cache on a machine with 256 MB of memory, assuming half the memory was available for caching. At worst, the slowdown observed is only linear, not at all comparable to the quadratic slowdown caused by linear directory searching. However, cache efficiency remains a desirable goal. I carried out preliminary analysis work suggesting it is possible to reduce the cache footprint logarithmically, by carefully controlling the allocation policy of inode numbers. This approach is attractive in that it does not require changes in the underlying directory index structure. It is considered practical to defer this work for the time being.

For completeness, I examined alternative index designs to determine whether there exists an equivalently good indexing strategy which is not susceptible to inode table thrashing for common operations. Such an index would necessarily record directory entries in the same order as the corresponding inodes. Recalling that the HTree design uses one index entry per block, this hypothetical design would multiply the number of index entries by the average number of directory entries per block, a factor of 200, conservatively. Since index entries are small, this is not as bad as it sounds. In effect, this would increase the size of the index to somewhat less than half the size of the leaf blocks, in total. Again this is not as bad as it sounds, because there would be no slack space in the leaf blocks. This would narrow the HTree method's size advantage to about 30 percent. However, the individual-index method imposes a new requirement: free space in each leaf block must be tracked. Some kind of persistent free space map would be needed and updating this map would require extra IO operations. The speed of fsck would be decreased measurably by the requirement to examine more index blocks. These size and performance disadvantages considered together leads to the conclusion that the fine-grained index approach's cache advantage in mass directory operations is not sufficient reason to prefer it over the method presented in this paper.

Nonlocal Splitting

A with HTrees, BTree blocks must be split as they become full. A B*Tree is a BTree variant that reduces the amount of unused space in split blocks by splitting groups of two blocks into three. This leaves each block approximately two thirds full, compared to half full in a normal BTree.

This same technique could be used with an HTree, and in fact the implementation is simpler because rebalancing is not required. The expectation would be to improve average block fullness from 3/4 to 5/6. It is for consideration whether this improvement warrants the additional complexity and slightly increased cost of the split operation.

7 Conclusions

The HTree—a uniform-depth hash-keyed tree—is a new kind of data structure that has been employed with apparent success to implement a directory index extension for Linux's Ext2 filesystem. Besides offering good performance and a simple implementation, the HTree structure allows Ext2's traditional directory file format to be retained, providing a high degree of both backward and forward compatibility. After a period of further refinement and testing, it is considered likely that the HTree directory indexing extension will become a standard part of Linux's Ext2 and Ext3 filesystems.

8 Acknowledgements

My heartfelt thanks to all those who helped make this work possible, especially:

innominate AG for employing me to help make Linux better

Uli Luckas for surviving many hours of code walkthroughs and debugging sessions

Andreas Dilger, for whom no detail was too unimportant to go uninvestigated

Mathew Wilcox, for being the first to proofread this paper

Stephen Tweedie for generally being encouraging, knowledgable and an all round nice guy

Ted Ts'o for putting me up to this in the first place

Anna just for being Anna

9 References

References

[1] Design and Implementation of the Second Extended Filesystem, `http://e2fsprogs.sourceforge.net/ext2intro.html`

[2] Analysis of the Ext2fs structure, `http://step.polymtl.ca/~ldd/ext2fs/ext2fs_toc.html`

[3] [rfc] Near-constant time directory index for Ext2, `http://search.luky.org/linux-kernel.2001/msg00117.html`

[4] [RFC] Ext2 Directory Index - File Structure `http://lwn.net/2001/0412/a/index-format.php3`

[5] HTree Performance: `http://nl.linux.org/~phillips/htree/performance.png`

[6] Journal File Systems, Juan I. Santos Florido, `http://www.linuxgazette.com/issue55/florido.html`

[7] BSD Dirhash: `http://groups.yahoo.com/group/freebsd-hackers/message/62664`

[8] XFS White Paper: `http://oss.sgi.com/projects/xfs/papers/xfs_usenix/index.html`

[9] ReiserFS Resources: `http://www.namesys.com/`

A Distributed Security Infrastructure for Carrier Class Linux Clusters

Makan Pourzandi, Ibrahim Haddad, Charles Levert
Miroslaw Zakrzewski, Michel Dagenais
Open Systems Lab, Ericsson Research Canada
8400 Décarie Blvd, Town of Mount-Royal (QC) Canada H4P 2N2
Makan.Pourzandi@ericsson.ca, Ibrahim.Haddad@ericsson.com, Charles.Levert@ericsson.ca
Miroslaw.Zakrzewski@ericsson.ca, Michel.Dagenais@polymtl.ca

Abstract

Traditionally, the telecom industry has used clusters to meet its carrier-class requirements of high availability, reliability, and scalability, while relying on cost-effective hardware and software. Efficient cluster security is now an essential requirement and has not yet been addressed in a coherent fashion on clustered systems. This paper presents an approach for distributed security architecture that supports advanced security mechanisms for current and future security needs, targeted for carrier-class application servers running on clustered systems.

Keywords: Linux, Security, Carrier Class Clusters, Distributed Infrastructure, IPSec, LSM.

1 Introduction

The interest in clustering from the telecommunication industry originates from the fact that clusters address carrier-class characteristics such as guaranteed service availability, reliability, and scaled performance, using cost-effective hardware and software. There are several efforts on going to use Linux as basic block for building next generation telecom clusters [12, 7]. These carrier-class characteristics have evolved and now include requirements for advanced levels of security. However, there are few efforts to build a coherent distributed framework to provide advanced security levels in clustered systems.

Our work targets implementing security mechanisms for soft real-time distributed carrier-grade applications running on large-scale Linux clusters. These clusters are dedicated to run only one application. They must provide five nines availability (99.999% uptime) that includes hardware upgrade and maintenance and operating system and applications upgrades. In such clusters, software and hardware configurations are under the tight control of administrators. The communications between the nodes inside the cluster and to external computers are restricted.

In this paper, we present the rationale behind developing a new architecture, named Distributed Security Infrastructure (DSI). We describe the main elements of this architecture, and discuss our preliminary results. DSI supports different security mechanisms to address the needs for telecom application servers running on clustered systems. DSI provides applications running on clustered systems with distributed mechanisms for access control, authentication, integrity of communications, and auditing.

The paper is organized as follows: Section 2 illustrates the need for a new approach to security requirements for carrier-class clustered servers. Sections 3 and 4 discuss the DSI architecture and its characteristics. Sections 5 to 12 present the main elements of the design. Section 13 compares our approach to other related work. Section 14 presents some preliminary results. Section 15 concludes with our ongoing work and future plans.

2 The need for a new approach

There exist many security solutions for Linux clustered servers ranging from external to cluster solutions, such as firewalls, to internal solutions such as integrity checking software. However, there is no solution dedicated for clusters. The most commonly used security approach is to package several existing solutions. Nevertheless, the integration and management of these different packages is very complex, and often results in the absence of interoperability between different security mechanisms. Additional difficulties are also raised when integrating these many packages, such as the ease of system maintenance and upgrade, and the difficulty of keeping up with numerous security patches and upgrades.

Carrier-class clusters have very tight restrictions on performance and response time. Therefore, much pressure is put on the system designer while designing security solutions. In fact, many security solutions cannot be used due to their high resource consumption.

Currently implemented security mechanisms are based on user privileges and do not support authentication and authorization checks for interactions between two processes belonging to the same user on different processors. However, for carrier-class applications, there are only a few users running the same application for a long period without any interruption. Applying the above concept will grant the same security privileges to all processes created on different nodes for a long period of time. This is due to the fact that the granularity of the basic entity for the above security control is the user. For carrier-class applications, some classes of actions require fine-grained access control to some resources, or enforcement of specific security policies, or both. Therefore, the user-based granularity is not sufficient. By consequence, DSI is based on a more fine-grained basic entity: the individual process.

3 DSI characteristics

As part of a carrier-class clusters, DSI must comply with carrier-class requirements such as reliability, scalability, and high availability. Furthermore, DSI to answer the needs explained in 2 supports the following requirements:

- Coherent framework: Security must be coherent through different layers of heterogeneous hardware, applications, middleware, operating systems, and networking technologies. All mechanisms must fit together to prevent any exploitable security gap in the system. Therefore, DSI aims at integrating together different security solutions and adapting them to soft real-time applications.
- Process level approach: DSI is based on a fine-grained basic entity: the individual process.
- Maximum performance: The introduction of security features must not impose high

performance penalties. Performance can be expected to degrade slightly during the first establishment of a security context; however, the impact on subsequent accesses must be negligible. It is possible to disable security by security administration decision.

- Pre-emptive security: Any changes in the security context will be reflected immediately on the running security services. Whenever the security context of a subject changes, the system will re-evaluate its current use of resources against this new security context.

- Dynamic security policy: It must be possible to support runtime changes in the distributed security policy. Carrier-class server nodes must provide continuous and long-term availability and thus it is impossible to interrupt the service to enforce a new security policy.

- Transparent key management: Cryptographic keys are generated in order to secure connections. This results in numerous keys that must be securely stored and managed.

- Framework supports fast detection and reaction to security incidents.

4 Architecture

DSI targets clusters and, in doing so, introduces original contributions to their security. Some of its parts, however, such as its Access Control Service and its use of security contexts and identifiers, owe much to existing propositions, such as Security Enhanced (SE) Linux [5].

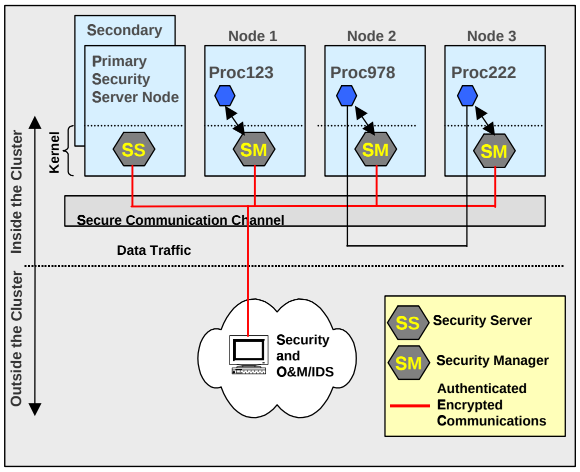


Figure 1: Distributed Architecture of DSI

4.1 Distributed architecture: Inside the cluster

DSI has two types of components: the management components and security service components. DSI management components define a thin layer of components that includes a security server, security managers, and a security communication channel (Figure 1). The service layer is a flexible layer, which can be modified or updated through adding, replacing, or removing services according to the needs of the cluster.

The security server is the central point of management in DSI, the entry point for secure operation and management, and alarms coming from the intrusion detection systems from outside the cluster. It is the central security authority for all the security components in the system. It is responsible for the distributed security policy. It also defines the dynamic security environment of the whole cluster by broadcasting changes in the distributed security policy to all security managers.

Security managers enforce security at each node of the cluster. They are responsible for locally enforcing changes in the security environment. Security managers only exchange

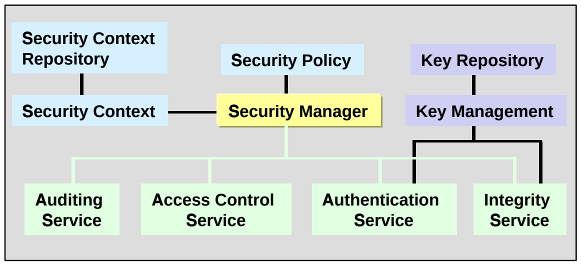


Figure 2: DSI Services

security information with the security server. The secure communication channel provides encrypted and authenticated communications between the security server and the security managers. All communications between the security server and the outside of the cluster take place through the secure communication channel. To avoid a single point of failure, the security services run on an equally hardened secondary security server as hot swappable services. These nodes are security hardened versions of Linux distributions to maximize security. All connections from and to these nodes are encrypted and authenticated.

The security mechanisms are on widely known, proved, and tested algorithms.

For the security mechanisms to be effective, users must not be able to bypass them. Hence, the best place to enforce security is at the kernel level; all security decisions, when necessary, are implemented at kernel level through DSI Security Module (DSM) [9]. This module is loaded on each node by the security manager upon its initialization.

4.2 Service based approach

The DSI architecture at each node is based on a set of loosely coupled services (Figure 2).

The security manager controls different security services on the node. This service-based architecture has the following advantages:

- The service implementation is separated from the rest of the system. By keeping the same API, the service implementation can be changed without affecting the application. However, an API for accessing security services is provided at user level for applications with special security needs (Section 10.1).
- It runs only predefined services according to the needs, performance issues, or security environment. In addition, services can be replaced on run time without major drawback on the running application. This enables the architecture to be modified and to resist changes throughout the system's lifetime.
- It is possible to add, remove, or update different services without administrative intervention. This reduces configuration errors due to the numerous security patches that need to be applied manually.

The security manager discovers the different services. Each service, upon its creation, sends a presence announcement to the local security manager, which registers these services and provides their access mechanisms to the internal modules. There are two types of services: security services (access control, authentication, integration, auditing) and security service providers that provide services to security managers.

The security management is implemented at all levels of DSI. There is a complete chain of commands from security administrators (human beings) of the cluster to different DSI components inside each nodes kernel. The administartors access the Security Server through the SCC. The Security server interprets the commands and propagate them to the Security Managers. Security managers translate these to control settings for different security services.

For example, security administrator detecting a back door on some software used indicates that some external IP can not accessed from cluster nodes. This is sent to the Security Server and translated as modification in DSP forbidding all connection to defined IP address. Propagated through SCC to security managers, this policy decision is enhanced at DSM level.

5 Security Server

The security server is the reference for all security managers and has the authority to declare a node as compromised. It subscribes to all updates to keep its cache of different security contexts up to date, which makes it the ideal candidate for running Intrusion Detection Systems (IDSs). It has a local certification authority[1] (CA). This last issues the certificates for secondary certification authorities run by the security managers. The primary tasks for security server include auditing, triggering alarms and warnings to inside and outside the cluster, managing the distributed security policy, receiving, interpreting and propagating security management operations to security managers.

6 Security Manager

The security manager enforces security on each node. It is primarily a lookup service to register different security services and service providers and connect them together. The security manager is instantiated at boot time with digital signatures to make certain that it is not replaced with a malicious security manager. Upon its creation, it joins the DSI framework and exchanges keys with the security server. Each security manager must publish any change to the security contexts of its local entities involved with remote entities and subscribe to changes in the security contexts of remote, related entities (see Section 8). The primary tasks for security managers include access control, process authentication, audit management, alarm publication, key management, as well as maintenance, and update of the locally stored distributed security policy.

[1]We mean that the CA is used for using inside the cluster.

7 Secure Communication Channel (SCC)

The secure communication channel provides secure communications for the security components inside and outside the cluster. Within the cluster, it provides with authenticated and encrypted communications among security components (Figure 3). It supports priority queuing to send and receive out-of-band alarms. It is coupled to the security manager by an event dispatching mechanism. For large-scale clusters, an event driven approach based on subscription to events from defined channels reduces the system load compared to the polling mechanisms. Further more, the benefits of this approach are:

- It does not present a single point of failure.
- It gives the possibility of event filtering, therefore less bandwidth used, and less time used for treating irrelevant information before discarding it.

The secure communication channel provides channels for alarms and warnings, security management, service discovery, and distribution of the security policy. It also provides a portability layer to avoid dependency on the low-level communication mechanisms.

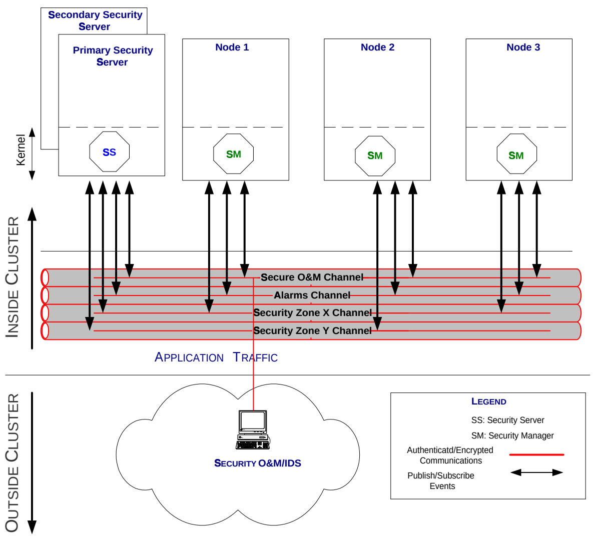


Figure 3: SCC is based on an event-driven logic and different channels

8 Security Context

For efficiency, a security identifier (SID) is defined as an integer that corresponds to a security context. All entities in the cluster have a SID. This SID is added at kernel level and cannot be tampered by users. For example, a structure containing SID is added to the structure presenting the process in kernel [9].

We define Cluster SID (CSID) as the pair of SID associated to the subject and the node where the subject belongs to. CSID is transferred across processors by security managers and interpreted through the whole cluster. Once the security context for a subject is needed outside of the local processor (for instance if this process accesses a remote object), its CSID is sent to the security manager of the node containing the object. The CSID propagation inside the cluster is based on SelOpt open source software implementation [8]. To avoid retransmissions, security managers rely on caching mechanisms.

To ensure the pre-emptive access control, the security manager of the node containing object subscribes through SCC to the event of a possible change in the security context of the access initiator entity.

9 A Coherent Vision: Security Contexts and the Distributed Security Policy (DSP)

Security configuration must be kept simple. Following this approach, DSI relies on a centralized security policy stored and managed on the security server. However, to maintain the cluster's scalability, read-only copies of the policy are pushed from the security server to the individual security managers through the SCC. This Distributed Security Policy (DSP) is an explicit set of rules that governs the configurable behavior of DSI. Each node, at secure boot time, relies on a minimal security policy that is either stored in flash memory or downloaded along with its digital signature. As soon as the DSP becomes available on a node, it prevails.

DSP allows a configurable behavior for security services. The DSI administrator (a human being) manipulates the primary copy of the DSP that resides on the security server. Thus, it must be represented in a human readable format. The basic update mechanism for DSP is to push a full copy of each new version of the policy through the SCC. However, given the mere size that the policy can take, an incremental update mechanism will be made available.

There can be several possible originating sources for the security policy rules. Manual configuration by the DSI administrator allows the most flexibility, but it rapidly becomes cumbersome. Thus, default policy rules are inferred from the very nature of the various software packages that are installed and running on the system. These default rules codify good security practices. The DSP should only need to be updated because of events such as the installation of new software components, but it should not be updated whenever ordinary recurring events occur. A security session manager handles this kind of events by updating the security context repository. A security context defines privileges associated with each entity. It is defined uniquely through the whole cluster, but it is the responsibility of the security manager who created it.

10 Access Control Service (ACS)

Access control can be defined as the prevention of unauthorized use of a resource [4]. It relies on the notions of subject (or access request initiator), object (or target), environment, decision, and enforcement. The Access Control Service (ACS) assumes that the subjects have been properly authenticated (see Section 11). DSI allows verifying the access control privileges even when subjects and objects are located on different nodes in the cluster. In order to simplify, we handle the access control in two levels: local when subject and object are on the same node and remote when they are on different nodes.

The local access control at each node is based on SID added to the structure in the kernel which, represents each entity (e.g., process, socket...). For local access control, the access rights are the functions of the security IDs of the subject (SSID) and the object (TSID). They are enhanced through DSI Secure Module, which we implemented [9].

10.1 Remote access control

For remote access control, we extend the local access control mechanisms by adding a new parameter: the security node ID. Therefore, the access rights are no more just the functions of the subject and target security IDs, but as well, the function of the security node ID (NSID). The SSID along with the NSID are sent to the node containing the object added to each IP packet as IP options (Figure 4).

A first level of access control based on SSID is done at this level, by the security manager on the source node. This is completely transparent to the process initiating the communication.

The second level of security check is done by the security manager on the target node based on SSID and NSID. This is also transparent to the process receiving the communication. Till this point, the access control decision is transparent. It is granted or denied based on SSID and NSID (i.e., CSID). This level of access control is enough for majority of the applications.

For applications with needs for finer grained access control, DSI provides an API allowing to take into account the SSID and NSID when making access decision to resource on target node.

We believe that it is not possible to implement an enough flexible and usable fine grained access control based only on platform. For fine grained security the application needs to collaborate with the security mechanisms provided by the platform. The application through DSI API asks to be set a new SID based on the SSID and NSID. Notice that the SSID and NSID are not revealed to the application. The application asks to change the SID based on the communication mechanism. For example, for a server process it means to pass the socket as an argument to DSI API. The security manager sets the new SID for the server process based on the original SID of the server (TPSID), SSID and NSID contained in each IP packet.

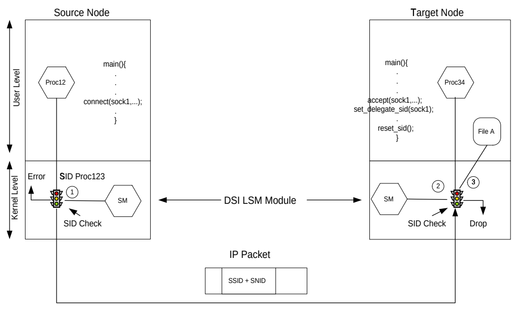


Figure 4: **Secure Remote access control:** Following security checks are done: 1) local check by the security manager (SM), if access agreed the Node ID (NSID) and Process SID (SSID) are added to each IP packet sent to the node 2) Application transparent check by the SM based on NSID and SSID, if access agreed the connection is established 3) For some applications with special security needs, the application can choose to further enhance the security by taking into account the SSID and NSID. To do this, the application needs to use DSI API (i.e., *set_delegate_sid)*.

Therefore the final access control decision is based on SSID, NSID, TPSID, and TSID.

Remark that DSI targets the carrier class platforms, with software environment under tight control with few applications running on the cluster. Therefore, even if the support of an additional API for security is a burden for application developers, we believe that only a small percentage of applications need to be modified. As for the majority of applications, transparent support of grant/deny based on SSID and NSID provided by the security managers is enough.

10.2 ACS architecture

The ACS that runs on the cluster's processors is comprised of two parts:

- A kernel-space part: This part is responsible for implementing both the enforcement and the decision-making tasks of access control. These two responsibilities are separated, as advocated by [3]. The kernel-space part maintains an internal representation of the information upon which it bases its decisions. This part is implemented as a Linux Security Module (LSM): DSI Security Module (DSM) [6].
- A user-space part: This part has many responsibilities. It takes the information from the Distributed Security Policy and from the Security Context Repository, combines them together, and feeds them to the DSM in an easily usable form. It also takes care of propagating back alarms from the kernel space part to the security manger, which will feed them to the Auditing and Logging Service and if necessary propagate to the security server through SCC.

Both parts are started and monitored by the local Security Manager (SM). The SM also introduces them to other services and subsystems of the infrastructure with which they need to interact.

10.3 ACS principles of operation

The ACS aims to provide fine-grained access control (at a sub-system call level). It respects the minimization principles of least privilege to limit the propagation and damage caused by eventual security breaches. As such, it provides defense in depth. The ACS that is running on a processor must make as little assumptions as possible about other processors, including whether they have been compromised. For that reason, an ACS instance is always the one making access decisions about resources that are local to its processor. For the initial design of the ACS, only grant/deny decision will be considered. Other more involved decisions would involve rate limiting and total usage limiting. Actions other than access control decision, such as interposition and active reactions, are not implemented either.

11 Authentication and communication integrity services

The authentication standard for now is the authentication by assertion. It means that the program accessing resources on remote processors asserts that it does this in behalf of a user. Neither the user schema nor the assertion only can be trusted seriously in an environment exposed to external attacks.

Local authentication in DSI is based on local verification by the DSM of each subject at node level.

The remote authentication of a process is the result of the local authentication of the process at the source node by DSM and the authentication of the node containing the subject to the target node.

IPSec is used for authenticating each node inside the cluster.

Developing for carrier class clusters, we have strong constraints on performance. IPSec has the advantage of covering both TCP and UDP[2]. To avoid applying the same policy to all IP traffic between two nodes (in particular, to avoid encrypting all data between two nodes), three IP addresses corresponding to three different subnetworks are assigned to each node. Each subnetwork defines a security policy: No security, authenticated only (IPSec AH mode), authenticated and encrypted (IPSec ESP mode). Filtering rules are further more used at networks elements (switches…) and at network interfaces of each node to enhance further the security rules.

The security manager at each node, based on SID of the sending process and the target node ID, and the target SID[3] according to the DSP transparently defines the security policy (e.g., subnetwork) to use: No Security, AH, or ESP.

Integrity and confidentiality of communications between two nodes is supported by use of IPSec ESP mode when necessary.

FreeS/WAN IPSec implementation has been used between nodes [2], however the support for IPSec AH mode is an issue. FreeS/WAN uses opportunistic encryption, which means that ESP mode is used even when AH mode is asked for. somehow in some cases for performance issues, it is though preferable to support AH mode without encryption load. At the end, we hope that the support for certificates will integrate the FreeS/WAN and will be not only supported as a patch.

12 Auditing Service

The auditing services are responsible for monitoring and auditing data and reporting security related information. This information may be used for several different purposes: intrusion and denial of service attacks detections, providing evidence in case of litigation or inquiries.

Auditing service for each node is responsible for analyzing the logs, detect the possible attack patterns, trigger the alarms, and propagate them through SCC. This service is responsible for functionality related to the lawful intercept.

This service has increase functionality on security server. It also monitors the internal net-

[2]The necessity of support of efficient security protocol for UDP is one of the main reasons why we have chosen IPSec.

[3]Target SID is defined by the port number on the target node.

work for the cluster and the distributed logs in order to detect attacks using Snort IDS [13]. This service on security server is related to external IDS through SCC.

The auditing service is connected to external log servers when needed. The connection between the auditing service and external loggers is not through SCC for performance reasons.

The requirements for this service are currently being defined.

13 Related work

This work distinguishes itself by being focused on the design of a security infrastructure targeted for clustered servers as compared to previous work that is focused on single computers or on clusters of general-purpose Linux machines. In addition, DSI takes into account all the issues related to security management starting at the design level. Some of the related work includes CorbaSec, the CORBA security service that handles the security issues regarding access control and authentication for interactions between different objects. CorbaSec does not take into account all aspects of security for example detection and reaction mechanisms like DSI and guarantees the security at middleware level independently from platform considerations.

On the other hand, Security Enhanced (SE) Linux from the National Security Agency (NSA) [5] or the Linux Security Module [6] (LSM) effort run on a single computer; they do not extend to a cluster.

Finally, Grid Security Infrastructure (GSI) was subsequently developed, based on existing standards, to address the security requirements that arise in Grid environments [1]. The DSI approach is more fine-grained and is based on modifying the OS to enhance security mechanism (as explained in Section 10.1). The approach of DSI is possible because the software and hardware configuration in the cluster is under tight control. In practice, DSI supports a coherent vision of security throughout the whole cluster as GSI supports secure interoperable mechanisms between different trust domains for multiple users.

14 Results

We performed preliminary experiments on a cluster of Linux nodes. The fact that the source code of the Linux kernel is available and well documented is a major advantage for developing DSI on Linux based clusters.

So far, a secure boot mechanism for a diskless Linux system was implemented. Using secure boot with digital signatures, a distributed trusted computing base (DTCB) will be available as of the boot of the cluster nodes. The kernel at secure boot is small enough to be thoroughly tested for vulnerabilities. Furthermore, the use of digital signatures for binaries and a local certification authority will prevent malicious modifications to the DTCB.

We also implemented a security module based on Linux Security Module (LSM) that enforces the security policy as part of the DSI access control service [6]. This module is integrated with SCC and provides distributed access control mechanisms. DSI currently supports preemptive and dynamic security policy at the process level throughout the whole cluster for some operations. As future work, we will extend these capabilities to all operations on the cluster.

At this time, we are implementing the distributed security policy. In order to ease administration and maintenance of this policy, we completed a study to devise methods to reuse information already contained in package management systems (such as RPM for Linux) in order to generate part of the security policy, or to push such information to the package [10]. Specification of the exact language used to express the policy and of the compilation and loading mechanisms remains to be completed.

We implemented a secure communication channel based on OmniORB, an open-source implementation of Corba [11]. The implementation of SCC is independent from the communication middleware used. As mentioned in Section 7, SCC logics are implemented on top of a portability layer. This makes the implementation independent of any communication middleware used. The choice of CORBA as communication middleware for SCC was motivated by the following factors:

- Support from CORBA standard and implementations for distributed real-time and embedded systems,
- Support for advanced security mechanisms by CorbaSec,
- Interoperability.

15 Conclusions and future works

In this paper, we presented the need for a new security approach for carrier-class applications running on Linux clusters. Based on our motivations to develop a coherent solution addressing the security needs of carrier-class servers, we proposed a new design for a secure distributed infrastructure. We presented the main elements of this design and discussed some of the preliminary results. We believe that this design is a practical approach to enhance security for large-scale Linux clusters with carrier-class needs.

To complete DSI, we plan to collaborate with Open Source projects and initiatives, and other organizations on the design and development of this secure infrastructure.

Acknowledgements

We thank David Gordon and Dominic Pellerin for their contributions to DSI. Also, we thank Marc Chatel and Bruno Hivert for their sanity checks on our design and implementation.

References

[1] *Foster I., Kesselman C., Tsudik G., Tuecke G., A Security Architecture for Computational Grids*, 5th ACM Conference on Computer and Communication Security.

[2] *Linux FreeS/WAN*, `http://www.freeswan.org`.

[3] *ISO 10181-3: Security Frameworks for Open Systems: Access Control Framework*, ISO, (1996).

[4] *ITU-U Recommendation X.800: Security Architecture for Open Systems Interconnection for CCITT Applications*, ITU-T (then CCITT), (1991).

[5] *Loscocco P.: Security-Enhanced Linux*, Linux 2.5 Kernel Summit, San Jose (Ca) USA, (2001), `http://www.nsa.gov/selinux/docs.html`.

[6] *Linux Security-Module (LSM) framework*, (2001), `http://lsm.immunix.org`.

[7] *MontaVista Linux Carrier Grade Edition 2.1*, `http://www.mvista.com/products/mvl_cge/mvlcge_overview.html`

[8] *Morris, J. Selopt: Labeled IPv4 networking for SE Linux,* `http://www.intercode.com.au/jmorris/selopt`

[9] *M. Zakrzewski. Mandatory Access Control for Linux Clustered Servers*, In Proceedings of Ottawa Linux Symposium, June 2002.

[10] *C. Levert, M. Dagenais, Security Policy Generation through Package Management*, In Proceedings of Ottawa Linux Symposium, June 2002.

[11] *omniORB,* `http://www.uk.research.att.com/omniORB/`

[12] *Open Source Development Lab, Carrier-Grade Linux Working Group,* `http://www.osdl.org/projects/cgl/`

[13] *SNORT*, `http://www.snort.org/`

EVMS: A Common Framework for Volume Management

Steven Pratt
Linux Technology Center
IBM
Austin, TX 78758
slpratt@us.ibm.com
http://evms.sf.net

Abstract

The Enterprise Volume Management System (EVMS) brings a new model of volume management to Linux. EVMS integrates all aspects of volume management into a single cohesive package. By introducing a new pluggable architecture, EVMS provides extendibility while ensuring consistency and cooperation across multiple volume management schemes.

EVMS consists of two main components, the Runtime, which resides in the kernel and handles discovery and I/O functions, and the Engine, which resides in User Space and handles setup and configuration. Packaged with the Engine are three user interfaces, a GTK based GUI, a command line interpreter, and an ncurses based interface.

EVMS borrows from the existing Linux volume management technologies, combining them into a single easy to use package. Imagine being able partition your disk, create mirrors and raid devices, define volume groups and logical volumes, all from one integrated, easy to use interface. With EVMS you can do this and more.

EVMS provides immediate benefit to system administrators who wish to get a handle on their storage configurations, as well as less technical users who have not memorized all of the various commands and config files. No more scanning through raidtab files or issuing multiple LVM commands just to find out how a system is configured. Just bring up the EVMS GUI and have all of this information at your fingertips. Not only can you see what disks or partitions make up which volumes, but you can also see if these volumes are formatted or mounted.

1 Introduction

Volume management is an integral part of any operating system. Every major server operating system has some form of volume management capability. The methods vary from simple DOS partitioning to complex volume groups and everything in between. What stands out about each operating system is that they generally have a single method for performing volume management and with that a single consistent interface for configuring storage. When we look at Linux we see that, as with most components, it offers multiple choices for volume management. In most cases having competing products that give the user different capabilities and functions is good for Linux as it fosters competition and technical advancement. In the cases where the user chooses only one of the competing technologies there is really no downside to this approach. However, in the case where similar or competing technologies do not completely overlap, users may desire to use multiple technologies in order to get a combination of function not found in any one technology. In these cases some problems can arise.

This is the case with volume management in Linux today. The three major volume management schemes available in Linux today (partitions, LVM and MD) each provide exclusive features which are not available in the other volume managers. This means that the user, instead of choosing one method of volume management, may instead use many. This puts the burden on the user/administrator to learn how to use each set of tools. Not only that, but he must figure out any interdependencies that exist and work around them.

To further complicate the task of managing storage, it is not enough to define the storage layout. Usually you must also associate a file system with the volume. This adds another layer in which multiple choices are available, each with slightly different features and interfaces. Today there is little to no coordination between volume managers and the file systems for operations such as resize, where issuing the commands in the wrong order can cause data loss.

EVMS attempts to make sense out of this jumble of components by providing a consistent architecture and framework in which all of the varying technologies can exist in harmony with each other. By providing a consistent set of APIs and common services, EVMS allows new technologies to be added to the existing framework while ensuring that they will interact correctly with all of the current functionality.

2 EVMS Architecture

In EVMS the task of volume management has been divided into three main components, the Engine, the Runtime, and the User Interfaces. EVMS uses a layered, plug-in model in the Engine and Runtime to provide flexibility and extensibility for managing storage. This method of managing storage allows for easy expansion or customization of various levels of volume management. The EVMS framework is fully 64 bit enabled and architecture independent. It is also endian neutral, except for certain compatibility plug-ins for which the original implementation used native layouts.

Runtime

The EVMS Runtime refers to the in-kernel portion of the system. The Runtime has two primary purposes:

1. Coordinating the discovery of logical volumes and creating the necessary block devices to represent those volumes in user-space.
2. Handling I/O to the volumes.

The EVMS kernel component consists of two parts, the common services and the plug-ins. The common services provide the framework which makes up the heart of the EVMS architecture. This framework coordinates the loading and registration of plug-ins for each of the four plug-in classes supported and the in-kernel discovery process. The common services also provide IOCTL routing and helper routines for the plug-ins to use. Before looking at the common services we should first describe each of the classes of plug-ins and how they work.

EVMS kernel plug-ins are built as standard Linux kernel modules and are compiled as part

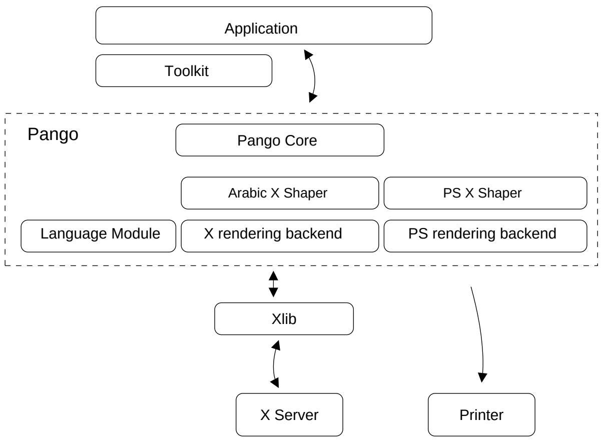


Figure 1: EVMS kernel architecture

of the Linux kernel build process. These plug-ins add functionality to the EVMS runtime by providing support for specific features of volume management. The plug-ins register with the EVMS common services at init or module load time.

The first class of plug-in is the Device Managers. Device managers are responsible for determining the available devices on the system and presenting them to the EVMS framework. Device managers are also responsible for determining attributes of the devices (such as hardsector size) as well as detecting if a device has been removed or modified. This is the only class of plug-in which deals directly with device drivers or device driver queues. There is currently only one Device Manager implemented (Local Device Manager) which manages all devices found on the gendisk list. It remains to be seen if other device managers are required for features such as multipath IO, NAS, and SAN, or whether the current device manager is sufficient.

The second class of plug-ins is the Segment (or Partition) Managers. Segment Managers are responsible for dividing logical disks into physically contiguous pieces or Segments. A separate Segment Manager is required for each partitioning scheme supported (DOS, S390, Amiga, ...) and only one Segment Manager may be assigned to a disk at a time. It is possible however, to 'stack' Segment Managers, which results in one partitioning scheme being imbedded within a segment created using another partitioning scheme.

The third class of plug-ins is the Region Managers. Region Managers consume disks and/or segments and produce regions which represent logically contiguous storage space. The region layer is the layer in which containers or groups are implemented, making this the place where AIX and Linux LVM plug-ins are found. Also, since this is the last layer before EVMS specific on-disk data structures are introduced, and the first which supports logically contiguous space, it is where most of the compatibility plug-ins are found, including MD. Region Managers, like Segment managers can be stacked, allowing for configurations such as LVM on RAID.

It should be noted that the Device Managers, Segment Managers and Region Managers have no constraints on their size, placement, or format of metadata. Depending on what the plug-in is trying to accomplish, such as compatibility with DOS partitions or Linux LVM, the plug-in can layout metadata in any format desired or required. The advantage of this is that these types of EVMS plug-ins can support any native metadata format. The downside is that with some non-EVMS metadata formats, redundancy and error checking may not be as complete as desired.

The fourth class of plug-ins is Features. This layer contains plug-ins which are designed specifically for EVMS and make use of metadata layouts provided by and enforced by the EVMS common code. Features consume one or more objects created by any of the layers and produce a new feature object. All features share some common metadata, called Feature Headers, on disk. Thus, it is possible to do consistency checking and to even detect if a feature which was configured on disk is for some reason not compiled into the kernel. The Feature Header is where common information like the volume name, volume minor number, feature id, feature depth or count, and versioning information is kept.

Kernel Discovery process

EVMS supports in kernel discovery. This means that each kernel plug-in contains the code required to probe for and recognize the metadata on disk and build volumes without user space intervention. This capability allows an EVMS enabled kernel to recognize and export all volumes that exist in a system at boot time without requiring a RAM disk or any user scripts to be run. This also means that EVMS is not dependent on any user space configuration files that could be accidentally deleted or damaged by file system or other errors. There is some debate as to whether this function belongs in the kernel or in user space. The main reasons for putting it in the kernel is that it make boot setup much easier (no scripts or RAM disks required) and it makes upgrades easier since the kernel is not dependent on user tools to discover volumes; thus the kernel can be upgraded independently of the user tools.

The kernel discovery process starts with the device manager examining the gendisk list for acceptable devices. The device manager creates a object representing each device to be managed and adds this node to the discover list. This list is returned to the common services which then passes it on to each Segment Manager. Each Segment Manager can examine each node and remove it from the list if it claims the disk. Once a Segment Manager claims a device, it processes this device and creates new objects for each Segment on the device. These new objects are placed on the discover list for further processing. Once each Segment manager has had a chance to claim devices, if any new objects were placed on the list, this process is repeated with the new list. This allows for nested partitioning. This process continues until no new objects are created and then the common services takes the object list (which now consists of disk and segments) and repeats this process with Region Managers.

Once Region discovery is completed, the kernel starts feature discovery. Due to the common Feature Headers for EVMS features this process is much more streamlined and efficient. Each object is examined for the existence of Feature Headers and if one is found, that object is grouped together with other objects with the same volume serial number (found in the Feature Header). Next the Feature Header depth is checked in each of the objects for the volume, and the objects with the deepest or bottom most count are given to the plug-in which will own the objects. This is done based on a feature ID stored in the Feature Header. This process repeats for each level of the feature stack, or until an error is encountered.

EVMS has the concept that only volumes and not intermediate objects are actually exported from the kernel. This helps prevent accidental access to objects in the middle of a volume stack. Additionally, each storage object can be marked as being a volume or not. If a storage object is not marked as a volume via a bit in the Feature Header, then it is not exported for access from the kernel.

After all features have been applied, the common services adds the volume to the global volume list and makes it available through the EVMS block device using the minor number and name stored in the Feature Header. For volumes which do not have EVMS features applied (compatibility volumes), they are assigned their Linux legacy device name (i.e. hda3, group1/lv1) and are assign the next available minor number.

IO Path

The IO path in EVMS is slightly different from many other block devices in Linux due to the plug-in implementation. The entire EVMS subsystem, including all plug-ins, functions as a single block device driver. What this means is that EVMS does not use the driver queue interface to drive IOs from one EVMS layer, or plug-in, to the next. Instead, each EVMS plug-in exports a function table as part of its initialization process. In this function table are read and write entry points. When an IO is received by EVMS on a volume via the EVMS block device, the common services route this request to the appropriate read (or write) entry point of the topmost object in the volume stack. The plug-in owning that object processes the request by modifying (duplicating, splitting, changing offset, etc.) it and calling the entry point of the object(s) to which the request is now destined. This process repeats for each object or layer in the call stack for this volume, and only when the request gets to the Device Manager is it then re-queued to the device queue of the actual disk driver. This is possible due to the EVMS framework, and removes the requirement for each plug-in to consume valuable system resources such as device major numbers and thus allows an infinite number of internal objects such as partitions.

IOCTLs

EVMS supports two types of IOCTLs, global and volume specific. Global IOCTLs are sent to the EVMS block device and are for commands which are not specific to any one volume or plug-in. They are handled entirely by the EVMS common services. These include commands used by the Engine such as get version, set debug info level, and rediscover. Volume specific IOCTLs are actually targeted at the minor number of the volume in question. These IOCTLs may be processed by the common services in some cases (such as get block size) or the IOCTL may be passed down the volume stack and processed by each plug-in in the stack (such as delete volume). One Global IOCTL which is worth mentioning is the direct plug-in communication IOCTL. This IOCTL allows for an instance of a plug-in in the Engine to communicate with its corresponding plug-in in the kernel. This allows a plug-in writer to code whatever communication is required, although in most cases this support is not needed by the plug-ins.

3 Engine

The EVMS Engine is the core of the user-space administration tools. The Engine is implemented as a shared library. Like the kernel component of EVMS, the Engine implements the same layered plug-in model. The Engine itself provides the common services and framework, while plug-ins, which are also shared objects, provide the real functionality. The Engine has three sets of APIs defined. First, the Engine library provides the front end API set which is used by the User Interfaces. This application interface provides a single well defined set of entry points through which all manipulation of volumes is done, regardless of the type of volume being configured. Second, the Engine defines the Plug-in APIs. This is the set of functions which must be implemented by each plug-in. This standard API set for plug-ins ensures that new plug-in will integrate with the rest of the system. The third API set is the common services provided by the Engine to the Plug-ins to make their job easier.

Changes to a system with EVMS are accomplished by using one of the user interfaces to open and interact with the Engine. When the Engine is opened, it gets a list of devices via an IOCTL to the EVMS kernel component. The Engine then performs a discovery process which is very similar to that found in the kernel. Each device is probed by the plug-ins and a complete in memory representation of entire system is built. As each plug-in discovers devices or objects that it manages, a tree structure or graph is created.

When reading or writing data to or from devices, the IO request is passed to the plug-in appearing below the current plug-in in the volume tree. The IO request is transformed just as it would be in the kernel IO path. This is done to allow for a complete virtualization of new configurations to occur. For example, an LVM container or group can be created using a RAID5 array which has not been committed to disk, nor is running in the kernel. This allows the user to make any number of changes and configure the system exactly as they would like it before writing any changes to the actual on disk metadata or kernel configuration. If the user decides that they do not like the changes, they can simply quit the Engine session without saving and nothing will have been modified. The user may also commit changes at any point in the configuration process if he so desires. Changes to the system are done by writing metadata to disk by calling down the stack of plug-ins until the device manager is reached, The device manager then writes the data to disk by calling a kernel IOCTL. Once the new metadata is committed to disk, the kernel is told to purge deleted or modified volumes from memory and to go rediscover them from changed metadata on disk. During this process the volumes that have been modified are quiesced temporarily by the common services of the EVMS runtime.

Each plug-in inside the Runtime has a corresponding Plug-in inside the EVMS Engine. These Plug-ins provide administrative capabilities for various kinds of logical volumes. For example, the DriveLinking plug-in inside the Runtime is responsible for I/O through DriveLink devices, and the corresponding DriveLinking plug-in inside the Engine handles creation, modification, and maintenance of DriveLink volumes.

Plug-ins in the Engine are typically more complex than their counterparts in the kernel. This is due to the greater number of APIs required for configuration as well as support for the sophisticated user interface. The Engine plug-ins contain all of the parameter validation code for creating and modifying storage objects. The Engine plug-ins must also duplicate the read/write logic found in the kernel plug-ins in order to support the complete virtulization of volume configuration in the Engine.

In addition to the four classes of plug-ins found in the kernel, there is one additionsl class which is unique to the Engine. This class is File System Interface Modules or FSIMS. FSIMS are a special plug-in class and have their own unique function table. FSIMS are used to allow EVMS to communicate with various File Systems to coordinate changes to volumes. Having an FSIM not only allows actions like mkfs and fsck to be performed directly from the EVMS interface, but it also ensure that actions such as expanding or shrinking a volume are properly coordinated with the File System without the user being required to know the order in which operation must be performed. This capability does not exists anywhere else in Linux.

4 User Interfaces

Because the EVMS Engine provides a programmatic interface instead of a direct user interface, multiple user interfaces can be written or tailored to a particular style or group of tasks. Currently, four different user interfaces have been developed:

1. A general command line which is useful for automating tasks.
2. A graphical user interface (GUI) for easy administration using a GUI desktop.
3. A text-mode, ncurses-based interface.
4. A set of command line utilities for emulating the Linux LVM (logical volume management) command set.

The first three of these interfaces are general purpose and support all existing plug-ins. The fourth, the command line utilities that emulate Linux LVM, is one example of how the Engine application programming interface (API) can be used to create a tailored user interface. As the name implies, these utilities will only interact with the LVM plug-in to manage LVM containers and regions.

All of the user interfaces interact with EVMS through the Engine application APIs. These APIs abstract the specific options and parameters of each Engine plug-in into a well defined interface known as the task interface. A set of well known tasks have been defined (create, expand, shrink, modify, ...) to allow for a negotiation to occur between the user interface and the plug-in. This allows the user interface to write one set of screens or functions for each task, but be able to use this code with any plug-in. It is the plug-in's responsibility to return to the user interface all objects that can be used in a task, and once an object(s) is selected, to return a list of options to the user interface. Each option is described by an option descriptor data structure which allows the user interface to select the proper display method and entry type for the particular option. The descriptor also indicates any restriction or limits on the option such as minimum or maximum values. As each option is set by the user interface, the information is passed to the plug-in for validation. As part of the negotiation or validation, the plug-in can enable or disable other options, or change the values possible for other options.

One of the additional advantages of this approach is the ability for the user interfaces to enforce limits based on actual constraints instead of theoretical ones. For example if there is only 100M of freespace available for a create command, the GUI will not let you enter any value greater that 100M, rather then letting you enter a larger value and fail the command.

5 Future Work

Two main work items will affect the EVMS architecture in the coming year. The first is a generic move capability. New APIs will be added to allow the moving of one storage object to another while the volume is mounted. For example this capability will allow for the moving of data on a mounted partition to be moved to a LVM region or vise versa. This same technology will be used within plug-ins to provide functionality equivalent to the Linux LVM's pvmove.

The second, and larger work item is cluster support. EVMS is currently working on adding cluster support for configuration of shared storage within a cluster. With this support, plug-ins such as snapshotting and bad block relocation will be able to work on volumes backed by shared storage and access from multiple nodes in a cluster.

6 References

1. The Home Page for the EVMS Project
   `http://evms.sf.net`

2. EVMS HowTo
   `http://evms.sf.net/howto`

3. Linux Partition HowTo
   `http://tldp.org/HOWTO/mini`
   `/Partition.html`

4. Software Raid HowTo
   `http://tldp.org/HOWTO`
   `/Software-Raid-HOWTO.html`

5. Linux LVM
   `http://www.Sistina.com`
   `/products_lvm.htm`

Automatic Regression testing of network code: User-Mode Linux and FreeSWAN

Michael C. Richardson
Sandelman Software Works Inc.
mcr@sandelman.ottawa.on.ca   http://www.sandelman.ca/mcr

Abstract

The Linux FreeSWAN project (IPsec for Linux) produces rather complicated networking code. The successful application of the protocol results in all network data being encrypted. The use of dynamic keying means that it nearly impossible for an observer (even a trusted one trying to test) to know what is going on. The need for multiple systems (often as many as 6) to be properly configured creates an environment nearly impossible to test regularily.

The emergence of virtual machine technology, particularly, User Mode Linux, has provided a solution to the testing problem: create as many virtual machines as needed and control them using standard testing scaffolding technology: expect(1). This paper describes the scaffolding and the resulting testing regime which is used.

The focus is around a modified network switch emulator, "uml_netjig" which provides the ability to play and capture network packets through a single User-Mode Linux virtual machine.

A second iteration of this tool is also described, combining more complicated expect scripts, and a command mode for uml_netjig, permitting coordination of the multiple virtual machines that are needed when doing fully negotiated IPsec sessions.

1 Background: What is this about

The Linux FreeS/WAN project is a funded project. It has the mandate to produce an IPsec implementation for Linux. The ultimate goal of this effort is to provide systems and software to permit citizens for the world to keep all of their Internet traffic private.[1].

Testing networking protocols is often difficult. By definition there is at least one network involved and often several independent systems attached to the network.

With many application layer network protocols (e.g. http) one can cheat—the network is the virtual "loopback" device, and multitasking permits both ends of a protocol to run on the same host. It is therefore common to see people doing all sorts of network development using a garden-variety notebook.

The situation is not the same for transport and network layer protocols such as IP, TCP, and IPsec. These layers of the protocol are more fundamental. They are typically implemented inside a system kernel. This makes development work as difficult as generic kernel work.

If one is to test them on one's notebook or desktop, one risks putting one's own development environment at risk. It is common experience that doing kernel development is

[1]See http://www.freeswan.org/[2]

much easier with at least two machines—one machine is crashed every ten minutes and the other machine is used as the development host. The split between development and testing is much better understood in embedded system work—the machine under test is often of a totally different type than the development machine. Historically, the machine under test (a VCR or a modem) is incapable of even running a development environment.

Network protocol development work is further complicated by the need to have more than one machine involved.

1.1 eXtreme Programming

The growing discipline of eXtreme Programming[Bec01] has a number of fundamental principles

- rapid feedback
- assume simplicitiy
- incremental change
- embracing change
- quality work

[Bec01] goes on to explain that the fundamental activities are coding, testing, listening, and designing. XP tries to reach the point where one writes the test cases before the code. To do this, the cost (in effort and time) of testing must be reduced such that all tests can be run frequently—several times a day if possible. This very rapid feedback reduces the risk of introducing problems—permitting developers to program more efficiently and with more confidence.

This paper describes the typical requirements for doing network testing. The reasons why it is expensive and why it is difficult to automate are explained. Our solution uses User-Mode-Linux to turn machines into processes. The scaffolding is then used to control these processes, to put them through their paces on a regular basis.

No solution is perfect on the first pass—XP actually encourages partial solutions to be implemented and feedback to be received—so we describe our second pass, which at the time of writing, is still in the design phase.

2 How to test with physical hardware

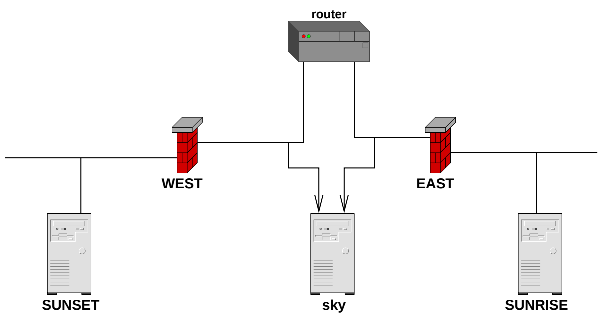


Figure 1: Basic Physical Network configuration

The basic network is shown in Figure 1. The taxonomy for our test setup is that the Sun rises in the east and the sun sets in the west. Thus one can easily remember where each host is.

EAST and WEST, shown with firewall icons, are FreeS/WAN IPsec gateway boxes.

SUNRISE and SUNSET are just ordinary hosts whose traffic will be protected by their respective gateways.

The machine SKY is used to do network analysis ("sniff"). There are frequently problems that occur when trying examine the traffic produced by a machine itself, so a separate machine to make unbiased observations if necessary.[3]

The two gateway boxes are not directly attached, but rather are connected via a router. There are two reasons for this:

- the current implementation of FreeS/WAN requires a default route to operate correctly.
- a common operational issue is with links where the Maximum Transmission Unit (MTU) is restricted, and this router provides a place to cause such an impairment[4]

This setup is very representative of the typically deployed scenario for FreeS/WAN systems in a VPN. It does not cover every single situation—most of the most difficult-to-reproduce bugs have occured in other setups. More machines are needed to create such setups.

Aside from the space and cost involved in providing each developer with six machines (it is often the case that `sky` is the developer's desktop), there are a number of other factors that make this difficult.

The major problem is maintaining this setup. There are many machines with many files that must be maintained. The systems must be kept up-to-date so that the latest kernels can be tested, yet at the same time, testing against older kernels is necessary. Different distributions need to be tested. The combinatorics are quite high.

The other major problem is work environment. Sitting in a room with six computers is a lot of noise. Getting access to each system's console is difficult (one can not rely upon network logins!). If a monitor is attached to each system (vs a monitor switch), then the developer probably gets too much exercise.

One answer to this is serial consoles. See Figure 2. Terminals attached to serial ports was the primary way that people used Unix until the advent of the X-terminal, and Linux continues this grand tradition.

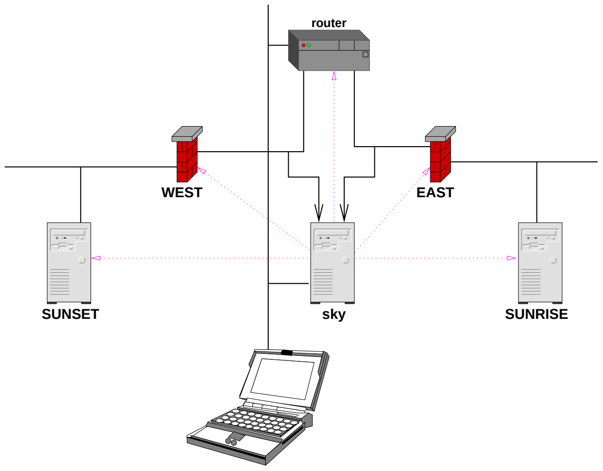


Figure 2: Basic Network with console access

One simply puts the following in `/etc/lilo.conf`:

```
serial=0,38400n
...
image=/boot/vmlinuz-2.4.18-6mdk
   label=linux2418
   root=/dev/hda1
   append="devfs=mount \
     console=ttyS0,38400 \
```

[3]In particular, on Linux 2.2 or lower, turning on the packet capture mechanism changes the control structures attached to the traffic and causes faults relating to policy for the keying channels' control packets. PR#48 at `http://bugs.freeswan.org:81/bugs/gnatsweb.pl?&database=freeswan&cmd=view&pr=48` 2.4 has solved this problem

[4]FreeS/WAN has adopted the term "impairment" to denote any challenges which are introduced to a system or network to permit another part of the system to be tested

```
    console=tty0"
  read-only
```

The console then appears on both "COM1" and on the VGA screen. In this situation, the machines may be located in another room, connected to a console server. One logs in from one's (quiet) desktop to the console server, accessing each machine via a serial port. Serial interfaces are readily available with either PCI or USB interfaces. This makes building a 6-port console server rather easy.

The developer now has ready access to each machine, can reboot each machine, select different kernels, and can configure it without even having networking on. In addition, kernel panics ("kernel oops") or other strange output on the console can be cut and pasted into emails, etc..

2.1 Still challenging to test

The serial consoles do not solve the other problems—managing the very many different configurations, or coordinating the systems to perform a test case.

The author has used such a setup for many years with many Unix operating systems. Using the "expect" program and the serial consoles one can automate some of the tests. Some of tests are harder to deal with—ones that fail can cause the system to hang—this will require operator intervention. Further use of more hardware can solve this problem as well—relays can toggle reset switches or even power cycles.

The result, however, is a very complicated testing environment—it can take weeks to configure it, and mere hours to break. There is far too much specialized hardware involved, not to mention the software.

There is a better way which will be described, but first, the requirements for the testing environment will be examined in a bit more detail.

3 What do we really need

3.1 A brief primer on IPsec

IPsec[TDG98],[KA98a] consists of three transport layer protocols: AH[KA98b], ESP[KA98c] and IPcomp[DNP99]. There is one management protocol in existence at this time, ISAKMP[MSST98]/IKE[Pip98],[HC98].

These transport protocols can be applied to upper layers of TCP, UDP, or any other transport protocol. When the upper layer is the "IPIP"[Per96], then the protocol is said to be in "tunnel" mode. For most Virtual Private Network (VPN) usages, tunnel mode is the preferred method since it hides the origin source/destination address. VPNs are often treated as being virtual leased lines.

Each of the transport protocols provide session-layer encryption. They are referred to as "security associations." These are unidirectional concepts—a pair is usually needed for bidirectional communications.

3.1.1 Authentication Header (AH)

The Authentication Header provides origin authentication and integrity of the headers and of the data portion. No privacy is provided.

3.1.2 Encapsulating Security Payload (ESP)

The ESP header provides origin authentication, integrity and optional privacy of the data portion only. Normally, this privacy option is provided by encryption, but the specification permits a "null" encryption to be used in some circumstances.

3.1.3 IP compression header (IPcomp)

A good encryption algorithm produces cyphertext that is evenly distributed. This makes it difficult to compress. If one wishes to compress the data it must be done prior to encrypting. The IPcomp header provides for this.

One of the problems of tunnel mode is that it adds 20 bytes of IP header, plus 28 bytes of ESP overhead to each packet. This can cause large packets to be fragmented. Compressing the packet first may make it small enough to avoid this fragmentation.

3.1.4 Internet Security Association Key Management Protocol (ISAKMP)

ISAKMP is a framework for doing Security Association Key Management. It can, in theory, be used to produce session keys for many different systems, not just IPsec.

3.1.5 Internet Key Daemon (IKE)

IKE is a profile of ISAKMP that is for use by IPsec. It is often called simply "IKE." IKE creates a private, authenticated key management channel. Using that channel, two peers can communicate, arranging for sessions keys to be generated for AH, ESP or IPcomp. The channel is used for the peers to agree on the encryption, authentication and compression algorithms that will be used. The traffic to which the policies will applied is also agreed upon.

3.2 Testing KLIPS

In FreeSWAN, the session layer encryption, security association management and traffic selection is done by a kernel component called KLIPS (Kernel Level IP Security). This component can be built as a loadable kernel module or statically built in.

As the security associations are unidirectional one can effectively separate the encrypt/encapsulate and decrypt/decapsulate operations for testing purposes.

For ease of thinking, the encryption operations are always done on EAST and the decryption operations are always done on WEST.

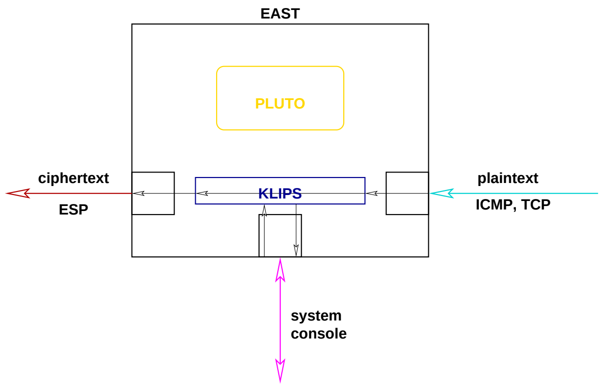


Figure 3: How to test KLIPS

As indicated in Figure 3, a source of plaintext packets is needed, a way to examine the ciphertext packets is needed, and a way to configure the system is needed. In the physical setup of the previous section, the source of plaintext packets is provided by the machine SUNRISE, and the examination of the packets is provided by SKY.

A typical initialization script for KLIPS is shown in Figure 4.

The term SPI means "Security Parameters Index." Each security association is indexed by a SPI. Note that a separate SPI is setup for

Figure 4: A typical initialization script for KLIPS

```
#!/bin/sh
TZ=GMT export TZ

ipsec spi --clear
ipsec eroute --clear

enckey=0x4043434545464649494a4a4c4c4f4f515152525454575758
authkey=0x87658765876587658765876587658765

ipsec klipsdebug --set pfkey
ipsec klipsdebug --set verbose

ipsec spi --af inet --edst 192.1.2.45 --spi 0x12345678 \
  --proto esp --src 192.1.2.23 --esp 3des-md5-96 \
  --enckey $enckey --authkey $authkey

ipsec spi --af inet --edst 192.1.2.45 --spi 0x12345678 \
  --proto tun --src 192.1.2.23 --dst 192.1.2.45 --ip4

ipsec spigrp inet 192.1.2.45 0x12345678 tun inet \
  192.1.2.45 0x12345678 esp

ipsec eroute --add --eraf inet --src 192.0.2.0/24 \
  --dst 192.0.1.0/24 --said tun0x12345678@192.1.2.45

ipsec tncfg --attach --virtual ipsec0 --physical eth1
ifconfig ipsec0 inet 192.1.2.23 netmask 0xffffff00 \
  broadcast 192.1.2.255 up

# magic route command
route add -host 192.0.1.1 gw 192.1.2.45 dev ipsec0

ipsec look
```

the ESP operation and for the tunnel operation. The two are then grouped together.

The `eroute` (Extended Route) command then selects traffic by source and destination address for processing by the aforementioned group. [KA98a] defines other selectors, including TCP and UDP port numbers, but those selectors are not implemented in KLIPS at this time.

The `tncfg` command attaches the IPsec pseudo to a physical device. This is necessary in 2.0 and prior kernels to provide a path for the resulting packets to actually leave the system. Otherwise, the `route` command at the end can cause packets to loop internally. Eliminating this problem—we refer to it as "stoopid routing tricks™"—is the major goal of revisions to KLIPS.

The `ipsec klipsdebug` commands turn on various debugging output. This debugging output is important for diagnosing what has really happened when the system fails.

Finally, the `ipsec look` command produces a short summary of resulting system setup. The output of this appears in Figure 5.

At this point, the system is ready to have packets sent through it. If the packets match the criteria for the SA, then they will be encrypted with the provided key.

3.2.1 KLIPS hassles

The observant will notice a number of numbers in the above output which were not in the script: the IV field (`0x24a4a14e81ee960e`), the lifetime values (it has been 9 seconds between the SA was created and the look command occured), and the date.

These variances cause two problems: the console output is not consistent on every run, and the resulting encrypted packets will have different ciphertext on each run.

3.3 Testing Pluto

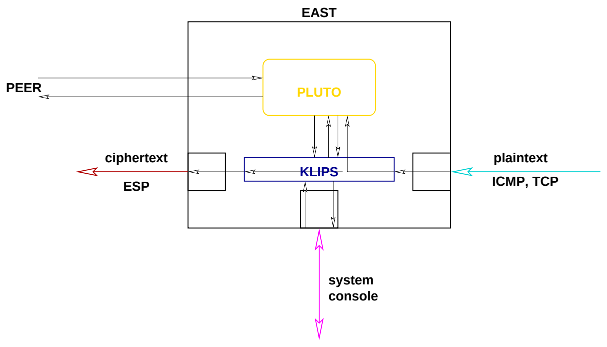


Figure 6: How to test Pluto

4 The first virtual attempt

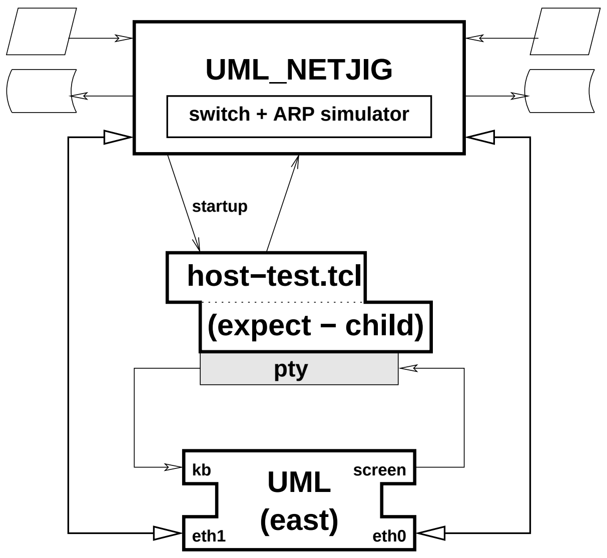


Figure 7: NetJig interface diagram

Figure 5: Output of `ipsec look`

```
east Tue Apr  2 04:32:28 GMT 2002
192.0.2.0/24       -> 192.0.1.0/24
       => tun0x12345678@192.1.2.45 esp0x12345678@192.1.2.45  (0)
ipsec0->eth1 mtu=16260(1500)->1500
esp0x12345678@192.1.2.45 ESP_3DES_HMAC_MD5: dir=out src=192.1.2.23
iv_bits=64bits iv=0x24a4a14e81ee960e alen=128 aklen=128 eklen=192
life(c,s,h)=addtime(9,0,0)
tun0x12345678@192.1.2.45 IPIP: dir=out src=192.1.2.23 life(c,s,h)=addtime(9,0,0)
Kernel IP routing table
Destination  Gateway      Genmask         Flags   MSS Window  irtt Iface
192.0.1.1    192.1.2.45   255.255.255.255 UGH      40 0          0 ipsec0
192.1.2.0    0.0.0.0      255.255.255.0   U        40 0          0 eth1
192.1.2.0    0.0.0.0      255.255.255.0   U        40 0          0 ipsec0
192.0.1.0    192.1.2.45   255.255.255.0   UG       40 0          0 eth1
192.0.2.0    0.0.0.0      255.255.255.0   U        40 0          0 eth0
0.0.0.0      192.1.2.254  0.0.0.0         UG       40 0          0 eth1
```

4.1 How to configure to use "make check"

4.1.1 What is "make check"

"make check" is a target in the top level makefile. It takes care of running a number of unit and system tests to confirm that FreeSWAN has been compiled correctly, and that no new bugs have been introduced.

"make check" expects to be able to build User-Mode Linux kernels with FreeSWAN included. To do this it needs to have some files downloaded and extracted prior to running "make check". This is described in the FreeSWAN documentation, under UML testing[5].

4.2 Running "make check"

"make check" works by walking the FreeSWAN source tree invoking the "check" target at each node. At present there are tests defined only for the `klips` directory. These tests will use the UML infrastructure to test out pieces of the `klips` code.

[5]http://www.freeswan.org/freeswan_snaps /CURRENT-SNAP/doc/umltesting.html

The results of the tests can be recorded. If the environment variable REGRESSRESULTS is non-null, then the results of each test will be recorded. This is used as part of a nightly regression testing system.

"make check" otherwise prints a minimal amount of output for each test, and indicates pass/fail status of each test as they are run. Failed tests do not cause failure of the target in the form of exit codes.

5 The second virtual attempt

This attempt is illustrated in Figure 8.

6 Conclusions

Use of virtual testing environment massively simplifies automated tests.

Limitations are that one can only test Linux 2.4 and beyond kernels.

It takes a lot of RAM and a lot of CPU, but still is cheaper than coordinating many physi-

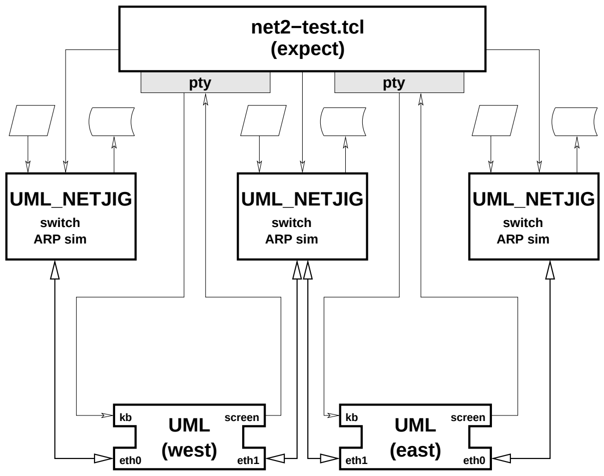


Figure 8: NetJig for multiple machines

cal machines.

References

[Atk95a] R. Atkinson. RFC 1825: Security architecture for the Internet Protocol, August 1995. Obsoleted by RFC2401 [KA98a]. Status: PROPOSED STANDARD.

[Atk95b] R. Atkinson. RFC 1826: IP authentication header, August 1995. Obsoleted by RFC2402 [KA98b]. Status: PROPOSED STANDARD.

[Atk95c] R. Atkinson. RFC 1827: IP encapsulating security payload (ESP), August 1995. Obsoleted by RFC2406 [KA98c]. Status: PROPOSED STANDARD.

[Bec01] Kent Beck. *eXtreme Programming: explained*. Addison-Wesley, 2001.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. RFC 2507: IP header compression, February 1999. Status: PROPOSED STANDARD.

[HC98] D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE), November 1998. Status: PROPOSED STANDARD.

[KA98a] S. Kent and R. Atkinson. RFC 2401: Security architecture for the Internet Protocol, November 1998. Obsoletes RFC1825 [Atk95a]. Status: PROPOSED STANDARD.

[KA98b] S. Kent and R. Atkinson. RFC 2402: IP authentication header, November 1998. Obsoletes RFC1826 [Atk95b]. Status: PROPOSED STANDARD.

[KA98c] S. Kent and R. Atkinson. RFC 2406: IP Encapsulating Security Payload (ESP), November 1998. Obsoletes RFC1827 [Atk95c]. Status: PROPOSED STANDARD.

[MSST98] D. Maughan, M. Schertler, M. Schneider, and J. Turner. RFC 2408: Internet Security Association and Key Management Protocol (ISAKMP), November 1998. Status: PROPOSED STANDARD.

[Per96] C. Perkins. RFC 2003: IP encapsulation within IP, October 1996. Status: PROPOSED STANDARD.

[Pip98] D. Piper. RFC 2407: The Internet IP security domain of interpretation for ISAKMP, November 1998. Status: PROPOSED STANDARD.

[TDG98] R. Thayer, N. Doraswamy, and R. Glenn. RFC 2411: IP security document roadmap, November 1998. Status: INFORMATIONAL.

PILS: A Generalized Plugin and Interface Loading System

Alan Robertson
International Business Machines Corporation
alanr@unix.sh OR alanr@us.ibm.com

Abstract

Many modern Linux application systems make extensive use of dynamically loadable object modules (plugins). However, most of these systems implement their plugin and interface management systems in a way that satisfies their own immediate needs, and is not generally directly usable by other projects.

PILS is an generalized and portable open source Plugin and Interface Loading System. PILS was developed as part of the Open Cluster Framework reference implementation, and is designed to be directly usable by a wide variety of other applications. PILS is available under the terms of the GNU Lesser General Public License (LGPL). Since it is written in C, and built with automake and libtool, it is portable to most modern operating systems. PILS manages both plugins (loadable objects), and the interfaces these plugins implement. PILS is designed to support any number of plugins implementing any number of interfaces.

This paper describes the philosophy and goals of PILS, presents an example of how to use PILS, and discusses a few implementation details of the PILS system.

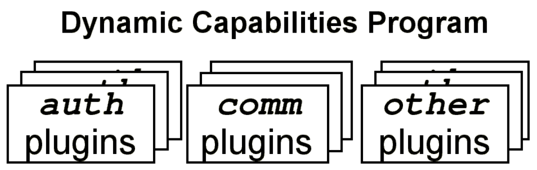


Figure 1: Plugin-enabled Program

1 Introduction

Many modern Linux application systems make extensive use of dynamically loaded object modules, oftentimes called plugins.

Plugins can be used for many purposes, and a complex program may use several different types of plugins for different purposes. A program which uses plugins can implement a variety of dynamic capabilities which were not explicitly planned for when the program was compiled. This situation is illustrated by Figure 1. The sample program has communications plugins, authentication plugins and other types of plugins. Such a program can take advantage of new types of communication systems, or authentication systems, etc. without recompiling or relinking the entire system. In some cases, the program can begin using newly-written code without even being restarted.

This is ideal when one wishes to create a general platform for many different people and organizations to build on. The Open Cluster Framework (OCF) reference implementation is such a system. It is not known how many types of plugins the system may eventually need, nor how many different implementations of each there might eventually be. Plugins are ideal building blocks for such general systems.

On most Linux-like systems, the `dlopen(3)` [dlopen] suite of calls are sufficient to load and unload shared objects (`.so` files) and to find symbols. However, there is much more to managing such plugins than is provided by either `dlopen(3)` or libtool [libtool]. PILS provides the following capabilities which are not provided by either `dlopen(3)` or libtool:

- Determining what capabilities or interfaces are implemented by a particular shared object
- Determining which plugins provide a particular interface
- Registering exported interfaces
- Importing interfaces for the use of the plugins
- Tracking the reference counts of interfaces

Additionally, the implementation of `dlopen(3)` varies from platform to platform, and is not available at all on some platforms. PILS uses libtool to take hide `dlopen(3)` idiosyncrasies.

PILS was written to provide basic capabilities for the Open Cluster Framework [OCF] reference implementation. OCF is intended to allow proprietary and closed software to coexist in the same framework, with contributions coming from many people, and to support plugins which were not compiled as part of the reference platform. As a result, the ideal model is to drop a suitable plugin into the correct directory, and have it simply work in every respect.

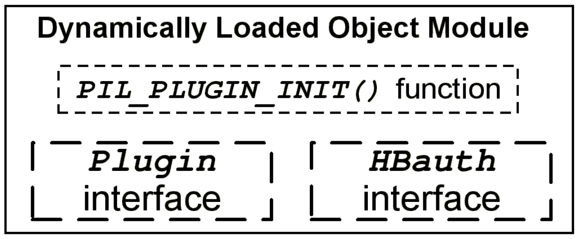


Figure 2: Dynamic Objects and Interfaces

As a result, simple automatic determination of the type of plugin and its capabilities must be supported.

PILS also standardizes certain common functions, such as setting the debugging level, and logging functions through mandantory plugin interfaces. This standardization makes plugins more manageable and flexible than would otherwise be the case. PILS has similar goals to the Glib 2.0 C class loader, but does not require the plugins to use the GTK class hierarchy, and provides some additional features.

2 PILS Model and Terminology

Before presenting more about PILS, it is necessary to define some terminology which is used in this paper. Many of the terms which PILS uses do not have universally accepted meanings. For the purposes of this document the following definitions are assumed:

- **Dynamically Loadable Object Module**. A dynamically loadable object module is an independent object file which can be linked at run time into a running program, executed, and then unloaded when desired. On Linux-like systems, dynamically loadable object modules are typically stored as shared object (**.so**) files. The relationship between a shared object file, its interfaces, and its `INIT` function is illustrated in Figure 2.

- **Plugin**. A plugin is a dynamically loadable object module which implements the `Plugin` interface described later. In addition to providing the `Plugin` interface, plugins typically implement one or more other interfaces.

- **Interface**. An interface is the set of exported and imported functions and data items which are shared by all implementations of these interfaces. For example, a communications interface might export functions to read and write packets, and import a function to lock a communications device. The exported functions are defined by a structure with pointers to the various functions (and optionally data items) which the plugin wishes to make public. The imported functions are similarly defined.

  Each interface type defines a unique set of imported and exported functions that are part of the interface which implementations of this interface must meet. PILS defines the **Plugin** interface, and allows others to be defined. PILS supports an arbitrary number of types of interfaces.

- **Exports**. The *exports* of an interface are the set of functions and/or data items which are provided by the plugin for the use of the system loading the plugin. These exported functions are provided through a single pointer to a structure containing all the individual functions. A typical interface defintion is a C structure consisting of a number of pointers to functions in a structure. Here is a sample interface from the example we will present in detail later.

  ```
  struct HBAuthOps {
   int (*auth)(struct HBauth_info*
           authinfo,
     const char* data,
     char*       result,
     int         resultlen);
     int (*needskey) (void);
  };
  ```

  In this example, the HBauth authentication exports are defined as a `struct HBAuthOps`. This structure in turn contains two function pointers, the `auth()` function, and the `needskey()` function. All implementations of the `HBauth` interface export this exact set of functions.

- **Imports**. The *imports* of an interface are the set of functions and/or data items which are provided by the loader of a plugin for the use of the plugin. The plugin implementation is then able to use these interfaces to accomplish its purpose. Most plugin loading systems do not provide for importing capabilities into a plugin. The provision of imports to the plugin increases the reusability of plugins in other contexts, and minimizes the use of external symbols by plugins (which is problematic on some platforms). These imported functions are provided through a single pointer to a structure containing all the individual functions, similar to the `HBauth` example in the Exports definition.

- **Type**. The word *type* is used in two closely-related senses in this document. In the most proper sense, *type* refers to the type of an interface. All implementations which share the same interface name are constrained to implement the same interface. This interface name is called the type of the interface, and also the type of the implementation.

  The word type is also used to refer to the *type* of a plugin. Although, technically plugins don't inherently have distinct types, there is a convention that a plugin named *bar* in directory *foo* provides the *bar* implementation of interface type *foo*. This convention is assumed by software which automatically loads plugins in order to load the particular interfaces.

- **Implementation**. An *implementation* of an interface is a particular set of exported functions and data which conform to the definition of the type of interface which it implements. When a plugin is loaded, it registers its interface implementations. Any given plugin can register as many implementations of as many different types as it wishes. Normally, applications provide multiple implementations of an interface, and each is generally contained in a separate plugin. As a shorthand for referring to interfaces (and sometimes plugins), we use a simple pathname convention. The string "HBauth/md5" is a shorthand notation for the *md5* implementation of the *HBauth* interface. This is consistent with the way the implementations are arranged on disk - with all the plugins of a given type being in the same directory.

3 Basic PILS Capabilities

The basic capabilities which PILS provides include the following:

- Loading a plugin
- Managing Reference counts
- Unloading a plugin (by reference count)
- Registration of interface implementations
- Provision of interface imports

4 Loading a PILS plugin

The process of loading a PILS plugin goes through the following steps:

1. **Request** The application requests the loading of a particular interface of a particular type using the `PILLoadPlugin()` function. Normally an application loads a particular plugin assuming that it provides an interface of the same type as the name of the directory in which it resides. Plugins which provide more than one interface are not fully supported at this time. More about this can be found in the Status and Future work section of this paper.

   If, as part of its configuration, the application needs to ask the user which particular implementation of a particular plugin should be loaded, the application can use the `PILListPlugins()` function to return a list of plugins of the given type. If it wishes to validate whether a particular plugin exists, it can use the `PILPluginExists()` call

2. **Load Shared Object** The PILS system then asks the libtool `lt_dlopen()` function to load the shared object into memory. `lt_dlopen()` then uses the native library loading system (commonly `dlopen(3)` to load the object into memory.

3. **Initialize Shared Object** Each plugin has a single initialization function which is then called to initialize the plugin. The name of this function is computed on the basis of its type and its name. This function name is created by the `PIL_PLUGIN_INIT` macro.

4. **Register Plugin** When the plugin's initialization (`PIL_PLUGIN_INIT`) function is called, it is is passed the Imports portion of the Plugin interface as a parameter. This Imports structure includes the following functions:

   - `register_plugin()` A function to call to register oneself as a plugin
   - `register_interface()` A function to call to register an exported interface
   - `log()` The preferred logging function - to be used by all the interfaces in the plugin.

   Once the plugin initialization function is called, the plugin then calls `register_plugin()` to register itself as a plugin. The register_plugin() function is where the exports portion of the Plugin interface is provided to the system. These standard exported Plugin functions include:

   - `pluginversion()` Returns the version of the plugin as a string.
   - `getdebuglevel()` Returns the current plugin debugging level.
   - `setdebuglevel()` Sets the current plugin debugging level to its parameter.
   - `close()` Prepare to be unloaded from memory.

5. **Register Interfaces** After registering itself as a plugin, the plugin calls `register_interface()` to register each interface it implements.

   Each type of interface has its own import and export requirements. Pointers to these structures are exchanged in the `register_interface()` call. When the `register_interface()` call is made, the `InterfaceMgr` managing this interface type then makes the interface available to be called. The generic `InterfaceMgr` does this by adding an entry to a `GHashTable` for that interface type. At this point, all the public interfaces of the plugin are available to be called.

5 Interface Managers

When interfaces are loaded, a plugin of type `InterfaceMgr` is invoked to manage the registered interfaces and make them available to the calling program. PILS provides the capability for each different type of interface to export its capabilities in a unique fashion, because each interface may have different policies and mechanisms for using them and making them accessible to the application. PILS allows these `InterfaceMgr`s to be plugins because they implement a single interface, and managing them as plugins is consistent with the design philosophy of the remainder of PILS. `InterfaceMgr` interfaces are managed much like other interface types, with the exception that the `InterfaceMgr/InterfaceMgr` interface manager is not dynamically loaded, but is linked to a set of built-in functions which are required to load other interface managers.

Any given type of interface can be managed either using a type-specific `InterfaceMgr` (named "`InterfaceMgr/`*type*"), or the generic interface manager. A type-specific interface manager may register the plugin with some other database or registry according to the needs of the application. The generic `InterfaceMgr` registers all the plugins it manages in a set of `GHashTable`s. One `GHashTable` is maintained for each interface type it manages. Many applications will find that the generic interface manager meets most common needs. This process sounds somewhat tedious but in practice most of this tedium is hidden and it is reasonably easy to use.

6 Sample Plugin

In this section, code for providing a sample plugin are provided and explained. This example code is based on the linux-ha [linux-ha] authentication plugins. In this case, authentication operations are exported as a `struct HBAuthOps`, which defines two exported functions, one for calculating a signature value, and another specifying whether or not the signature method requires a key. This is the same set of exported functions described earlier. In this example, these functions are called `md5_auth_calc()` and `md5_auth_needskey()` respectively.

The first thing to do is set a few #defines which are used by later macros. PIL_PLUGINTYPE defines the interface being implemented, and PIL_PLUGIN and PIL_PLUGIN_S define the name of our implementation.

```
#define PIL_PLUGINTYPE HBauth
#define PIL_PLUGIN      md5
#define PIL_PLUGIN_S    "md5"
/* Our plugin is called
   "HBauth/md5.so" */
```

Next, declare the set of operations to be exported to the world. In this case, an authentication plugin only needs to export two functions, so declare them, and set up the appropriate structure to point to them, so they can be exported later on.

```
static int
md5_auth_calc(const struct
   HBauth_info *t,
   const char * text, char * result,
   int resultlen);
static int md5_auth_needskey(void);

/* Authentication plugin
   operations */
static struct HBAuthOps md5ops =
{       md5_auth_calc,
        md5_auth_needskey
};
```

Now, define a couple of shutdown functions for managing the unloading of the interface and plugin. These two are provided separately, one or both of them may not have anything to do.

```
/* Shut down the plugin */
static void
md5closepi(PILPlugin* pi)
{
}

/* Shut down the interface */
static PIL_rc
md5closeintf(PILInterface* pi,
             void* pp)
{
        return PIL_OK;
}
```

The plugin needs to invoke a magic boilerplate macro which provides some common defaults for a number of things that the plugin requires.

Next comes declarations about the information to be exchanged when the plugin and interface are registered.

```
PIL_PLUGIN_BOILERPLATE("1.0", Debug,
                        md5closepi);

static const PILPluginImports*  PluginImports;
static PILPlugin*               OurPlugin;
static PILInterface*            OurInterface;
static void*                    OurImports;
static void*                    interfprivate;
```

Next comes the plugin initialization and registration function which gets called when the plugin is loaded. The `PIL_PLUGIN_INIT` macro gives the initialization function a name based on the plugin type and name, to avoid symbol clashes.

```
PIL_rc
PIL_PLUGIN_INIT(PILPlugin* us
,       const PILPluginImports* imports)
{
        /* Save away imports for later */
        PluginImports = imports;
        OurPlugin = us;

        /* Register ourselves as a plugin */
        imports->register_plugin(us
        ,    &OurPIExports);

        /*  Register an HAauth/md5 interface */
        return imports->register_interface(us
        ,       "HBauth" , "md5"
        ,       &md5ops
        ,       md5closeintf
        ,       &OurInterface
        ,       &OurImports
        ,       interfprivate);
}
```

This is the end of all the PILS-specific code. The real work of the plugin follows. Note the use of the imported `log()` function. This allows the plugin to use the same logging method as the application which loads it uses. The plugin neither knows nor cares how logging is to be done in the particular application in which it has been loaded.

```
/* Real work (should) happen here... */
```

```
static int
md5_auth_calc(const struct HBauth_info *t
,       const char * text, char * result
,       int resultlen)
{
        /* UhOh, No Code yet! <8-O */

        OurImports->log(PIL_FATAL
        ,   "UhOh! forgot to write code!");

        /*NOTREACHED*/
        /* Compute md5 authentication  */
        return 0;
}
static int
md5_auth_needskey(void)
{
       /* md5 authentication requires a key */
       return 1;
}
```

7 Plugin Usage Code

Plugin code doesn't need to be aware of which `InterfaceMgr` is managing it, but code that needs to access the loaded functions must be aware of how to interact with the interface manager, in order to be able to find the exported interfaces. For this example, the generic `InterfaceMgr` code is assumed.

First, declare variables to hold a reference to the plugin system, the loaded authentication functions, and some authentication information which the HBauth system needs.

```
/* Sample code ignores errors ;-) */

PILPluginUniv* PluginLoadingSystem = NULL;
GHashTable*    AuthFuncs = NULL;
char                 result[64];
struct HBAuthOps*    Auth;

struct HBauth_info   authinfo =
{NULL, "md5", "TopSecretKey!"};

PILGenericIfMgmtRqst RegisterRqsts[]= {
 {"HBauth", &AuthFuncs, NULL, NULL, NULL}
 { NULL, NULL, NULL, NULL, NULL}
};
```

Next, initialize the plugin system, telling it where to look to find plugins.

```
PluginLoadingSystem = NewPILPluginUniv
  ("/usr/lib/heartbeat/plugins");
```

Load the generic plugin manager, telling it (through `RegisterRqsts`) to update `Authfuncs` whenever an HBauth plugin is registered or unregistered.

```
PILLoadPlugin(PluginLoadingSystem
,   "InterfaceMgr", "generic"
,   &RegisterRqsts));
```

At this point, the plugin system is completely ready to go, and plugins can be loaded and unload at will.

```
PILLoadPlugin(PluginLoadingSystem
,   "HBauth", "md5", NULL);
```

Now, the plugin is loaded, and can be accessed. The generic plugin loader stashed a pointer to the interface the `AuthFuncs` `GHashTable`.

```
/* Get the interface for the "md5" plugin */
Auth = g_hash_table_lookup(AuthFuncs,"md5");

/* Compute signature and put it in 'result' */
Auth->auth(&authinfo, "ImportantStuffToSign"
,      result, sizeof(result));
```

When the authentication plugin is no longer needed, decrement the reference count, and the plugin will automatically be unloaded.

```
Auth = NULL;

PILIncrIFRefCount(
     PluginLoadingSystem
,    "HBauth", "md5", -1);
```

Although the code to prepare the application is somewhat more complex than the example of the plugin itself, most of this code won't be repeated for each plugin or each plugin type.

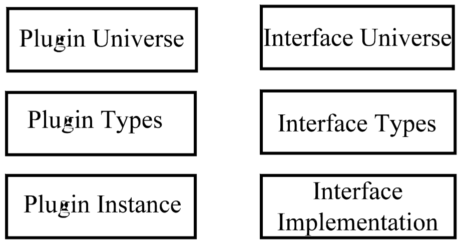


Figure 3: Layers of Abstractions in PILS

8 PILS Implementation Overview

8.1 PILS Data Relationships

PILS is written using the Glib [Glib] library, and makes extensive use of `GHashTable`s. There are basically two related abstraction stacks which PILS maintains: the Plugin Universe, and the Interface Universe. These are, in effect, parallel representations of related information, or for the more pun-minded, parallel universes. This is illustrated by Figure 3.

Each universe consists of a set of types, and each type contains a set of instances of the fundamental object (a Plugin or an Interface). Most of the work is keeping the relationships between the two layers and these two universes synchronized, so that it is known what interfaces were instantiated from any given plugin, and which plugin any particular interface was loaded from. There are a number of reference counts, and more complexity than one might expect.

8.2 InterfaceMgr: Managing Interfaces

From the perspective of PILS, the most interesting part of the world consists of interfaces. `InterfaceMgr` is PILS' name for the type of interface which is presented by plugins which manage interfaces.

Interfaces are where the variation and interesting behaviors are generally implemented. So, PILS implements an interface for the management of interfaces. This interface management function of PILS is believed to be unique. In PILS, the interface which manages interfaces is called the `InterfaceMgr` interface. So the `InterfaceMgr` interface which manages other the `InterfaceMgr` interfaces is the `InterfaceMgr/InterfaceMgr` implementation. The `InterfaceMgr/InterfaceMgr` is built-in (not dynamically loaded) since it is necessary for bootstrapping the dynamic loading management system. It loads and manages the other interface managers (including the generic interface manager).

Normally, the name of an interface manager is the same as the name of the type of interface it manages. Not so for the generic interface manager, which can manage any number of types of interfaces. When it is loaded, it is passed a parameter to tell it which types of interfaces it should manage. Since it can register any number of implementations, it then registers itself as the manager for each of these interface types it was passed when it started up.

9 Security Considerations

There are a few additional security considerations associated with plugin-enabled programs. Programs which use plugins to provide capabilities have more files and directories which need to be properly secured in order to ensure the application is not compromised. All programs have files and directories which must be properly secured in order for them to be secure, but software which uses plugins typically have a few more such directories and files. In addition to the location of the binary, and the normal libraries, the location of the plugins and the plugins themselves (and there may be many of them) must also be properly secured.

With an improperly secured system, and plugins which meet well-known interfaces, it is a very simple matter to create a plugin which meets the well-known interface, but which opens a wide security hole which can go completely undetected for a long period of time.

Plugins which provide security functions and which provide extremely simple interfaces (like the authentication example presented earlier) make extraordinarily tempting targets for intruders. It is prudent to assume that attackers *will* exploit such interfaces if they are improperly secured.

Plugins run in the address space of the loading program, so they can easily do any thing which the program itself has permissions to do. There is a difficult issue of trust associated with a collection of plugins which come from different sources.

It is necessary for software which uses plugins (whether from PILS or some other source) to ensure that they install their software in properly secured locations. Although some of the security enhancements described later will help this problem somewhat, the need to properly install and administer systems is still fundamental.

10 Status and Future Work

The current implementation of PILS is functional, and is currently used by the Linux-HA project and part of the OCF reference framework. It is currently generally available as part of the Linux-HA distribution [ha-dist], but is not currently available as a separate subpackage (but this will probably have occured by the publication of this paper). The source to PILS can also be directly viewed in CVS [ha-cvs].

Although PILS is a powerful tool providing a rich set of capabilities, the area of plugin management is broad and quite interesting, and PILS is in the early stages of its evolution. As a result, there are a number of needs which PILS does not fully satisfy. Since it is an open source project, all interested parties are invited to contribute these or other enhancements. The following is a list of features which are under consideration for the inclusion in future versions.

- **Aliases.** Add an alias capability. Although each plugin can provide more than one interface, the current implementation of the "tell me all the plugins which implement interface X", assumes that each plugin actually only implements its main interface. To remedy this limitation, it is desirable to add an alias capability.

  On many Linux filesystems, symbolic links could be used, but it is believed that even symbolic links would require some additional implementation effort and then they would be limited to being stored on filesystems which implement symbolic links.

- **PATH support.** Allow PATH-like searches for the location of plugins.

- **Porting.** Complete and verify the ports to other operating systems.

- **Default InterfaceMgr.** Add the ability for PILS to set an automatic default `InterfaceMgr`, rather than expecting the application to declare in advance which plugins they wish the current generic manager to manage.

- **InterfaceMgr management.** Extend the InterfaceMgr paradigm to add a new function to ask a particular InterfaceMgr to manage a particular Interface type.

- **InterfaceMgr interface** Add the ability to add new interfaces to manage after an interface manager is loaded. This is mainly for the generic interface manager, but may also be necessary for plugins whose interfaces have become inaccessable but whose plugin is still loaded. Note that this may overlap or interact with the previous item.

- **Security awareness.** Enhance PILS security awareness. For example, verify who owns plugins, plugin directories, and so forth.

- **Signed plugins.** Add support for cryptographic signatures for plugins.

- **Plugin licenses.** Add the license and licenseURL functions as standard member functions for plugins.

- **Independence.** If sufficient interest is shown, it would be good to make PILS a completely independent open source project. In any case, PILS needs to be a completely independent package which can be installed without any of the rest of the OCF software.

- **Gtype support.** Extend PILS to load GTK C types. If this were reasonable, then it might be a better solution than the C class loader which is scheduled to be released with the 2.0 version of Glib.

- **C++ class support.** Support loading of C++ classes.

- **Non-native languages.** Generalize the idea of plugins in such a way that PILS could also load plugins written in arbitrary languages like Perl, Python, or Java.
- **Interface version management.** There is currently no built-in capability or convention for managing interface versions (including the plugin interface).

11 Acknowledgments

Special thanks go to Neal McBurnett who contributed both to the early design stages of PILS, and also to the original design stages of the HBauth plugin which is used in the examples. Thanks also go out to Cliff White, Ramachandra Pai and Xiaoxiang Liu who spent time reviewing and critiquing the paper. The author also wishes to thank the developers of the Glib library, who have created an extraordinarily useful and functional C library. PILS development would have been much more difficult without Glib. The author notes that PILS is not a commercial product and this paper represents the views of the author, and does not necessarily represent the views of his employer.

12 Trademarks

References

[ha-cvs] *PILS CVS Repository*, Robertson, et al, `http://cvs.linux-ha.org /viewcvs/viewcvs.cgi/linux-ha /lib/pils/`, `http://cvs.linux-ha.org /viewcvs/viewcvs.cgi/linux-ha /include/pils/`

[dlopen] *Linux dlopen(3) manual page*, Linux community. `http://www.freebsd.org/cgi /man.cgi?query=dlopen &apropos=0&sektion=0& format=html&manpath=SuSE+ Linux%2Fi386+7.3`

[Glib] *Glib Reference Manual*, Gnome Project. `http://developer.gnome.org/doc /API/glib/index.html`

[ha-dist] *High-Availability Linux Distribution*, Robertson, et al, High-Availability Linux Project. `http://linux-ha.org/download/`

[libtool] *Libtool Reference Manual*, Free Software Foundation. `http://www.gnu.org/software /libtool/manual.html`

[linux-ha] *High-Availability Linux Home Page*, Robertson, et al, High-Availability Linux Project. `http://linux-ha.org/`

[OCF] *Open Cluster Framework Project Home Page*, Robertson, et al, `http://opencf.org/`

Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux

Hubertus Franke
IBM Thomas J. Watson Research Center
frankeh@watson.ibm.com

Rusty Russell
IBM Linux Technology Center
rusty@rustcorp.com.au

Matthew Kirkwood
matthew@hairy.beasts.org

Abstract

Fast userlevel locking is an alternative locking mechanism to the typically heavy weight kernel approaches such as fcntl locking and System V semaphores. Here, multiple processes communicate locking state through shared memory regions and atomic operations. Kernel involvement is only necessary when there is contention on a lock, in order to perform queueing and scheduling functions. In this paper we discuss the issues related to user level locking by following the history of ideas and the code to the current day. We present the efficacy of "futexes" through benchmarks, both synthetic and through adaptations to existing databases. We conclude by presenting the potential future directions of the "futex" interface.

1 Introduction

Linux™[1] has seen significant growth as a server operating system and has been successfully deployed in enterprise environments for Web, file and print serving. With the deployment of Version 2.4, Linux has seen a tremendous boost in scalability and robustness that makes it now feasible to deploy even more demanding enterprise applications such as high end databases, business intelligence software and application servers. As a result, whole enterprise business suites and middleware such as SAP™, Websphere™, Oracle, DB2™[2], etc., are now available for Linux.

For these enterprise applications to run efficiently on Linux, or on any other operating system for that matter, the OS must provide the proper abstractions and services. Enterprise applications and applications suites are increasingly built as multi process / multithreaded applications. Multi-threaded applications can take better advantage of SMP hardware, while multiple processes allows for higher degrees of fault tolerance, i.e., a single process abort does not necessarily bring the entire application down. Furthermore, applications suites are often a collection of multiple independent subsystems.

Despite their functional separation, the processes representing these subsystems often must communicate with each other and share state amongst each other. Examples of this are database systems, which typically maintain shared I/O buffers in user space. The buffers

[1]Linux is a trademark of Linus Torvalds

[2]All third party trademarks are the property of their respective owners.

are concurrently accessed by various database engines and prefetching processes.

Access to such shared state must be properly synchronized through either exclusive or shared locks. Exclusive locks allow only one party access to the protected entity, while shared locks allow multiple reader – single writer semantics. Synchronization implies a shared state, indicating that a particular resource is available or busy, and a means to wait for its availability. The latter one can either be accomplished through busy-waiting or through a explicit / implicit call to the scheduler.

In traditional UNIX™ [3] systems, System V IPC (inter process communication) such as *semaphores*, *msgqueues*, *sockets* and the file locking mechanism (`flock()`) are the basic mechanisms for two processes to synchronize. These mechanisms expose an opaque handle to a kernel object that naturally provides the shared state and atomic operations in the kernel. Services must be requested through system calls (e.g., `semop()`). The drawback of this approach is that every lock access requires a system call. When locks have low contention rates, the system call can constitute a significant overhead.

One solution to this problem is to deploy user level locking, which avoids some of the overhead associated with purely kernel-based locking mechanisms. It relies on a user level lock located in a shared memory region and modified through atomic operations to indicate the lock status. Only the contended case requires kernel intervention. The exact behavior and the obtainable performance are directly affected by how and when the kernel services are invoked. The idea described here is not new. Some of the foundation of this paper are described in [4], [7] and [6]. In [2] the impact of locking on JVM performance is discussed.

[3]UNIX is a trademark of The Open Group

In this paper we are describing a particular fast user level locking mechanism called *futexes* that was developed in the context of the Linux operating system. It consists of two parts, the user library and a kernel service that has been integrated into the Linux kernel distribution version 2.5.7.

The paper is organized as followed. In section 2 we describe the basic behavioral and functional requirements of a user level locking mechanism. In section 3 we describe some of the earlier approaches that led to the current design of *futexes* and the futexes themselves. In section 4 we provide a performance assessment on a synthetic and a database benchmark. In section 5 we elaborate on current and future efforts and in 6 we conclude.

2 Requirements

In this section we are stating some of the requirements of a fast userlevel locking mechanism that we derived as part of this work and that were posted to us as requirements by middleware providers.

There are various behavioral requirements that need to be considered. Most center around the fairness of the locking scheme and the lock release policy. In a **fair** locking scheme the lock is granted in the order it was requested, i.e., it is handed over to the longest waiting task. This can have negative impact on throughput due to the increased number of context switches. At the same time it can lead to the so called **convoy problem**. Since, the locks are granted in the order of request arrival, they all proceed at the speed of the slowest process, slowing down all waiting processes. A common solution to the convoy problem has been to mark the lock available upon release, wake all waiting processes and have them recontend for the lock. This is referred to as **random fairness**, although higher priority tasks will usually have an advantage over lower priority ones. However, this also leads to the well known **thundering herd problem**. Despite this, it can work quite well on uni-processor systems if the first task to wake releases the lock before being preempted or scheduled, allowing the second herd member to obtain the lock, etc. It works less spectacularly on SMP. To avoid this problem, one should only wake up one waiting task upon lock release. Marking the lock available as part of releasing it, gives the releasing task the opportunity to reacquire the lock immediately again, if so desired, and avoid unnecessary context switches and the convoy problem. Some refer to these as **greedy**, as the running task has the highest probability of reacquiring the lock if the lock is hot. However, this can lead to starvation. Hence, the basic mechanisms must enable both fair locking, random locking and greedy or convoy avoidance locking (short ca-locking). Another requirement is to enable spin locking, i.e., have an application spin for the availablilty of the lock for some user specified time (or until granted) before giving up and resolving to block in the kernel for its availability. Hence an application has the choice to either (a) block waiting to be notified for the lock to be released, or (b) yield the processor until the thread is rescheduled and then the lock is tried to be acquired again, or (c) spin consuming CPU cycles until the lock is released.

With respect to performance, there are basically two overriding goals:

- avoid system calls if possible, as system calls typically consume several hundred instructions.

- avoid unnecessary context switches: context switches lead to overhead associated with TLB invalidations etc.

Hence, in fast userlevel locking, we first distinguish between the uncontended and the contended case. The uncontended case should be efficient and should avoid system calls by all means. In the contended case we are willing to perform a system call to block in the kernel.

Avoiding system calls in the uncontended case requires a shared state in user space accessible to all participating processes/task. This shared state, referred to as the *user lock*, indicates the status of the lock, i.e., whether the lock is held or not and whether there are waiting tasks or not. This is in contrast to the System V IPC mechanisms which merely exports a handle to the user, and performs all operations in the kernel.

The user lock is located in a shared memory region that was create via `shmat()` or `mmap()`. As a result, it can be located at different virtual addresses in different address spaces. In the uncontended case, the application atomically changes the lock status word without entering into the kernel. Hence, atomic operations such as `atomic_inc()`, `atomic_dec`, `cmpxchg()`, and `test_and_set()` are neccessary in user space. In the contended case, the application needs to wait for the release of the lock or needs to wake up a waiting task in the case of an unlock operation. In order to wait in the kernel, a *kernel object* is required, that has *waiting queues* associated with it. The waiting queues provide the queueing and scheduling interactions. Of course, the aforementioned IPC mechanisms can be used for this purpose. However, these objects still imply a heavy weight object that requires a priori allocation and often does not precisely provide the required functionality. Another alternative that is commonly deployed are *spinlocks* where the task spins on the availability of the user lock until granted. To avoid too many cpu cycles being wasted, the task yields the processor occasionally.

It is desirable to have the user lock be handle-free. In other words instead of handling an oqaque *kernel handle*, requiring initialization and cross process global handles, it is desirable to address locks directly through their virtual address. As a consequence, kernel objects can be allocated dynamically and on demand, rather than apriori.

A lock, though addressed by a virtual address, can be identified conceptually through its *global lock identity*, which we define by the memory object backing the virtual address and the offset within that object. We notate this by the tuple [B,O]. Three fundamental memory types can be distinguished that represent B: (a) anonymous memory, (b) shared memory segment, and (c) memory mapped files. While (b) and (c) can be used between multiple processes, (a) can only be used between threads of the same process. Utilizing the virtual address of the lock as the kernel handle also provides for an integrated access mechanism that ties the virtual address automatically with its kernel object.

Despite the atomic manipulation of the user level lock word, race conditions can still exists as the sequence of lock word manipulation and system calls is not atomic. This has to be resolved properly within the kernel to avoid deadlock and inproper functioning.

Another requirement is that fast user level locking should be simple enough to provide the basic foundation to efficiently enable more complicated synchronization constructs, e.g. semaphores, rwlocks, blocking locks, or spin versions of these, pthread mutexes, DB latches. It should also allow for a clean separation of the blocking requirements towards the kernel, so that the blocking only has to be implemented with a small set of different constructs. This allows for extending the use of the basic primitives without kernel modifications. Of interest is the implementation of mutex, semaphores and multiple reader/single writer locks.

Finally, a solution needs to be found that enables the recovery of "dead" locks. We define unrecoverable locks as those that have been acquired by a process and the process terminates without releasing the lock. There are no convenient means for the kernel or for the other processes to determine which locks are currently held by a particular process, as lock acquisition can be achieved through user memory manipulation. Registering the process's "pid" after lock acquisition is not enough as both operations are not atomic. If the process dies before it can register its pid or if it cleared its pid and before being able the release the lock, the lock is unrecoverable. A protocol based solution to this problem is described in [1]. We have not addressed this problem in our prototypes yet.

3 Linux Fast User level Locking: History and Implementations

Having stated the requirements in the previous section, we now proceed to describe the basic general implementation issues. For the purpose of this discussion we define a general opaque datatype `ulock_t` to represent the userlevel lock. At a minimum it requires a status word.

```
typedef struct ulock_t {
        long status;
} ulock_t;
```

We assume that a shared memory region has been allocated either through `shmat()` or through `mmap()` and that any user locks are allocated into this region. Again note, that the addresses of the same lock need not be the same across all participating address spaces.

The basic semaphore functions `UP()` and `DOWN()` can be implemented as follows.

```
static inline int
usema_down(ulock_t *ulock)
{
    if (!__ulock_down(ulock))
            return 0;
    return sys_ulock_wait(ulock);
}

static inline int
usema_up(ulock_t *ulock)
{
    if (!__ulock_up(ulock))
            return 0;
    return sys_ulock_wakeup(ulock);
}
```

The `__ulock_down()` and `__ulock_up()` provide the atomic increment and decrement operations on the lock status word. A non positive count (status) indicates that the lock is not available. In addition, a negative count *could* indicate the number of waiting tasks in the kernel. If a contention is detected, i.e. `(ulock->status <= 0)`, the kernel is invoked through the `sys_*` functions to either wait on the wait queue associated with `ulock` or release a blocking task from said waitqueue.

All counting is performed on the lock word and race conditions resulting from the non-atomicity of the lock word must be resolved in the kernel. Due to such race conditions, a lock can receive a wakeup before the waiting process had a chance to enqueue itself into the kernel wait queue. We describe below how various implementation resolved this race condition as part of the kernel service.

One early design suggested was the explicit allocation of a kernel object and the export of the kernel object address as the handle. The kernel object was comprised of a wait queue and a unique security signature. On every wait or wakeup call, the signature would be verified to ensure that the handle passed indeed was referring to a valid kernel object. The disadvantages of this approach have been mentioned in section 2, namely that a handle needs to be stored in `ulock_t` and that explicit allocation and deallocation of the kernel object are required. Furthermore, security is limited to the length of the key and hypothetically could be guessed.

Another prototype implementation, known as *ulocks* [3], implements general user semaphores with both fair and convoy avoidance wakeup policy. Mutual exclusive locks are regarded as a subset of the user semaphores. The prototype also provides multiple reader/single writer locks (rwlocks). The user lock object `ulock_t` consists of a lock word and an integer indicating the required number of kernel wait queues. User semaphores and exclusive locks are implemented with one kernel wait queue and multiple reader/single writer locks are implemented with two kernel wait queues.

This implementation separates the lock word from the kernel wait queues and other kernel objects, i.e., the lock word is never accessed from the kernel on the time critical wait and wakeup code path. Hence the state of the lock and the number of waiting tasks in the kernel is all recorded in the lock word. For exclusive locks, standard counting as described in the general `ulock_t` discussion, is implemented. As with general semaphores, a positive number indicates the number of times the semaphore can be acquired, "0" and less indicates that the lock is busy, while the absolute of a negative number indicates the number of waiting tasks in the kernel.

The "premature" wakeup call is handled by implementing the kernel internal waitqueues using kernel semaphores (`struct semaphore`) which are initialized with a value 0. A premature wakeup call, i.e. no pending waiter yet, simply increases the kernel semaphore's count to 1. Once the pending wait arrives it simply decrements the count back to 0 and exits the system call without waiting in the kernel. All the wait queues (kernel semaphores) associated with a user lock are encapsulated in a single kernel object.

In the rwlocks case, the lock word is split into three fields: write locked (1 bit), writes waiting (15 bits), readers (16 bits). If write locked, the `readers` indicate the number of tasks waiting to read the lock, if not write locked, it indicates the numbers of tasks that have acquired read access to the lock. Writers are blocking on a first kernel wait queue, while readers are blocking on a second kernel wait queue associated with a ulock. To wakeup multiple pending read requests, the number of task to be woken up is passed through the system call interface.

To implement rwlocks and ca-locks, atomic compare and exchange support is required. Unfortunately on older 386 platforms that is not the case.

The kernel routines must identify the kernel object that is associated with the user lock. Since the lock can be placed at different virtual addresses in different processes, a lookup has to be performed. In the common fast lookup, the virtual address of the user lock and the address space are hashed to a kernel object. If no hash entry exists, the proper global identity $[B, O]$ of the lock must be established. For this we first scan the calling process's vma list for the vma containing the lock word and its offset. The global identity is then looked up in a second hash table that links global identities with their associated kernel object. If no kernel object exists for this global identity, one is allocated, initialized and added to the hash functions. The `close()` function associated with a shared region holding kernel objects is intercepted, so that kernel objects are deleted and the hash tables are cleaned up, once all attached processes have detached from the shared region.

While this implementation provides for all the requirements, the kernel infrastructure of multiple hash tables and lookups was deemed too heavy. In addition, the requirement for compare and exchange is also seen to be restrictive.

3.1 Futexes

With several independent implementations [8, 9, 10] in existence, the time seemed right in early 2002 to attempt to combine the best elements of each to produce the minimum useful subset for insertion into the experimental Linux kernel series.

There are three key points of the original futex implementation which was added to the 2.5.7 kernel:

1. We use a unique identifier for each futex (which can be shared across different address spaces, so may have different virtual addresses in each): this identifier is the "struct page" pointer and the offset within that page. We increment the reference count on the page so it cannot be swapped out while the process is sleeping.

2. The structure indicating which futex the process is sleeping on is placed in a hash table, and is created upon entry to the futex syscalls on the process's kernel stack.

3. The compression of "*f*ast *u*serspace mu*tex*" into "*futex*" gave a simple unique identifier to the section of code and the function names used.

3.1.1 The 2.5.7 Implementation

The initial implementation which was judged a sufficient basis for kernel inclusion used a single two-argument system call, `“sys_futex(struct futex *, int op)”`. The first argument was the address of the futex, and the second was the operation, used to furthur demultiplex the system call and insulate the implementation somewhat from the problems of system call number allocation. The latter is especially important as the system call is expand as new operations are required. The two valid op numbers for this implementation were `FUTEX_UP` and `FUTEX_DOWN.`

The algorithm was simple, the file *linux/kernel/futex.c* containing 140 code lines, and 233 in total.

1. The user address was checked for alignment and that it did not overlap a page boundary.
2. The page is pinned: this involves looking up the address in the process’s address space to find the appropriate “`struct page *`”, and incrementing its reference count so it cannot be swapped out.
3. The “`struct page *`” and offset within the page are added, and that result hashed using the recently introduced fast multiplicative hashing routines [11], to give a hash bucket in the futex hash table.
4. The “op” argument is then examined. If it is `FUTEX_DOWN` then:
   (a) The process is marked *INTERRUPTIBLE*, meaning it is ready to sleep.
   (b) A “`struct futex_q`” is chained to the tail of the hash bucket determined in step 3: the tail is chosen to give FIFO ordering for wakeups. This structures contains a pointer to the process and the “`struct page *`” and offset which identify the futex uniquely.
   (c) The page is mapped into low memory (if it is a high memory page), and an atomic decrement of the futex address is attempted,[4] then unmapped again. If this does not decrement the counter to zero, we check for signals (setting the error to `EINTR` and going to the next step), schedule, and then repeat this step.
   (d) Otherwise, we now have the futex, or have received a signal, so we mark this process *RUNNING*, unlink ourselves from the hash table, and wake the next waiter if there is one, and return `0` or `-EINTR`. We have to wake another process so that it decrements the futex to -1 to indicate that it is waiting (in the case where we have the futex), or to avoid the race where a signal came in just as we were woken up to get the futex (in the case where a signal was received).
5. If the op argument was `FUTEX_UP:`
   (a) Map the page into low memory if it is in a high memory page
   (b) Set the count of the futex to one (“available”).
   (c) Unmap the page if it was mapped from high memory

[4]We do not even attempt to decrement the address if it is already negative, to avoid potential wraparound. We do the decrement even if the counter is zero, as “-1” indicates we are sleeping and hence has different semantics than 0.

   (d) Search the hash table for the first "`struct futex_q`" associated with this futex, and wake up that process.

6. Otherwise, if the op argument is anything else, set the error to EINVAL.

7. Unpin the page.

While there are several subtleties in this implementation, it gives a second major advantage over System V semaphores: there are no explicit limits on how many futexes you can create, nor can one futex user "starve" other users of futexes. This is because the futex is merely a memory location like any other until the `sys_futex` syscall is entered, and each process can only do one `sys_futex` syscall at a time, so we are limited to pinning one page per process into memory, at worst.

3.1.2 What about Read-Write Locks?

We considered an implementation of "FUTEX_READ_DOWN" et. al, which would be similar to the simple mutual exclusion locks, but before adding these to the kernel, Paul Mackerras suggested a design for creating read/write lock in userspace by using two futexes and a count: *f*ast *u*serspace *r*ead/*w*rite l*ocks*, or *furwocks*. This implementation provides the benchmark for any kernel-based implementation to beat to justify its inclusion as a first-class primitive, which can be done by adding new valid "op" values. A comparision with the integrated approach chosen by ulocks is provided in Section 4.

3.1.3 Problems with the 2.5.7 Implementation

Once the first implementation entered the mainstream experimental kernel, it drew the attention of a much wider audience. In particular those concerned with implementing POSIX(tm)[5] threads, and attention also returned to the fairness issue.

- There is no straightforward way to implement the pthread_cond_timedwait primitive: this operation requires a timeout, but using a timer is difficult as these must not interfere with their use by any other code.
- The pthread_cond_broadcast primitive requires every process sleeping to be woken up, which does not fit well with the 2.5.7 implementation, where a process only exits the kernel when the futex has been successfully obtained or a signal is received.
- For N:M threading, such as the Next Generation Posix Threads project [5] an asynchronous interface for finding out about the futex is required, since a single process (containing multiple threads) might be interested in more than one futex.
- Starvation occurs in the following situtation: a single process which immediately drops and then immediately competes for the lock will regain it before any woken process will.

With these limitations brought to light, we searched for another design which would be flexible enough to cater for these diverse needs. After various implemenation attempts and discussions we settled on a variation of *atomic_compare_and_swap* primitive, with

[5]POSIX is a trademark of the IEEE Inc.

the atomicity guaranteed by passing the expected value into the kernel for checking. **?** To do this, two new "op" values replaced the operations above, and the system call was changed to two additional arguments, "int val" and "struct timespec *reltime".

FUTEX_WAIT: Similar to the previous FUTEX_DOWN, except that the looping and manipulation of the counter is left to userspace. This works as follows:

1. Set the process state to *INTERRUPTIBLE*, and place "struct futex_q" into the hash table as before.
2. Map the page into low memory (if in high memory).
3. Read the futex value.
4. Unmap the page (if mapped at step 2).
5. If the value read at step 3 is not equal to the "val" argument provided to the system call, set the return to `EWOULDBLOCK`.
6. Otherwise, sleep for the time indicated by the "reltime" argument, or indefinitely if that is NULL.
   (a) If we timed out, set the return value to `ETIMEDOUT`.
   (b) Otherwise, if there is a signal pending, set the return value to `EINTR`.
7. Try to remove our "`struct futex_q`" from the hash table: if we were already removed, return 0 (success) unconditionally, as this means we were woken up, otherwise return the error code specified above.

FUTEX_WAKE: This is similar to the previous `FUTEX_UP`, except that it does not alter the futex value, it simple wakes one (or more) processes. The number of processes to wake is controlled by the "int val" parameter, and the return value for the system call is the number of processes actually woken and removed from the hash table.

FUTEX_AWAIT: This is proposed as an asynchronous operation to notify the process via a SIGIO-style mechanism when the value changes. The exact method has not yet been settled (see future work in Section 5).

This new primitive is only slightly slower than the previous one,[6] in that the time between waking the process and that process attempting to claim the lock has increased (as the lock claim is done in userspace on return from the FUTEX_WAKE syscall), and if the process has to attempt the lock multiple times before success, each attempt will be accompanied by a syscall, rather than the syscall claiming the lock itself.

On the other hand, the following can be implemented entirely in the userspace library:

1. All the POSIX style locks, including pthread_cond_broadcast (which requires the "wake all" operation) and pthread_cond_timedwait (which requires the timeout argument). One of the authors (Rusty) has implemented a "non-pthreads" demonstration library which does exactly this.

2. Read-write locks in a single word, on architectures which support cmpxchg-style primitives.

[6]About 1.5% on a low-contention tdbtorture, 3.5% on a high-contention tdbtorture

3. FIFO wakeup, where fairness is guaranteed to anyone waiting (see 3.1.4).

Finally, it is worthwhile pointing out that the kernel implementation requires exactly the same number of lines as the previous implementation: 233.

3.1.4 FIFO Queueing

The naive implementation of "up" does the following:

1. Atomically set the futex to 1 ("available") and record the previous value.
2. If the previous value was negative, invoke sys_futex to wake up a waiter.

Now, there is the potential for another process to claim the futex (without entering the kernel at all) between these two steps: the process woken at step 2 will then fail, and go back to sleep. As long as this does not lead to starvation, this unfairness is usually tolerable, given the performance improvements shown in Section 4

There is one particular case where starvation is a real problem which must be avoided. A process which is holding the lock for extended periods and wishes to "give way" if others are waiting cannot simple to "futex_up(); futex_down();", as it will always win the lock back before any other processes.

Hence one of us (Hubertus) added the concept of "`futex_up_fair()`", where the futex is set to an extremely negative number ("passed"), instead of 1 ("available"). This looks like a "contended" case to the fast userspace "futex_down()" path, as it is negative, but indicates to any process after a successful return from the `FUTEX_WAIT` call that the futex has been passed directly, and no further action (other than resetting the value to -1) is required to claim it.

4 Performance Evaluation

In this section we assess the performance of the current implementation. We start out with a synthetic benchmark and continue with a modified database benchmark.

4.1 MicroBenchmark: UlockFlex

Ulockflex is a synthetic benchmark designed to ensure the integrity and measure the performance of locking primitives. In a run, *Ulockflex* allocates a finite set (typically one) of global shared regions (shmat or mmap'ed files) and a specified number of user locks which are assigned to the shared region in a round robin fashion. It then clones a specified number of tasks either as threads or as processes and assigns each task to one particular lock in a round robin fashion. Each cloned task, in a tight loop, computes two random numbers $nlht$ and lht, acquires its assigned lock, does some work of lock hold time lht, releases the lock, does some more work of non-lock hold time $nlht$ and repeats the loop. The mean lock hold time $lht(mean)$ and non-lock hold times $nlht(mean)$ are input parameters. lht and $nlht$ are determined as random numbers over a uniform distribution in the interval $[0.5..1.5]$ of their respective mean. The tool reports total cummulative throughput (as in number of iterations through the loop). It also reports the coefficient of variance of the per task througput. A higher coefficient indicates the potential for starvation. A small coefficient indicates fairness over the period of execution. A data structure associated with each lock is updated after obtaining the lock and verified before releasing the lock, thus allowing for integrity checks.

In the following we evaluate the performance of various user locking primitives that were built on the basics of the futex and the ulock implementations. We consider the basic two wakeup policies for both futexes and ulocks, i.e. fair wakeup and regular wakeup (i.e. convoy avoidance), yielding the 4 cases *futex_fair, futex, ulocks_fair* and *ulocks*. For these cases we also consider a spinning lock acquisition in that the task tries to acquire the lock for 3 $\mu secs$ before giving up and blocking in the kernel, yielding the 4 cases of *futex_fair(spin,3), futex(spin,3), ulocks_fair(spin,3)* and *ulocks(spin,3)*. For reference we also provide the measurements for a locking mechanism build on System V semaphores, i.e., each lock request results in a system call. This variant is denoted as *sysv*, resulting in 9 overall locking primitives being evaluated.

All experiments were performed on a dual Pentium-III 500 MHz, 256MB system. A data point was obtained by running ulockflex for 10 seconds with a minimum of 10 runs or until a 95% confidence interval was achieved.

In the first experiment we determine the basic overhead of the locking mechanims. For this we run with one task, one lock and $nlht == lht == 0$. Note that in this case all user locking mechanisms never have to enter into the kernel. Performance is reported as % efficiency of a run without lock invocations. The *sysv* was 25.1% efficient, while all 8 user level locking cases fell within 84.6% and 87.9%. When the $(nlht+lht)$ was increased to 10$\mu secs$, the efficiency of *sysv* was still only 82.2%, while those of the user level locks ranged from 98.9% to 99.1%.

When executing this setup with two tasks and two locks the efficiency of *sysv* drops to 18.3% from 25.1% indicating a hot lock in the kernel. At the same time the user level primitives all remain in the same range, as expected. The same effect can be described as follows. With this setup we would expect twice the throughput performance as compared to the 1 task, 1 lock setup. Indeed, for all user primitives the scalability observed is between 1.99 and 2.02, while *sysv* only shows a scalability of 1.51.

In the next set of experiments we fixed the total loop execution time $nlht + lht$ to 10$\mu secs$, however we changed the individual components. Let $(nlht, lht)$ denote a configuration. Four configuration are observed: (0,10), (5,5), (7,3), (9,1). The (0,10) represents the highly contended case, while (9,1) represents a significantly less contended case. The exact contention is determined by the number of tasks accessing a shared lock. Contention numbers reported are all measured against the fair locking version of ulocks in a separate run. The contention measurement does not introduce any significant overhead.

Figures 1..5 show the comparision of the 9 locking primitives for the four configurations under various task counts (2,3,4,100,1000). The percentage improvements for each configuration and task count over the *sysv* base number for that configuration are reported in Table 1 for the fair futexes and ulocks without and with spinning (3 $\mu secs$) and in Table 2 for the regular futexes and ulocks.

The overall qualitative assessment of the results presented in these figures and tables is as follows. First comparing the fair locking mechanisms, fair ulocks, in general, have an advantage over fair futexes. Furthermore, fair futexes perform worse than *sysv* for high contention scenarios. Only in the high task count numbers do fair futexes outperform (substantially) *sysv* and fair ulocks. Spinning only showed some decent improvement in the low contention cases, as expected. For the regular versions (ca-locks), both futexes and ulocks

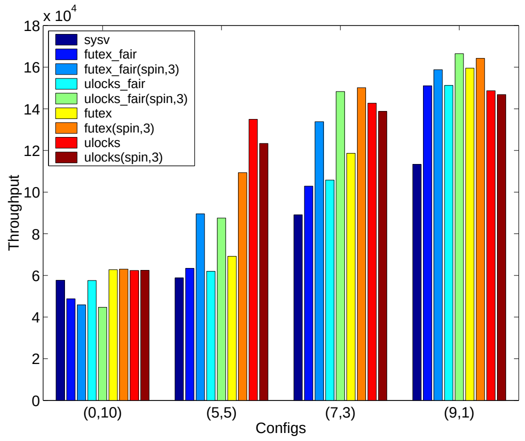


Figure 1: Throughput for various lock types for 2 tasks, 1 lock and 4 configurations

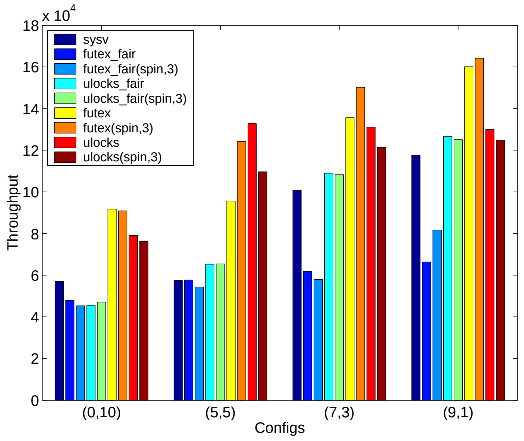


Figure 3: Throughput for various lock types for 4 tasks, 1 lock and 4 configurations

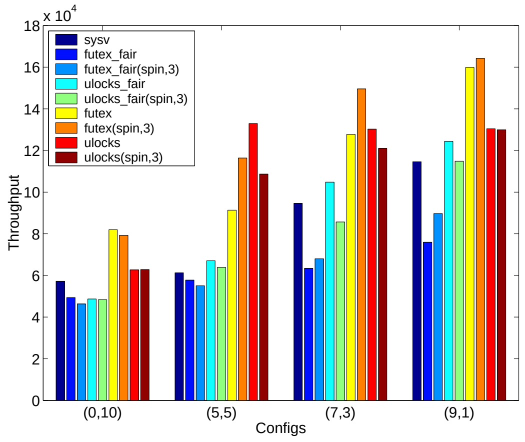


Figure 2: Throughput for various lock types for 3 tasks, 1 lock and 4 configurations

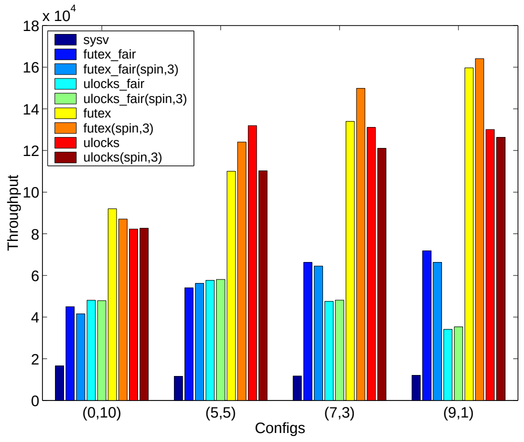


Figure 4: Throughput for various lock types for 100 tasks, 1 lock and 4 configurations

always outperform the *sysv* version. The general tendency is for ulocks to achieve their performance at the `(5,5)` configuration with little additional benefits. Though futexes in general lack the ulock performance at the `(5,5)` configuration, they outperform ulocks at the `(7.3)` and the `(9,1)` configurations. In contrast to futexes, spinning for ulocks does not help.

Figure 1 shows the results for 2 tasks competing for 1 lock under four contention scenarios. The lock contention for the 4 configurations were 100%, 97.8%, 41.7% and 13.1%. The lock contention observed for Figure 2.. 5 are all above 99.8%.

We now turn our attention to the multiple reader/single writer (rwlock) lock primitives. To recall, furwocks implement the rwlock functionality ontop of two regular futexes, while ulocks implement them directly in the interface through atomic compare and exchange manipulation of the lock word. *Ulockflex* allows the specification of a `share-level` for rwlocks. This translates into the probability of a task requesting a read lock instead of a write lock while iterating through the tight loop.

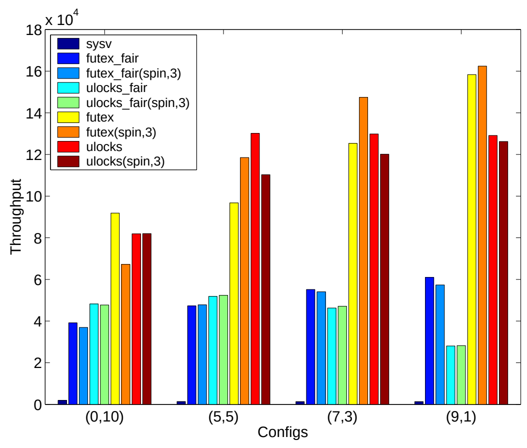


Figure 5: Throughput for various lock types for 1000 tasks, 1 lock and 4 configurations

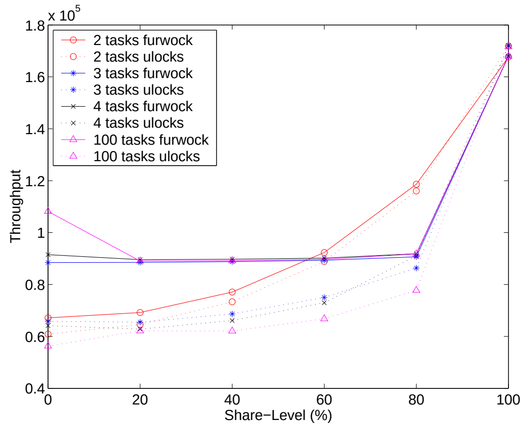


Figure 6: Throughput of furwocks and shared ulocks for (2,3,4,100) tasks competing for a single lock under different read share ratios

Figure 6 shows the achieved throughput of furwocks and shared ulocks for 2, 3, 4 and 100 tasks competing for a single lock under different read share ratios. The general observation is that the furwocks (solid lines) outperform the ulocks (dashed lines) for their respective task numbers. In general the lower the share level and/or the higher the task numbers the better the improvements that can be achieved with furwocks over shared ulocks. Only in the 100% share-level (only read accesses) do shared ulocks outperform furwocks by $\tilde{2}$-3%.

We now analyze the fairness of the user locking. We monitor the global fairness by computing the coefficient of variance *coeff* of the per task throughput. Note this should not be compared with the fair locking itself. The *coeff* of *sysv* is typically below 0.01. Only the 1000 task case showed a *coeff* of 9.1, indicating that tasks did not all properly get started. The *coeff* for fair futexes and fair ulocks for small task numbers (2,3,4) is in general below 0.01 (as expected). For large task number (100,1000), the *coeff* remains very low for futexes, while ulocks experience a *coeff* as high as 1.10. For furwocks, the general observation is that the *coeff* is less than 0.16 in both furwocks and shared ulocks. Only for the 100 task case does the *coeff* reach 0.45. Overall the mean of coeff for all scenarios is 0.068 for furwocks and 0.054 for shared ulocks. In general we can state that at these level of contention, global starvation is not a problem.

We now turn our attention to the degree of local fairness for the ca-locks. We do this by investigating how many times a task is capable of reacquiring the lock before some other task locks it. To do so, we examine a high contention case of 100 tasks and the (9,1) configuration. The kernel lock and the fair futexes showed perfect fairness, 99.99% of the task could never reacquire its lock without losing it to some other task. The fair ulocks only 92.1% failed to reacquire, 3.6% was able to grab the lock twice in a row and 0.4% three times. The maximum times a lock was able to be reacquired was 1034 times. For futexes these numbers are 79.0%, 21.0% and maximum of 575 and for ulocks they are 82.4%, 17.54% and maximum of 751. To some degree it confirms that futexes and ulocks have a higher degree of instant reacquisition, however this analysis fails to shed more light on why futexes are so much better than ulocks.

4.2 TDB Torture Results

The Trivial DataBase (TDB) is a simple hash-chain-based on-disk database used by SAMBA and other projects to store persistent internal data. It has a similar interface to the classic dbm library, but allows multiple readers and writers and is less than 2000 lines long. TDB normally uses fcntl locks: we replaced these with futex locks in a special part of the memory-mapped file. We also examined an implementation using "spin then yield" locks, which try to get the lock 1000 times before calling yield() to let other processes schedule.

tdbtorture is one of the standard test programs which comes with TDB: we simplified it to eliminate the cleanup traversal which it normally performs, resulting in a benchmark which forks 6 processes, each of which does 200000 random search/add/delete/traverse operations.

To examine behavior under high contention, we created a database with only one hash chain, giving only two locks (there is one lock for the free records chain). For the low contention case, we used 4096 chains (there is still some contention on the allocation lock). For the no contention case, we used a single process, rather than 6. The results shown in Table 3 were obtained on a 2-processor 350MHz Pentium II.

It is interesting that the fcntl locks have different scaling properties than futexes: they actually do much worse under the low contention case, possibly because the number of locks the kernel has to keep track of increases.

Another point to make here is the simplicity of the transformation from fcntl locks to futexes within TDB: the modification took no longer than five minutes to someone familiar with the code.

5 Current and Future Directions

Currently we are evaluating an asynchronous wait extension to the futex subsystem. The requirement for this arises for the necessity to support global POSIX mutexes in thread packages. In particular, we are working with the NGPT (next generation pthreads) team to derive specific requirements for building global POSIX mutexes over futexes. Doing so provides the benefit that in the uncontended case, no kernel interactions are required. However, NGPT supports a $M : N$ threading model, i.e., M user level threads are executed over N tasks. Conceptually, the N tasks provide virtual processors on which the M user threads are executing.

When a user level thread, executing on one of these N tasks, needs to block on a futex, it should not block the task, as this task provides the virtual processing. Instead only the user thread should be descheduled by the thread manager of the NGPT system. Nevertheless, a `waitobj` must be attached to the waitqueue in the kernel, indicating that a user thread is waiting on a particular futex and that the task group needs a notification wrt to the continuation on that futex. Once the thread manager receives the notification it can reschedule the previously blocked user thread.

For this we provide an additional operator `AFUTEX_WAIT` to the `sys_futex` system call. Its task is to append a *waitobj* to the futex's kernel waitqueue and continue. Compared to the synchronous calls described in Section 3, this `waitobj` can not be allocated on the stack and must be allocated and deallocated dynamically. Dynamic allocations have the disadvantage that the `waitobjs` must be freed even during an irregular program exit. It further poses a denial of service attack threat in that a malicious applications can continously call `sys_futex(AFUTEX_WAIT)`. We are contemplating various solutions to this problem.

The general solutions seem to convert to the usage of a */dev/futex* device to control resource consumption. The first solution is to allocate a file descriptor *fd* from the */dev/futex* "device" for each outstanding asynchronous `waitobj`. Conveniently these descriptors should be "pooled" to avoid the constant opening and closing of the device. The private data of the file would simply be the `waitobj`. Upon completion a SIGIO is sent to the application. The advantage of this approach is that the denial of service attack is naturally limited to the file limits imposed on a process. Furthermore, on program death, all `waitobjs` still enqueued can be easily dequeued. The disadvantage is that this approach can significantly pollute the "fd' space. Another solution proposed has been to open only one *fd*, but allow multiple `waitobj` allocations for this *fd*. This approach removes the fd space pollution issue but requires an additional tuning parameter for how many outstanding `waitobjs` should be allowed per fd. It also requires proper resource management of the `waitobjs` in the kernel. At this point no definite decisions has been reached on which direction to proceed.

The question of priorities in futexes has been raised: the current implementation is strictly FIFO order. The use of nice level is almost certainly too restrictive, so some other priority method would be required. Expanding the system call to add a priority argument is possible, if there were demonstrated application advantage.

6 Conclusion

In this paper we described a fast userlevel locking mechanism, called *futexes*, that were integrated into the Linux 2.5 development kernel. We outlined the various requirements for such a package, described previous various solutions and the current futex package. In the performance section we showed, that futexes can provide significant performance advantages over standard System V IPC semaphores in all cases studies.

7 Acknowledgements

Ulrich Drepper (for feedback about current POSIX threads and glibc requirements), Paul Mackerras (for furwocks and many ideas on alternate implementations), Peter Waechtler and Bill Abt for their feedback on asynchronous notifications.

References

[1] Philip Bohannon and et. al. Recoverable User-Level Mutual Exclusion. In *Proc. 7th IEEE Symposium on Parallel and Distributed Systems*, October 1995.

[2] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java Server Performance: Acase study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[3] Hubertus Franke. Ulocks: Fast Userlevel Locking. Available at http://lse.sourceforge.net.

[4] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[5] NGPT: Next Generation Pthreads. Available at http://oss.software.ibm.com/pthreads.

[6] Michael Scott and William N. Scherer III. Scalable Queue-Based Spin Locks with Timeouts. In *Proc. 11th ACMSIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'01*, 2001.

[7] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, 1995.

[8] Message-ID: <Pine.LNX.4.33.0201071902070.5064-101000@sphinx.mythic-beasts.com>.

[9] Message-ID: <20020211143841.A1674@elinux01.watson.ibm.com>.

[10] Message-ID: <E16gRe3−0006ak−00@wagner.rustcorp.com.au>.

[11] Message-ID: <20020106183417.L10326@holomorphy.com>.

Conf	no-spin		spin	
	futex	ulock	futex	ulock
2 tasks				
(0,10)	-15.5	-0.7	-20.5	-22.9
(5,5)	7.9	4.6	52.4	47.7
(7,3)	15.5	18.7	50.2	66.4
(9,1)	33.2	33.1	40.1	46.5
3 tasks				
(0,10)	-13.7	-15.2	-19.1	-15.9
(5,5)	-5.7	8.9	-10.1	3.8
(7,3)	-33.0	11.0	-28.2	-9.2
(9,1)	-33.7	7.5	-21.7	-0.7
4 tasks				
(0,10)	-15.8	-20.0	-20.4	-17.5
(5,5)	0.6	13.3	-5.3	13.5
(7,3)	-38.6	8.0	-42.5	7.3
(9,1)	-43.6	7.7	-30.6	6.4
100 tasks				
(0,10)	172.3	190.8	151.4	189.5
(5,5)	367.6	393.9	386.4	397.6
(7,3)	464.0	300.5	449.0	305.5
(9,1)	495.7	180.3	449.1	190.0
1000 tasks				
(0,10)	1900.4	2343.9	1787.2	2317.9
(5,5)	3363.7	3752.5	3403.7	3792.1
(7,3)	3972.5	3295.2	3891.1	3357.3
(9,1)	4393.7	1971.5	4127.7	1985.3

Table 1: Percentage improvement of Fair locking (spinning and non-spinning) over the base *sysv* throughput

Conf	no-spin		spin	
	futex	ulock	futex	ulock
2 tasks				
(0,10)	8.8	7.6	9.3	7.8
(5,5)	17.7	127.8	86.0	108.2
(7,3)	33.2	60.1	68.5	55.7
(9,1)	40.8	30.9	44.9	29.3
3 tasks				
(0,10)	43.2	9.0	38.5	9.3
(5,5)	49.1	116.0	89.9	76.5
(7,3)	35.0	38.0	58.0	28.1
(9,1)	39.5	12.8	43.3	12.3
4 tasks				
(0,10)	61.2	38.8	59.7	33.7
(5,5)	66.6	130.5	116.3	90.5
(7,3)	34.7	29.9	49.1	20.3
(9,1)	36.1	10.5	39.6	6.2
100 tasks				
(0,10)	456.8	397.1	426.9	399.7
(5,5)	852.3	1030.2	973.4	844.5
(7,3)	1040.4	1003.9	1175.2	919.5
(9,1)	1223.7	967.7	1260.4	936.5
1000 tasks				
(0,10)	4591.7	4047.9	3333.1	4055.2
(5,5)	6989.5	9570.0	8583.8	8095.9
(7,3)	9149.7	9427.1	10781.5	8714.6
(9,1)	11569.6	9437.7	11869.9	9223.3

Table 2: Percentage improvement of regular (ca) locking (spinning and non-spinning) over the base *sysv* throughput

Locktype	Contention Level		
	High	Low	None
FCNTL	1003.69	1482.08	76.4
SPIN	751.18	431.42	67.6
FUTEX	593.00	111.45	41.5

Table 3: Completion times (secs) of tdbtorture runs with different contention rates and different lock implementations

Evaluation and Improvement of IPv6 Protocol Stack by USAGI Project

Yuji Sekiya
Keio University
sekiya@linux-ipv6.org

Hideaki Yoshifuji
The University of Tokyo
yoshfuji@linux-ipv6.org

Mitsuru Kanda
Toshiba Corporation
mk@linux-ipv6.org

Kazunori Miyazawa
Yokogawa Electric Corporation
miyazawa@linux-ipv6.org

Abstract

IPv6 protocol stack has been implemented in Linux kernel since 1996. In spite of the early implementation of IPv6 in the kernel, the stack wasn't maintained for a long time and became out of date. For instance, Linux host couldn't get IPv6 addresses by stateless address autoconfiguration. It was caused by poorly implementation of neighbor discovery protocol. Considering the situation we started USAGI project in October 2000. Our goal is to develop, integrate and provide high quality, RFC compliant, and free IPv6 stack including IPsec and Mobile IPv6. Finally we want to integrate our improvements into the original kernel.

At the beginning we evaluated the original Linux IPv6 stack by TAHI tool. The first evaluation was performed on linux-2.2.15 kernel and the result showed that Linux IPv6 protocol stack wasn't compliant to latest RFCs and didn't have IPsec function which is mandatory for IPv6. As compared with KAME IPv6 protocol stack which passed almost all items of the test, Linux had many problems in kernel. For example, the kernel failed 38 of 58 test items for Neighbor Discovery Protocol and Stateless Address Autoconfiguration failed 55 of 77 items. Then we summarized the results and began to start improving.

Ever since starting the project, we have been continuing to evaluate and improve Linux IPv6 protocol stack. As a result, we have achieved a lot of improvements and released our snapshot code every two weeks and stable code four times. Nowadays the results of evaluation become better and almost all of test item have been passed. Furthermore IPv6 IPsec functions begin to work.

From the experiences, we describe improving methods and evaluation results of USAGI IPv6 protocol stack in this paper. Lastly we describe our future development and merge plans.

1 Introduction

Establishment of IPv6, as a next-generation internet protocol to IPv4, has started since the beginning of the 1990's. The aspect of IPv6 is on providing the solution to the protocol scalability, the greatest problem IPv4 was facing as the Internet grew larger. In detail, IPv6 differ from IPv4 in following ways.

- 128bit address space.

- Forbidding of packet fragmentation in intermediate routers.
- Flexible feature extension using extension headers.
- Supporting security features by default.
- Supporting Plug & Play features by default.

Currently, IPv6 is at the final phase of standardization. Fundamental specifications are almost fixed and commercial products which supports IPv6 has started to show up in the market. International leased lines for IPv6 are out as well. IPv6 has expanded the existing Internet by providing solutions to protocol scalability and beginning to grow as a standard for connecting everything, not just existing computers.

Considering above circumstances, USAGI Project was lunched in October, 2000. USAGI Project is a project which aims to provide improved IPv6 stack on Linux. There are similar organization called KAME, which provides IPv6 stack on BSD Operating systems such as FreeBSD, NetBSD, OpenBSD, and BSD/OS. However, KAME Project does not target their development on Linux. It is important to provide high-quality IPv6 stack on Linux, which is one of the most popular free open-source operating systems in the world, for IPv6 to propagate.

2 Linux IPv6 Implementation

Linux kernel has IPv6 protocol stack by default. However, this IPv6 protocol stack has several problems. In this section, we describe basic IPv6 functions which are needed for IPv6 host and router. Additionally, evaluations on functions mentioned below are done using existing IPv6 protocol stack.

Following utilization patterns are assumed for using Linux, as an IPv6 host, to connect to IPv6 networks.

1. IPv6 host
2. IPv6 gateway inside the house
3. IPv6 mobile host

There, problems on Linux IPv6 protocol stack are described and evaluated in details following the categories mentioned above. Then following evaluations are performed on Linux kernel 2.2.15, 2.2.20 and 2.4.18.

2.1 Linux as an IPv6 Host

Following features are required for using Linux as an IPv6 host.

- IPv6 address autoconfiguration
- Reachability and unreachability detection of neighbor hosts and routers
- IPv6 TCP/UDP socket communication

"IPv6 address autoconfiguration" is a indispensable feature of IPv6 to enable plug & play function. Additionally, "Reachability and unreachability detection of neighbor hosts and routers" are required for a IPv6 host to detect and switch default routers quickly. Furthermore, "IPv6 TCP/UDP socket communication" is a feature required for using IPv6 applications on Linux.

From these perspectives, evaluation on features which Linux IPv6 protocol stack on kernel 2.2.15, 2.2.20 and 2.4.18. The 2.2.15 kernel was released before USAGI Project initialization. The evaluations were done using tools provided by TAHI Project[9].

IPv6 address autoconfiguration First of all, evaluation on IPv6 address autoconfiguration feature is described. IPv6 stateless address autoconfiguration is a function which is defined in RFC2462[10]. It is a function which configures IPv6 address and default router automatically when receiving router advertisements from IPv6 routers. This function is needed for IPv6 Plug & Play feature. The evaluation result of 2.2.15 kernel on this function is shown in Table 2, result of 2.2.20 kernel is shown in Table 3 and result of 2.4.18 kernel is shown in Table 4.

In result of 2.2.15 kernel, there were 30 items that failed and 22 items with warnings of the 54 items in the test. Only 1 of the 54 items passed. There are several causes for this. First, the Duplicate Address Detection (DAD) described in RFC2462[10], which detects duplicated address, may not work correctly. This can be seen from test numbers 2, 20, 21, 23 showing failure in Table 2.

Moreover, the Router Advertisement (RA), which auto-configures the address and the default route, can be one of the causes for the messages may not have processed correctly. The numbers 30, 31 and 40 show this in Table 2.

In result of 2.2.20 kernel, there were 25 failed items, 15 warned items and 13 passed items. The number of passed items are increased from 1 to 13. However, many of DAD function were failed and didn't work correctly.

In result of 2.4.18 kernel, many functions were improved. There were 42 passed items, 10 failed items and 1 warned item.

As compared with 2.2 kernel, DAD functions were improved on Linux 2.4 kernel. However, there was a lack in the number of error processes when receiving irregular messages, which caused the test to failed for abnormal behavior.

Figure 6 summarizes the result of comparing the three kernel results.

Reachability and unreachability detection Second, evaluation on Neighbor Discovery Protocol is mentioned. Neighbor Discovery Protocol is a function defined in RFC2461[7]. It is a function which detects appearance and disappearance of hosts and routers. This function is needed for IPv6 hosts to communicate with neighbor hosts. The evaluation is also performed on Linux kernel 2.2.15, 2.2.20 and 2.4.18. The result on this function is shown in Table 5, Table 6, and Table 7.

Of the 58 items in the test, there were 36 items that failed, 2 items with warnings and 20 items that passed in result of kernel 2.2.15. From failures in test numbers 15, 16, 17, 19, 20, 22, 23 and 24, we see that the state transition in Neighbor Discovery Protocol did not follow the specifications defined in RFC2461.

Same as result of 2.2.15 kernel, there were many failed items in result of 2.2.20 kernel. Regarding NDP functions there wan no difference between 2.2.15 and 2.2.20 kernels.

In result of 2.4.18 kernel, the number of failed items was decreased to 27 items and the number of succeeded items was increased to 29. As a result, there were some improvements between 2.2.20 and 2.4.18 kernels. However, it wasn't good result for working correct IPv6 host.

Figure 7 summarizes the comparison of the three kernel results.

IPv6 TCP/UDP socket communication Last, to enable IPv6 applications, it is necessary to have communication features for both

TCP and UDP in IPv6. GNU libc(glibc)[5] and IPv6 protocol stack in Linux has both TCP and UDP, and applications using Application Program Interface may access both TCP and UDP socket in IPv6.

However, GNU libc and IPv6 protocol stack in Linux made in the beginning of year 2000 before the USAGI Project was founded, were implemented as IPv6 socket API based on RFC2133[3]. The latest IPv6 socket API at that time was RFC2553[4] revised from RFC2133, thus both GNU libc and IPv6 protocol stack in Linux were implemented not based on the latest specifications.

From the evaluation results above, there are number of changes required to use Linux as IPv6 host. Among all, neighbor discovery protocol and IPv6 address autoconfiguration are base features in IPv6 communication. If these features does not function correctly, it will not function as IPV6 host.

Moreover, from the security and error process point of view, protocol stack should be build not cause effects on the function by discarding illegal packets. From this point of view, IPv6 protocol stack in Linux is has not reached the level to use as a stable IPv6 host.

2.2 Linux as an IPv6 Gateway Inside the House

Next, we evaluate Linux IPv6 protocol stack on the features necessary when Linux is used as IPv6 router in homes.

- IPv6 packet forwarding according to routing table
- IPv6 over IPv4 tunneling

The "IPv6 packet forwarding according to routing table" is a feature to forward packets to the next hop according to the routing information in the routing table when IPv6 packets are received. This in a required feature in a IPv6 router. The "IPv6 over IPv4 tunneling" is a technology to structure IPv6 Internet by establishing a virtual link over the IPv4 Internet.

We evaluated the Linux IPv6 protocol stack using Linux kernel 2.2.15 on the above 2 features. We tested if IPv6 packets are correctly forwarded using the topology shown in Figure 1.

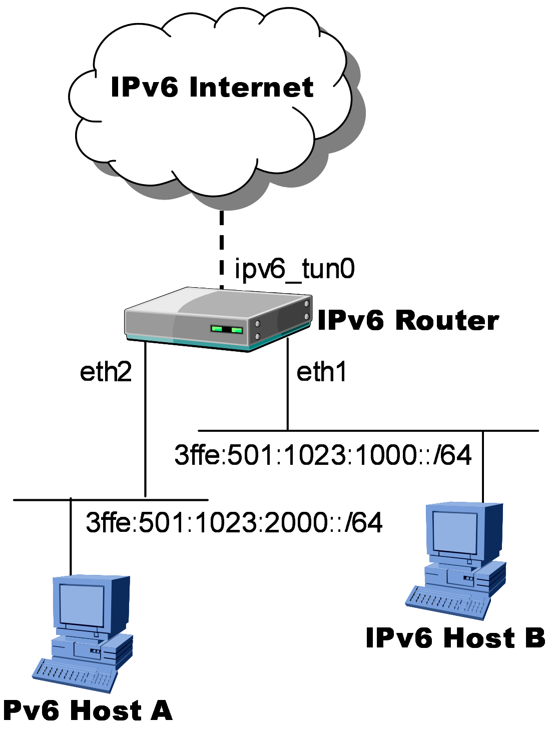


Figure 1: Topology for Routing Evaluation

Also, the routing table on the IPv6 router is shown in the Figure 1. IPv6 hosts A and B receives RA message from IPv6 router and auto-configures IPv6 address and the default route. Additionally, this IPv6 router and the IPv6 router on the other side are connected using IPv6 over IPv4 tunnel. It is assumed that IPv6 router on the other side has correct routing. IPv6 router on the other side uses

Kernel IPv6 routing table		
Destination	Next Hop	Iface
::1/128	::	lo
3ffe:501:1023::/48	::	ipv6_tun0
3ffe:501:1023:1000::1/128	::	lo
3ffe:501:1023:1000::/64	::	eth1
3ffe:501:1023:2000::1/128	::	lo
3ffe:501:1023:2000::/64	::	eth2
fe80::/64	::	eth0
fe80::/64	::	eth1
fe80::/64	::	eth2
fe80::/64	::	ipv6_tun0
ff00::/8	::	eth0
ff00::/8	::	eth1
ff00::/8	::	eth2
ff00::/8	::	ipv6_tun0
::/0	fe80::290:27ff:fe3a:d8	ipv6_tun0

Table 1: IPv6 Routing Table

FreeBSD with KAME snap kit.

Following four kinds of tests are done using above network environment.

TEST #1 Communication from IPv6 router on the other side to IPv6 hosts A and B.

TEST #2 Communication between IPv6 hosts A and B.

TEST #3 Communication between IPv6 router and IPv6 router on the other side.

TEST #4 Communication from IPv6 router to any IPv6 hosts on the Internet.

As a result, test 1 was proven to be successful. Test2, passing through IPv6 router, host A and B was able to communicate each other. On test 3, with the use of IPv6 tunnel, communication using IPv6 was proven to be possible.

However, test 4 failed. The cause was IPv6 router discarding the packet sent from A to fe80::290:27ff:fe3a:d8, which is the next hop of the default route(::/0) without forwarding.

This results from the IPv6 protocol stack specification on Linux. From the point of routing control mechanism, Linux IPv6 protocol stack differs from IPv4 protocol stack with no similarity. On IPv4 protocol stack, routing table is kept in the hash table[1], on the other hand, IPv6 routing table is kept using radix tree[8].

In routing table using radix tree, the top of the tree is the host which possesses the information regarding default route. However, as shown in Figure 2, Linux IPv6 protocol stack has a radix tree with fixed node information on top and it points to ipv6_null_entry. Therefore, when default route is added, the information is attached next to the rt6_info structure which contains ipv6_null_entry. This causes default route not to be referred.

In conclusion, "IPv6 packet forwarding according to routing table" works correctly except for the default route part. Also, "IPv6 over IPv4 tunneling" is proven to work correctly as well. For IPv6 routers inside the houses, it is critical that default route is not functioning. It is hard to say that IPv6 router inside the house possess full routes and it is recom-

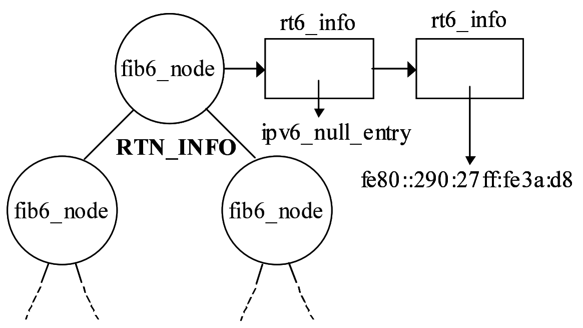


Figure 2: Linux IPv6 Routing Table Structure

mended that routes outside the house is held as default route. Therefore, it is hard to use Linux as a IPv6 gateway inside the house without the change in protocol specification.

2.3 Summary of Linux IPv6 Protocol Stack

From the evaluations mentioned above, Linux IPv6 protocol stack is not suitable in its features as an IPv6 node. It is hard to believe Linux IPv6 protocol to function as an IPv6 host and router using the current Linux IPv6 protocol stack.

3 Activities of USAGI Project

As mentioned in Section 2, IPv6 protocol stack in Linux carry several fatal problems. Thus, USAGI Project was launched, as mentioned in Section 1, to improve the IPv6 protocol stack in Linux. Improvement made by USAGI Project is mentioned in this section.

3.1 Improvement by USAGI Project

Improvements by USAGI Project are mentioned below based on the evaluation and analysis in Section 2.

- Reinforcing illegal NDP message check
- Improving control timer for NDP state
- Following Latest API
- Improving IPv6 routing table structure
- Imprementing IPsec for IPv6

Each improvements is mentioned in detail.

Reinforce illegal NDP message check First of all, reinforced illegal message check for Neighbor Advertisement(NA), Neighbor Solicitation(NS), Router Advertisement(RA), and Router Solicitation(RS) as defined in NDP specifications. The checks are mentioned below in detail.

- Make an unified management function for above messages
- Discarding messages with Hop Limit other than 255
- Discarding NA messages with solicited flag and multicast address as source address
- Selecting proper source address when sending DAD messages
- Improving router renewal algorithm for RA messages with link-local unicast address as source address
- Discarding RA messages with source address other than link-local unicast address
- Discarding NA messages with destination address as multicast address
- Improving the number to send RS message

These simple improvement prevented abnormal process.

Improving control timer for NDP state As mentioned before, NDP is a feature to control status of neighbor nodes. NDP checks on regular intervals if neighbor nodes are reachable or not. The specification require to allocate NDP entry in memory for every node in neighbor, and control status for each neighbor node.

However, the existing Linux IPv6 protocol stack checks reachability of neighbor nodes with a single kernel timer, as shown in Figure 3. Consequently, reachability were checked in constant intervals, regardless of the status for each node.

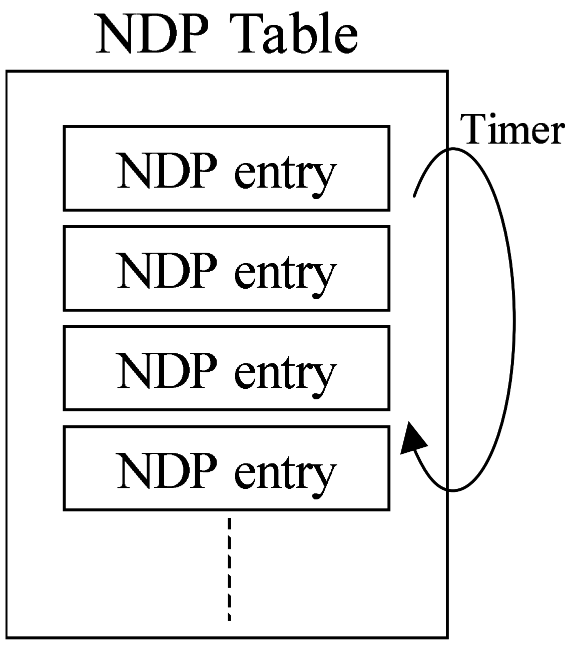


Figure 3: Linux NDP Table

Therefore, USAGI Project improved this kernel timer to check each NDP entry independently as shown in Figure 4. Thus, it is possible to enable and disable timer separately for each NDP entry, and prevent check made to unnecessary NDP entries. Moreover, it is possible to exchange messages correspondent to the status of each NDP entry as defined in the NDP specifications.

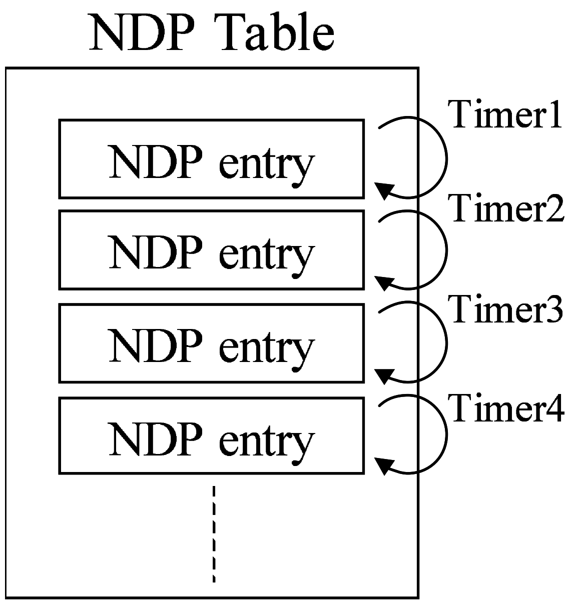


Figure 4: USAGI NDP Table

Following Latest API Linux IPv6 protocol stack and GNU libc, which corresponds to RFC2133, was updated to comply with RFC2533. Change in kernel is adding member sin6_scope_id to sockaddr_in6 structure.

Improving IPv6 routing table structure Problem on recognizing default route is fixed. As shown in Figure 2, modification has made it possible to change pointer from fib6_node structure to rt6_info structure and also made it to adapt routing table lookup functions to new tree structure. With all the improvements, Linux IPv6 protocol stack can recognize IPv6 default route and forward IPv6 packet properly.

Implementing IPsec for IPv6 We implement IPsec for IPv6 in USAGI kernel and our IPsec stack was derived from IABG IPsec[6]. We have improved and changed based on IABG's IPsec and currently our IPsec stack is independent from IABG's IPsec.

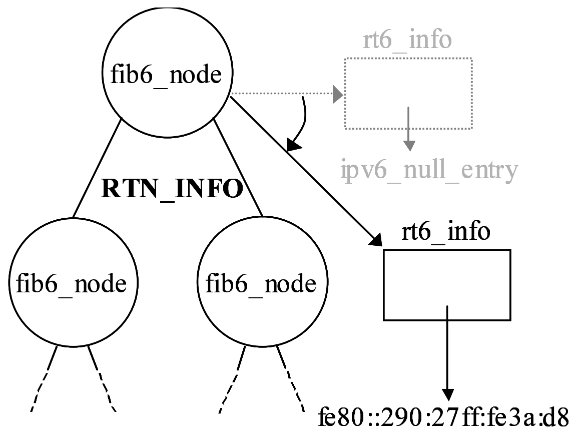


Figure 5: USAGI IPv6 Routing Table Structure

Our stack supports AH/ESP transport mode and manual keying, key negotiation by IKE which is ported from FreeS/WAN[2]. We have already attended some interoperability test events and our stack could communicate with other IPsec implementations. Because our implementation is in still progress, we will continue to work with IPsec for IPv6.

3.2 Evaluation of USAGI Improvement

The same TAHI IPv6 conformance test was done on USAGI snapshot release at the point of Feb. 18th, 2002, which contains five improvements mentioned earlier. The kernel is based on Linux kernel 2.4.17.

The test result of IPv6 address autoconfiguration on USAGI kernel is shown in Table 8.

The result showed that it passed almost all tests regarding IPv6 stateless address autoconfiguration. There were 52 passed items and only 1 warned item.

Then the test result of NDP on USAGI kernel is shown in Table 9.

It reduced the number of failed items. There were 45 succeeded items, 3 warned items and 10 failed items.

Summary of the IPv6 conformance tests on Linux kernels and USAGI IPv6 kernel regarding IPv6 stateless address autoconfiguration and neighbor discovery is shown in Figure 8 and Figure 9.

4 Availability

The outcome of the USAGI Project can be obtained from http://www.linux-ipv6.org/. We release snapshot and stable USAGI kit. Stable USAGI kits are released several times a year and snapshot USAGI kits are released once in two weeks. We have already released stable kit four times. The first was on Nov. 1st, 2000 and the second was on Feb. 5th, 2001. The third was on Jan. 1st, 2002 and The forth, it was a bug fix release of third stable release, was released on Apr. 8th, 2002.

5 Summary

We evaluated Linux IPv6 protocol stack and analyzed the problems in detail. As a result, we were able to pointed out the points which needs improvements. The results enabled to pass many items listed in IPv6 conformance test, and use the features required for IPv6 host.

From the evaluation results, it is obviously that the quality of Linux IPv6 protocol stack has been improved by USAGI Project. We are now working for making patches in order to contribute our improvements to original Linux kernels. However, we have to divide our improvements into small patches which are splitted by each improvement. It takes somewhat span to complete making patches and contribute them.

6 Future Works

Finally, we list our future plans in USAGI Project in this section.

The analysis and improvements mentioned in this paper, were those only in IPv6 protocol specification in IPv6 protocol stack.

Concerning IPv6 protocol specifications, we plan to continue activity especially focused on passing TAHI IPv6 conformance Test, implementing IPv6 features not yet implemented, IPsec Protocol, and IPv6 performance tuning.

For our future plan, we have the below developing items.

- IPsec tunnel mode
- Generic tunnel device for IPv4 and IPv6
- Introduce scope semantics
- DHCPv6
- Prefix Delegation Protocol
- IPv4/IPv6 Translator

References

[1] Jon Crowcroft and Iain Phillips. *TCP/IP and Linux Protocol Implementation: Systems Code for the Linux Internet*. WILEY Publishers, October 2001.

[2] FreeS/WAN Project. Linux FreeS/WAN Project. `http://www.freeswan.org/`.

[3] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC2133, April 1997.

[4] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC2553, March 1999.

[5] GNU Project. GNU C library. `http://www.gnu.org/software /libc/libc.html`.

[6] IABG. IPv6 at IABG. `http://www.ipv6.iabg.de/`.

[7] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC2461, December 1998.

[8] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, pages 93–104, 1991.

[9] TAHI Project. Test and Verification for IPv6. `http://www.tahi.org/`.

[10] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC2462, December 1998.

7 Tables & Figures

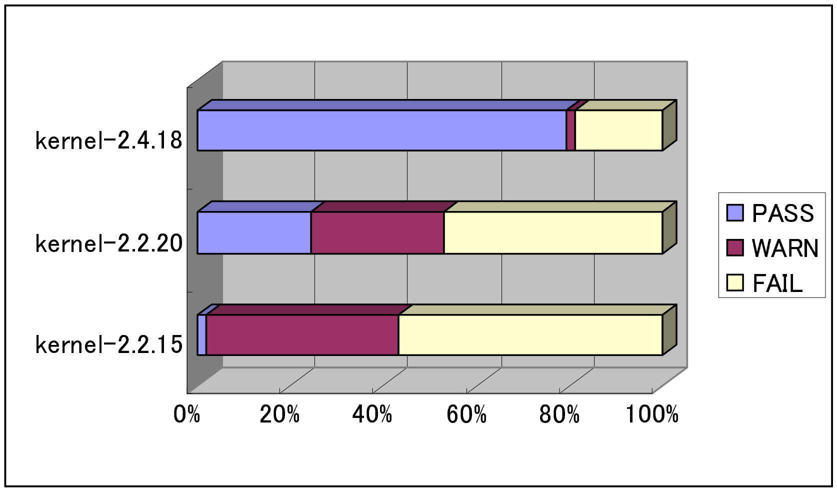


Figure 6: Result of IPv6 Address Autoconfiguration Test on Linux Kernel

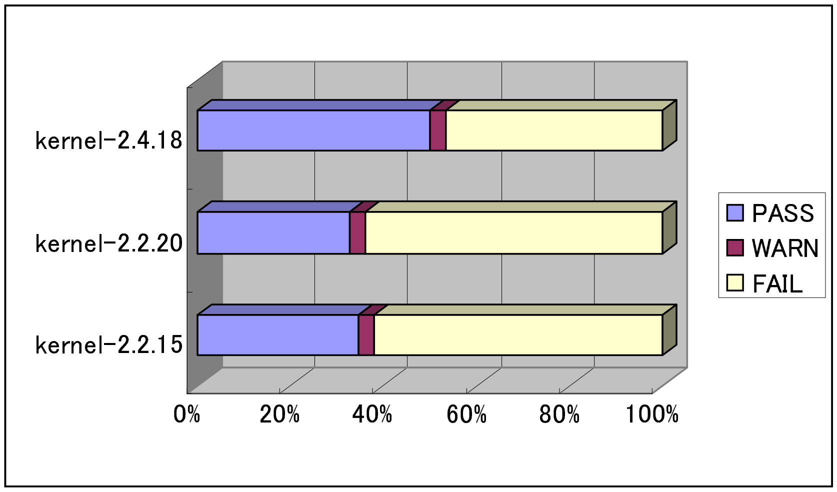


Figure 7: Result of IPv6 NDP Test on Linux Kernel

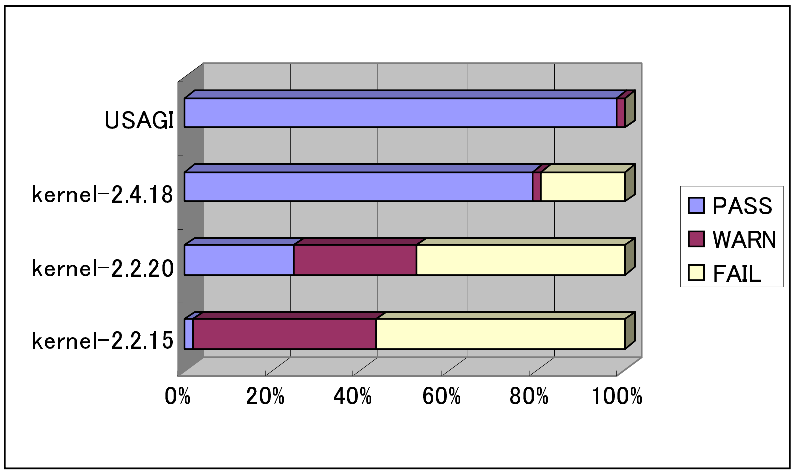


Figure 8: Result of IPv6 Address Autoconfiguration Test on USAGI

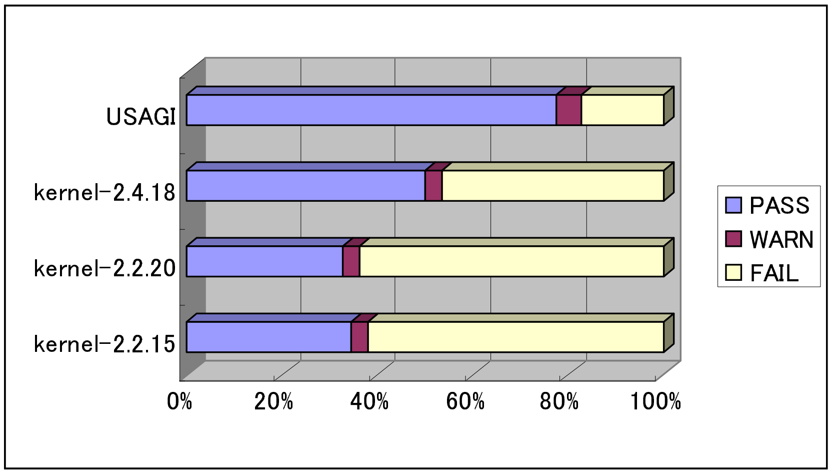


Figure 9: Result of IPv6 NDP Test on USAGI

Table 2: IPv6 Conformance Test For Stateless Address Configuration on Linux kernel 2.2.15

Test No.	Title	Result
1	DAD is performed on NUT by Stateless Link-local address autoconfiguration	WARN
2	DAD Success when NUT received no packet on Stateless Link-local address autoconfiguration	FAIL
3	DAD Fail when NUT received Valid NS in random delaying phase on Stateless Link-local address autoconfiguration	PASS
4	DAD Fail when NUT received Valid NS (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	WARN
5	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	WARN
6	DAD Fail when NUT received Surprise NS (Prefix Option) on Stateless Link-local address autoconfiguration (Surprise test)	WARN
7	DAD Fail when NUT received Valid NA (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	WARN
8	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	WARN
9	DAD Fail when NUT received NA (No TLL option) on Stateless Link-local address autoconfiguration	WARN
10	DAD Fail when NUT received NA (dst addr == solicited node multicast) on Stateless Link-local address autoconfiguration	WARN
11	DAD Fail when NUT received Surprise NA (Many Options) on Stateless Link-local address autoconfiguration (Surprise test)	WARN
12	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Stateless Link-local address autoconfiguration	FAIL
13	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Stateless Link-local address autoconfiguration	FAIL
14	DAD Success when NUT received Invalid NS (Hoplimit != 255) on Stateless Link-local address autoconfiguration	FAIL
15	DAD Success when NUT received Invalid NS (Include SLL opt) on Stateless Link-local address autoconfiguration	FAIL
16	DAD Success when NUT received NS (Src addr is Unicast) on Stateless Link-local address autoconfiguration	FAIL
17	DAD Success when NUT received Invalid NA (Hoplimit != 255) on Stateless Link-local address autoconfiguration	FAIL
18	DAD Success when NUT received Invalid NA (S flag == 1) on Stateless Link-local address autoconfiguration	FAIL
19	DAD Success when NUT received NA (Dst addr is unicast) on Stateless Link-local address autoconfiguration	FAIL
20	DAD is performed on NUT by Manual Link-local address configuration	FAIL
21	DAD Success when NUT received no packet on Manual Link-local address configuration	FAIL
22	DAD is performed on NUT by Manual Global address configuration	WARN
23	DAD Success when NUT received no packet on Manual Global address configuration	FAIL
24	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Manual Global address configuration	WARN
25	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on	

Table continues on next page…

Table 2 continued…

Test No.	Title	Result
	Manual Global address configuration	WARN
26	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Manual Global address configuration	FAIL
27	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Manual Global address configuration	FAIL
28	DAD is performed on NUT by Stateless Global address autoconfiguration	WARN
29	DAD is performed on NUT by Stateless Global address autoconfiguration after DAD Failed for Link-local address autoconfiguration	WARN
30	ADDRCONF Success when NUT received Valid RA (Global address)	FAIL
31	ADDRCONF Success when NUT received Valid RA (Site-local address)	FAIL
32	ADDRCONF Success when NUT received Surprise RA (Many link-layer options) (Surprise test)	FAIL
33	NUT ignores prefixopt if PreferredLifeTime > ValidLifeTime	WARN
34	NUT ignores prefixopt if Prefixlen > 64 (interface ID len is 64)	WARN
35	NUT ignores prefixopt if Prefixlen < 64 (interface ID len is 64)	WARN
36	NUT ignores prefixopt if A flag is 0	WARN
37	NUT ignores prefixopt if prefix is Link-local	WARN
38	NUT ignores prefixopt if Prefixlen > 128	WARN
39	NUT ignores prefixopt if ValidLifeTime is 0 (unknown prefix)	WARN
40	NUT ignores prefixopt if ValidLifeTime is 0 (known prefix but without IPSEC authentication)	FAIL
41	NUT ignores prefixopt if prefix is Global Multicast (Surprise test)	WARN
42	Probe PrefixOptions processing order of same prefixes in one RA (Surprise test)	FAIL
43	Check if ValidLifetime is reset on NUT by RA with same prefix (before expiry, greater VLT)	FAIL
44	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, same VLT)	FAIL
45	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, same VLT)	FAIL
46	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, less VLT)	FAIL
47	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, less VLT)	FAIL
48	Packet receiving and Global address lifetime expiry (valid preferred, valid deprecated, invalid)	FAIL
49	Packet receiving and Site-local address lifetime expiry (valid preferred, valid deprecated, invalid)	FAIL
50	Source address selection and address lifetime expiry (valid deprecated VS valid preferred)	FAIL
51	Source address selection and address lifetime expiry (valid deprecated VS valid deprecated)	FAIL
52	Source address selection and address lifetime expiry (invalid VS valid deprecated)	FAIL
53	Source address selection and address lifetime expiry (invalid VS invalid)	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 3: IPv6 Conformance Test For Stateless Address Configuration on Linux kernel 2.2.20

Test No.	Title	Result
1	DAD is performed on NUT by Stateless Link-local address autoconfiguration	PASS
2	DAD Success when NUT received no packet on Stateless Link-local address autoconfiguration	PASS
3	DAD Fail when NUT received Valid NS in random delaying phase on Stateless Link-local address autoconfiguration	PASS
4	DAD Fail when NUT received Valid NS (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
5	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
6	DAD Fail when NUT received Surprise NS (Prefix Option) on Stateless Link-local address autoconfiguration (Surprise test)	PASS
7	DAD Fail when NUT received Valid NA (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
8	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
9	DAD Fail when NUT received NA (No TLL option) on Stateless Link-local address autoconfiguration	PASS
10	DAD Fail when NUT received NA (dst addr == solicited node multicast) on Stateless Link-local address autoconfiguration	PASS
11	DAD Fail when NUT received Surprise NA (Many Options) on Stateless Link-local address autoconfiguration (Surprise test)	PASS
12	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Stateless Link-local address autoconfiguration	FAIL
13	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Stateless Link-local address autoconfiguration	FAIL
14	DAD Success when NUT received Invalid NS (Hoplimit != 255) on Stateless Link-local address autoconfiguration	FAIL
15	DAD Success when NUT received Invalid NS (Include SLL opt) on Stateless Link-local address autoconfiguration	FAIL
16	DAD Success when NUT received NS (Src addr is Unicast) on Stateless Link-local address autoconfiguration	FAIL
17	DAD Success when NUT received Invalid NA (Hoplimit != 255) on Stateless Link-local address autoconfiguration	FAIL
18	DAD Success when NUT received Invalid NA (S flag == 1) on Stateless Link-local address autoconfiguration	FAIL
19	DAD Success when NUT received NA (Dst addr is unicast) on Stateless Link-local address autoconfiguration	FAIL
20	DAD is performed on NUT by Manual Link-local address configuration	WARN
21	DAD Success when NUT received no packet on Manual Link-local address configuration	FAIL
22	DAD is performed on NUT by Manual Global address configuration	WARN
23	DAD Success when NUT received no packet on Manual Global address configuration	FAIL
24	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Manual Global address configuration	WARN
25	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on	

Table continues on next page...

Table 3 continued...

Test No.	Title	Result
	Manual Global address configuration	WARN
26	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Manual Global address configuration	FAIL
27	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Manual Global address configuration	FAIL
28	DAD is performed on NUT by Stateless Global address autoconfiguration	WARN
29	DAD is performed on NUT by Stateless Global address autoconfiguration after DAD Failed for Link-local address autoconfiguration	PASS
30	ADDRCONF Success when NUT received Valid RA (Global address)	FAIL
31	ADDRCONF Success when NUT received Valid RA (Site-local address)	FAIL
32	ADDRCONF Success when NUT received Surprise RA (Many link-layer options) (Surprise test)	FAIL
33	NUT ignores prefixopt if PreferredLifeTime > ValidLifeTime	WARN
34	NUT ignores prefixopt if Prefixlen > 64 (interface ID len is 64)	WARN
35	NUT ignores prefixopt if Prefixlen < 64 (interface ID len is 64)	WARN
36	NUT ignores prefixopt if A flag is 0	WARN
37	NUT ignores prefixopt if prefix is Link-local	PASS
38	NUT ignores prefixopt if Prefixlen > 128	WARN
39	NUT ignores prefixopt if ValidLifeTime is 0 (unknown prefix)	WARN
40	NUT ignores prefixopt if ValidLifeTime is 0 (known prefix but without IPSEC authentication)	FAIL
41	NUT ignores prefixopt if prefix is Global Multicast (Surprise test)	WARN
42	Probe PrefixOptions processing order of same prefixes in one RA (Surprise test)	FAIL
43	Check if ValidLifetime is reset on NUT by RA with same prefix (before expiry, greater VLT)	FAIL
44	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, same VLT)	FAIL
45	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, same VLT)	FAIL
46	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, less VLT)	FAIL
47	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, less VLT)	FAIL
48	Packet receiving and Global address lifetime expiry (valid preferred, valid deprecated, invalid)	FAIL
49	Packet receiving and Site-local address lifetime expiry (valid preferred, valid deprecated, invalid)	FAIL
50	Source address selection and address lifetime expiry (valid deprecated VS valid preferred)	WARN
51	Source address selection and address lifetime expiry (valid deprecated VS valid deprecated)	WARN
52	Source address selection and address lifetime expiry (invalid VS valid deprecated)	WARN
53	Source address selection and address lifetime expiry (invalid VS invalid)	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 4: IPv6 Conformance Test For Stateless Address Configuration on Linux kernel 2.4.18

Test No.	Title	Result
1	DAD is performed on NUT by Stateless Link-local address autoconfiguration	PASS
2	DAD Success when NUT received no packet on Stateless Link-local address autoconfiguration	PASS
3	DAD Fail when NUT received Valid NS in random delaying phase on Stateless Link-local address autoconfiguration	PASS
4	DAD Fail when NUT received Valid NS (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
5	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
6	DAD Fail when NUT received Surprise NS (Prefix Option) on Stateless Link-local address autoconfiguration (Surprise test)	PASS
7	DAD Fail when NUT received Valid NA (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
8	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
9	DAD Fail when NUT received NA (No TLL option) on Stateless Link-local address autoconfiguration	PASS
10	DAD Fail when NUT received NA (dst addr == solicited node multicast) on Stateless Link-local address autoconfiguration	PASS
11	DAD Fail when NUT received Surprise NA (Many Options) on Stateless Link-local address autoconfiguration (Surprise test)	PASS
12	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Stateless Link-local address autoconfiguration	FAIL
13	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Stateless Link-local address autoconfiguration	PASS
14	DAD Success when NUT received Invalid NS (Hoplimit != 255) on Stateless Link-local address autoconfiguration	PASS
15	DAD Success when NUT received Invalid NS (Include SLL opt) on Stateless Link-local address autoconfiguration	FAIL
16	DAD Success when NUT received NS (Src addr is Unicast) on Stateless Link-local address autoconfiguration	PASS
17	DAD Success when NUT received Invalid NA (Hoplimit != 255) on Stateless Link-local address autoconfiguration	PASS
18	DAD Success when NUT received Invalid NA (S flag == 1) on Stateless Link-local address autoconfiguration	PASS
19	DAD Success when NUT received NA (Dst addr is unicast) on Stateless Link-local address autoconfiguration	PASS
20	DAD is performed on NUT by Manual Link-local address configuration	PASS
21	DAD Success when NUT received no packet on Manual Link-local address configuration	PASS
22	DAD is performed on NUT by Manual Global address configuration	PASS
23	DAD Success when NUT received no packet on Manual Global address configuration	PASS
24	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Manual Global address configuration	PASS
25	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on	

Table continues on next page...

Table 4 continued. . .

Test No.	Title	Result
	Manual Global address configuration	PASS
26	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Manual Global address configuration	FAIL
27	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Manual Global address configuration	PASS
28	DAD is performed on NUT by Stateless Global address autoconfiguration	PASS
29	DAD is performed on NUT by Stateless Global address autoconfiguration after DAD Failed for Link-local address autoconfiguration	PASS
30	ADDRCONF Success when NUT received Valid RA (Global address)	PASS
31	ADDRCONF Success when NUT received Valid RA (Site-local address)	PASS
32	ADDRCONF Success when NUT received Surprise RA (Many link-layer options) (Surprise test)	PASS
33	NUT ignores prefixopt if PreferredLifeTime > ValidLifeTime	PASS
34	NUT ignores prefixopt if Prefixlen > 64 (interface ID len is 64)	PASS
35	NUT ignores prefixopt if Prefixlen < 64 (interface ID len is 64)	PASS
36	NUT ignores prefixopt if A flag is 0	PASS
37	NUT ignores prefixopt if prefix is Link-local	PASS
38	NUT ignores prefixopt if Prefixlen > 128	PASS
39	NUT ignores prefixopt if ValidLifeTime is 0 (unknown prefix)	PASS
40	NUT ignores prefixopt if ValidLifeTime is 0 (known prefix but without IPSEC authentication)	PASS
41	NUT ignores prefixopt if prefix is Global Multicast (Surprise test)	PASS
42	Probe PrefixOptions processing order of same prefixes in one RA (Surprise test)	WARN
43	Check if ValidLifetime is reset on NUT by RA with same prefix (before expiry, greater VLT)	FAIL
44	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, same VLT)	FAIL
45	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, same VLT)	FAIL
46	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, less VLT)	PASS
47	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, less VLT)	FAIL
48	Packet receiving and Global address lifetime expiry (valid preferred, valid deprecated, invalid)	FAIL
49	Packet receiving and Site-local address lifetime expiry (valid preferred, valid deprecated, invalid)	FAIL
50	Source address selection and address lifetime expiry (valid deprecated VS valid preferred)	PASS
51	Source address selection and address lifetime expiry (valid deprecated VS valid deprecated)	PASS
52	Source address selection and address lifetime expiry (invalid VS valid deprecated)	PASS
53	Source address selection and address lifetime expiry (invalid VS invalid)	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 5: IPv6 Conformance Test For Neighbor Discovery on Linux kernel 2.2.15

Test No.	Title	Result
1	Verify that the NUT send NSs (link-local ==> link-local)	FAIL
2	Verify that the NUT send NSs (global ==> global)	FAIL
3	Verify that the NUT send NSs (link-local ==> global)	FAIL
4	Verify that the NUT send NSs (global ==> link-local)	FAIL
5	Multicast NS w/ Default Config.	PASS
6	Multicast NS w/ RetransTimer=3sec.	PASS
7	Unicast NS w/ Default Config.	PASS
8	Unicast NS w/ RestransTier=3sec.	PASS
9	Address Resolution Queue (one entry for an address ?)	PASS
10	Address Resolution Queue (more then one entry for an address ?)	PASS
11	Address Resolution Queue (one entry per an address ?)	FAIL
12	Receiving valid NSs	WARN
13	Receiving invalid NSs	FAIL
14	NS vs. IsRouter flag	PASS
15	NS vs. NONCE	FAIL
16	NS vs. INCOMPLETE	FAIL
17	NS vs. REACHABLE	FAIL
18	NS vs. STALE	PASS
19	NS vs. PROBE	FAIL
20	R flag vs. IsRouter flag	FAIL
21	NA vs. NONCE	PASS
22	NA vs. INCOMPLETE	FAIL
23	NA vs. REACHABLE	FAIL
24	NA vs. STALE	FAIL
25	NA vs. PROBE	FAIL
26	Sending RS	FAIL
27	Sending RS after receiving unsolicited RA	FAIL
28	Not sending RS after receiving solicited RA	FAIL
29	Ignoring RS	PASS
30	RA set IsRouter flag	FAIL
31	Receiving multiple RAs #1	FAIL
32	Receiving multiple RAs #2	FAIL
33	Ingnoring invalid RAs	FAIL
34	ReachableTIme vs BaseReachableTime	FAIL
35	RouterLifetime=0	PASS
36	RouterLifetime=5	FAIL
37	Next-hop Determination	WARN
38	The Default Router List vs Unreachability Detection	FAIL
39	RA vs NONCE	PASS
40	RA vs INCOMPLETE	PASS
41	RA vs REACHABLE	FAIL
42	RA vs STALE	PASS
43	RA vs PROBE	FAIL
44	Redirect vs NONCE	PASS
45	Redirect vs INCOMPLETE	PASS

Table continues on next page…

Table 5 continued. . .

Test No.	Title	Result
46	Redirect vs REACHABLE	PASS
47	Redirect vs STALE	PASS
48	Redirect vs PROBE	FAIL
49	Invalid Redirect vs Neighbor Cache State	FAIL
50	Redirect vs Destination Cache; Redirect to a host	FAIL
51	Redirect vs Destination Cache; Redirect to a better router	FAIL
52	Redirect vs Neighbor Unreachability Detection; Redirect to a host	FAIL
53	Redirect vs Neighbor Unreachability Detection; Redirect to a better router	FAIL
54	Redirect vs NA w/ RFlag=0 #1	PASS
55	Redirect vs NA w/ RFlag=0 #2	FAIL
56	Redirect vs RA w/ RouterLifetime=0 #1	PASS
57	Redirect vs RA w/ RouterLifetime=0 #2	FAIL
58	Redirect vs NONCE	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 6: IPv6 Conformance Test For Neighbor Discovery on Linux kernel 2.2.20

Test No.	Title	Result
1	Verify that the NUT send NSs (link-local ==> link-local)	FAIL
2	Verify that the NUT send NSs (global ==> global)	FAIL
3	Verify that the NUT send NSs (link-local ==> global)	FAIL
4	Verify that the NUT send NSs (global ==> link-local)	FAIL
5	Multicast NS w/ Default Config.	PASS
6	Multicast NS w/ RetransTimer=3sec.	PASS
7	Unicast NS w/ Default Config.	PASS
8	Unicast NS w/ RestransTier=3sec.	PASS
9	Address Resolution Queue (one entry for an address ?)	PASS
10	Address Resolution Queue (more then one entry for an address ?)	PASS
11	Address Resolution Queue (one entry per an address ?)	FAIL
12	Receiving valid NSs	WARN
13	Receiving invalid NSs	FAIL
14	NS vs. IsRouter flag	PASS
15	NS vs. NONCE	FAIL
16	NS vs. INCOMPLETE	FAIL
17	NS vs. REACHABLE	FAIL
18	NS vs. STALE	PASS
19	NS vs. PROBE	FAIL
20	R flag vs. IsRouter flag	FAIL
21	NA vs. NONCE	PASS
22	NA vs. INCOMPLETE	FAIL
23	NA vs. REACHABLE	FAIL
24	NA vs. STALE	FAIL
25	NA vs. PROBE	FAIL
26	Sending RS	FAIL
27	Sending RS after receiving unsolicited RA	FAIL
28	Not sending RS after receiving solicited RA	FAIL
29	Ignoring RS	PASS

Table continues on next page. . .

Table 6 continued. . .

Test No.	Title	Result
30	RA set IsRouter flag	FAIL
31	Receiving multiple RAs #1	FAIL
32	Receiving multiple RAs #2	FAIL
33	Ingnoring invalid RAs	FAIL
34	ReachableTIme vs BaseReachableTime	FAIL
35	RouterLifetime=0	PASS
36	RouterLifetime=5	FAIL
37	Next-hop Determination	WARN
38	The Default Router List vs Unreachability Detection	FAIL
39	RA vs NONCE	FAIL
40	RA vs INCOMPLETE	PASS
41	RA vs REACHABLE	FAIL
42	RA vs STALE	PASS
43	RA vs PROBE	FAIL
44	Redirect vs NONCE	PASS
45	Redirect vs INCOMPLETE	PASS
46	Redirect vs REACHABLE	PASS
47	Redirect vs STALE	PASS
48	Redirect vs PROBE	FAIL
49	Invalid Redirect vs Neighbor Cache State	FAIL
50	Redirect vs Destination Cache; Redirect to a host	FAIL
51	Redirect vs Destination Cache; Redirect to a better router	FAIL
52	Redirect vs Neighbor Unreachability Detection; Redirect to a host	FAIL
53	Redirect vs Neighbor Unreachability Detection; Redirect to a better router	FAIL
54	Redirect vs NA w/ RFlag=0 #1	PASS
55	Redirect vs NA w/ RFlag=0 #2	FAIL
56	Redirect vs RA w/ RouterLifetime=0 #1	PASS
57	Redirect vs RA w/ RouterLifetime=0 #2	FAIL
58	Redirect vs NONCE	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 7: IPv6 Conformance Test For Neighbor Discovery on Linux kernel 2.4.18

Test No.	Title	Result
1	Verify that the NUT send NSs (link-local ==> link-local)	FAIL
2	Verify that the NUT send NSs (global ==> global)	FAIL
3	Verify that the NUT send NSs (link-local ==> global)	FAIL
4	Verify that the NUT send NSs (global ==> link-local)	FAIL
5	Multicast NS w/ Default Config.	PASS
6	Multicast NS w/ RetransTimer=3sec.	PASS
7	Unicast NS w/ Default Config.	PASS
8	Unicast NS w/ RestransTier=3sec.	PASS
9	Address Resolution Queue (one entry for an address ?)	PASS
10	Address Resolution Queue (more then one entry for an address ?)	PASS
11	Address Resolution Queue (one entry per an address ?)	FAIL
12	Receiving valid NSs	WARN
13	Receiving invalid NSs	FAIL
14	NS vs. IsRouter flag	PASS

Table continues on next page. . .

Table 7 continued…

Test No.	Title	Result
15	NS vs. NONCE	FAIL
16	NS vs. INCOMPLETE	FAIL
17	NS vs. REACHABLE	PASS
18	NS vs. STALE	PASS
19	NS vs. PROBE	FAIL
20	R flag vs. IsRouter flag	FAIL
21	NA vs. NONCE	PASS
22	NA vs. INCOMPLETE	FAIL
23	NA vs. REACHABLE	FAIL
24	NA vs. STALE	FAIL
25	NA vs. PROBE	FAIL
26	Sending RS	PASS
27	Sending RS after receiving unsolicited RA	PASS
28	Not sending RS after receiving solicited RA	PASS
29	Ignoring RS	PASS
30	RA set IsRouter flag	FAIL
31	Receiving multiple RAs #1	PASS
32	Receiving multiple RAs #2	PASS
33	Ingnoring invalid RAs	PASS
34	ReachableTIme vs BaseReachableTime	PASS
35	RouterLifetime=0	PASS
36	RouterLifetime=5	FAIL
37	Next-hop Determination	WARN
38	The Default Router List vs Unreachability Detection	FAIL
39	RA vs NONCE	PASS
40	RA vs INCOMPLETE	PASS
41	RA vs REACHABLE	PASS
42	RA vs STALE	PASS
43	RA vs PROBE	FAIL
44	Redirect vs NONCE	PASS
45	Redirect vs INCOMPLETE	PASS
46	Redirect vs REACHABLE	PASS
47	Redirect vs STALE	PASS
48	Redirect vs PROBE	FAIL
49	Invalid Redirect vs Neighbor Cache State	FAIL
50	Redirect vs Destination Cache; Redirect to a host	FAIL
51	Redirect vs Destination Cache; Redirect to a better router	FAIL
52	Redirect vs Neighbor Unreachability Detection; Redirect to a host	FAIL
53	Redirect vs Neighbor Unreachability Detection; Redirect to a better router	FAIL
54	Redirect vs NA w/ RFlag=0 #1	PASS
55	Redirect vs NA w/ RFlag=0 #2	FAIL
56	Redirect vs RA w/ RouterLifetime=0 #1	PASS
57	Redirect vs RA w/ RouterLifetime=0 #2	FAIL
58	Redirect vs NONCE	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 8: IPv6 Conformance Test For Stateless Address Configuration on USAGI kernel

Test No.	Title	Result
1	DAD is performed on NUT by Stateless Link-local address autoconfiguration	PASS
2	DAD Success when NUT received no packet on Stateless Link-local address autoconfiguration	PASS
3	DAD Fail when NUT received Valid NS in random delaying phase on Stateless Link-local address autoconfiguration	PASS
4	DAD Fail when NUT received Valid NS (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
5	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
6	DAD Fail when NUT received Surprise NS (Prefix Option) on Stateless Link-local address autoconfiguration (Surprise test)	PASS
7	DAD Fail when NUT received Valid NA (dst MAC addr != MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
8	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on Stateless Link-local address autoconfiguration	PASS
9	DAD Fail when NUT received NA (No TLL option) on Stateless Link-local address autoconfiguration	PASS
10	DAD Fail when NUT received NA (dst addr == solicited node multicast) on Stateless Link-local address autoconfiguration	PASS
11	DAD Fail when NUT received Surprise NA (Many Options) on Stateless Link-local address autoconfiguration (Surprise test)	PASS
12	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Stateless Link-local address autoconfiguration	PASS
13	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Stateless Link-local address autoconfiguration	PASS
14	DAD Success when NUT received Invalid NS (Hoplimit != 255) on Stateless Link-local address autoconfiguration	PASS
15	DAD Success when NUT received Invalid NS (Include SLL opt) on Stateless Link-local address autoconfiguration	PASS
16	DAD Success when NUT received NS (Src addr is Unicast) on Stateless Link-local address autoconfiguration	PASS
17	DAD Success when NUT received Invalid NA (Hoplimit != 255) on Stateless Link-local address autoconfiguration	PASS
18	DAD Success when NUT received Invalid NA (S flag == 1) on Stateless Link-local address autoconfiguration	PASS
19	DAD Success when NUT received NA (Dst addr is unicast) on Stateless Link-local address autoconfiguration	PASS
20	DAD is performed on NUT by Manual Link-local address configuration	PASS
21	DAD Success when NUT received no packet on Manual Link-local address configuration	PASS
22	DAD is performed on NUT by Manual Global address configuration	PASS
23	DAD Success when NUT received no packet on Manual Global address configuration	PASS
24	DAD Fail when NUT received Valid NS (dst MAC addr == MAC addr of NUT) on Manual Global address configuration	PASS
25	DAD Fail when NUT received Valid NA (dst MAC addr == MAC addr of NUT) on	

Table continues on next page...

Table 8 continued…

Test No.	Title	Result
	Manual Global address configuration	PASS
26	DAD Success when NUT received Invalid NS (Dst addr is Allnodes) on Manual Global address configuration	PASS
27	DAD Success when NUT received Invalid NS (Dst addr is Tentative) on Manual Global address configuration	PASS
28	DAD is performed on NUT by Stateless Global address autoconfiguration	PASS
29	DAD is performed on NUT by Stateless Global address autoconfiguration after DAD Failed for Link-local address autoconfiguration	PASS
30	ADDRCONF Success when NUT received Valid RA (Global address)	PASS
31	ADDRCONF Success when NUT received Valid RA (Site-local address)	PASS
32	ADDRCONF Success when NUT received Surprise RA (Many link-layer options) (Surprise test)	PASS
33	NUT ignores prefixopt if PreferredLifeTime > ValidLifeTime	PASS
34	NUT ignores prefixopt if Prefixlen > 64 (interface ID len is 64)	PASS
35	NUT ignores prefixopt if Prefixlen < 64 (interface ID len is 64)	PASS
36	NUT ignores prefixopt if A flag is 0	PASS
37	NUT ignores prefixopt if prefix is Link-local	PASS
38	NUT ignores prefixopt if Prefixlen > 128	PASS
39	NUT ignores prefixopt if ValidLifeTime is 0 (unknown prefix)	PASS
40	NUT ignores prefixopt if ValidLifeTime is 0 (known prefix but without IPSEC authentication)	PASS
41	NUT ignores prefixopt if prefix is Global Multicast (Surprise test)	PASS
42	Probe PrefixOptions processing order of same prefixes in one RA (Surprise test)	WARN
43	Check if ValidLifetime is reset on NUT by RA with same prefix (before expiry, greater VLT)	PASS
44	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, same VLT)	PASS
45	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, same VLT)	PASS
46	Check if ValidLifetime is NOT reset on NUT by RA with same prefix (before expiry, less VLT)	PASS
47	Check if ValidLifetime is reset on NUT by RA with same prefix (after expiry, less VLT)	PASS
48	Packet receiving and Global address lifetime expiry (valid preferred, valid deprecated, invalid)	PASS
49	Packet receiving and Site-local address lifetime expiry (valid preferred, valid deprecated, invalid)	PASS
50	Source address selection and address lifetime expiry (valid deprecated VS valid preferred)	PASS
51	Source address selection and address lifetime expiry (valid deprecated VS valid deprecated)	PASS
52	Source address selection and address lifetime expiry (invalid VS valid deprecated)	PASS
53	Source address selection and address lifetime expiry (invalid VS invalid)	PASS

This Report was generated by TAHI IPv6 Conformance Test Suite

Table 9: IPv6 Conformance Test For Neighbor Discovery on USAGI kernel

Test No.	Title	Result
1	Verify that the NUT send NSs (link-local ==> link-local)	PASS
2	Verify that the NUT send NSs (global ==> global)	PASS
3	Verify that the NUT send NSs (link-local ==> global)	PASS
4	Verify that the NUT send NSs (global ==> link-local)	PASS
5	Multicast NS w/ Default Config.	PASS
6	Multicast NS w/ RetransTimer=3sec.	PASS
7	Unicast NS w/ Default Config.	PASS
8	Unicast NS w/ RestransTier=3sec.	PASS
9	Address Resolution Queue (one entry for an address ?)	PASS
10	Address Resolution Queue (more then one entry for an address ?)	PASS
11	Address Resolution Queue (one entry per an address ?)	PASS
12	Receiving valid NSs	WARN
13	Receiving invalid NSs	PASS
14	NS vs. IsRouter flag	PASS
15	NS vs. NONCE	PASS
16	NS vs. INCOMPLETE	PASS
17	NS vs. REACHABLE	PASS
18	NS vs. STALE	PASS
19	NS vs. PROBE	PASS
20	R flag vs. IsRouter flag	WARN
21	NA vs. NONCE	PASS
22	NA vs. INCOMPLETE	PASS
23	NA vs. REACHABLE	PASS
24	NA vs. STALE	PASS
25	NA vs. PROBE	PASS
26	Sending RS	PASS
27	Sending RS after receiving unsolicited RA	PASS
28	Not sending RS after receiving solicited RA	PASS
29	Ignoring RS	PASS
30	RA set IsRouter flag	FAIL
31	Receiving multiple RAs #1	PASS
32	Receiving multiple RAs #2	PASS
33	Ingnoring invalid RAs	PASS
34	ReachableTIme vs BaseReachableTime	PASS
35	RouterLifetime=0	PASS
36	RouterLifetime=5	FAIL
37	Next-hop Determination	WARN
38	The Default Router List vs Unreachability Detection	PASS
39	RA vs NONCE	PASS
40	RA vs INCOMPLETE	PASS
41	RA vs REACHABLE	PASS
42	RA vs STALE	PASS
43	RA vs PROBE	PASS
44	Redirect vs NONCE	PASS
45	Redirect vs INCOMPLETE	PASS

Table continues on next page…

Table 9 continued. . .

Test No.	Title	Result
46	Redirect vs REACHABLE	PASS
47	Redirect vs STALE	PASS
48	Redirect vs PROBE	PASS
49	Invalid Redirect vs Neighbor Cache State	FAIL
50	Redirect vs Destination Cache; Redirect to a host	FAIL
51	Redirect vs Destination Cache; Redirect to a better router	FAIL
52	Redirect vs Neighbor Unreachability Detection; Redirect to a host	FAIL
53	Redirect vs Neighbor Unreachability Detection; Redirect to a better router	FAIL
54	Redirect vs NA w/ RFlag=0 #1	PASS
55	Redirect vs NA w/ RFlag=0 #2	FAIL
56	Redirect vs RA w/ RouterLifetime=0 #1	PASS
57	Redirect vs RA w/ RouterLifetime=0 #2	FAIL
58	Redirect vs NONCE	FAIL

This Report was generated by TAHI IPv6 Conformance Test Suite

GNU Bayonne: telephony application server of the GNU project

David Sugar
Open Source Telecom.
Somerset, NJ, 08873
sugar@gnu.org
http://www.gnu.org/software/bayonne

Abstract

GNU Bayonne is a middleware telephony server that can be used to create and deploy script driven telephony application services. These services interact with users over the public telephone network. GNU Bayonne can be used to create carrier applications like Voice Mail and calling card systems, as well as enterprise applications such as unified messaging. It can be used to provide voice response for e-commerce systems and has been used in this role in various e-gov projects. GNU Bayonne can also be used to telephony enable existing scripting languages such as perl and python.

1 Introduction

Our goal in GNU Bayonne was to make telephony services as easy to program and deploy as a web server is today. We choose to make this server easily programmable thru server scripting. We also desired to have it highly portable, and allow it to integrate with existing application scripting tools so that one could leverage not just the core server but the entire platform to deliver telephony functionality and integrate with other resources like databases.

GNU Bayonne, as a telephony server, also imposes some very real and unique design constraints. For example, we must provide interactive voice response in real-time. “realtime” in this case may mean what a person might tolerate, or delay of 1/10th of a second, rather than what one might measure in milliseconds in other kinds of real-time applications. However, this still means that the service cannot block, for, after all, you cannot flow control people speaking.

Since each vendor of telephony hardware has chosen to create their own unique and substantial application library interface, we needed GNU Bayonne to sit above these and be able to abstract them. Ultimately we choose to create a driver plugin architecture to do this. What this means is that you can get a card and api from Aculab, for example, write your application in GNU Bayonne using it, and later choose, say, to use Intel telephony hardware, and still have your application run, unmodified. This has never been done in the industry widely because many of these same telephony hardware manufacturers like to produce their own middleware solutions that lock users into their products.

2 GNU Common C++

To create GNU Bayonne we needed a portable foundation written in C++. I wanted to use C++ for several reasons. First, the highly abstract nature of the driver interfaces seemed very natural to use class encapsulation for. Second, I found I personally could write C++ code faster and more bug free than I could write C code.

Why we choose not to use an existing framework is also simple to explain. We knew we needed threading, and socket support, and a few other things. There were no single framework that did all these things except a few that were very large and complex which did far more than we needed. We wanted a small footprint for Bayonne, and the most adaptable framework that we found at the time typically added several megs of core image just for the runtime library.

GNU Common C++ (originally APE) was created to provide a very easy to comprehend and portable class abstraction for threads, sockets, semaphores, exceptions, etc. This has since grown into its own and is now used as a foundation of a number of projects as well as being a part of GNU.

3 GNU ccScript

In addition to having portable C++ threading, we needed a scripting engine. This scripting system had to operate in conjunction with a non-blocking state-transition call processing system. It also had to offer immediate call response, and support several hundred to a thousand instances running concurrently in one server image.

Many extension languages assume a separate execution instance (thread or process) for each interpreter instance. These were unsuitable. Many extension languages assume expression parsing with non-deterministic run time. An expression could invoke recursive functions or entire subprograms for example. Again, since we wanted not to have a separate execution instance for each interpreter instance, and have each instance respond to the leading edge of an event callback from the telephony driver as it steps thru a state machine, none of the existing common solutions like tcl, perl, guile, etc, would immediately work for us. Instead, we created a non-blocking and deterministic scripting engine, GNU ccScript.

GNU ccScript is unique in several ways. It is step executed, and is non-blocking. Statements either execute and return immediately, or they schedule their completion for a later time with the executive. A given “step” is executed, rather than linearly. This allows a single thread to invoke and manage multiple interpreter instances. While GNU Bayonne can support interacting with hundreds of simultaneous telephone callers on high density carrier scale hardware, we do not require hundreds of native “thread” instances running in the server, and we have a very modest cpu load.

Another way GNU ccScript is unique is in support for memory loaded scripts. To avoid delay or blocking while loading scripts, all scripts are loaded and parsed into a virtual machine structure in memory. When we wish to change scripts, a brand new virtual machine instance is created to contain these scripts. Calls currently in progress continue under the’ old vm and new callers are offered the new vm. When the last old call terminates, the entire old vm is then disposed of. This allows for 100% uptime even while services are modified.

Finally, GNU ccScript allows direct class extension of the script interpreter. This allows one to easily create a derived dialect specific to a given application, or even specific to a given GNU Bayonne driver, simply by deriving it from the core language thru standard C++ class extension.

4 TGI support and plugins

While GNU Bayonne offers a ccScript virtual interpreter for creating telephony applications, we wanted to be able to integrate support for databases and other things. There are systems and scripting environments such as Perl which already offer database connectivity. So we created a concept called "TGI," which, like CGI, allows external executables to be invoked from within a call flow script, and the results to be recorded so that information can be passed both to and from the user.

The TGI model for GNU Bayonne is very similar to how CGI works for a web server. In TGI, a separate process is started, and it is passed information on the phone caller thru environment variables. Environment variables are used rather than command line arguments to prevent snooping of transactions that might include things like credit card information and which might be visible to a simple "ps" command.

The TGI process is tethered to GNU Bayonne thru stdout and any output it generates is used to invoke server commands. These commands can do things like set return values, such as the result of a database lookup, or they can do things like invoke new sessions to perform outbound dialing. A "pool" of available processes are maintained for TGI gateways so that it can be treated as a restricted resource, rather than creating a gateway for each concurrent call session. It is assumed gateway execution time represents a small percentage of total call time, so it is efficient to maintain a small process pool always available for quick TGI startup and desirable to prevent stampeding if say all the callers hit a TGI at the exact same moment.

TGI does involve a lot of overhead, and so in addition we have the ability to create direct command extensions to the native GNU Bayonne scripting languages. These command extensions can be processed thru plugin modules which can be loaded at runtime, and offer both scripting language visible interface extensions, and, within the plugin, the logic necessary to support the operation being represented to the scripting system. These are much more tightly coupled to the internal virtual machine environment and a well written plugin could make use of thread pools or other resources in a very efficient manner for high port capacity applications.

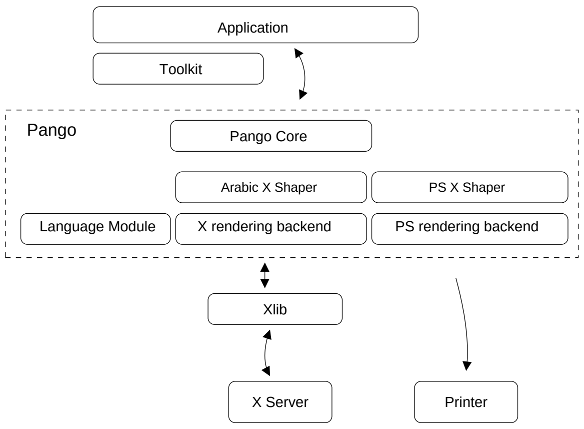


Figure 1: Architecture of GNU Bayonne

5 Architecture

As can be seen, we bring all these elements together into a GNU Bayonne server, which then executes as a single core image. The server itself exports a series of base classes which are then derived in plugins. In this way, the core server itself acts as a "library" as well as a system image. One advantage of this scheme is that, unlike a true library, the loaded modules and core server do not need to be relocatable, since only one instance is instantiated in a specific form that is not shared over arbitrary processes.

When the server comes up, it creates gateways and loads plugins. The plugins themselves use base classes found in the server and derived objects that are defined for static storage. This means when the plugin object is mapped thru dload, its constructor is immediately executed, and the object's base class found in the server image registers the object with the rest of GNU Bayonne. Using this method, plugins in effect automatically register themselves thru the server as they are loaded, rather than thru a separate runtime operation.

The server itself also instantiates some objects at startup even before main() runs. These are typically objects related to plugin registration or parsing of the config file.

6 Hardware Requirements

Since GNU Bayonne has to interact with telephone users over the public telephone network or private branch exchange, there must be hardware used to interconnect GNU Bayonne to the telephone network. There are many vendors that supply this kind of hardware and often as PC add-on cards. Some of these cards are single line telephony devices such as the Quicknet LineJack card, and others might support multiple T1 spans. Some of these cards have extensive on-board DSP resources and TDM busses to allow interconnection and switching.

GNU Bayonne tries to abstract the hardware as much as possible and supports a very broad range of hardware already. GNU Bayonne offers support for /dev/phone Linux kernel telephony cards such as the Quicknet LineJack, for multiport analog DSP cards from VoiceTronix and Dialogic, and digital telephony cards including CAPI 2.0 (CAPI4Linux) compliant cards, and digital span cards from Intel/Dialogic and Aculab. We are always looking to broaden this range of card support.

At present both voice modem and OpenH323 support is being worked on. Voice modem support will allow one to use generic low cost voice modems as a GNU Bayonne telephony resource. The openh323 driver will actually require no hardware but will enable GNU Bayonne to be used as an application server for telephone networks and softswitch equipment built around the h323 protocol family. At the time of this writing I am not sure if either or both of these will be completed in time for the 1.0 release.

7 GNU Bayonne and XML Scripting

Some people have chosen to create telephony services thru web scripting, which is an admerable ambition. To do this, several XML dialects have been created, but the idea is essentially the same. A query is made, typically to a web server, which then does some local processing and spits back a well formed XML document, which can then be used as a script to interact with the telephone user. These make use of XML to generate application logic and control much like a scripting language, and, perhaps, is an inappropriate use of XML, which really is designed for document presentation and inter- exchange rather than as a scripting tool. However, given the popularity of creating services in this manner, we do support them in GNU Bayonne.

GNU Bayonne did not choose to be designed with a single or specific XML dialect in mind, and as such it uses a plugin. The design is implimented by dynamically transcoding an XML document that has been fetched into the internal ccScript virtual machine instructions, and then execute the transcoded script as if it were a native ccScript application. This allows us to transcode different XML dialects and run them on GNU Bayonne, or even support multiple dialects at once.

Since we now learn that several companies are trying to force thru XML voice browsing standards which they have patent claims in, it seems fortunate that we neither depend on XML scripting nor are restricted to a specific dialect at this time. My main concern is if the W3C will standardize voice browsing itself only to later find out that the very process of presenting a document in XML encoded scripting to a telephone user may turn out to have a submarine patent, rather than just the specific attempts to patent parts of the existing W3C voice browsing standard efforts.

8 Current Status

At the time of this paper's publication, the 1.0 release of GNU Bayonne should already be in active distribution. This release represents several years of active development and has been standardized in how it operates and how it is deployed. Even before this point, and for the past 6 months, active development has happened on a second generation GNU Bayonne server, and snapshots of this new server are currently available for download. Where GNU Bayonne is evolving will be explained further on.

9 GNU Bayonne the Meta Projects

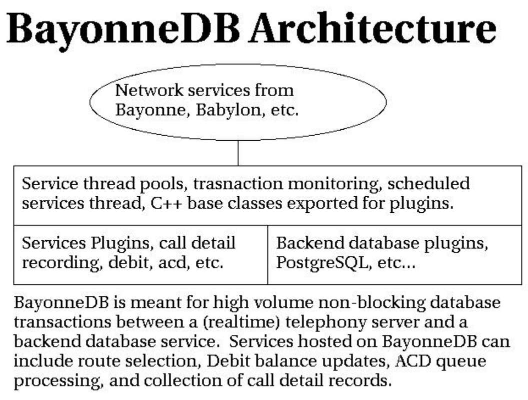
GNU Bayonne does not exist alone but is part of a larger meta-project, "GNUCOMM." The goals of GNUCOMM is to provide telephony services for both current and next generation telephone networks using freely licensed software. These services could be defined as services that interact with desktop users such as address books that can dial phones and softphone applications, services for telephone switching such as the IPSwitch GNU softswitch project and GNU oSIP proxy registrar, services for gateways between current and next generation telephone networks such as troll and proxies between firewalled telephone networks such as Ogre, realtime database transaction systems like preViking Infotel and BayonneDB, and voice application services such as those delivered thru GNU Bayonne.

10 Transactional Databases

BayonneDB is mentioned briefly for transactional services. When we conceived of the need for a transactional database server, we considered that database queries might be slow. The telephony server does not want to do nothing while a transaction is completing, especially if it takes many seconds to happen. Maybe the caller needs to be played music on hold or given other options.

To accomplish non-blocking transactions that allow the telephony server to continue call processing, we choose a peer messaging architecture. A request would be sent to an external server for a transaction, and when the transaction completes, a result message would be sent to the server. There can be time-out and retransmission controls which allow this to be conduced thru UDP packets rather than potentially blocking TCP sessions. This set of protocols and specifications was created initially by Zaheer Milari and myself and published early last year.

BayonneDB was an attempt to implement the concepts in an operational server. Like Bayonne, BayonneDB offers abstraction thru plugins and is based on GNU Common C++. In the case of BayonneDB, it is designed to abstract the interface to the underlying database server used to complete the transaction request. Being threaded, BayonneDB can maintain a persistent threadpool of database connections to optimize overall query performance. A short di-


BayonneDB Architecture
Network services from Bayonne, Babylon, etc.
Service thread pools, trasnaction monitoring, scheduled services thread, C++ base classes exported for plugins.
Services Plugins, call detail recording, debit, acd, etc.
Backend database plugins, PostgreSQL, etc...

BayonneDB is meant for high volume non-blocking database transactions between a (realtime) telephony server and a backend database service. Services hosted on BayonneDB can include route selection, Debit balance updates, ACD queue processing, and collection of call detail records.

Figure 2: Architecture of BayonneDB

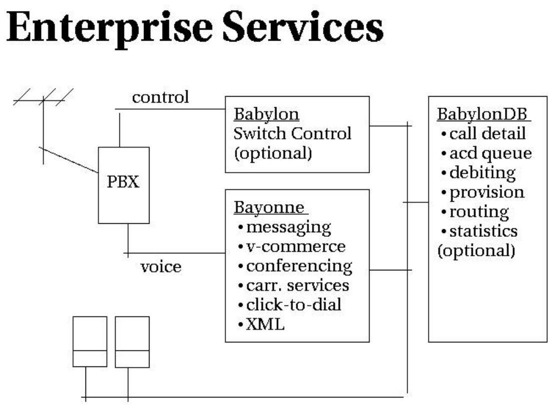


Figure 3: Enterprise Applications Today

agram of BayonneDB architecture is presented below:

11 Enterprise Applications

In our broadest view of enterprise telephony applications, we can see using GNU Bayonne as a part of an overall solution. GNU Bayonne must be able to interact with enterprise data, whether thru transaction monitors such as BayonneDB or thru perl scripts executed via TGI. It may need to interact with other services such as email when delivering voice messages to a unified mailbox, or the local phone switch thru

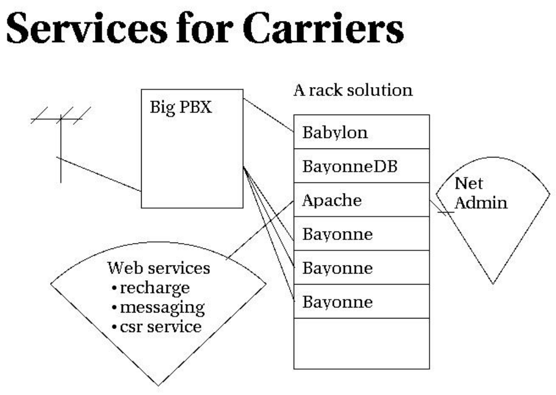


Figure 4: Carrier Applications Today

a resource such as Babylon. We will explain Babylon a bit later.

Our view of GNU Bayonne and telephony application services are that it is a strategic and integral part of the commercial enterprise. Proprietary solutions that are in common use today have often been designed from the question of how to lock a user into a specific OEM product family and control what a user or reseller can do or integrate such products, rather than from the question of what the enterprise user needs and how to provide the means to enable it. This has often kept telephony separate and walled off from the rest of the enterprise. We do not wish to see it separate but a natural extension, whether of web services, of contact management, of customer relations, etc.

12 Carrier Applications

When we look at carrier class applications for GNU Bayonne today, we typically consider applications like operator assistance systems, prepaid calling platforms, and service provider voice mail. Each of these has different requirements. What they have in common is that a front end central office switch might be used, such as a Lucent Excel or even a full ESS

5 switch. Application logic and control for voice processing would then be hosted on one or more racks of GNU Bayonne servers most likely interconnected with multiple T1 spans. If database transactions are involved, such as in pre-paid calling, perhaps we would distribute a BayonneDB server to provide database connectivity for each rack. A web server may also exist if there is some web service component.

Operator assist services are probably the easiest to understand. Very often a carrier might need to provide directory assistance or some other form of specialized assist service. A call will come in from the switching center to a GNU Bayonne server, which will then decide what to do with the call. If the caller is from a location that is known, perhaps the call will be re-routed by GNU Bayonne thru an outgoing span to a local service center. Online operator assistance might be done by creating an outgoing session to locate an operator and then bridge the callers, all on a GNU Bayonne server.

In service provider voice mail one doesn't have to bridge calls. Service provider voice mail is typically much simpler than enterprise voice mail; there is no company voice directory, there is no forwarding or replying between voice mailboxes, there may be no external message notification. All these things make it an easy to define application on first apperance. What it must be is reliable, and ideally scalable.

The problem with service provider voice mail is where to store the potentially large pool of message boxes. We don't know what callers might call in for messages or when. If the call capacity is larger than a single server can handle even with multiple T1 spans, then we might need to deal with a reliable message store hosted on a machine outside the GNU Bayonne servers. We could also scatter mailboxes over multiple machines by hashing the mailbox address into a GNU Bayonne server address, and load balance over multiple servers that way.

If we have a common external message store, perhaps we can have it on a fibre channel bus. GNU Bayonne doesn't like blocking, and traditional network file systems, like NFS, can have long timeout and blocking intervals. Messages can also be transported from a central store over different procotols. One thought I had was a UDP based transport protocol for voice messaging. Since the need is not for full duplex voice, many of the issues in regard to latency and packet size can be relaxed for transporting a voice stream over what is typically required to make VoIP systems work. With a network addressable message store, GNU Bayonne can provide a reliable platform for service provider voice mail.

Many applications carriers wish to deploy do not nessisarly require "carrier grade" Linux to appear before they can be used. In fact, IDT Corp, a major provider of prepaid calling in the world today, uses over 500 rack mounted commodity PC's running things including a standard distribution of "RedHat" GNU/Linux to reliably service over 20 million call minutes per day in their main switching center. This does not mean there is no value in the carrier grade kernel work, just that it is not nessisary to create and sell some types of GNU/Linux voice processing solutions for carriers today. We have looked at the issues involved in high reliability/carrier grade enhanced Linux and we intend to address those as described a little further.

13 GNU Bayonne clustering

In England one enterprising fellow is working on GNU Bayonne tandem switching nodes. A tandem switching node essentially routes call traffic between spans based on various rules, perhaps to achieve a low interconnection count or to find the least cost available route in a telephone network. This touches upon an interesting and unique feature of GNU Bayonne which we have not yet talked about; GNU Bayonne servers talk to each other.

When Bayonne servers talk with each other, they do two things. Each node elects a "buddy" node to act as a failover for itself. Elections are held every few minutes and the design of this is that a single node will only elect itself to buddy up to two additional nodes in the network. Buddies are useful in failover, since they are aware of all transactions and the state of each GNU Bayonne server, and can complete transactions if a given machine (node) goes down. By having a limited set of buddies chosen thru election, we assure there is no network stampede when a node goes down on the part of other nodes wishing to complete transactions for it.

Since global call state is shared among GNU Bayonne servers, each server knowns what the other one is doing and what its current utilization is like. This can be very useful in a tandem switching application where one needs to know where available endpoints are and if there are ports available at each end point for a given call request. GNU Bayonne cluster networking is still in its infancy, and we are looking for ways to express networking thru the application scripting language.

The main use of clustering at the moment is to overcome the inherit limits of system reliability for acceptance of GNU Bayonne in developing carrier class applications. Over time, this need will be lessened as we take advantage of the work being done on carrier grade GNU/Linux.

14 The NG Server

Even before GNU Bayonne 1.0 had been finalized, work had been started by late last year on a successor to GNU Bayonne. This successor attempts to simplify many of the architectural choices that were made early on in the project to make it easier to adept and integrate GNU Bayonne in new ways.

One of the biggest challegnes in the current GNU Bayonne server is the creation of telephony card plugins. These often involve the implementation of a complete state machine for each and every driver, and very often the code is duplicated. GNU Bayonne "2" solves this by pushing the state machine into the core server and making it fully abstract thru C++ class extension. This allows drivers to be simplified, but also enabled us to build multiple servers from a single code base.

Another key difference in GNU Bayonne "2" is much more direct support for carrier grade Linux solutions. In particular, unlike GNU Bayonne, this new server can take ports in and out of service on a live server, and this allows for cards to be hotplugged or hot swapped. In a carrier grade platform, the kernel will provide notification of changeover events and application services can listen for and respond to these events. GNU Bayonne is designed to support this concept of notification for management of resources it is controlling.

Finally, GNU Bayonne "2" is designed from the ground up to take advantage of XML in various ways. It uses a custom XML dialect for a configuration language. It also acts as a web service with both the ability to request XML content that describe the running state of GNU Bayonne services and the ability to support XMLRPC. This fits into our vision for making telephony servers integrate with web services, and will be described further in a seperate paper.

15 Acknowledgments

There are a number of contributors to GNU Bayonne. These include Matthias Ivers who has provided a lot of good bug fixes and new scheduler code. Matt Benjamin has provided a new and improved tgi tokeniser and worked on Pika outbound dialing code. Wilane Ousmane helped with the French phrasebook rulesets and French language audio prompts. Henry Molina helped with the Spanish phrasebook rulesets and Spanish language audio prompts. Kai Germanschewski wrote the CAPI 2.0 driver for GNU Bayonne, and David Kerry contributed the entire Aculab driver tree. Mark Lipscombe worked extensivily on the Dialogic driver tree. There have been many additional people who have contributed to and participated in related projects like GNU Common C++ or who have helped in other ways.

Prospect: A Sampling System Profiler for Linux Design, Implementation, and Internals

Alex Tsariounov
Hewlett-Packard Company
3404 E. Harmony Rd., MS42
Fort Collins, CO 80528
alex_tsariounov@hp.com

Bob Montgomery
Hewlett-Packard Company
3404 E. Harmony Rd., MS42
Fort Collins, CO 80528
bob_montgomery@hp.com

Abstract

Prospect is a developer's profiling tool for the Linux operating system. Prospect is an instruction-pointer-sampling flat profiler for obtaining code profiles in a non-intrusive way. One can obtain profiles (both symbol-level and assembly-level) without undue requirements on the target application. For example, there is no need to specially instrument, rebuild, or relink the application. In fact, the only requirement is that the application not be stripped.

Prospect has a history on the HPUX operating system where it was invented in 1988. In the last year's time frame, we have moved this profiler to Linux with the aid of the sampling module oprofile released under GPL by John Levon while at the Victoria University of Manchester, UK. We describe the interface to oprofile, the data structures and algorithms used to collect and store the instruction pointer and system event data, and the symbol profile generation.

1 Introduction

In 1988 Doug Baskins at HP wanted to know exactly what an HP-UX machine was doing during operations. He ended up designing and implementing the Kernel Instrumentation (KI) package (a kernel tracing facility) and a testing tool for it. The testing tool became valuable in its own right and was named "Prospect" after the gold prospectors of the past. Just as when one prospects for gold and finds the occasional nugget, so does one also use Prospect to find nuggets of performance data. The KI and Prospect live on to this day in modern HP-UX systems.

In the last year's time frame we have moved this idea to Linux where we needed such a performance analysis tool. Prospect produces flat symbol and assembly-level profiles through instruction pointer (IP) sampling. All applications running on the system are profiled and produce both user and kernel profiles. Prospect also generates kernel-only profiles in its reports. One can obtain profiles without undue requirements on the target applications. For example, there is no need to specially instrument the application and there is no need to rebuild or relink. In fact, the only requirement is that the applications not be stripped. Shared libraries escape this requirement for the most part as do assembly-level profiles.

Prospect on Linux uses the oprofile module developed by John Levon [Levon] while at the Victoria University of Manchester, UK. Oprofile is a neat project that uses the P6 performance counters to clock an NMI sampler and is an active GPL project at this time with growing contributions. The goals of oprofile and Prospect are similar though parallel and thus the two tools complement each other nicely.

Hewlett Packard has recently allowed Prospect to become Open Source and released it under the GNU General Public License Version 2. Hosting arrangements have not been determined yet for the project at the time of this paper.

In your exploration of HP's web sites you may find reference to an HP-UX version of Prospect. This is still an active project currently maintained by one of the authors but it is not Open Source. However, it *is* available for free for the HP-UX operating system and can be downloaded from `ftp://ftp.cup.hp.com` in the `dist/networking/tools/prospect` subdirectory. The HP-UX version of Prospect does not use oprofile or GDB in any way.

2 Architecture

Figure 1 shows the overall architecture of Prospect. Note that not all parts are used all the time. In fact, there are several phases that Prospect goes through in the typical run that we describe later on.

As can be seen from Figure 1, Prospect has the following major architectural modules:

Oprofile Sampling Module This is the interface that provides us with most of our data. Prospect uses `/proc` upon initialization to record all current activity on the system, however, all data from that point on is provided by oprofile. Oprofile comes with a user-space daemon that provides some parallel functionality to Prospect. Prospect takes the place of the oprofile user-space daemon and only uses the oprofile sampling module. Most of the attractive properties of Prospect – properties such as non-intrusiveness, no special build nor link requirements, total system picture, accurate kernel profiles – are in fact provided by the oprofile module.

Prospect System Model All processes on the system are modeled as data structures of some process-specific information and a virtual address space consisting of a doubly linked list of executable regions. These regions have path name and symbol information encoded in them. The symbol information is filled in after the sampling run when all regions that had instruction pointer hits are read in.

Data Storage and Retrieval Most data in Prospect is stored in a digital tree (also called a trie, see [Fredkin] and [Knuth]) data structure. This type of data structure provides a way to store sparse data without undue storage or management requirements. Retrieval performance is acceptable for most cases. In one case we implement a cache in front of the tree to improve retrieval performance.

Symbol Mapping with ELF All symbol information that Prospect requires is located in each executable ELF file. Prospect uses `libelf` to read this symbol information out of the executable files (applications, shared libraries, and kernel). The `libelf` library provides a nice, platform-independent way to get at this information. For each region that had instruction pointer hits, Prospect creates a searchable symbol table from this information: local (static) symbols are promoted to global for searching purposes and duplicate symbols are noted. At the end of the run in the profile generation phase, Prospect then has the addresses of all instruction pointer hits stored for

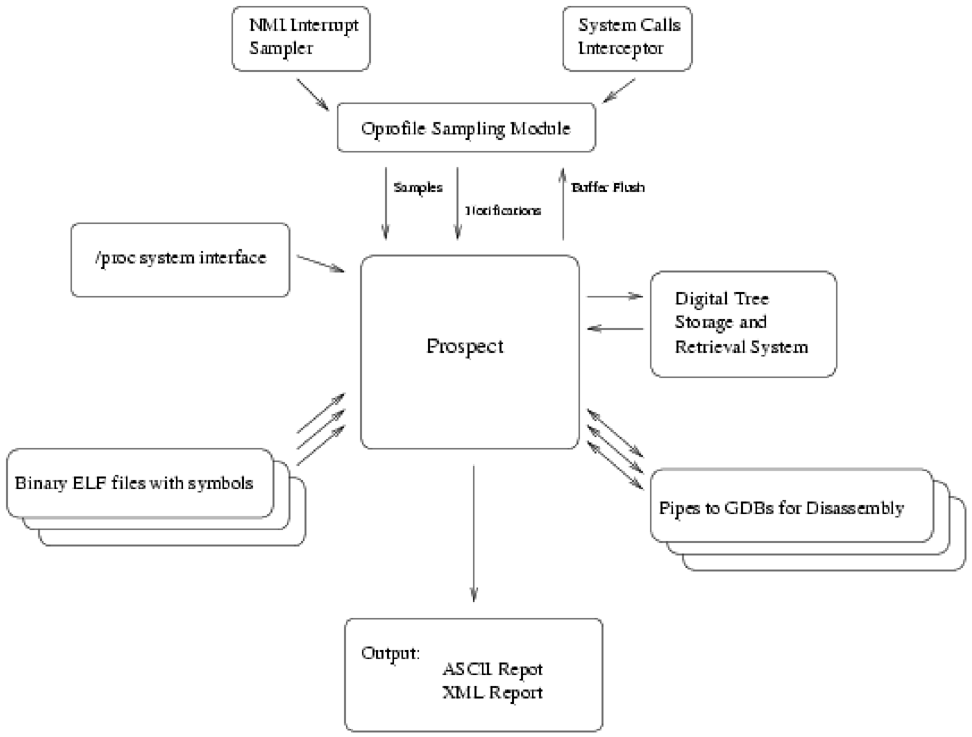


Figure 1: Architecture of Prospect

each process in its hit trie as well as a sorted array of symbols. Prospect then `bsearch`es the symbol array for each distinct instruction pointer address to produce the profile.

Stripped Shared Libraries If the executable file is stripped, then there are no symbols to read out. This is one of the few restrictions that Prospect places on executables for generating symbol-level profiles (note that assembly-level profiles are always available). For shared libraries, it turns out that Prospect can use the dynamic symbol table even if the file has been stripped. The caveat is that local (static) symbols are removed. Since a lot of hits are to local symbols, Prospect attempts to produce useful data by bracketing the hits between the closest global symbols. For example, this produces profiling symbols such as `bsearch->qsort` for a hit to a static routine defined somewhere between the global symbols of `bsearch` and `qsort` in `libc`. While this is not a perfect solution, it is hoped that it at least points to where to look further for more information and it is better than not producing any data at all. The assumption of course is that the static symbol is related to one of the two global symbols in the bracket.

GDB for Assembly-Level Profiles Assembly-level profiles have been a feature of Prospect from the early days. Originally, these profiles were more of a test than anything because they were relatively easy to produce. Prospect already had the exact addresses of the instruction pointer hits and all that was necessary was to read out the instructions at those addresses and display them. It turned out that this functionality became very useful – even in light of the fact that modern processors make this type of profile almost impossible to produce with 100% accuracy because of pipeline effects and such. Nevertheless, if one is familiar with the architecture and behavior of the processor, the disassembled profile produced by Prospect becomes a valuable aid in determining cache and TLB effects in the code being profiled. Prospect wraps GDB with a lightweight pipe communication wrapper to actually perform the disassembly. This allows Prospect to use GDB as a library and escape disassembly complexities such as variable instruction size on IA32 systems. Since a GDB process can take approximately 3.2 MB in memory, Prospect keeps a user-configurable queue of open pipes to GDB processes for the disassembly phase.

The short descriptions above give an overall picture of the functional modules involved in Prospect. To complete this picture, it is useful to see an enumeration of run time phases that the software shifts through for a typical run:

1. Load and initialize the oprofile sampling module.
2. Go through all processes in `/proc` and set up process data structures with virtual address space (VAS) information.
3. Attach to and activate the oprofile module.
4. Set up periodic alarm signal to flush the oprofile buffer.
5. Exec the child process.
6. Go into a blocking read loop on the oprofile device file.
    - Every alarm, flush the oprofile buffer and check for child exit.
    - The flush allows us to read the oprofile buffers and store that information in memory.
    - Should the oprofile buffer fill before the alarm signal goes off, then the read will just go through allowing the information to be stored and Prospect then reblocks in the read loop.
7. If child has exited, stop profiling and empty the oprofile buffer.
8. Go through the storage structures of all processes and extract all program counter hits. Match these to the files stored in each process's VAS and create the report.
9. Shutdown and leave the oprofile module in memory.

3 Detailed Descriptions

Now that we have a general idea of the Prospect architecture and control flow, this section will describe a few functional modules in greater detail. Three modules are of interest: the oprofile interface, the `dtree` digital tree implementation, and the GDB interface for assembly-level profile generation.

3.1 The Oprofile Module Interface

The oprofile project is a very active GPL project (see http://oprofile.sf.net) that provides an instruction pointer sampling mechanism that is clocked off the Pentium P6[1] series performance counters.[2] The performance counter events are routed through the APIC to the NMI pin of the processor. The module registers a short sampling routine to the NMI vector that

[1]P6 series covers Pentium Pro through Pentium III

[2]There are two configurable counters on the P6; AMD chips are also supported

actually does the sampling. The instruction pointer samples are kept in a trick constant-size hash table in kernel memory and dumped to user space though a device interface file. Normal operation is to have the user-space daemon block on reading this file. When the kernel buffer nears capacity, or when a "`1`" is sent to the buffer flush `/proc` oprofile interface file, the read succeeds and the data is transfered. In addition to instruction pointer samples, the oprofile module also intercepts certain system calls and passes this trace through another device interface file. Finally, oprofile also keeps track of opened files in a hash table in kernel memory for determining the path of `exec`'ed and `mmap`'ed files. This hash table is accessible from user space and is normally memory mapped. All of this information taken in its entirety allows Prospect to generate its profiles.

Prospect acts as the user-space daemon functionality wise (and in fact replaces the oprofile user-space daemon for the purpose of Prospect), but is not a daemon at all and is removed from memory after every run (the oprofile module is left resident). Both Prospect and the oprofile tools have similar but different goals and so the tools complement each other nicely. The goal of Prospect was two-fold: (1) make oprofile trivially easy to use; (2) create a profile of all system activity specified within a certain time interval. This interstice of time is specified by the child of the Prospect process in the same manner as for the `/bin/time` command.

The oprofile tools were designed to unobtrusively run in the background, sampling system information continuously. Thus one would have a long term profiling record of all processes on the system. Also, the effects of short-lived processes are merged into single profile files identified by filenames in a `/var` location when you use the oprofile tools.

While there are a number of events that can be used to clock the oprofile sampling module, Prospect always uses the CPU_CLK_UNHALTED event by default. However, the setup code was written in such a way that if Prospect detects that oprofile was set up already at initialization time, then the existing set up is not changed. Thus you can use a different event to clock the sampler by using the `op_start` script supplied with the oprofile distribution. In effect, you can generate profiles based on any of the performance monitor events – see the oprofile documentation for details.

Using the oprofile modules entails the following interesting phases:

Initialization Upon initialization, Prospect first finds the oprofile module and if not loaded, loads it. Prospect then attempts to open the hash map device: if successful, then Prospect has control of the oprofile module. Only one user can have the oprofile device files open at a time. If this was a fresh load, then the performance counter 0 is set up to clock CPU_CLK_UNHALTED at a default frequency of 200 Hz (unless specified otherwise with the `-H <Hertz>` command line parameter). If the oprofile module was already loaded, then the counter setup is not touched with exception of the frequency. We do not open the sample device yet since that starts profiling.

Next, Prospect goes through all processes currently existing in `/proc` and creates data structures for all of them indexed by PID. Prospect then reads in the `System.map` file and locates an uncompressed kernel image. If Prospect can't read in the `System.map` file then kernel profiles are disabled; if an uncompressed kernel image is not available, then disassembled kernel profiles will not be produced. You can specify both of these files with the `-K` and `-L` arguments – however, Prospect will check the specified `System.map` against `/proc/ksyms` and if they don't match, kernel profiles will not be available.

At this point, the oprofile devices are opened which starts profiling. Prospect next sets up the periodic alarm signal to flush the oprofile buffer. Now, Prospect `fork`'s and `exec`'s the command line that was passed to it and goes into the read-block loop on the oprofile device files.

Periodic Flush Every two seconds by default, Prospect flushes the oprofile buffer. This causes oprofile to allow its profiling data to be read out of the kernel holding buffers through the oprofile device files. This continues for the duration of the child run. After Prospect blocks on the read on the samples oprofile device, one of two things can happen: (1) the alarm signal goes off and takes us out of block, or (2) the notification and/or the samples buffer fills to high water causing the read to succeed. If the alarm goes off, then Prospect writes a "`1`" to the oprofile flush device file and re-reads the sample file. This time the read will go through for both the samples buffer and the notifications buffer.

The tradeoff when using periodic flushing with oprofile is deciding how much data the kernel gets to keep in the instruction pointer hash table by adjusting the periodic flush rate and balancing that against the bounding time interval. The further apart the periodic flush command is, the more data is stored in kernel memory and the longer a read will block. This rate can be adjusted with the `-M <value>` command line parameter. The `<value>` is in hundredths of a second; for example, 200 is every two seconds. If the flush rate is set to a very short interval (0.01 seconds is the minimum allowed), then Prospect will stop sampling very shortly after the child application terminates. If however, the flush rate is longer then it can take up to that amount of time before Prospect stops sampling after the child application exits. And of course, there is more system intrusion for faster flush rates.

Shutdown Every time a read succeeds Prospect checks for the child exit with a `waitpid`. Upon child exit, Prospect then empties the oprofile buffer by issuing a flush and read combination twice in succession. The buffers are read as many times as there are CPUs for each cycle. Next, the oprofile device files are closed. This stops profiling. Prospect leaves the oprofile module in memory on assumption that it will be used later on. At this point, Prospect has all the data it needs and can generate the report.

The files `linux_module.c` and `linux_module.h` hold the functions and definitions that interface with the oprofile module. The file `rec_proc.c` orchestrates the sequence for data collection.

3.2 The `dtree` Module

Most data used in Prospect is kept in a data structure called a Digital Tree and is called `dtree` in the code. The name trie is synonymous with this. This data structure is discussed by Knuth[Knuth] in his third volume of *The Art of Computer Programming*. The trie lends itself very well to managing sparse data, and the data that Prospect collects is sparse consisting mainly of: (1) process structures indexed by PID, and (2) instruction pointer hits indexed by memory address. The concept is easily stated:

the value of the index is implied in the structure of the trie itself. The subdirectory `dtree` holds all the files for this implementation.

3.2.1 Basic Description

Prospect happens to use a quaternary trie. This means that there are four items in each node and so 2 bits translate each node path. This equates into a trie that can be 16 levels deep for a 32-bit entity (32 levels for 64 bits). Because of the quaternary nature, five consecutive indices will extend the 32-bit trie branch to the maximum level. This also means that five consecutive indices will extend the 64-bit version to its maximum level too since the current implementation does not double the order of the trie along with the word size for the 32- to 64-bit transition. This has not proven to be a performance burden yet.

Traversing the trie to find a value consists of repeatedly using the top two bits to choose which quaternary branch to follow and shifting up by two after the choice until a value is arrived at. If a node contains values, it is called a flower and since it is a quaternary trie, it can have up to four values in a flower.

3.2.2 Operation and Use

The `dtree` module is used to store a `void` pointer at the leaf. This can be an actual pointer that points to some structure (as in the process structures indexed by PID), or it can just be used as a `long` variable to store a count (as for the number of hits indexed by memory address). The `dtree` functions return a `void*` that should be cast appropriately.

The `dtree.h` header file defines the following shortcut functions for using the structure easily:

```
Insert:   DTI(Pdt, Idx)
Get:      DTG(Pdt, Idx)

First:    DTF(Pdt, Idx)
Next:     DTN(Pdt, Idx)
Prev:     DTP(Pdt, Idx)
Last:     DTL(Pdt, Idx)
```

In these definitions, `Pdt` should be a `void` pointer to a dtree variable and `Idx` should be a `long` for index value.

Since the `dtree` module handles all memory management, setting up a trie is as easy as declaring a `NULL void` pointer and inserting the first value. For example, to add a certain amount of hits to a counter variable indexed by address:

```
void *hit_tree=NULL;
void add_hits(int hits,
              char *addr) {
  int *hptr;
  hptr = (int*)DTI(hit_tree,
         (unsigned long) addr);
  if (hptr) *hptr += hits;
}
```

The `DTI` insert method (which stands for "dtree insert") will return a pointer to a non-null quantity if an item at that index (`addr` in the example above) exists, otherwise the returned pointer points to `NULL`. The function will return an actual `NULL` only if `sbrk` fails.

Using this method to store a pointer to an arbitrary structure is very similar as the following example shows.

```
struct big_struct **bs_ptr;
void *tree=NULL;

bs_ptr = (struct big_struct*)
         DTI(tree, 451);
if (*bs_ptr==NULL) {
    /* first insert */
    *bs_ptr=malloc(
            sizeof(big_struct));
    *bs_ptr->bs_member = 42;
}
else {
     *bs_ptr->bs_member += 42;
}
```

The next method is DTG which stands for "dtree get". This method is used to query the trie if a value at an index is present. This function will return a NULL if there is no value present, and a pointer to the value if it is. For example:

```
int *var;
void *trie;
unsigned long PC;

/* some code here */

var = (int*)
      DTG(trie, 0x80004533);
if (var)
   printf("Value = %d\n,*var);
else
   printf(
    "No value at 0x80004533\n");
```

The next two dtree functions form the mechanism for the common way to extract all information out of the trie. These are DTF (for "dtree first") and DTN (for "dtree next"). An actual code example from Prospect follows. Here we pull out all instruction pointer hits for a particular region for a particular process.

```
#typedef unsigned long ul;
process_t *p;
ul *P;
...
/* extract profile and build table */
for (Index=0L,
     P = (ul*) DTF(p->pr_profile, Index);
     P != (ul*) NULL;
     P = (ul*) DTN(p->pr_profile, Index)
    )
{
  if (*P == 0) {
    mBUG("*P=0 on user profile extract");
    continue;
  }
  BuildUserSymbolTbl(Index, (ul) *P, p);
}
```

The function BuildUserSymbolTbl does the work of building a profile table and is called for every address in the profile with the amount of hits to that address. Note that these dtree methods (DTF,DTN,DTP,DTL) expect a variable for the Index argument. This is because the methods set that variable to the index where the returned value is found.

Note also that all the dtree access methods actually return a pointer to a pointer, thus casting is necessary to get things right.

3.2.3 Caching for Better Performance

Not much performance analysis was done on the dtree structure itself, however, recognizing some patterns in the incoming data allowed performance upgrading of the method overall by putting a small cache in front of the trie. This is especially useful for the process structure indexed by PID trie, and as it happens, the cache is not used for the hits indexed by memory address tries.

The software cache size was selected to match the cacheline size of the machine: the cache is used to hold four indices and four corresponding values. At machine word length, this corresponds to a 32-byte cacheline for a 32-bit machine, and a 64-byte cacheline for a 64-bit machine. This cache can then store up to four index/value pairs. Every time through the cached lookup routine, we first see if the index that is asked for is in the cache. If it is, the corresponding value is returned out of the cache. If it is not, then we do a true lookup and afterwards, we choose a index/value pair at random in the cache and replace it with the new value. The value looked up is then returned. The random generator is the lower two bits of the free-running counter – the TSC on an IA32 chip, and AR44 register on IPF.

Even though the item replacement is random, the cache is quite effective. You can turn on some statistics output by uncommenting the #define PERFTEST line in the prospect.h header file and running prospect on any load.

3.3 Controlling GDB: The dass_gdb Wrapper

The initial HP-UX implementation of Prospect had a built-in disassembler for the PARISC instruction set. This was a fixed-size instruction set and table driven. For the IA32 port the instruction set was rather different and instructions were no longer of fixed size. After realizing that GDB provides a nice disassembly facility and that when Prospect was disassembling instructions and generating the IPD (Instruction Profile Disassembly) profiles, it was in a non-performance-critical stage, we decided to create a wrapper around this GDB facility.

Use of this wrapper allows the programmatic symbolic disassembly of any binary file on the system and its use is almost transparent when moving from IA32 to IPF as well. The wrapper uses normal Unix pipes to open a GDB process, send it disassembly commands, and read back the disassembled instructions. Prospect includes code for managing many such open pipes to separate GDB processes. The subdirectory dass_gdb contains the files that implement the wrapper.

3.3.1 Interface Description

The dass_gdb wrapper defines the following functions as its interface:

```
void  *dass_open(const char
                   *filename);
char **dass(void *handle,
            char *begin,
            char *end);
void   dass_free(char **array);
int    dass_close(void *handle);
```

These functions perform the following services.

dass_open() This is the initialization function that takes a file name to disassemble as argument and upon success returns a void pointer to a control structure. The function make sure the file exists and that GDB is accessible. It then fork/execs a GDB on the file and sets GDB ready to accept disassembly directives. The returned control structure pointer can then be used for subsequent management of this process.

dass() This is the main work function in the wrapper. It accepts as arguments: the process control pointer, a starting address, and a finishing address. The instructions disassembled are inclusive of the begin address and exclusive of the end address. For example, a call with X begin address and X+1 ending address will return the disassembled instruction at address X. Note that for IPF, instructions are grouped in bundles of three. Prospect always prints out the full bundle for every disassembled instruction on IPF.

The dass() function returns a list of strings as the return value. Or rather, a char** through which the caller can access this list. It is up to the caller to free the memory used for the list of strings once the caller is done with it. Freeing this memory is done with the next function das_free().

dass_free() This function will free a block of memory that serves as a list of strings for the return value of the function `dass()`. Argument is the `char**` variable that was returned from `dass()`.

dass_close() This function accepts a control structure pointer to a running GDB. The process is killed and all pipes are closed.

3.3.2 The Rolling Queue

Each GDB process can take 3.2 to 3.6 megabytes of memory on average. Since a Prospect output can have potentially hundreds (perhaps even thousands) of separate files associated with regions for all processes that ran during the sampling period, having this many running GDB processes at once can bog the system down. At the same time, you don't want to start and stop a single GDB process every time the profile switches to a different executable or library since that would add a distasteful overhead to this process.

If we further examine Prospect output, then we see some more things that can be used to advantage. For example, a large number of files are used repeatedly for many processes (the shared libraries), and if processes are repeatedly run, all of their disassembled output comes from the same files. What we needed was a way to hold open a number of GDB processes, set at a reasonable default and configurable by the user. Thus, Prospect uses a most-used-first queue of open pipes to running GDB processes with the number of simultaneously opened pipes set by the `-g <number>` parameter.

The way this rolling queue works is as follows. When a file is to be disassembled, Prospect first tries to find an open pipe to a GDB process for that file in the queue by linear search through the queue. If a pipe tied to the desired filename is found, then Prospect moves that pipe to the head of the queue. If the pipe is not found, then Prospect opens a pipe to a GDB for that file with a call to `das_open()`. The pipe is then inserted at the head of the queue. At this point, Prospect checks if there are too many open pipes according to the `-g <number>` parameter, and if there are, Prospect closes the pipe held at the tail of the queue.

Thus, the most often used files for disassembly gravitate toward the front of the queue and are found quicker than the less often used files. The length of the queue determines how many simultaneous pipes to GDB processes are held open and is user configurable. If you set the `-g <number>` parameter to 1 (one), then this mechanism will open and close a GDB process for every file encountered even if it's the same file. This saves memory but costs time in the open/close process overhead. If you set the parameter to a high number, then that many open pipes (and hence running GDB processes) will be held open. This will improve run-time performance, but will cost memory. The default amount of slots in the queue is set to 8. See the file `incache.c` for details on this queue implementation.

4 Example Use

Besides its obvious use in profiling single benchmark and application programs, Prospect is also useful for insight into the behavior of the whole system during interesting workloads. In this example, the netperf benchmark will be used to create a multi-process kernel-intensive workload and we'll explore Prospect's ability to see what's going on.

Our workload is created by this script:

```
for i in 1 2 3 4
do
 netperf -t TCP_RR -H isv204 \
         -l 60 -P 0 -v 0 &
done
```

```
  wait
```

This starts four simultaneous netperf request-response runs to remote server isv204, instructing each to run for 60 seconds. The script ends when all the netperf processes have exited.

A timed run of the netperf4.sh script shows:

```
      2.94user    57.08system
   1:02.10elapsed 96%CPU
```

To profile everything on the system while this script runs, a typical Prospect command line would be:

```
 $ prospect -f prospect.out -V4 \
     -H1000 \
     ./netperf4.sh >netperf.out 2>&1
```

The -V4 option says to trace all active processes on the system during the duration of the command ./netperf4.sh. Prospect will produce a user and kernel profile for each process that meets the CPU time threshold, plus a global profile of the time spent in the kernel. The -H1000 option sets a sample rate of 1000 samples/sec.

Alternately, if the workload were ongoing, or not easily startable from Prospect, you could obtain the full system profile by "profiling" a sleep command of the desired duration:

```
  $ prospect -f prospect.out -V4 \
           -H1000 sleep 60
```

The oprofile module provides Non-Maskable Interrupt (NMI) sampling on hardware platforms that support it, and Prospect normally sets it up to use CPU clock cycle counting to generate the interrupts that collect the IP samples. On platforms where oprofile doesn't support NMI sampling, the Real-Time Clock (RTC) hardware can be used to generate sample interrupts. RTC sampling is most useful for profiling user-mode execution on uniprocessor systems. Since it doesn't use a Non-Maskable Interrupt, it has a blind-spot in kernel mode that will be illustrated in our example: RTC sampling can't catch execution in other interrupt handlers.

4.1 Sampling System Idle Time

What do CPUs do when there is nothing useful to do? They execute an idling routine in the kernel and wait to be interrupted with useful work. On most i386 Linux systems, the cpu_idle and default_idle routines in the kernel are where they wait. The implementation of default_idle can create an interesting problem for Prospect with NMI sampling. The default_idle routine uses the HLT (halt) instruction to stop the CPU while it waits for something to do. But the CPU_CLK_UNHALTED counter used to generate the NMI sampling is so named because it does not advance when the CPU is halted. Sampling just stops when the CPU halts and Prospect doesn't know it. That part of the idle time simply disappears from the profile. The interrupts used by RTC sampling are driven by elapsed time, so the halted CPU is awakened from its halted state and sampled, effectively caught in the default idle routine. But, as mentioned before, RTC sampling has other problems sampling the kernel.

Because of some hardware bug of long ago, Linux provides a boot parameter that allows NMI sampling to see idle time. Appending "no-hlt=1" to the boot string in lilo.conf causes default_idle to become part of a time-wasting loop instead of halting the CPU, and allows Prospect to show us the idle time as hits in the cpu_idle and default_idle routines.

The results below came from three different

Prospect runs: Default NMI mode, RTC mode, and NMI mode on a no-hlt=1 kernel. Since the RTC sample rates are limited to powers of 2, we used -H1024 for all runs so we can compare hit counts as well as equivalent times. Here are some highlights from the Prospect output for the netperf workload:

4.2 Statistics of Run

The "Statistics of Run" section has stats on the operation of Prospect itself. Three counts are of interest here:

	Num samples	System Hits	User hits
Default	61667	58579	3088
RTC	65024	61798	3226
No-hlt	65657	62658	2999

Note that the Default case appears to be missing 3 or 4 seconds worth of 1024 Hz samples. We'll see why later. We can also see that user time is not a significant part of this workload.

The output that Prospect provides is quite extensive. For clarity of presentation, we only reproduce the relevant parts of the output in this document.

4.3 Extrapolated Summary of Processes

This section summarizes all processes seen during the profiling period, dropping the ones that don't meet the CPU usage thresholds (configurable with -k and -m options). The two methods produce similar results:

Default NMI sampling

Process	User	Hits/ Time	System	Hits/ Time
netperf	728	0.7109	14838	14.4902
netperf	709	0.6924	14197	13.8643
netperf	679	0.6631	14140	13.8086
netperf	666	0.6504	14557	14.2158
prospect	301	0.2939	769	0.7510
bash	5	0.0049	2	0.0020

RTC sampling

Process	User	Hits/ Time	System	Hits/ Time
netperf	776	0.7578	14634	14.2910
netperf	768	0.7500	14253	13.9189
netperf	699	0.6826	14442	14.1035
netperf	698	0.6816	14335	13.9990
prospect	278	0.2715	28	0.0273
bash	6	0.0059	4	0.0039

The No-hlt NMI case is not significantly different than the Default case in this section. The missing System Time in the Default case does not show up in the process summary.

Note also that Prospect sees and reports its own overhead, but some parts of the oprofile module necessarily run with sampling disabled, so indirect methods would be required to assess the complete effect of profiling.

4.4 Details of Processes

After the summary, each eligible process is profiled in user mode and in kernel (or system) mode. Although we see from the summary that user mode execution is not a significant contributor to CPU time in this workload, there are a couple of interesting points.

The four netperf user profiles are expectedly similar, showing these three routines and their files:

```
send_tcp_rr in
     /usr/local/netperf/netperf
recv->recvfrom in /lib/libc-2.2.4.so
send->sendmsg  in /lib/libc-2.2.4.so
```

The libc shared library is stripped, so Prospect shows the enclosing symbols from the dynamic symbol table as a range to emphasize that there is some uncertainty. In other words, samples occurred between the recv and recvfrom entries, and between send and sendmsg, but without the normal symbol table, Prospect can't know what static functions might exist in that range. A greater (and possibly too much greater) level of detail within these ranges could be seen by specifying disassembly. Otherwise, linking static or building unstripped shared libraries would also reveal more detail in the user mode profile.

To compare the NMI and RTC sampling methods, here are the user hits reported for the top routines of each netperf process in the USER portion of profile:

Default NMI sampling

Routine name	Hits	Hits	Hits	Hits
send_tcp_rr	410	401	382	376
recv->recvfrom	167	158	155	158
send->sendmsg	145	140	134	126
Total	722	699	671	660

RTC sampling

Routine name	Hits	Hits	Hits	Hits
send_tcp_rr	387	373	359	352
recv->recvfrom	222	194	178	189
send->sendmsg	163	194	158	153
Total	772	761	695	694

Results with the No-hlt kernel are not significantly different than the NMI results shown here. The big differences in the three methods show up in the Kernel portions of the process profiles and in the Global Kernel profile.

Here are the kernel (or system) hits reported for the top kernel routines of each netperf process in the KERNEL portion of profile:

Default NMI sampling

Routine name	Hits	Hits	Hits	Hits
speedo_interrupt	1411	1313	1354	1345
tcp_sendmsg	686	659	616	654
speedo_rx	517	529	544	587
. . .				
speedo_start_xmit	482	491	452	465
. . .				
do_softirq	223	200	226	219
. . .				

RTC sampling

Routine name	Hits	Hits	Hits	Hits
do_softirq	2693	2698	2709	2681
speedo_start_xmit	1422	1383	1425	1339
tcp_sendmsg	665	700	696	650
. . .				

Once again, NMI no-hlt sampling was not significantly different from Default NMI sampling. But as you can see, the profile of the top routines in the RTC version differs significantly from that in the NMI version. While the top two RTC routines do show up in the NMI version with lower hit counts, the two speedo routines in the top three of the NMI version don't show up in the RTC profile at all.

The routines that are missing from the RTC profile are part of the interrupt handler for the eepro100 network driver. They show up under the netperf process KERNEL profile because the interrupt came in when that process was on the CPU. Other routines shown under the process KERNEL profile are actually there as a result of system calls made by the process, but you can't currently tell which are which without knowing something about the code.

As an example of how an "innocent" process can have its profile "corrupted" in this way, a background cpuspin program was run at a nice priority throughout another netperf workload. When running alone on a system, the cpuspin program normally gets 0.06 seconds of system time in a run that consumes 58 seconds of user CPU time. But during the netperf run interval, Prospect reported that the background cpuspin program picked up 7.3 seconds of user time and 3.9 seconds of system time. The KERNEL portion of cpuspin's profile showed these routines at the top:

```
Routine name      Hits
----------------------
speedo_interrupt   714
speedo_rx          397
schedule           269
uhci_interrupt     259
tcp_v4_rcv         210
net_rx_action      173
...
----------------------
```

You can see how this could mislead you into believing something about the cpuspin program that wasn't actually true.

4.5 Global KERNEL Profile

Now it's time to get back to the missing system time in the Default NMI sampling method. The last section of a -V4 Prospect report is the Global KERNEL Profile. It provides a single profile of all system hits in the run. Here are the top routines:

```
Default NMI sampling:
Routine name      Hits
----------------------
speedo_interrupt  5436
tcp_sendmsg       2615
speedo_rx         2182
uhci_interrupt    1978
speedo_start_xmit 1890
schedule          1777
tcp_recvmsg       1623
__rdtsc_delay     1366
...
default_idle        11
...
----------------------

No-hlt NMI sampling:
Routine name      Hits
----------------------
speedo_interrupt  5556
tcp_sendmsg       2612
speedo_rx         2202
default_idle      2070
cpu_idle          1945
speedo_start_xmit 1944
uhci_interrupt    1929
schedule          1716
tcp_recvmsg       1596
__rdtsc_delay     1414
...
----------------------
```

Hit counts are pretty similar except for the two idle routines. The No-hlt kernel allows Prospect to see 4015 hits in the idle routines, while it only saw 11 hits in the Default NMI case. The reported difference in system hits between the two runs was 4079, so we have the culprit. The RTC version still insists that do_softirq is the top kernel routine, but at least it does attribute 4010 hits to default_idle.

A disassembly (-e) run with RTC sampling showed that in the do_softirq routine, a huge concentration of hits occurred in a three-instruction cluster consisting of:

Hits	Address	Instruction
4117	<do_softirq+77>:	lea 0x0(%esi),%esi
94	<do_softirq+80>:	test $0x1,%bl
2961	<do_softirq+83>:	je <do_softirq+93>

This doesn't make a lot of sense until you look at the two instructions just before these:

```
<do_softirq+74>: sti
<do_softirq+75>: mov %ebp,%esi
```

The sti instruction allows the processor to start responding to external maskable interrupts after the next instruction is executed. With RTC sampling, the sample is taken when the interrupt is allowed to occur, not when the real time clock wants it to occur. In this case it was held off until after the sti instruction in do_softirq. NMI isn't held off at all, so the real kernel profile can be seen.

These runs were performed on a 2.4.14 uniprocessor kernel. NMI sampling works well on SMP as well, reporting times that are multiplied by the number of CPUs running. RTC sampling is not as reliable on SMP systems, since each interrupt is processed by only one of the CPUs. The accuracy of the resulting profile depends on the RTC interrupts being distributed evenly across the CPUs.

The drivers on this kernel were built in, not loaded as modules. Prospect can not currently provide profiles of kernel modules, but that capability should be available Real Soon Now.

5 Acknowledgments

A number of people have contributed in many ways to this project, not the least of which are: Doug Baskins, Keith Fish, Michael Morrell, and Bob Metzger, all of whom are at HP.

We would also like to thank John Levon for writing such cool software and putting it under GPL.

References

[Fredkin] Fredkin, E. H., *Trie Memory*, CACM 3:9 (September), pp. 490-500, (1960).

[Knuth] Donald E. Knuth, *The Art of Computer Programming*, Volume 3 Second Edition, pp. 492-507, (1998).

[Levon] John Levon, *The Oprofile System Profiler*. `http://oprofile.sf.net` (2001).

How NOT to write kernel drivers

Arjan van de Ven
Red Hat, Inc.
arjanv@redhat.com, http://people.redhat.com/arjanv

Abstract

Quit a few tutorials, articles and books give an introduction on how to write Linux kernel drivers. Unfortionatly the things one should NOT do in Linux kernel code is is either only a minor appendix or, more commonly, completely absent. This paper tries to briefly touch the areas in which the most common and serious bugs and do-nots are encountered.

1 Introduction

Quit a few tutorials, articles and books give an introduction on how to write Linux kernel drivers. Unfortionatly the things one should NOT do in Linux kernel code is is either only a minor appendix or, more commonly, completely absent.

With the growing popularity of Linux in the last few years, more and more vendors are trying to create linux drivers, and quite often that is done by giving an engineer the windows driver code and the assignment to have a linux driver ready in 4 weeks.

In my job as Red Hat Linux kernel maintainer such drivers quite often end up in my inbox with the request to include it. Surprisingly quite often these drivers share the same bugs and "please don't do *that*" things.

This text and the corresponding talk is intended to show several such bugs and why they are bad. Quite a few will be of the "Oh but of course" caliber but that's usually the case with bugs in hindsight.

My hope is that explaining why certain things are wrong is enough to prevent such things from cluttering up my inbox too much.

All of the code examples in this paper are from real code, however they have been edited slightly to fit the layout and non-essential bits are removed for clarity.

2 Allocating memory

The Linux kernel has a diverse API for allocating memory, unlike operating systems such as Microsoft Windows and SCO Unixware. Linux uses this set of functions and flags one the one hand to be able to more aggressively optimize the VM algorithms, and on the other hand to provide safeguards against out-of-memory deadlocks.

Quite often drivers that are ported from other operating systems try to abstract the multitude of allocators and flags into one function. Not only does this void the VM tuning optimisations, it also leads to subtle and hard to debug bugs.

2.1 GFP_KERNEL and GFP_ATOMIC

Most kernel tutorials describe that you shouldn't use GFP_KERNEL in interrupt context because it can schedule (which is correct), and as a result the following "abstraction" is found quite often:

```
static int uhci_submit_iso_urb(
urb_t *urb)
{
...
   tdm = kmalloc(some_size,
        in_interrupt()
      ? GFP_ATOMIC : GFP_KERNEL);
...
}
```

from usb-uhci.c in kernel 2.4.9

which appears to take this "interrupt context" requirement into account. However the entire cunning plan falls appart when spinlocks enter the picture: in_interrupt() might return false even though scheduling is not allowed. Since scheduling by kmalloc() is not the common case, this bug won't often show up in casual testing.

2.2 vmalloc

Most tutorials warn about using kmalloc for allocating big areas of memory, and it seems a lot of people also notice these limits in practice:

```
static inline void *
osi_malloc(unsigned int size)
{
   void *ptr;
   if (size > 2*PAGE_SIZE)
      return vmalloc(size);
   ptr = kmalloc(size, GFP_NOFS);
   if (ptr)
      return ptr;
   return vmalloc(size);
}
```

from OpenGFS 0.99.2

There are several problems with such an approach. The first one is performance: `vmalloc()`'ed memory requires more TLB's, which are a rare resource on most CPUs.

A more serious problem arises when the `osi_free()` routine gets involved: it calls `vfree()` for the `vmalloc()`ed memory, and it's illegal to call `vfree()` from interrupt context. This abstraction makes it okay to call the `osi_free()` function from interrupt context sometimes, but not for big allocations.

A third issue that most tutorials don't mention is that while you can use `vmalloc()` to allocate larger chunks of memory, the total amount you can allocate is rather limited, in the order of 64 Megabytes on modern machines (depending on what PCI hardware is present).

2.3 Other subtleties

Abstractions of the Linux memory allocators often also hides the deadlock avoidance mechanisms Linux provides. In the write path of a filesystem, using `GFP_KERNEL` or `GFP_ATOMIC` will lead to deadlocks eventually. Such deadlocks might not show up in your testing, but Murphy's law guarantees that your first big customer will hit the deadlock on December 24th at 6pm. `GFP_NOFS` is there for a reason, as is `GFP_NOIO` for block device drivers. Using an abstraction for the allocator might appear to make your life easier by not having to understand these issues, but sooner rather than later it'll come back and haunt you—or, worse, your users.

3 Synchronisation primitives

The Linux kernel provides a reasonably complete set of synchronisation primitives in the form of semaphores and spinlocks (both in the normal form as the reader/writer variant). Rusty's hamster even wrote documentation about when to use what primivive. These primitives however do not form a perfect match with that Microsoft Windows or Unixware provide as primitives.

Not seldom do I find self-written synchronisation primitives in submitted drivers, and as a general rule they are all buggy.

```
/*
 * EnterCriticalSection:
 */
unsigned long
EnterCriticalSection(
DevInfo_pt pDev)
{
  int retval;
#ifdef __SMP__
  for (;;) {
    retval=test_and_set_bit(SEMA_SRL,
      &pDev->Sema);
    if (!retval)
       break;
    sleep_on(&pDev->WaitQ);
  }
  return(retval);
#else /* __SMP__ */
  if (pDev->Sema)
    return(-1);
  pDev->Sema=-1;
  return(0);
#endif
}
```

The code above is a typical example of driver-code that tries to emulate a primitive from another OS. The counterpart was as follows:

```
/*
 * LeaveCriticalSection
 */
void LeaveCriticalSection(DevInfo_pt
pDev)
{
#ifdef __SMP__
  clear_bit(SEMA_SRL,&pDevi->Sema);
  wake_up(&pDev->WaitQ);
#else
  pDev->Sema0;
#endif
}
```

It's tempting to leave the "what's wrong with that" as an excercise to the reader; however it's better to nip such code in the but so here goes a two-cpu example with 2 tasks, task A is holding the lock on cpu 1 and task B is trying to acquire the lock on cpu 2. Figure 1 shows how the mentioned code will leave task B sleeping without it ever getting the critical section.

	CPU 1	CPU 2	State
T0	....	....	Task A has has the sema bit set
T1	....	test_and_set	Task B tries and sees someone has the sema bit
T2	clear_bit	....	Task A releases with a clear_bit()
T3	wake_up	....	Task A waits up all waiters on the waitqueue
T4	....	sleep_on	Task B puts itself on the waitqueue and sleeps
T5	....	....	There's nobody left to wake task B ....

Figure 1: Timing diagram of the deadlock

Now it's very well possible to try to fix the above code to not have this deadlock. However, it's far simpler to just use the linux semaphores which provide all the functionality required for this case.

Another problem with "home made locks" is that they do not work on architectures that reorder instructions and/or memory accesses aggressively, such as PowerPC or Power4.

One thing pops up rather often in drivers ported from other operating systems: recursive locks. The linux kernel provides no recursive locks (with the exception of the Big Kernel Lock, see section 4). The philosophy is that locking should be taken into account when designing your code, and in that case you don't need recursive semaphores or spinlocks. Recursive locks have all sorts of subtle deadlock issues which are beyond the scope of this text and while you can write correct recursive locks, just say no.

4 SMP

Despite popular belief, SMP safety is not something you "weld" into your code as a hindsight. SMP safety is something you need to take into account right from the start. Makeing a (largish) piece of code SMP safe in hindsight leads to all kinds of lock-ordering nightmares, makes you wish there were recursive locks, and generally results in a suboptimal solution. While I could give numerous drivers as examples of how to not do it, the main kernel with the Big Kernel Lock (BKL) is the best example of this. The BKL was put in to make the kernel work on SMP, in hindsight—and well, it still results with nightmares with dozens and dozens of races. It has been taking 5 years so far to fix all the core subsystems to have proper locking of their own.

With the merging of the preemptible kernel patch in the 2.5 series of the kernel, SMP safety is even more important than before. With preemption turned on, your code can be interrupted at any point and acts as if it's running on an SMP machine.

There's a simple list of questions you should ask yourself for all code you write:

- What prevents the data I'm working on from beeing freed under me
- What prevents my module from being unloaded under me
- What prevents a user from opening/closing my device here
- What do I not want to happen to the data/device I'm working with right now...
- ...and what makes sure that that doesn't happen

Such a list can never be complete obviously, and the only weapon you can use to make your code SMP safe is your brain. Of course it helps if you have access to SMP hardware, and even booting an SMP kernel if you have only one CPU will allow you to find certain types of deadlocks.

5 All the world is not a VAX

```
#ifdef ALPHA
#define U32     unsigned int
#else
#define U32     unsigned long
#endif
```

```
from drivers/scsi/inia100.h,
kernel 2.4.9
```

The Linux kernel works on several architectures, not just Intel x86. The kernel API has evolved in a way that makes drivers basically automatically portable between architectures. That is, if the API is followed and no strange assumptions are made, as was done in the inia100.h example. Unfortunately, even if you don't follow the API the driver might at least *appear* to work on a normal PC, since the PC architecture makes certain consistency guarantees and has certain behavior that other platforms can't provide.

5.1 PCI posting

One of the most common, and hardest to debug, problems in PCI device drivers is the lack of dealing with *PCI posting*. *PCI posting* is the effect where the cpu writes some data to a certain PCI device, the PCI bridge (or, rather, any component between the CPU and the actual device) is allowed to buffer this write as long as it wants. The only constraint to this buffering is that a read operation from a device will not complete before all pending writes to this device are completed, hence effectively forcing a “flush” of all pending writes.

A typical example of how this can easily go wrong is shown below:

```
//—————————————————————-
// Procedure:    eeprom_stand_by
//
// Description: This routine lowers the
//     EEPROM chip select (EECS) for a
//     few microseconds.
//—————————————————————-
static void
eeprom_stand_by(struct e100_priv
*adapter)
{
   u16 x;

   x = readw(&CSR_EEPROM(adapter));
   x &= ~(EECS | EESK);
   writew(x, &CSR_EEPROM(adapter));
   udelay(EEPROM_STALL_TIME);
   x |= EECS;
   writew(x, &CSR_EEPROM(adapter));
   udelay(EEPROM_STALL_TIME);
}
```

`from the Intel e100 driver in kernel 2.5.6`

The intent of the programmer clearly is to clear some bit in the cards memory space, wait a certain amount of time, enable it again and wait some more time, effectively creating a periodic waveform if the routine is called in sequence. However, due to posting, the first writew() might not reach the card for a long time and the udelay() is therefore totally missing the intended goal, it just warms the cpu a bit. The end result in this case is that the eeprom contents is written out incorrectly to the card.

The fix is obvious (and present in later kernels); just adding a `readl()` immediately after both `writew()` to read and discard some data from the card is enough.

Only recently started the more advanced PC chipsets to do more aggressive posting, so this bug probably will not, or only seldom, show up on home PC’s. IA64 and other architectures have had more aggressive posting in the chipsets for a longer time already.

5.2 GFP_DMA

The GFP_DMA memory allocation flag was originally intended to allocate ISA bus DMA-able memory, however several architectures give a different meaning to it nowadays. Since the demise of the ISA bus, GFP_DMA should not be used in drivers *at all*; the PCI DMA API has well defined ways of allocating and mapping memory for PCI use. The example below shows the Intel e100 driver abuse GFP_DMA to work around a performance issue in early IA64 architecture code that deals with a missing IOMMU chip.

```
#if (defined __ia64__)
   new_skb =
      __dev_alloc_skb(skb_size,
      GFP_ATOMIC|GFP_DMA);
   // Try to alloc non-DMA skb if
   // failed to get from the DMA zone

   if (new_skb==NULL){
      new_skb =
        dev_alloc_skb(skb_size);
   }
#else
   new_skb =
        dev_alloc_skb(skb_size);
#endif
```

`Intel e100 1.8.38` While the code will work correctly for current IA64 systems, it's not guaranteed that newer IA64 machines won't have an IOMMU and that GFP_DMA doesn't actually result in PCI-reachable memory. The obviously right thing here was to fix the performance bug in the architecture code, or even making the architecture use the highmem mechanism.

5.3 Assuming PC resources

```
static void
qla2100_putc(int8_t c)
{
...
   /* BAUD rate divisor LSB. */
   OUTB(0x3f8+3, 0x83);
   /* BAUD rate divisor MSB. */
   /* 0xC = 9600 baud */
   OUTB(0x3f8, 0xc);
...
}
```

`Qlogic 2x00 v5.31 fiber channel driver` The above example is taken from a fibre channel driver. The purpose is to have some sort of serial console for tracing and debugging (let's pretend for a minute that Linux doesn't already have generic serial console code).

Hardcoding architecture constants like `0x3f8` (even if they haven't changed in the last 20 years) is just inviting trouble. Some day someone will want to run your code on an IA64, or a MIPS or even an ARM machine, and all hell breaks loose. In addition, these "constants" actually recently started changing with the advent of the legacy-free PC's.

Another serious issue is that this code uses IO resources without registering them with the kernel; one can assume that even the most expensive intelligent UPS gets confused when a driver like this interferes with the monitoring application which happens to be connected to this serial port (and yes, sysadmins do get grumpy when in the middle of the night a bad sector results the driver turning off the UPS due to error handling calling this debug code).

5.4 On-disk and wire formats

Two of the major differences between architectures that are actually visible to kernel drivers are byte order and structure padding.

Both these items you normally don't have to care about; however, they do become important when you put information on persistent storage (when writing a filesystem for example) or when sending information over a network.

The JFFS2 filesystem was sloppy in this respect and stored its metadata on the (flash) device in CPU byte order. After the filesystem was in use for several months, on several architectures, people started complaining that they couldn't take their device from one system (powerpc) to another (x86) while still being able to read the data. David Woodhouse can testify that making a filesytem auto-sensing, bi-endian is not something you want to do.

Using a defined byte order for the on disk format is certainly preferable. Verifying the correctness of code in this respect is normally nontrivial if you only have access to x86 machines; EXT3 uses the cunning trick to specify all on-disk data (in the journal) in Big Endian order so that any missing byte-order correction will be noticed immediatly on a PC.

Structure padding is a similar issue:

```
struct example {
    char foo;
    u64  bar;
};
```

in the example above the size of `struct example` will depend on the architecture. Different architectures have different requirements for minimal alignment of data and the compiler will add invisible padding between `foo` and `bar` to satisfy the architecture requirements.

This has 3 important consequences when this structure needs to be written to persistent storage or the network:

1. The size of the structure differs per architecture
2. The location of the `bar` data is in a different place in the bytes that make up `struct example`
3. The code below is *not* enough to clear the entire structure

```
struct example *blah;
...
blah = kmalloc(sizeof(*blah),
       GFP_KERNEL);
if (!blah)
    return -ENOMEM;
blah->foo = 3;
blah->bar = 40;
...
```

The code above does not clear the (invisible) padding, and for in-kernel use that's not a problem. However when exposing this struct outside the kernel, it is very important to realize that the padding bytes are uninitialized and hence can contain just about anything that ever was in memory, including the root password or parts of gpg keys. Things like that usually make sure that your module ends up on bugtraq just before the meeting with a big potential customer.

6 Using `int` for flags

A common bug that causes portability issues is the use of an `int` variable for storing the CPU flags in with `spin_lock_irqsave`.

```
void
mgsl_sppp_tx_timeout(
struct net_device *dev)
{
   struct mgsl_struct *info =
      dev->priv;
   int flags;
...
   spin_lock_irqsave(
        &info->irq_spinlock, flags);
   usc_stop_transmitter(info);
   spin_unlock_irqrestore(
        &info->irq_spinlock, flags);
}
```

`from drivers/char/synclink.c kernel 2.4.9`

On 64 bit architectures, the CPU flags are 64 bit, and the code such as quoted above will fail to work quite spectacularly. `spin_lock_irqsave` is specified to take an `unsigned long` variable for flags, and thankfully kernels 2.5.10 and later will cause a compiler warning if this isn't the case.

7 Files

Just about all kernel tutorials and books warn you to not open configuration files from inside the kernel. Unfortionatly that doesn't seem to stop people from doing so anyway.

```
static void chandev_read_conf(void)
{
#define CHANDEV_FILE \
        "/etc/chandev.conf"
...
```

```
    set_fs(KERNEL_DS);
    if(stat(CHANDEV_FILE,
       &statbuf)==0)
    {
       set_fs(KERNEL_DS);
       if((fd=open(CHANDEV_FILE,
              O_RDONLY,0))!=-1)
       {
          curr=0;
          left=statbuf.st_size;
          while((len=read(fd,
               &buff[curr],left))>0)
          {
             curr+=len;
             left-=len;
          }
....
}
```

`drivers/char/s390/misc/chandev.c in kernel 2.4.9/s390`

Apart from the aesthetic issues involving the kernel setting policy on userspace, and the problem that writing a secure parser is non-trivial (both of which doesn't seem to impress people enough to not write code like this), there is the issue of *which* file is it.

There are several reasons why this isn't as clear as it might look on first sigh. In recent 2.4 and 2.5 kernels, each process has its own namespace (and hence effectively its own root directory). Autoloading the driver will result in the module using `init`'s namespace, while manually loading it by root will result in the module using the namespace root currently happens to be in. `chroot` will have similar surprise effects, as will loading the module from an initial ramdisk.

The initial ramdisk case just makes a system harder to administer, but the other two cases actually carry a slight security risk: while one needs to be root to load modules, root might not realize he is not in the `init` namespace; this gives a non-root user the ability to feed a config file to the kernel (and either merely affect the settings, or worse, exploit vulnerabilities in the parser).

8 Just yuck

8.1 Overriding system calls

There are certain things that should just not be done in kernel space. The most evil one is overriding or adding system calls from modules. For historical reasons, the `sys_call_table` symbol is exported for the use in the Linux ABI patches that allow the kernel to run binaries from other linux-like operating systems (Unixware, Solaris etc). Linux ABI only needed this to call existing function calls, and has long since been fixed to do so directly.

Unfortionatly this export seems to have opened the door for other modules to override existing system calls with different behavior. The mix of modules that do this consists of several filesystems of IBM origin, binary only security tools, and oprofile, a performance monitoring tool.

```
void oplock_init(void)
{
  TRACE0(TRACE_SMB, 5,
    TRCID_OPLOCK_INIT,
    "Loading kernel"
    " oplock support\n");
  old_sys_fcntl =
     sys_call_table[__NR_fcntl];
  old_sys_fcntl64 =
     sys_call_table[__NR_fcntl64];
  sys_call_table[__NR_fcntl] =
     (long)&newsys_fcntl;
  sys_call_table[__NR_fcntl64] =
     (long)&newsys_fcntl64;
}
```

```
glue layer for IBM GFPS binary
only (C++) filesystem
```

Apart from the SMP races in such code (it's impossible to get the locking against module unloading working), overriding system calls is just too evil for words.

8.2 Depending on userspace program names

Writing kernel code that depends on filenames of programs in userspace prevents people from arranging their system the way they want, and also can lead to "interesting" surprises.

Sometimes filenames are unavoidable; the kernel has a series of filenames for `init` it tries when booting, while allowing a command line override. `modprobe` and `/sbin/hotplug` are also called by the kernel to provide improved plug-and-play behavior; however the filenames of both are /proc settings, and the bootup scripts are expected to set the proper values if the provided defaults are not correct.

A different kind of problem is if the kernel decides to behave differently based on the name of the program that is running:

```
Boolean
cxiIsSambaThread()
{
   return (!strcmp(current->comm,
      "smbd"));
}
```

```
glue layer for IBM GFPS binary
only (C++) filesystem
```

There is no guarantee that Samba is the only binary that is named *smbd*, and changing behavior based on such a test can lead to interesting surprises for other programs. There even can be a security angle if this test is used for enhancing permissions.

8.3 Long udelay

I have been told that drivers for Microsoft Windows can sleep in a lot more places than Linux drivers. Several drivers I've seen that are ported from Windows or are shared with Windows depended on this and the Linux variant of the driver gets into trouble as a result. The common solution between such drivers seems to be to just use the busy-waiting `udelay()` macro.

```
static uint8_t
qla2100_mailbox_command(....)
{
...
  cnt = 0x100000*2;   /* 22 secs */
  for( ; cnt > 0 && !ha->flags;
    cnt-- )
  {
    /* Check for pending interrupts. */
     data = RD_REG_WORD(
       &reg->istatus);
     ...
     udelay(10);
  }
}
```

```
from Qlogic qla2x00 driver
version 4.28
```

In the above driver code snippet the `qla2100_mailbox_command()` function, which has to be called with the `io_request_lock` spinlock acquired and local interrupts disabled, busy waits for about 20 seconds. This is thankfully not the common case, but since the spinlock in question is required for doing any IO at all, 22 seconds is usually enough to kill a system (especially if hardware watchdogs are involved). And even if it doesn't, writing a driver like this will not make you popular with the low latency people.

There is no clear-cut way to fix such problems in an afternoon. In the qla2x00 case, it required a complete redesign of all the locking and some surgery in the structure of the driver. As a side effect the lock hold times and contention reduced significantly (even ignoring the long busy waits since those are not really in fast paths); fixing these issues pays off in more than one way.

8.4 Floating point

Everybody knows using floating point in the kernel isn't allowed (with the exception of carefully selected places where MMX is used to speed up RAID XOR performance and such). However it's rather easy to put floating point in by accident:

```
#define NTSC 14.31818
#define calc_freq(n,m,k) \
   ((NTSC * (n+8))/((m+2)*(1<<k)))
...
int fi;
fi = calc_freq(n,m,k);
```

`drivers/video/trident_fb.{c,h}`
`2.4.18pre`

The first define is actually in the header file and all the use is in the C file, where the author probably no longer realized the NTSC constant was a float. Compiling the kernel with the `-msoft-float` flag finds such problems fast before random userspace floating point results get corrupted.

9 Conclusion

This paper only touches the tip of the iceberg of problems found in kernel modules; however I hope that it has become clear that adding abstraction layers for the Linux kernel API to glue drivers from other operating systems to Linux, is a bad idea. Beyond that it's mostly a matter of common sense and having a good design of a driver; due to the open source nature of the Linux kernel there's plenty of good (but also of bad) examples available for borrowing a design idea.

10 Legalese

The IBM GFPS glue layer code is governed by the following license:

all other code is copyrighted by the respec-

Metanet: Message-Passing Network Daemons

Erik Walthinsen

omega@temple-baptist.com, http://www.omegacs.net/˜omega/

Abstract

MetaNet is a message-passing architecture designed for the construction of network services daemons. Services are implemented as a series of mailboxes with attached functions to provide the functionality. The mailbox namespace is global across all services, allowing new daemons to hook into existing daemons to modify their behavior. A simple example is a stock DHCP daemon hooked by an external application that does an LDAP lookup for a machine's IP before falling back to the normal DHCP allocation scheme. This paper covers the architecture of the MetaNet system, and uses the example of a Captive Portal as used for public wireless network control to show how multiple services can be quite easily tied together to provide more complex services. Other possible projects using MetaNet will also be explored.

1 Introduction

Network services on Unix machines are provided today by separate daemons designed for each protocol and service. Each has its own internal structure, configuration files, behaviors, history, and so on. This works well in most environments because each service is designed to be completely standalone, as this is one of the driving principles behind Unix: "Do one thing, and do it well." For the most part, each of these daemons does indeed do its job reasonably well.

Unfortunately, there are situations where this myriad of separate daemons can make it difficult to accomplish the task at hand, or more likely they are simply too big to fit on the target device. The platform example used in this paper is that of an intelligent wireless access point (AP) running Linux. The M1 from Musenki [Musenki] is just such an access point, consisting of a Motorola MPC8241 embedded PPC processor, 32MB of RAM, 8MB of flash, a 10/100 NIC, and a MiniPCI slot for a wireless card. It runs Linux natively and must be able to provide all the normal network services, as well as be useful to the PersonalTelco Project [PTP] in its city-wide free networking projects. This drives some of the more interesting requirements, pushing well beyond the boundaries of current software.

MetaNet is an attempt at creating an infrastructure that enables these services to be not only small, but capable of extremely complex interactions. MetaNet itself is only an architectural design, not a specific implementation. The reference implementation is currently (as of this writing) written in Perl for convenience, but could be and will be implemented in other languages such as C and Python. More advanced features ensure that these languages can be mixed freely in larger systems, allowing high-speed services to be written in C and other services or glue code to be written in a scripted language.

2 Unusual Requirements

The PersonalTelco Project (PTP) has several (4 as of this writing, more on the way) open, public access points covering several very public locations in Portland, OR. The most visible of these is the node at Pioneer Courthouse Square, aka "Portland's Living Room", in the center of downtown. A wireless AP covers the Square and is connected to the Internet by redundant T-1's that are otherwise heavily underutilized by the company donating the space and bandwidth. Because the signal is unencrypted and usable by anyone with a laptop and a wireless card, some steps are necessary in order to keep usage under control, and limit or avoid liability. The solution used is called a "Captive Portal." The idea behind such a portal is that new clients on the wireless network are by default completely firewalled off from the Internet. In order for the user to gain access to the network they must log in after agreeing to a usage policy, etc. Custom client login software is unworkable because it would have to be distributed to clients and installed, which most savvy users wouldn't even consider. Instead, the portal uses software the user already has: a browser. Linux Netfilter [Netfilter] transparent redirects are used to shuttle all HTTP connections to a web server contained in the portal software itself. When the user successfully logs in, the firewall is modified to allow that client access to the Internet.

Once the client is associated and logged into the portal, there are several other aspects that have to be dealt with. The first is the fact that a spammer could trivially send mass quantities of email through such a wide-open connection, and could do so untraceably. Another is making sure that no single client abuses the upstream connection and effectively locks other clients off the network. Statistics logging and "Extrusion Detection" are also critical components that have to be built into such a system.

3 An Example

Rather than explaining how one might implement a complete Captive Portal with fully discreet daemons and why it would be nearly impossible, a simpler example will suffice to explain the basic MetaNet concepts.

On many large networks, an LDAP database is used to hold information about each machine and user. If this database includes the MAC address of a machine and the intended IP address, this information must be made available to the DHCP server.

With ISC dhcpd [ISC] as shipped with almost all Linux distributions, this would have to be accomplished by extracting the relevant pieces from the LDAP database and constructing a dhcpd.conf. This would have to be done every time the LDAP database changes, must be pushed to each DHCP server, and dhcpd has to be restarted. On a large network this could become quite a hassle to manage.

4 The MetaNet Approach

MetaNet is designed to split the implementation of network services into discreet components which communicate by passing messages. The degree to which the software is split into pieces determines the degree to which it can be integrated with other services.

The fundamental entity is a "message", which is simply a list of tag/type/value tuples sent to a specific "mailbox". All control and data transfer takes place via these messages. Objects that encompass various services such as sockets or web servers are simply a set of mailboxes with which other objects interact.

The mailbox names are strings that use a filesystem-style path syntax, allowing for an effectively unlimited namespace if used properly. Common messages include `"/system/socket/new"` to create a new socket, or `"/httpd/request"` when the web server gets a remote request. Most such mailbox names are indeed derived from the name of the object that created them, such as in the previous example where the object is simply named `"/httpd"`.

In order to capture messages sent to a mailbox, a "listener" is attached. This is a function pointer (or reference, in Perl), coupled with another list of tuples to allow the specific instance of that function pointer to be uniquely identified. (Mailboxes themselves in fact have tuples associated with them for the same reason.) Each mailbox maintains an ordered list of these listeners, so that when a message is sent to the mailbox, these listeners are called in order. As a special case, a listener can return an error code that indicates that no more listeners should be called.

There are two conceptually different kinds of mailboxes, with no actual technical difference between them. The difference lies in who is listening and who is sending the messages. In the case of a socket, the `"read"` message is sent by the socket code itself, and the nominal owner of the socket supplies a listener to catch the data received on the socket. The `"write"` request on the other hand works the other way around, with the owner sending the message and the socket's listener responsible for write()ing the data to the socket. In some cases both the object and external entities can provide listeners, for instance as a means of keeping track of what an object is doing or being told to do.

5 The Main Loop

At the core of any MetaNet application is the main loop, which is responsible for dealing with all the external events the application may listen for. This is basically a glorified select() loop capable of sending messages whenever a socket is available for reading or writing, as well as handling timeout routines. In some cases messages may actually be queued up for delivery directly from the main loop as well. The specific implementation details of the main loop depend entirely on the language and general program style used to construct the MetaNet library. The current Perl prototype does not yet use messages to indicate readable file descriptors or timeouts, though there is no technical reason this cannot be changed to a more consistent style.

6 DHCP and LDAP

In our example scenario, the DHCP server would be written in such a way as to expose many mailboxes through which its internal control passes. Upon creation, it would instantiate a socket to listen for requests. When a request packet arrives, the socket will send a `"read"` message for the server to pick up and translate from the packed DHCP format to a more readable set of tuples, which is then sent as a DHCP-specific message. The DHCP server then must maintain the whole of the DHCP state machine internally, likely using the machine's MAC address ("chaddr" in dhcp) as key.

The state machine will at some point need to find an IP address for the host. Normally this is done by sending a message to a mailbox that might be named `"lookup"` or similar, which another part of the stock DHCP server would be listening to. This code would perform the usual lookup of a free IP address in the pool, or find an address recently associated with that MAC address. It then sends this information back to the state machine by sending a `"lookup-response"` message, for instance. This re-engages the state machine and eventually results in the client being successfully configured.

In order to integrate this DHCP server with and LDAP database, only a few lines of code must be written. The first function is attached as a listener to the `"lookup"` mailbox, and is responsible for triggering the LDAP query. It simply sends a message to a previously instantiated LDAP client object requesting a specific lookup based on the MAC address of the client. It then returns an error code that indicates that it should be the last listener called for this particular message. Since the listener would have been prepended to the mailbox, this precludes the DHCP server's normal lookup routine from being called. The second function listens to the LDAP response mailbox and constructs the necessary message to be sent back into the DHCP state machine.

The only thing missing from this design is the ability for a failed LDAP lookup to fall back to a lookup from the pool. As a quick hack, this could be accomplished by triggering the `"lookup"` message a second time with a tuple in place to indicate to the LDAP glue code that it should not make another attempt to look in the database, but rather fall through and defer to the original DHCP server's listener.

7 Synchronous Operations

The MetaNet architecture as described is obviously highly asynchronous, since the only way to return data from a listener is by sending another message. In many cases, such as the above LDAP attempt, it is very advantageous to be able to execute certain operations synchronously, waiting for some kind of response. DNS lookups are an obvious candidate, since a large system may have to deal with both names and IP addresses may be doing DNS requests in many places. A completely asynchronous system requires that every function that does a DNS request be split into the part before and the part after the request. This could very quickly become a coding nightmare, especially when error cases are introduced.

The solution is to implement a method for blocking on the receipt of a given message. This is done by attaching a special listener to the mailbox, then sleeping. The listener is responsible for waking up the suspended code and providing the tuples from the message that woke it up.

There are several related mechanisms that can be used to accomplish this. POSIX threads can be used to create a new thread into which the main loop can be "moved" while the original thread waits on a condition variable. A listener is attached to the mailbox being waited on, which wakes up the original thread and ensures that one of the threads finishes before the other continues the main loop. In cases where threads are rarely created, the initial thread must be the one to continue, as killing it would terminate the application.

8 Inter-application Cooperation

If all the services of a given machine were provided by a single daemon with various objects to handle different protocols, a bug in any one service could bring down the entire daemon. This lack of isolation would be the death of any such system. To solve this, there must be a way of separating each service into a separate process, while retaining the ability for these daemons to interact with each other.

In order to do this, MetaNet uses IPC in the form of Unix-domain sockets, with one socket per daemon, residing in a central location. The socket is named after the process, avoiding the pain of having a separate TCP port for each service just for message passing. When a given process needs to send messages to another, it opens up a connection to that socket, which is maintained for the life of the processes.

In order to send a message or attach a listeners to a mailbox in another process, the name of the process is appended with a colon, such as `"dhcpd:/dhcpd/lookup"`. Sending a message to that mailbox will cause a packet to be sent across this on-disk socket to the appropriate application, where it will trigger the listeners in that process. Listeners can be added to this mailbox remotely as well, which is done by inserting a dummy listener into the mailboxes stack with the necessary code to route that message back to the originating process.

Using this method, a standalone, fully self-contained daemon such as dhcpd can have its behavior modified from the outside. As such, the dhcpd could be written in a language such as C, while the glue code to bridge to LDAP could be written in Python or Perl, assuming there is a compliant MetaNet implementation for that language.

Longer term, this IPC mechanism can be extended to support sending messages between different daemons on separate machines. This can be used for statistics gathering, remote configuration, or even to create prototypes of new networking protocols.

9 Implementing a Captive Portal

The first test application for the Perl implementation of MetaNet was a captive portal. The central piece of the portal is a web server, which is easily created with:

```
MetaNet::send("/system/http/server/new",
  name => "/captive/server", port => 5280);
```

A listener is attached to capture the requests made by remote clients:

```
MetaNet::append_new_listener(
  "/captive/server/request",
  \&client_request);
```

Some firewall tables rules are put in place to enable this to function:

```
# drop forwarded packets
# unless allowed
iptables -P FORWARD DROP
# masquerade all the clients
iptables -t nat -A POSTROUTING
  -o $pubdev
  -s $pubnet -j SNAT --to $extIP
# allow any packets that have a
# MARK set
iptables -A FORWARD -i $pubdev
  -o $extdev
  -m mark --mark 0x1 -j ACCEPT
# redir all other HTTP connections
# to myself
iptables -t nat -A PREROUTING
  -i $pubdev -p tcp --dport 80
  -m mark ! --mark 0x1
  -j REDIRECT --to-port 5280
```

This blocks all forwarded traffic that doesn't have a firewall mark set, except for HTTP connections which are forwarded to the portal daemon on port 5280. Next, an HTTP request handler has to be constructed to handle two cases: the client requesting a random page from the Internet, and a client that has been redirected to the login server:

```
sub client_request {
  my ($mbox, $listener, %tuples) = @_;

  if ($tuples{Host} =~
      /$tuples{sockhost}:$tuples{sockport}) {
    serve_page($tuples{clientname},
               $tuples{path});
  } else {
   my $url =
      "http://$tuples{Host}$tuples{path}";
   redirect($tuples{clientname},
   "http://$tuples{sockhost}:5280/?url=$url");
  }

  MetaNet::send("$tuples{clientname}/close");
  return 1;
}
```

The first line gathers the arguments that a listener function gets: the mailbox reference, the listener reference, and a hash containing all the tuples sent as part of the message. In this particular message, the tuples are as follows:

- **$tuples{clientname}** The base name for all mailboxes associated with this client connection

- **$tuples{Host}** The "Host" HTTP header, indicating the intended destination

- **$tuples{path}** The path of the requested file on the site
- **$tuples{sockhost}** The host address of this end of the socket: the portal's address
- **$tuples{sockport}** The port connected to on the portal: 5280

The essence of the code is the check to determine whether the client actually intended to connect to the captive portal itself or not. If it did, it serves a page to the client as needed in order to show the client a login webpage, acceptable usage policy, logos, etc. If it intended to go elsewhere, it is redirected to the index page of the portal:

```
sub redirect {
  my ($client,$newurl) = @_;

  print "redirecting browser to '$newurl'\n";
  $headers{code} = 307;
  $headers{'Location'} = $newurl;
  $headers{'Refresh'} = "1;URL=$newurl";
  $headers{'Content-Type'} = "text/html";
  $headers{data} =
     "<html><head><title>Moved!</title>";
  $headers{data} .=
    "<meta http-equiv=\"Refresh\"
    content=\"1;URL=$newurl\"></head>";
  $headers{data} .= "<body>This page has been
    <a href=\"$newurl\">moved.</a></body>
    </html>";

  MetaNet::send("$client/response", %headers);
}
```

The redirect function uses several distinct tricks to try to get the client to jump to the new page. Once the client has done so, gone through the login sequence, etc., it is time to allow the client to surf the web:

```
sub client_login {
  my ($client) = @_;

  system("iptables -t mangle
    -I PREROUTING 1
    -i $pubdev -s $client->{host}
    -j MARK --set-mark 0x1");
}
```

This function obviously assumes the presence of an object containing various information about the client, or at least its IP address. A simple hash of these client objects is stored globally for quick reference, keyed by the final octet of the client's address.

If that were all there was to the captive portal, eventually every IP address in the range would be wide open, and the purpose would be lost. To avoid this we have to implement a timeout mechanism to determine if the client is still actively using their connection, and if they've been idle for some period of time, log them out automatically. To do this we'll create a periodic timeout:

```
MetaNet::add_timeout(time + 5,
                         \&idle_timeout);
```

The idle_timeout function is rather too involved in `iptables` parsing to reproduce here, but the overall structure is along the lines of:

```
  sub idle_timeout
    foreach client
      determine bytes used count
      compare to previous count
      if current != previous
        update last-active time
      if (curtime - last-active) >idle_timeout
        log client out
```

The logout function is simply the same as the login function, with -D instead of -I on the iptables commandline.

There is a long list of features that can be added to this basic portal design. The page handler can support status pages displaying the state of each client. The traffic statistics can be logged to an RRDtool database for future graphing. A DHCP server could be integrated to give the portal a head-start on new clients. MAC-based filtering could be done to make sure logged in IP addresses don't get hijacked when someone leaves.

10 Transparent Proxying of POP and IMAP

The spam problem is a little tougher to solve while still allowing legitimate users to go about their normal business. Not having the SMTP port open for anyone restricts users to webmail or perhaps secure SMTP, and that can be enough of a deterrent to some people to make the whole experience a waste of time. The goal would then be to find a way to only open up SMTP packet forwarding when it is a reasonably certainty that spamming is not going to be done. While this can never be foolproof, the simple fact that the portal is a complete chokepoint for all traffic makes things much easier to regulate.

A common technique used by corporate sites with mobile users, when a VPN is not available, is to open up SMTP relaying on their server for a short duration immediately following a successful POP or IMAP authentication from that IP address. This technique relies on the assumption that remote users will check their mail before sending mail, or can be easily trained to do so if their mail client doesn't already do this.

The same trick can be used to determine whether the SMTP port should be opened for a given client. If it were possible to detect when the client has made a successful connection attempt to a POP or IMAP server, the port can be opened. The task of determining whether this has actually occurred, however, can be problematic. It requires that the daemon in charge is able to watch the POP or IMAP traffic generated by both the client and the server.

The method attempted involves constructing a transparent proxy for the POP and IMAP protocols. In HTTP/1.1, transparent proxying is made possible by the fact that the protocol requires the host to send the intended destination of the connection as part of the connection itself. The proxy acts like a web server up until the point this information is sent (which happily is immediately upon connection), and promptly makes a connection to the final destination. This is required because iptables REDIRECT does not in any way give the server that handles the redirect any indication of the original destination. Unfortunately, neither POP nor IMAP (or almost any other protocol for that matter) are similarly capable of being transparent proxied as currently defined. In order to accomplish this, we can take advantage of a feature of the netfilter/iptables called "ipq", which is an iptables target that sends packets up to userspace via a netlink socket. There happens to be a Perl module to interface with this socket, making interfacing quite trivial. We can start with the necessary firewall rules:

```
# masquerade all the clients
iptables -t nat -A POSTROUTING -o $pubdev
  -s $pubnet -j SNAT --to $extIP
# redirect all IMAP connections to our proxy
iptables -t nat -A PREROUTING -i $pubdev
  -p tcp --dport 143 -j REDIRECT
  --to-port 65143
# queue all related SYN packets to userspace
iptables -t mangle -A PREROUTING -s $pubnet
  -p tcp --dport 143 --syn -j QUEUE
# by default, block all SMTP traffic outbound
iptables -A FORWARD -i $pubdev -p tcp
  --dport 25
  -j REJECT
```

To start off the transparent proxy we must create a connection to the netlink socket so we can acquire the SYN packets:

```
my $queue =
  new IPTables::IPv4::IPQueue(
    copy_mode => IPQ_COPY_PACKET,
    copy_range => 64);
MetaNet::add_fd($queue->get_fd(),
  \&ipq_read,
  undef, undef, queue => $queue);
```

The file descriptor for the netlink socket (patch to IPQueue module required) is added to the main loop's list of file descriptors to listen on.

Next, we create a socket to listen for the redirected IMAP connections:

```
MetaNet::send("/system/socket/new",
  name => "/transproxy/socket");
MetaNet::append_new_listener(
  "/transproxy/socket/new_client,
  \&new_client);
MetaNet::send(
  "/transproxy/socket/bind",
  protocol => "tcp", port => 65143);
```

Once the socket is bound and listening, we enter the main loop and wait for something to happen:

```
MetaNet::main();
```

When a packet arrives on the netlink socket, the ipq_read function is called:

```
sub ipq_read {
  my ($fd, %tuples) = @_;
  my ($msg, $ip_header, $src_ip, $dest_ip);

  $msg = $tuples->{queue}->get_message(-1);
  $ip_header = NetPacket::IP->decode(
               $msg->payload());
  $src_ip = $ip_header->{src_ip};
  $dest_ip = $ip_header->{dest_ip};
  $TransProxy::syn_attempts{"$src_ip"} =
     $dest_ip;
  $queue->set_verdict($msg->packet_id(),
                      NF_ACCEPT);
}
```

The client's IP address and the original intended destination are associated in a global hash for future reference. The source port would also be stored, but current experiments show that the REDIRECT target seems to cause the source port to change before it gets to the new destination, making it unusable for the purpose. This means that multiple connections on the same port in very close proximity are likely to be confused with each other.

Almost immediately after the SYN packet is processed, a connection will be established with the daemon via the redirect, triggering new_client:

```
sub new_client {
  my ($mbox, $listener, %tuples) = @_;

  $peerhost = $tuples{peerhost};
  if (defined($TransProxy::syn_attempts{
   "$peerhost"})) {
    build_tunnel($tuples{name},
      $TransProxy::syn_attempts{"$peerhost"},
      143);
    undef $TransProxy::syn_attempts{
          "$peerhost"};
  }
}
```

If the client address of the socket is found in the table of previous attempted connections, a socket tunnel is created between the new connection and the originally intended host. This is done by creating a new socket, connecting it to the server, then attaching functions to the two sockets' "read" mailboxes that send a "write" to the peer socket.

In order to determine if a successful authentication has occurred, the function that handles incoming packets from the server checks each packet for the string "OK LOGIN". If the string is found, the firewall is modified to allow SMTP traffic for that client:

```
if ($data =~ /OK LOGIN/) {
  system("iptables -I FORWARD 1
    -i $pubnet
    -p tcp --dport 25
    -s $client->{peerhost}
    -d $client->{desthost}
    -j ACCEPT");
}
```

A timeout mechanism similar to that used in the captive portal can be used to close the port after a certain amount of inactivity. Even more useful would be interaction between the transparent proxy and the captive portal, automatically closing these holes when the client as a whole times out.

11 Performance

The primary goal of MetaNet has been flexibility from the very beginning. It is not intended as the basis for large highly-scalable systems serving hundreds of clients per second. The highly unstructured string-based design doesn't necessarily lend itself to a highly-performant implementation, though it is entirely possible that some caching and other language tricks could be employed to improve the speed. If a highly-scalable server is needed with the ability to send or receive messages, a small subset could be implemented on top of an existing architecture.

12 Conclusion

The MetaNet architecture provides the ability to construct lightweight services very quickly by building off existing code. More importantly, it allows separate services to be integrated with a minimum of code. Completely new services can be built by gluing together otherwise unrelated subsystems.
The M1 platform from Musenki is the current major target of this work, with the goal of replacing all the software on the machine with MetaNet-based daemons capable of being glued together in previously unknown ways. Such a system would consist of a kernel, init, a shell and basic utilities, and the MetaNet-based daemons. Python is a logical choice for this platform, as the interpreter is less than half a megabyte, and implicitly allows developers to script the machine onboard. Such a machine could then be widely deployed to create the fabled city-wide free wireless network.

13 Acknowledgements

The PersonalTelco group deserves a significant amount of credit for getting me to think about these problems, and then actually *do* something about them. Discussions with Professor Jim Binkley at Portland State University, as well as his class on routing protocols, have been quite helpful. The GStreamer crew also deserves some credit for being "patient" while I worked on this project.

References

[Musenki] *Musenki*
`http://www.musenki.com/`

[PTP] *PersonalTelco Project*
`http://www.personatelco.net/`

[Netfilter] *Linux Netfilter*
`http://netfilter.samba.org/`

[ISC] *Internet Software Consortium*
`http://www.isc.org/`

How to replicate the fire: HA for netfilter based firewalls

Harald Welte
Netfilter Core Team + Astaro AG
laforge@gnumonks.org || *laforge@astaro.com*
http://www.gnumonks.org/

Abstract

With traditional, stateless firewalling (such as ipfwadm, ipchains) there is no need for special HA support in the firewalling subsystem. As long as all packet filtering rules and routing table entries are configured in exactly the same way, one can use any available tool for IP-Address takeover to accomplish the goal of failing over from one node to the other.

With Linux 2.4.x netfilter/iptables, the Linux firewalling code moves beyond traditional packet filtering. Netfilter provides a modular connection tracking susbsystem which can be employed for stateful firewalling. The connection tracking subsystem gathers information about the state of all current network flows (connections). Packet filtering decisions and NAT information is associated with this state information.

In a high availability scenario, this connection tracking state needs to be replicated from the currently active firewall node to all standby slave firewall nodes. Only when all connection tracking state is replicated, the slave node will have all necessarry state information at the time a failover event occurs.

The netfilter/iptables does currently not have any functionality for replicating connection tracking state accross multiple nodes. However, the author of this presentation, Harald Welte, has started a project for connection tracking state replication with netfilter/iptables.

The presentation will cover the architectural design and implementation of the connection tracking failover sytem. With respect to the date of the conference, it is to be expected that the project is still a work-in-progress at that time.

1 Failover of stateless firewalls

There are no special precautions when installing a highly available stateless packet filter. Since there is no state kept, all information needed for filtering is the ruleset and the individual, seperate packets.

Building a set of highly available stateless packet filters can thus be achieved by using any traditional means of IP-address takeover, such as Hartbeat or VRRPd.

The only remaining issue is to make sure the firewalling ruleset is exactly the same on both machines. This should be ensured by the firewall administrator every time he updates the ruleset.

If this is not applicable, because a very dynamic ruleset is employed, one can build a very easy solution using iptables-supplied tools iptables-save and iptables-restore. The output of iptables-save can be piped over ssh to iptables-restore on a different host.

Limitations

- no state tracking
- not possible in combination with NAT
- no counter consistency of per-rule packet/byte counters

2 Failover of stateful firewalls

Modern firewalls implement state tracking (aka connection tracking) in order to keep some state about the currently active sessions. The amount of per-connection state kept at the firewall depends on the particular implementation.

As soon as **any** state is kept at the packet filter, this state information needs to be replicated to the slave/backup nodes within the failover setup.

In Linux 2.4.x, all relevant state is kept within the *connection tracking subsystem*. In order to understand how this state could possibly be replicated, we need to understand the architecture of this conntrack subsystem.

2.1 Architecture of the Linux Connection Tracking Subsystem

Connection tracking within Linux is implemented as a netfilter module, called ip_conntrack.o.

Before describing the connection tracking subsystem, we need to describe a couple of definitions and primitives used throughout the conntrack code.

A connection is represented within the conntrack subsystem using *struct ip_conntrack*, also called *connection tracking entry*.

Connection tracking is utilizing *conntrack tuples*, which are tuples consisting out of (srcip, srcport, dstip, dstport, l4prot). A connection is uniquely identified by two tuples: The tuple in the original direction (IP_CT_DIR_ORIGINAL) and the tuple for the reply direction (IP_CT_DIR_REPLY).

Connection tracking itself does not drop packets[1] or impose any policy. It just associates every packet with a connection tracking entry, which in turn has a particular state. All other kernel code can use this state information[2].

2.1.1 Integration of conntrack with netfilter

If the ip_conntrack.o module is registered
with netfilter, it attaches to the
NF_IP_PRE_ROUTING,
NF_IP_POST_ROUTING,
NF_IP_LOCAL_IN and
NF_IP_LOCAL_OUT hooks.

Because forwarded packets are the most common case on firewalls, I will only describe how connection tracking works for forwarded packets. The two relevant hooks for forwarded packets are NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING.

Every time a packet arrives at the NF_IP_PRE_ROUTING hook, connection tracking creates a conntrack tuple from the packet. It then compares this tuple to the original and reply tuples of all already-seen

[1]well, in some rare cases in combination with NAT it needs to drop. But don't tell anyone, this is secret.

[2]state information is internally represented via the *struct sk_buff.nfct* structure member of a packet.

connections[3] to find out if this just-arrived packet belongs to any existing connection. If there is no match, a new conntrack table entry (struct ip_conntrack) is created.

Let's assume the case where we have already existing connections but are starting from scratch.

The first packet comes in, we derive the tuple from the packet headers, look up the conntrack hash table, don't find any matching entry. As a result, we create a new struct ip_conntrack. This struct ip_conntrack is filled with all necessarry data, like the original and reply tuple of the connection. How do we know the reply tuple? By inverting the source and destination parts of the original tuple.[4] Please note that this new struct ip_conntrack is **not** yet placed into the conntrack hash table.

The packet is now passed on to other callback functions which have registered with a lower priority at NF_IP_PRE_ROUTING. It then continues traversal of the network stack as usual, including all respective netfilter hooks.

If the packet survives (i.e. is not dropped by the routing code, network stack, firewall ruleset, ...), it re-appears at NF_IP_POST_ROUTING. In this case, we can now safely assume that this packet will be sent off on the outgoing interface, and thus put the connection tracking entry which we created at NF_IP_PRE_ROUTING into the conntrack hash table. This process is called *confirming the conntrack.*

The connection tracking code itself is not monolithic, but consists out of a couple of seperate modules[5]. Besides the conntrack core, there are two important kind of modules: Protocol helpers and application helpers.

Protocol helpers implement the layer-4-protocol specific parts. They currently exist for TCP, UDP and ICMP (an experimental helper for GRE exists).

2.1.2 TCP connection tracking

As TCP is a connection oriented protocol, it is not very difficult to imagine how conntection tracking for this protocol could work. There are well-defined state transitions possible, and conntrack can decide which state transitions are valid within the TCP specification. In reality it's not all that easy, since we cannot assume that all packets that pass the packet filter actually arrive at the receiving end, ...

It is noteworthy that the standard connection tracking code does **not** do TCP sequence number and window tracking. A well-maintained patch to add this feature exists almost as long as connection tracking itself. It will be integrated with the 2.5.x kernel. The problem with window tracking is its bad interaction with connection pickup. The TCP conntrack code is able to pick up already existing connections, e.g. in case your firewall was rebooted. However, connection pickup is conflicting with TCP window tracking: The TCP window scaling option is only transferred at connection setup time, and we don't know about it in case of pickup ...

2.1.3 ICMP tracking

ICMP is not really a connection oriented protocol. So how is it possible to do connection tracking for ICMP?

[3]Of course this is not implemented as a linear search over all existing connections.

[4]So why do we need two tuples, if they can be derived from each other? Wait until we discuss NAT.

[5]They don't actually have to be seperate kernel modules; e.g. TCP, UDP and ICMP tracking modules are all part of the linux kernel module ip_conntrack.o

The ICMP protocol can be split in two groups of messages

- ICMP error messages, which sort-of belong to a different connection ICMP error messages are associated *RELATED* to a different connection. (ICMP_DEST_UNREACH, ICMP_SOURCE_QUENCH, ICMP_TIME_EXCEEDED, ICMP_PARAMETERPROB, ICMP_REDIRECT).
- ICMP queries, which have a request->reply character. So what the conntrack code does, is let the request have a state of *NEW*, and the reply *ESTABLISHED*. The reply closes the connection immediately. (ICMP_ECHO, ICMP_TIMESTAMP, ICMP_INFO_REQUEST, ICMP_ADDRESS)

2.1.4 UDP connection tracking

UDP is designed as a connectionless datagram protocol. But most common protocols using UDP as layer 4 protocol have bi-directional UDP communication. Imagine a DNS query, where the client sends an UDP frame to port 53 of the nameserver, and the nameserver sends back a DNS reply packet from its UDP port 53 to the client.

Netfilter trats this as a connection. The first packet (the DNS request) is assigned a state of *NEW*, because the packet is expected to create a new 'connection'. The dns servers' reply packet is marked as *ESTABLISHED*.

2.1.5 conntrack application helpers

More complex application protocols involving multiple connections need special support by a so-called "conntrack application helper module". Modules in the stock kernel come for FTP and IRC(DCC). Netfilter CVS currently contains patches for PPTP, H.323, Eggdrop botnet, tftp ald talk. We're still lacking a lot of protocols (e.g. SIP, SMB/CIFS) - but they are unlikely to appear until somebody really needs them and either develops them on his own or funds development.

2.1.6 Integration of connection tracking with iptables

As stated earlier, conntrack doesn't impose any policy on packets. It just determines the relation of a packet to already existing connections. To base packet filtering decision on this sate information, the iptables *state* match can be used. Every packet is within one of the following categories:

- **NEW**: packet would create a new connection, if it survives
- **ESTABLISHED**: packet is part of an already established connection (either direction)
- **RELATED**: packet is in some way related to an already established connection, e.g. ICMP errors or FTP data sessions
- **INVALID**: conntrack is unable to derive conntrack information from this packet. Please note that all multicast or broadcast packets fall in this category.

2.2 Poor man's conntrack failover

When thinking about failover of stateful firewalls, one usually thinks about replication of state. This presumes that the state is gathered at one firewalling node (the currently active node), and replicated to several other passive standby nodes. There is, howeve, a very different approach to replication: concurrent state tracking on all firewalling nodes.

The basic assumption of this approach is: In a setup where all firewalling nodes receive exactly the same traffic, all nodes will deduct the same state information.

The implementability of this approach is totally dependent on fulfillment of this assumption.

- *All packets need to be seen by all nodes*. This is not always true, but can be achieved by using shared media like traditional ethernet (no switches!!) and promiscuous mode on all ethernet interfaces.

- *All nodes need to be able to process all packets*. This cannot be universally guaranteed. Even if the hardware (CPU, RAM, Chipset, NIC's) and software (Linux kernel) are exactly the same, they might behave different, especially under high load. To avoid those effects, the hardware should be able to deal with way more traffic than seen during operation. Also, there should be no userspace processes (like proxes, etc.) running on the firewalling nodes at all. WARNING: Nobody guarantees this behaviour. However, the poor man is usually not interested in scientific proof but in usability in his particular practical setup.

However, even if those conditions are fulfilled, ther are remaining issues:

- *No resynchronization after reboot*. If a node is rebooted (because of a hardware fault, software bug, software update, ..) it will loose all state information until the event of the reboot. This means, the state information of this node after reboot will not contain any old state, gathered before the reboot. The effect depend on the traffic. Generally, it is only assured that state information about all connections initiated after the reboot will be present. If there are short-lived connections (like http), the state information on the just rebooted node will approximate the state information of an older node. Only after all sessions active at the time of reboot have terminated, state information is guaranteed to be resynchronized.

- *Only possible with shared medium*. The practical implication is that no switched ethernet (and thus no full duplex) can be used.

The major advantage of the poor man's approach is implementation simplicity. No state transfer mechanism needs to be developed. Only very little changes to the existing conntrack code would be needed in order to be able to do tracking based on packets received from promiscuous interfaces. The active node would have packet forwarding turned on, the passive nodes off.

I'm not proposing this as a real solution to the failover problem. It's hackish, buggy and likely to break very easily. But considering it can be implemented in very little programming time, it could be an option for very small installations with low reliability criteria.

2.3 Conntrack state replication

The preferred solution to the failover problem is, without any doubt, replication of the connection tracking state.

The proposed conntrack state replication soltution consists out of several parts:

- A connection tracking state replication protocol
- An event interface generating event messages as soon as state information changes on the active node
- An interface for explicit generation of connection tracking table entries on the standby slaves
- Some code (preferrably a kernel thread) running on the active node, receiving state updates by the event interface and generating conntrack state replication protocol messages
- Some code (preferrably a kernel thread) running on the slave node(s), receiving conntrack state replication protocol messages and updating the local conntrack table accordingly

Flow of events in chronological order:

- *on active node, inside the network RX softirq*
  - connection tracking code is analyzing a forwarded packet
  - connection tracking gathers some new state information
  - connection tracking updates local connection tracking database
  - connection tracking sends event message to event API
- *on active node, inside the conntrack-sync kernel thread*
  - conntrack sync daemon receives event through event API
  - conntrack sync daemon aggregates multiple event messages into a state replication protocol message, removing possible redundancy
  - conntrack sync daemon generates state replication protocol message
  - conntrack sync daemon sends state replication protocol message (private network between firewall nodes)
- *on slave node(s), inside network RX softirq*
  - connection tracking code ignores packets coming from the interface attached to the private conntrac sync network
  - state replication protocol messages is appended to socket receive queue of conntrack-sync kernel thread
- *on slave node(s), inside conntrack-sync kernel thread*
  - conntrack sync daemon receives state replication message
  - conntrack sync daemon creates/updates conntrack entry

2.3.1 Connection tracking state replication protocol

In order to be able to replicate the state between two or more firewalls, a state replication protocol is needed. This protocol is used over a private network segment shared by all nodes for state replication. It is designed to work over IP unicast and IP multicast transport. IP unicast will be used for direct point-to-point communication between one active firewall and one standby firewall. IP multicast will be used when the state needs to be replicated to more than one standby firewall.

The principle design criteria of this protocol are:

- **reliable against data loss**, as the underlying UDP layer does only provide checksumming against data corruption, but doesn't employ any means against data loss
- **lightweight**, since generating the state update messages is already a very expensive process for the sender, eating additional CPU, memory and IO bandwith.
- **easy to parse**, to minimize overhead at the receiver(s)

The protocol does not employ any security mechanism like encryption, authentication or reliability against spoofing attacks. It is assumed that the private conntrack sync network is a secure communications channel, not accessible to any malicious 3rd party.

To achieve the reliability against data loss, an easy sequence numbering scheme is used. All protocol messages are prefixed by a seuqence number, determined by the sender. If the slave detects packet loss by discontinuous sequence numbers, it can request the retransmission of the missing packets by stating the missing sequence number(s). Since there is no acknowledgement for sucessfully received packets, the sender has to keep a reasonably-sized backlog of recently-sent packets in order to be able to fulfill retransmission requests.

The different state replication protocol messages types are:

- **NF_CTSRP_NEW**: New conntrack entry has been created (and confirmed[6])
- **NF_CTSRP_UPDATE**: State information of existing conntrack entry has changed
- **NF_CTSRP_EXPIRE**: Existing conntrack entry has been expired

[6]See the above description of the conntrack code for what is meant by *confirming* a conntrack entry

To uniquely identify (and later reference) a conntrack entry, a *conntrack_id* is assigned to every conntrack entry transferred using a NF_CTSRP_NEW message. This conntrack_id must be saved at the receiver(s) together with the conntrack entry, since it is used by the sender for subsequent NF_CTSRP_UPDATE and NF_CTSRP_EXPIRE messages.

The protocol itself does not care about the source of this conntrack_id, but since the current netfilter connection tracking implementation does never change the addres of a conntrack entry, the memory address of the entry can be used, since it comes for free.

2.3.2 Connection tracking state syncronization sender

Maximum care needs to be taken for the implementation of the ctsyncd sender.

The normal workload of the active firewall node is likely to be already very high, so generating and sending the conntrack state replication messages needs to be highly efficient.

- **NF_CTSRP_NEW** will be generated at the NF_IP_POST_ROUTING hook, at the time ip_conntrack_confirm() is called. Delaying this message until conntrack confirmation happens saves us from replicating otherwise unneeded state information.
- **NF_CTSRP_UPDATE** need to be created automagically by the conntrack core. It is not possible to have any failover-specific code within conntrack protocol and/or application helpers. The easiest

way involving the least changes to the conntrack core code is to copy parts of the conntrack entry before calling any helper functions, and then use memcmp() to find out if the helper has changed any information.

- **NF_CTSRP_EXPIRE** can be added very easily to the existing conntrack destroy function.

2.3.3 Connection tracking state syncronization receiver

Impmentation of the receiver is very straightforward.

Apart from dealing with lost CTSRP packets, it just needs to call the respective conntrack add/modify/delete functions offered by the core.

2.3.4 Necessary changes within netfilter conntrack core

To be able to implement the described conntrack state replication mechanism, the following changes to the conntrack core are needed:

- Ability to exclude certain packets from being tracked. This is a long-wanted feature on the TODO list of the netfilter project and will be implemented by having a "prestate" table in combination with a "NOTRACK" target.
- Ability to register callback functions to be called every time a new conntrack entry is created or an existing entry modified.
- Export an API to add externally add, modify and remove conntrack entries. Since the needed ip_conntrack_lock is exported, implementation could even reside outside the conntrack core code.

Since the number of changes is very low, it is very likely that the modifications will go into the mainstream kernel without any big hassle.

Multiple Page Size Support in the Linux Kernel

Simon Winwood [‡] [§]
[§]School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia
sjw@cse.unsw.edu.au

Yefim Shuf [‡] [¶]
[¶]Computer Science Department
Princeton University
Princeton, NJ 08544, USA
yshuf@cs.princeton.edu

Hubertus Franke [‡]
[‡]IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598, USA
{swinwoo, yefim, frankeh}@us.ibm.com

Abstract

The Linux kernel currently supports a single user space page size, usually the minimum dictated by the architecture. This paper describes the ongoing modifications to the Linux kernel to allow applications to vary the size of pages used to map their address spaces and to reap the performance benefits associated with the use of large pages.

The results from our implementation of multiple page size support in the Linux kernel are very encouraging. Namely, we find that the performance improvement of applications written in various modern programming languages range from 10% to over 35%. The observed performance improvements are consistent with those reported by other researchers. Considering that memory latencies continue to grow and represent a barrier for achieving scalable performance on faster processors, we argue that multiple page size support is a necessary and important addition to the OS kernel and the Linux kernel in particular.

1 Introduction

To achieve high performance, many processors supporting virtual memory implement a Translation Lookaside Buffer (TLB) [8]. A TLB is a small hardware cache for maintaining virtual to physical translation information for recently referenced pages. During execution of any instruction, a translation from virtual to physical addresses needs to be performed at least once. Thereby, a TLB is effectively reducing the cost of obtaining translation information from page tables stored in memory.

Programs with good spatial and temporal locality of reference achieve high TLB hit rates which contribute to higher application performance. Because of long memory latencies, programs with poor locality can incur a noticeable performance hit due to low TLB utilization. Large working sets of many modern applications and commercial middleware [12, 13] make achieving high TLB hit rates a challenging and important task.

Adding more entries to a TLB to increase its coverage and increasing the associativity of a TLB to reach higher TLB hit rates is not always feasible as large and complex TLBs make it difficult to attain short processor cycle times. A short TLB latency is a critical requirement for many modern processors with fast physically tagged caches, in which translation information (i.e., a physical page associated with a TLB entry) needs to be available to perform cache tag checking [8]. Therefore, many processors achieve wider TLB coverage by supporting large pages. Traditionally, operating systems did not expose large pages to application software, limiting this support to the kernel. Growing working sets of applications make it appealing to support large pages for applications, as well as for the kernel itself.

A key challenge for this work was to provide efficient support for multiple page sizes with only minor changes to the kernel. This paper discusses ongoing research to support multiple page sizes in the context of the Linux operating system, and makes the following contributions:

- it describes the changes necessary to support multiple page sizes in the Linux kernel;
- it presents validation data demonstrating the accuracy of our implementation and its ability to meet our design goals; and
- it illustrates non-trivial performance benefits of large pages (reaching more than 35%) for Java applications and (reaching over 15%) for C and C++ applications from well-known benchmark suites.

We have an implementation of multiple page size support for the IA-32 architecture and are currently working on an implementation for the PowerPC[1] architecture.

[1]This is for the PPC405gp and PPC440 processors, both of which support multiple page sizes.

The rest of the paper is organized as follows. In Section 2, we present an overview of the Linux virtual memory subsystem. We describe the design and implementation of multiple page size support in the Linux kernel in Section 3 and Section 4 respectively. Experimental results obtained from the implementation are presented and analyzed in Section 5. Related work is discussed in Section 6. Finally, we summarize the results of our work and present some ideas for future work in Section 7.

2 The Virtual Memory Subsystem in Linux

In this section, we give a brief overview of the Linux Virtual Memory (VM) subsystem[2]. Unless otherwise noted, this section refers to the 2.4 series of kernels after version 2.4.18.

2.1 Address space data structures

Each address space is defined by a `mm_struct` data structure[3]. The `mm_struct` contains information about the address space, including a list of *Virtual Memory Areas* (VMAs), a pointer to the page directory, and various locks and resource counters.

A VMA contains information about a single region of the address space. This includes:

- the address range the VMA is responsible for;
- the access rights — read, write, and execute — for that region;

[2]This section is meant to be neither exhaustive or complete.

[3]Note that multiple tasks can share the same address space

- the file, if any, which backs the region; and
- any performance hints supplied by an application, such as memory access behaviour.

A VMA is also responsible for populating the region at page fault time via its `nopage` method. A VMA generally maps a virtual address range onto a region of a file, or zero filled (anonymous) memory.

A VMA exists for each segment in a process's executable (e.g., its text and data segments), its stack, any dynamically linked libraries, and any other files the process may have mapped into its address space. All VMAs, except for those created when a process is initially loaded, are created with the **`mmap`** system call[4]. The **`mmap`** system call essentially checks that the process is allowed the desired access to the requested file[5] and sets up the VMA.

The *page directory* contains mappings from virtual addresses to physical addresses. Linux uses a three level *hierarchical page table* (PT), although in most cases the middle level is optimised out. Each leaf node entry in the PT, called a *page table entry* (PTE), contains the address of the corresponding physical page, the current protection attributes for that page[6], and other page attributes such as whether the mapping is dirty, referenced, or valid.

Figure 1 shows the relationship between the Virtual Address Space, the `mm_struct`, the VMAs, the Physical Address Space, and the page directory.

[4]This is not strictly true: the **`shmat`** system call is also used to create VMAs. It is, however, essentially a wrapper for the **`mmap`** system call.

[5]This check is trivial if the mapping is anonymous.

[6]The pages protection attributes may change over the life of the mapping due to copy-on-write and reference counting

2.2 The `page` data structure and allocator

The `page` data structure represents a page of physical memory, and contains the following properties:

- its usage count, which denotes whether it is in the page cache, if it has buffers associated with it, and how many processes are using it;
- its associated mapping, which indicates how a file is mapped onto its data, and its offset;
- its wait queue, which contains processes waiting on the page; and
- its various flags, most importantly:

  locked This flag is used to lock a page. When a page is locked, I/O is pending on the page, or the page is being examined by the swap subsystem.

  error This flag is used to communicate to the VM subsystem that an error occurred during I/O to the page.

  referenced This flag is used by the swapping algorithm. It is set when a page is referenced, for example, when the page is accessed by the **`read`** system call, and when a PTE is found to be referenced during the page table scan performed by the swapper.

  uptodate This flag is used by the page cache to determine whether the page's contents are valid. It is set after the page is read in.

  dirty This flag is used to determine whether the page's contents has been modified. It is set when the page is written to, either by an explicit `write` system call, or through a store instruction.

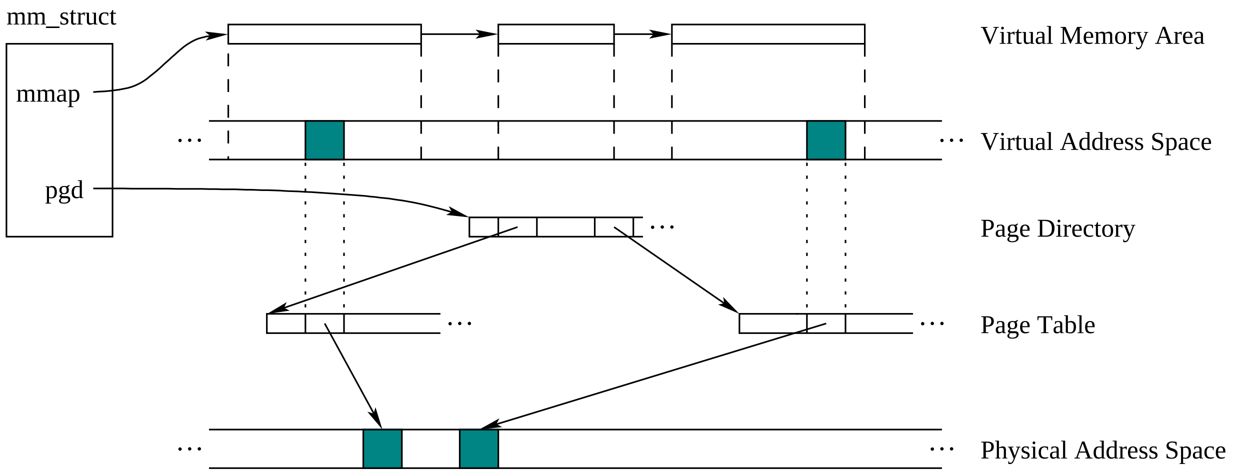


Figure 1: Virtual Address Space data structures

lru This flag is used to indicate that the page is in the LRU list.

active This flag is used to indicate that the page is in the active list.

launder This flag is used to determine whether the page is currently undergoing swap activity. It is set when the page is selected to be swapped out.

Pages are organised into *zones*; memory is requested in terms of the target zone. Each zone has certain properties: the *DMA* zone consists of pages whose physical address is below the 16MB limit required by some older devices, the *normal* zone contains pages that can be used for any purpose (aside from that fulfilled by the *DMA* zone), and the *highmem* zone contains all pages that do not fit into the kernel's virtual memory: when the kernel needs to access this memory, it needs to be mapped into a region of the kernels address space. Note that the *DMA* zone is only required for support of legacy devices, and the *highmem* zone is only required on machines with 32 bit (or smaller) address spaces.

Each zone uses a buddy allocator to allocate pages, so pages of different orders can be allocated. Although a client of the allocator requests pages from a specific list of zones and a specific page order, the pages that are returned can come from anywhere within the zone. This means that a page for the slab allocator[7] can be allocated between pages that are allocated for the page cache. The same can be said for other non-swappable pages such as task control blocks and page table nodes.

2.3 The page cache

The page cache implements a general cache for file data. Most filesystems use the page cache to avoid re-implementing the page cache's functionality. A filesystem takes advantage of the page cache by setting a file's `mmap` operation to `generic_file_mmap`. When the file is `mmap`ed, the VMA is set up such that its `nopage` function invokes `filemap_nopage`. The file's `read` and `write` operations will also go through the page cache.

[7]The *slab allocator*[5] provides efficient allocation of objects, such as `inodes`. Pages allocated to the slab allocator cannot be paged out.

The page cache uses the page's mapping and offset fields to uniquely identify the file data that the page contains — when an access occurs, the page cache uses this data to look up the page in a hash table.

2.4 The swap subsystem

Linux attempts to fully utilise memory. At any one time, the amount of available memory may be less than that required by an application. To satisfy a request for memory, the kernel may need to free a page that is currently being used. Selecting and freeing pages is the job of the swap subsystem.

The swap subsystem uses two lists to record the activity of pages: a list of pages which have not been accessed during in a certain time period, called the *inactive* list, and a list of pages which have been recently accessed, called the *active* list. The active list is maintained pseudo LRU, while the inactive list is used by the one-handed clock replacement algorithm currently implemented in the kernel. Whenever a page on the inactive list is referenced, it is moved to the active list.

The kernel uses a swapper thread to periodically balance the number of pages in the active and inactive lists: if a page in the active list has been referenced, it is moved to the end of the active list, otherwise it is moved to the end of the inactive list.

Periodically, the swapper thread sweeps through the inactive list looking for pages that can be freed. If the swapper thread is unable to free enough pages, it starts scanning page tables: for each PTE examined, the kernel checks to see whether the page has been referenced (i.e., whether the *referenced* bit is set in the PTE). If so, the page is moved to the active list, if it is not already a member. Otherwise, the page is considered a candidate for swapping. In this manner reference statistics are gathered from the page tables in the system, and used to select pages to be swapped out and freed.

The swapper thread may be woken up when the amount of memory becomes too low. The swapper functions may also be called directly when the amount of free memory becomes critical: when memory allocation fails, a task may attempt to swap out pages directly.

2.5 Anatomy of a page fault

When a virtual memory address is accessed, but a corresponding mapping is not in the TLB, a *TLB miss* occurs. When this happens, the *fault address* is looked up in the page table, by either the hardware in systems with a hardware loaded TLB, or via the kernel in systems with a software loaded TLB (note that this implies an interrupt occurs).

If a mapping exists in the page table, is valid, and matches permissions with the type of the access, the entry is inserted into the TLB, the page table is updated to reflect the access by setting the referenced bit[8], and the faulting instruction is restarted.

If a valid mapping does not exist, the kernel's page fault handler is invoked. The handler searches the current address space's VMA set for the VMA which corresponds to the fault address, and checks whether the access requested is allowed by the permissions specified in the VMA.

The kernel then looks up the PTE corresponding to the fault address and allocates a page table node if necessary. If the fault is a write to a PTE marked read-only, the address space requires a private copy of the page. A page is allocated, the old page is copied, and the dirty

[8]Note that architectures with a hardware loaded TLB whose page table doesn't map directly onto Linux's need to simulate this bit

bit is set in the PTE. If the PTE exists but isn't valid, the page needs to be swapped in, otherwise the page needs to be allocated and filled.

If the VMA does not define a `nopage` method, the memory is defined to be anonymous, i.e., zero-filled memory that is not associated with any device or file. In this case, the kernel allocates a page, zeroes it, and inserts the appropriate entry into the page table. If a valid `nopage` method exists, it is invoked and the resulting page is inserted into the PTE.

In the majority of filesystems, the `nopage` method goes to the page cache. The mapping and offset for the fault address are calculated — the information required for this calculation is stored in the VMA — and the page cache hash table is searched for the file data corresponding to the mapping and offset.

If an up-to-date page exists, then no further action is required. If the page exists but is not up-to-date, it is read in. Otherwise, a new page is allocated, inserted into the page cache, and read in. In all cases, the page's reference count is incremented, and the page is returned.

3 Design

This section discusses the approaches we considered and justifies our final design. This section is organised as follows: Section 3.1 discusses the goals that guided the design and the terminology used throughout this and future sections. Section 3.2 discusses the semantics of large pages: what aspects of the support for large pages the kernel exports to user space, the granularity at which page size decisions are made, and the high-level abstractions the kernel exports to the user.

It should be noted that this is an ongoing project, so the approaches describe here may have been improved upon by the time of publication.

3.1 Goals

This section discusses the design goals and guidelines which we attempt to adhere to in the design of our solution. We consider a good design to have the following properties:

Low overhead We do not wish to penalise applications that will not benefit from large pages, so we aim to minimise the performance impact of our modifications for these applications.

Generic The Linux kernel runs on numerous different architectures[9] and is usually ported quickly to new architectures. Any kernel enhancements such as ours should be easily adaptable to support existing and future systems, especially considering that many modern architectures feature MMUs which support multiple page sizes.

Flexible While a generic solution allows for easy portability, it does not indicate how well such a solution takes advantage of an architectures support for multiple page sizes. The design should be flexible enough to encompass any support.

Simple The more complex a solution is, the more likely it is to have subtle bugs, and the harder it is to understand. While we can foresee a point at which a more complex solution may be necessary, the initial design should be as simple as possible.

Minimal The Linux kernel is a large and complex system, so a minimalist approach is required: subsystem modifications that

[9]A count of the number of architectures in the mainline kernel reveals 15 implementations that are more or less complete

are not absolutely required may result in a solution that is overly complex and unwieldy. Therefore, we try to limit our changes to the VM subsystem only.

3.2 Semantics

This section discusses the semantics associated with supporting multiple page sizes: how the page size for a range of virtual addresses is chosen and whether the kernel considers this page size mandatory or advisory.

The following terms are used throughout this and later sections:

Base page A *base page* is the smallest page supported by the kernel, usually the minimum dictated by the hardware.

Superpage A *superpage* is a contiguous sequence of 2^n base pages.

Order A superpage's *order* refers to its size. A superpage of order n contains 2^n *base pages*.

Sub-superpage A *sub-superpage* is a superpage of order m, contained in a superpage of order n, such that $n \geq m$. Note that a base page is a sub-superpage with order $m = 0$

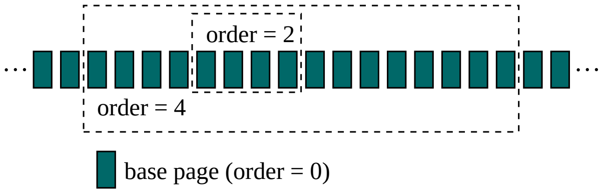


Figure 2: A superpage and sub-superpage

These concepts are illustrated in Figure 2 which shows a superpage of order 4 containing a sub-superpage of order 2.

3.2.1 Visibility

There are two basic approaches to supporting multiple page sizes: restrict knowledge of superpages to the kernel or export page size decisions to user space.

In the former approach, the kernel can create superpage mappings based on some heuristic, for example, a dynamic heuristic based on TLB miss information, or a static heuristic based on the type of mapping such as whether the mapping is for code, data, or whether it is anonymous. This approach is transparent to applications, and should result in all applications benefiting. It is, however, more complex, and would rely on effective heuristics to map a virtual address range with large pages.

In the latter approach, an application explicitly requests a section of its address space be mapped with superpages. This request could come in the form of programmer hints, or via instrumentation inserted by a compiler. While this approach requires applications to have specific knowledge of the operating system's support for large pages, it is much simpler from the kernels perspective. The major problem with this approach is that it requires the application programmer to have a good understanding of the applications memory behaviour.

We have decided on the latter approach, due to its simplicity: the former approach would necessitate developing heuristics that require fine-tuning and rewriting.

3.2.2 Granularity

This section discusses the granularity of control that the application has over page sizes. The approaches considered were:

per address space While making page sizes per address space would simplify some aspects of the implementation, it is too restrictive. We expect applications to have regions of their address space where the use of large pages would be a waste of memory;

per address space region type [10]
: This approach also has its drawbacks: there is no clear set of types, although the region's attributes (e.g., executable, anonymous) could be used, so again this approach is limited without any clear gains;

per address space region This approach is more flexible than either of the above approaches, however it does not allow for hotspot mapping within a region; or

over an arbitrary address space range.
: This is the most flexible approach, however, there are implementation issues: the kernel would need to keep track of the applications desired page sizes for the entire address space.

To allow maximum flexibility while minimising implementation overhead, we have decided upon a combination of the last two options: an application can dictate the page size for an arbitrary address range only if that range belongs to an address space region. This means that an application can map a region hotspot with large pages, but leave the rest of the region at the system's default page size.

3.2.3 Interface

This section discusses the guarantees given about the actual page size used to map an address space range.

[10]A region is a defined part of the address space that created by the **mmap** system call, for example.

The kernel can take a best-effort approach to mapping a virtual address with the applications indicated page size, falling back to a smaller page size if the larger page is not immediately available. Alternatively, the kernel can block the application until the desired page size becomes available, copying any existing pages to the newly allocates superpage.

Rather than mandating either behaviour, we have elected to allow the application to choose between the two alternatives. In situations where selecting a larger page size is merely an opportunistic optimisation for a relatively short running application, the first behaviour is desirable. In cases where the application is expected to execute for an extended period of time, however, the expected performance improvement may be greater than the expected wait time, and so waiting for a superpage to become available is justified. If an application is expected to reuse a large mapping over a number of invocations (a text page or a data file, for example), the application will benefit by waiting for the large page to be constructed.

4 Implementation

This section discusses the implementation of the design in Section 3.

4.1 Interface

An application requires some mechanism to communicate a desired page size to the kernel. A system call is the conventional mechanism for communicating with the kernel. In this section, we discuss our implementation of a system call interface for setting the page size for a region of the address space.

We considered three options: add a parameter to the **mmap** system call which specifying the page size for the new mapping; implement a new system call, **`setpagesize`**; and add another operation to the **`madvise`** system call.

Using the **`mmap`** system call would appear to be an obvious solution. It has, however, several negative aspects: firstly, the **`mmap`** system call is complex and is frequently used. Modifying **`mmap`**'s argument types would break existing code, as would adding extra parameters. Secondly, the application would be restricted to the one page size for that mapping, for the life of the mapping.

Using a new system call would be the cleanest alternative, however this requires significant modifications to all architectures, and is generally frowned upon where an alternative exists.

Using the **`madvise`** system call would allow an application to modify the page size at any point during its execution and would not affect existing applications, as any modification would be orthogonal to current operations.

We therefore added a *setregionorder*(n) operation to the **`madvise`** system call, where n is the new page order. We implemented this using the `advise` parameter of the **`madvise`** system call. The upper half of the parameter word contains the desired page order, while the lower half indicates that a *setregionorder* operation is to be performed.

Within the kernel, the **`madvise`** system call verifies that the requested page order is actually supported by the processor, and sets the VMA's `order` attribute accordingly.

4.2 Address space data structures

This section discusses the modifications made to the kernel's representation of a virtual address space. The application can modify the page size used by a VMA at runtime, either by an explicit **`madvise`** system call or by instructing the kernel to fall back to a smaller page size if a larger is not available. Consequently, the kernel needs to keep track of the following: firstly, the page size indicated by the application, which is associated with the VMA; secondly, the actual page size used to map a virtual address.

To communicate the requested page order to the VMA's `nopage` function, another parameter was added. This parameter indicates the desired page order at invocation, and contains the actual page size upon return. We rely upon the fact that subsystems which have not been modified will only return base pages.

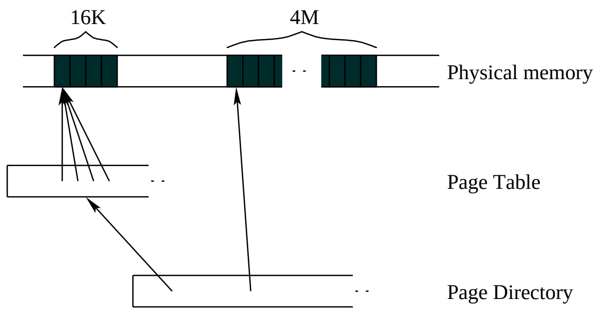


Figure 3: The modified page table structure

To store the superpage size that actually maps the virtual address range, the PTE includes the order of the mapping. To achieve this, we associated unused bits within the PTE with different page sizes, although the actual bits and sizes may be dictated by hardware.

The page table structure was also modified: superpages which span a virtual address range greater or equal to that of a non-leaf page directory entry are collapsed until they fit into a single page table node (see Figure 3). This means that we can now have valid page table elements at each level of the address translation hierarchy. This affects kernel routines which scan the page table, for example, the swap routine.

Although the main reason behind this was to conform to the page table structure defined by the x86 family, it also has other advantages: the kernel can use positional information to determine the page size, rather than relying solely on the information store in the PTE. This means that the number of page sizes supported by the kernel is not restricted by the number of unused bits in the PTE (which can be quite few). There may also be some performance advantage as the TLB refill handler does needs to traverse fewer page table levels.

4.3 Representing superpages in physical memory

This section discusses the representation of superpages in the `page` data structure. The kernel needs to keep track of various properties of the superpage, such as whether it is freeable, whether it needs to be written back, etc. The superpage can include sub-superpages which are in use: any superpage operation that affects the sub-superpage also affects the superpage, and this needs to be taken into consideration.

We considered the following representations of superpages: firstly, an explicit hierarchy of `page` data structures, with one level for each possible order. A superpage would then be operated on using the `page` data structure at the appropriate level. This implies that each operation would only have to look at a single instance of the `page` data structure.

This approach is the cleanest in terms of semantics. Unfortunately, the kernel makes certain assumptions about the one-to-one relationship between the `page` data structure and the actual physical page. Implementing this design would violate those assumptions and also involve significant modifications to the lower levels of the kernel.

The alternative involves a modification to the existing `page` data structure, such that each page contains the order of the superpage it belongs to. A superpage of order n would then be operated on by iterating over all 2^n base pages. This approach conforms to the kernels existing semantics. It is, however, subject to various race conditions, and is inelegant.

We implemented a combination of the two approaches presented: while we do not have an explicit hierarchy, there is an implicit hierarchy created by storing the superpage's order in each component base page. We logically partition the properties of a page into those associated with superpages, or with base pages.

This partitioning was guided by the usage of these properties: if the property was used in the VM subsystem only, it was usually put in the superpage partition. If the property was used for I/O, it was put into the base page partition. The properties were then partitioned as follows:

- the page's `usage count` is per superpage. As all allocation are done in terms of superpages, it follows that a superpage is only freeable if no sub-superpage is being used. This means that whenever a sub-superpage's usage count is modified, the actual modification is applied to the superpage;

- the `mapping` and `offset` properties are per base page, as they are only used to perform I/O on the page;

- the `wait queue` is per base page, as it is used to signal when I/O has completed;

- the `flags` are partitioned as follows:

  locked is per base page, as it is used primarily to indicate that a page is undergoing I/O;

  error is per base page, as it is used to indicate an I/O error in the page;

referenced is per superpage, as it is used by the VM subsystem only;

uptodate is per base page, as it is set when I/O successfully completes on a page;

dirty is per superpage, as it is primarily used in the VM subsystem;

lru is per superpage, as it indicates whether a page is in the LRU list, and the LRU list is now defined to contain superpages;

active is per superpage, as it indicates whether a page is in the active list, and the active list is now defined to contain superpages;

launder is per superpage, as it is only used in the swap subsystem, and the swap subsystem has to deal with superpages.

All other flags are per base page, as they reflect static properties of the page, (for example, whether the page is in the highmem zone).

Operations that iterate over each base page in a superpage are required to operate in ascending order to avoid deadlock or other inconsistencies.

4.4 Page allocation

The current page allocator supports multiple page sizes, however it has 2 major problems: firstly, non-swappable pages can be spread throughout each zone, causing memory fragmentation; secondly, if a large page is required, but a user (i.e. swappable) page is in the way, there is no efficient way to find all users of that page.

While the latter problem can be solved by Rik van Riel's reverse mapping patch[18], the former is still an issue. For this implementation, we have created another *largepage* zone, which is used exclusively for large pages. While this is not a permanent solution, it does aid in debugging, and solves the immediate problem for specialised users. The size of the *largepage* zone is fixed at boot time.

For maximum flexibility, the current allocator should be modified so that pages which are not pageable are allocated in so that they do not cause fragmentation. Also, pages which are allocated together will probably be freed together, so clustering pages at allocation time may also reduce fragmentation.

4.5 The Page Cache

To support mapping files with superpages, the page cache needs to be modified. The bulk of these modifications are in the `nopage` and affiliated functions, which attempt to allocate and read in a superpage of the requested order. To avoid any problems due to overlapping superpages, we require a superpage of order n also have file order n — that is, the alignment of the superpage in the virtual, physical, and file space is the same. For example, a 64K mapping of a file should be at a file offset that is a multiple of 64K, a virtual offset that is a multiple of 64K, and a physical offset of 64K[11].

The changes to the `nopage` function are essentially straightforward. If an application requests a superpage which contained in the page cache, it get back a sub-superpage whose order is the minimum of the requested order and the superpage's order. If a superpage does not exist, a page of the requested order is allocated, each base page is read in, and the superpage is added to the LRU and active queues.

Because reading in a large page can cause significant I/O activity (the amount of time re-

[11]The virtual and physical alignment constraints are common to most architectures.

quired to read in 4MB of data from a disk can be significant), we may need to read in base pages in a more intelligent fashion. One solution is to read in the sub-superpage which contains the address of interest first and schedule the remainder of the superpage to be read in after the first sub-superpage has completed. When the rest of the superpage has completed I/O, the address space can be mapped with the superpage. Note that this is similar to the early restart method used in some modern processors to fetch a cache line.

4.6 The swap subsystem

In our current implementation, a region mapped with superpages will not be swapped out. Swapping a superpage would negate any performance gained by its use due to the high cost of disk I/O. The superpage may need to be written back, however, and this is handled in an essentially iterative manner — when the superpage is not being used by any applications, and it is chosen by the swap subsystem to be swapped out (i.e. when it appears as a victim on the LRU list), each base page is flushed to disk, and the superpage is freed.

In the future, a number of approaches present themselves. The kernel may, for example, split up a superpage into smaller superpages over a series of swap events, until a threshold superpage order is met, and then swap that out. Alternatively, the kernel may just swap out the entire page.

4.7 Architecture specifics

This section discusses the architecture specific aspects of our implementation. Although our implementation attempts to be generic, the kernel requires knowledge of the architecture's support for multiple page sizes and the additional page table requirements.

The architecture specific layer in our implementation consists mainly of page table operations, i.e., creating and accessing a PTE. To constructed a PTE, the kernel now uses `mk_pte_order`, which is identical to `mk_pte`[12] except for an additional `order` parameter. This function creates a PTE with which maps a page of order `order`. To allow the kernel to inspect a PTE, a `pte_order` function is required. This function returns the order of a PTE.

On architectures which use an additional page table (usually because it is required by the hardware), the `update_mmu_cache` needs to be modified to take superpages into consideration. The kernel also requires a mechanism to verify that a page size is supported. This is achieved by implementing the `pgorder_supported` function.

4.8 Anatomy of a large page fault

In systems with a hardware loaded TLB, a TLB miss is transparent to the kernel, and so is not different in the case of a large page. In architectures with a software TLB refill handler, the new page table structure needs to be taken into consideration: the handler needs to check whether each level in the page table hierarchy is a valid PTE. The refill handler also needs to extract the page size from the entry and insert the correct $(VA, PA, size)$ entry into the TLB.

If there is no valid mapping in the page table, a page fault occurs. As with the standard kernel, the VMA is found and the access is validated. The PTE is then found, although a page table node is not created if it is required — the page table node is allocated later on in the page fault process. This postponement in allocating page table nodes is required as the kernel does not know what size the allocated page will be: this

[12]For backwards compatibility, `mk_pte` calls `mk_pte_order` with order 0

is determined when the page is allocated.

On a write access to a page marked read-only in the PTE, a private copy is created and replaces the read-only mapping. This involves copying the entire superpage, so it is a relatively expensive operation — as with all superpage operations, there will only be overhead if the operations would not have been done on each base page. For example, writing a single character to a 4Mb mapping will result in the whole 4Mb being copied, which would not have occurred if the region was mapped with 4K pages. Conversely, if most or all of the base pages are to be written to, copying them in one operation may reduce the total overhead due to caching effects and the reduced number of page faults.

If no mapping exists, the VMA's `order` field is consulted to determine the application's desired page size. If there are pages mapped into the region defined by this order and the fault address, and the application has elected to opportunistically allocate superpages, the kernel selects the largest supported order that contains the fault address, no mapped pages, and is less than or equal to the desired order. Otherwise, the application's desired page order is selected.

After the kernel has determined the correct page order, it examines the VMA's `nopage` method. If the `nopage` method is not defined, a zeroed superpage is allocated and inserted into the page table. Otherwise, the `nopage` method is called with the calculated page order, and the result is inserted into the page table.

If the file that backs the VMA is using the page cache to handle page faults, the kernel searches the page cache for the file data associated with the fault address. If a superpage is found, the minimum of the superpage's order and the requested order is used to determine the sub-superpage to be validated. The sub-superpage is then checked to ensure its contents are valid, and if so, it is returned. If the sub-superpage's contents is not valid, each base page is read in, and the sub-superpage is returned.

I-TLB 4K pages	128 entries, 4-way SA
I-TLB 4M pages	Fragmented into 4K I-TLB
I-L1 cache	12K micro-ops
D-TLB 4K pages	64 entries, FA
D-TLB 4M pages	Shared with 4K D-TLB
D-L1 cache	8K, 64 byte CL, 4-way SA
unified L2 cache	256K, 64-byte CLS, 8-way SA

Table 1: Pentium 4 processor's memory system characteristics (Notation: CL - cache lines; CLS - cache lines, sectored; SA - set associative; FA - fully associative).

5 Experimental Results

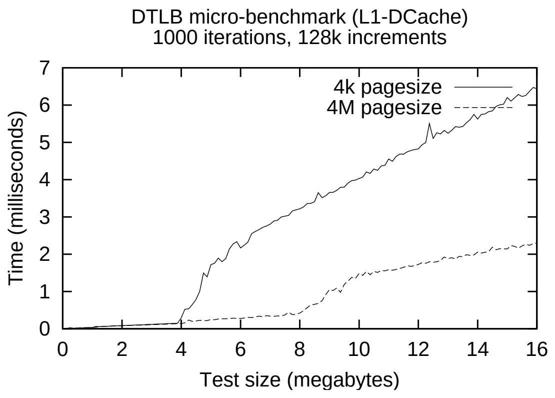
In this section, we present and analyze the experimental data from our implementation of multiple page size support in the Linux kernel.

All results in this section were generated on a 1.8GHz Pentium 4 system with 512M of RAM. The Pentium 4 processor has separate instruction and data TLBs and supports two different page sizes: 4K and 4M[13]. Table 1 shows the parameters of the memory system of Pentium 4.

5.1 Validating the Implementation with a Micro-benchmark

This section presents and discusses the data validating the accuracy of our implementation and demonstrating the benefits of multiple page size support for a simple microbenchmark. The use of a simple benchmark makes it possible to reason in detail about its memory

[13]Note that with large physical memory support (>4GB), the large page size on Pentium 4 processors is 2M.



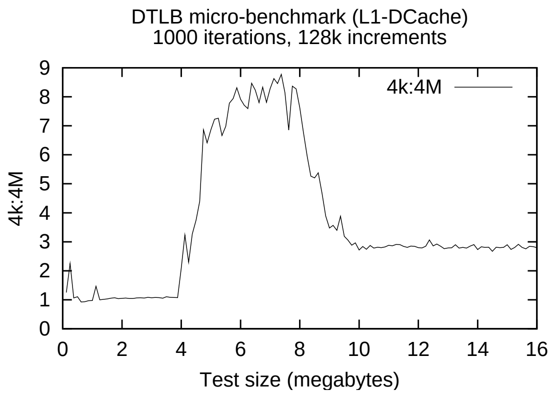


Figure 4: The execution times of the microbenchmark with small 4K pages and large 4M pages (left) and the ratios of execution times (right).

behavior and its interactions with the memory system.

The benchmark allocates a heap and initializes it with data. We vary the heap size from 128K to 32M in 128K increments in order to adjust the working set of the benchmark. The benchmark performs 1000 iterations during each of which it strides through the heap in the following manner: for each 4K page, it accesses one word of data. Assuming that caches and TLBs do not contain any information, each data access brings one cache line of PTEs and one cache line of data into the data L1 cache. To ensure that consecutive accesses do not compete for cache lines in the same cache set, we increment the offset at which we access data within a page by the size of a cache line. We also access every sixteenth page to ensure that we use only one PTE per L1 cache line[14].

We performed two sets of experiments. In the first set, the heap was mapped with 4K pages. In the second set, the heap was mapped with 4M pages. Both the 4K and the 4M cases have several inflection points. The first two inflection points for the 4K case are at 4M and 6M, and the first two inflection points for the 4M case are at 8M and 10M. The first inflection point indicates that the important working set (consisting of data and PTEs) can no longer fit in the fast L1 cache. Up to this point, the benchmark achieves full L1 cache reuse (both data and PTEs fit in the L1 cache)[15]. Between the first and the second inflection points, the benchmark achieves partial cache reuse (some of the data and PTEs remain in L1 across iterations). After the second inflection point, there is no L1 cache reuse (neither data nor PTEs remain in the L1 cache across iterations). The working set, however, still fits in the larger L2 cache. The performance of the 4K case degrades sooner than that of the 4M case due to the space overhead of PTEs[16]. The 4M case does not suffer from this behavior as it uses few PTEs and, hence, significantly less space in the L1 data cache; each cache line can accommodate 16 PTEs mapping a total of 64M of contiguous address space.

By extending the portion of the graph where

[14]On our Pentium 4 machine, one 64-byte cache line accommodates sixteen 4-byte PTE entries.

[15]Coincidentally, because we access one cache line of data per 4K page and access every sixteenth page, the 64-entry D-TLB begins thrashing at 4M, too.

[16]Namely, the PTEs occupy the same number of cache lines as the data. Consequently, the number of L1 misses begins to grow once the number of distinct pages we touch exceeds one half the number of cache lines in the L1 data cache.

the benchmark achieves full L1 cache reuse (i.e., past the first inflection point to the right), one can estimate the performance of the benchmark on a system with increasingly larger L1 cache. Similarly, by extending the portion of the graph where the benchmark experiences no L1 cache reuse, one can estimate the performance of the benchmark on a system with a slower L1 data cache (whose access time is equal to the access time of the L2 cache of our configuration). The next inflection point (not shown on the graph) will occur when the L2 cache starts to saturate.

5.2 Assessing Performance for Traditional Workloads

This section discuss the performance of multiple page size support in the context of the SPEC CPU2000 benchmark suite[16], specifically CINT2000, the integer component of SPEC CPU2000.

The CINT2000 benchmark suite was designed to measure the performance of a CPU and its memory subsystem. There are 12 integer benchmarks in the suite. These are the *gzip* data compression utility, *vpr* circuit placement and routing utility, *gcc* compiler, *mcf* minimum cost network flow solver, *crafty* chess program, *parser* natural language processor, *eon* ray tracer, *perlbmk*[17] perl utility, *gap* computational group theory, *vortex* object oriented database, *bzip2* data compression utility, and *twolf* place and route simulation benchmarks. All applications, except for *eon*, are written in C. The *eon* benchmark is written in C++.

We noted that the applications in the CINT2000 suite use the `malloc` family of functions to allocate the majority of their memory. To provide the application with memory backed by large pages via the `malloc` function, we modified the *sbrk* function. The memory allocator uses `sbrk` to allocate memory at page granularity; it then allocates portions of this memory to the application upon request. The `sbrk` function ensures that the pages it gives to memory allocator are valid; i.e., it grows the process's heap using the **`brk`** system call when required.

We modified the *sbrk* function so that it returns memory backed by large pages. At the first request, `sbrk` maps a large region of memory, and uses the **`madvise`** system call to map that region with large pages. Whenever the memory allocator requests a memory, `sbrk` returns the next free page in this region.

If the memory request is greater than some threshold (128K), the current memory allocator will allocate pages using the **`mmap`** system call. To ensure that the memory allocator returned memory backed by large pages, we disabled this feature so that the allocator always uses our `sbrk`.

To allow the applications to use our modified memory allocator and `sbrk` functions, we placed these functions in a shared library and used the dynamic linker's preload functionality. We set the `LD_PRELOAD` environment variable to out library, so the dynamic linker will resolve any `malloc` function calls in the application to our implementation. In this way, no recompilation is necessary for the applications to use large pages.

Table 2 shows the performance results we obtained using large pages. Overall, the results obtained are encouraging, many applications showing approximately 15% improvement in run time.

[17]Due to compilation difficulties, this benchmark was excluded from out results

Benchmark	Improvement (%)
164.gzip	12.31
175.vpr	16.72
176.gcc	9.29
181.mcf	9.43
186.crafty	15.22
197.parser	16.30
252.eon	12.07
254.gap	5.91
255.vortex	22.27
256.bzip2	14.37
300.twolf	12.47

Table 2: Performance improvements for SPEC CPU2000 integer benchmark suite using large pages

5.3 Assessing Performance with Emerging Workloads

This section discusses the impact of large pages on the performance of Java workloads. Java applications, and SPECjvm98 [15] applications in particular, are known to have to have poor cache and page locality of data references [11, 14]. To demonstrate the advantages of large pages for Java programs, we conducted a set of experiments with the *fast* configuration of Jikes Research Virtual Machine (Jikes RVM) [1, 2] configured with the mark-and-sweep memory manager (consisting of an allocator and a garbage collector) [3, 10].

To get the baseline numbers, i.e., where the heap is mapped with 4K pages, we ran the SPECjvm98 applications with the largest available data size on an unmodified Jikes RVM. The virtual address space in Jikes RVM consists of three regions: the *bootimage region*, the *small heap* (the heap region intended for small objects), and the *large heap* (for objects whose size exceeds 2K). We modified the *bootimage runner* of Jikes RVM[18] to ensure that the *small heap* is aligned to a 4M boundary and is mapped by 4M pages.

The decision to map only the *small heap* to large pages was based on the observation that, with a few exceptions, most objects created by SPECjvm98 are small. We then repeated the experiments by mapping all three heap regions to large pages. We also varied the size of the *small heap* from 16M to 128M and computed the performance improvements with 4M pages over a configuration that uses only 4K pages.

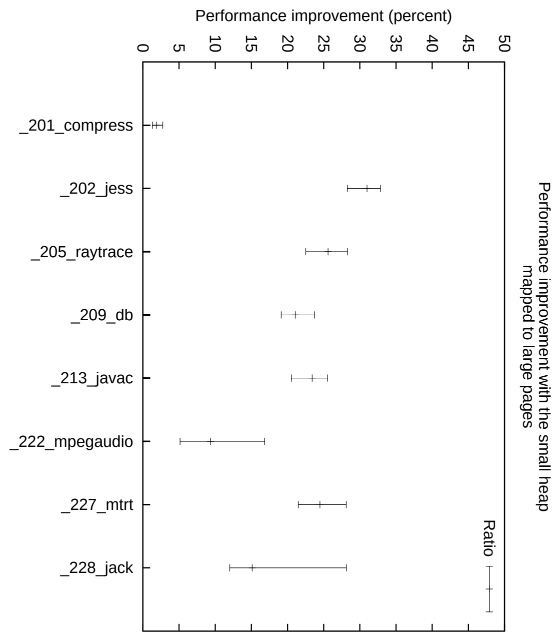
For each application, Figure 5 shows the minimum, the average, and the maximum performance improvements when the *small heap* is mapped to large pages (left) and when all three heap regions are mapped to large pages (right). It can be seen that for several applications the performance improvements are consistent and range from 15% to 30% even if only the *small heap* is mapped to large pages. The `compress` benchmark is the only one in the suite that creates a significant number of large objects and only a few small objects, and so does not benefit from large pages in this case.

When all three heap regions are mapped to large pages, we observe an additional 5% to 10% performance improvement. For many applications, the performance improvement ranges from 20% to 40% over the base case. It can also be seen that the `compress` benchmark enjoys a significant performance boost.

5.4 Discussion

The observed benefits of large page support can vary and depend on a number of factors such as the characteristics of applications and architecture. In this section, we discuss some of these factors.

[18]The *bootimage runner* is a program responsible for mapping memory for Jikes RVM and the heap, loading the core of the RVM into memory, and then passing control to the RVM.



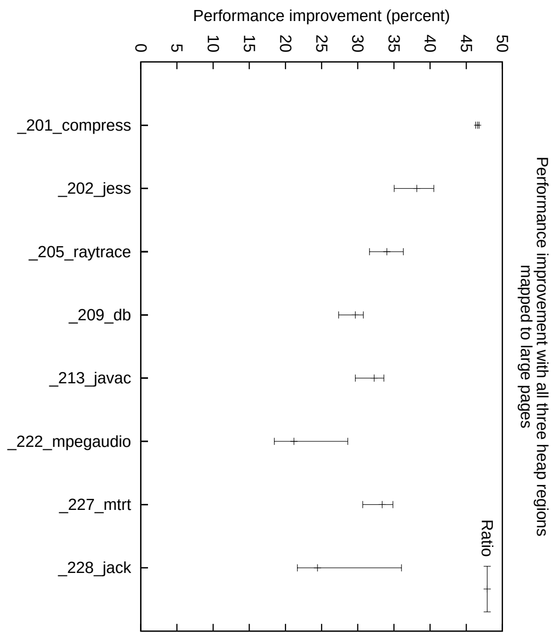


Figure 5: Summary of results for SPECjvm98.

A small number of TLB entries covering large pages[19] may not be sufficient for a realistic application to take full advantage of large page support. If the working set of an application is scattered over a wide range of address space, the application is likely to experiencing thrashing of a relatively small 4M page TLB, in some cases to a much larger extent than with the 64-entry 4K page data TLB. This is a problem on processors like Pentium II and Pentium III.

Applications executing on processors with software loaded TLBs are expected to benefit from large pages. The TLB miss overhead of an application executing on a processor that handles TLB misses in hardware (such as x86 processors) can also be significant unless most page tables of an application can fit in the L2 cache and co-reside with the rest of the working set. This is highly unlikely for applications of interest: assuming that each L2 cache line is 32 bytes and each PTE is 4 bytes, one L2 cache line can cover eight 4K pages (a total of 32k). Hence, a 512K L2 cache can accommodate PTEs to cover only 512M of address space (this does not leave any space for data in the L2 cache). Consequently, for applications with relatively large working sets, it is highly likely that a significant fraction of PTEs would not be found in the L2 cache on a TLB miss. Although hardware makes reloading a TLB from a L2 cache relatively inexpensive, many TLB misses may need to be satisfied directly from memory.

The Java platform [7] presents another set of challenges. For performance reasons, state-of-the-art JVMs compile Java bytecode into executable machine code [1, 2, 9]. In some virtual machines, such as Jikes RVM [1, 2], generated machine code is placed into the same heap as application data and is managed by the same memory manager. It has been observed that the code locality of Java programs tends to be better than their data locality [14]. This suggests that application code should reside in small

[19]In Pentium II and Pentium III microprocessors, there are eight 4M page data TLB entries some of which are used by the kernel.

pages while application data should reside in large pages. In Jikes RVM, generated code and data objects are indistinguishable from the memory manager's point of view and are intermixed in the heap. Because a memory region can only be mapped to either small or large pages, a tradeoff must be made. Mapping the entire heap region to large pages may not be effective since application code may not need to use large pages. What is worse is that some processors have a very small number of 4M page instruction TLB entries[20] which can lead to thrashing of an instruction TLB. Consequently, for best performance results, a JVM should be made aware of the constraints imposed by the underlying OS and hardware, and segregate application code and data into separate well-defined regions.

For Java programs, some performance gains are expected to come from better garbage collection (GC) performance. Much work during garbage collection is spent on chasing pointers in objects to find all reachable objects in the region of the heap that is being collected [4]. Many reachable objects can be scattered throughout the heap. As a result, the locality of GCs is often worse than that of applications [11]. This behavior is representative of systems employing non-moving GCs which have to be used when some objects cannot be relocated (e.g., when not all pointers can be identified reliably by a runtime). Consequently, large pages can improve TLB miss rates during GC (and overall GC performance). Applications that perform GC frequently, have a lot of live data at GC times, or whose live data are spread around the heap can benefit from large page support and achieve short GC pauses. Short pauses are critical for software systems that are expected to have relatively predictable response times.

[20]There are only two 4M page instruction TLB entries in Pentium II and Pentium III processors.

The availability of large pages can also be beneficial for programs that use data prefetching instructions. Modern processors squash a data prefetching request if the appropriate translation information is not available in the data TLB. Consequently, high TLB miss rates of applications can lead to many prefetching requests being squashed, thereby leading to ineffective utilization of memory bandwidth and reduced application performance[14]. The use of large pages can help reduce TLB misses and take full advantage of prefetching hardware. Further, a hardware performing automatic sequential data and code prefetching stops when a page boundary is crossed and has to be restarted at the beginning of the next page[21]. Large pages make it possible for such hardware to run for a longer period of time and to perform more useful work with fewer interruptions.

6 Related work

Ganapathy and Schimmel [6] discussed a design of general purpose operating system support for large pages. They implemented their design in the IRIX operating system for the SGI ORIGIN 2000 system that employs the MIPS R10000 processors (which handle TLB misses in software).

An important aspect of their approach is that it preserves the format of `pfdat` and PTE data structures of the IRIX OS. The `pfdat` structures represent pages of a base size and contain no page size information (just as in the original system). Large pages are simply treated as a collection of base pages. Consequently, only a few parts of the OS kernel need to be aware of large pages and need to be modified.

The PTEs contain the page size information but

[21]This is due to the fact that such automatic prefetching hardware uses physical addresses for prefetching.

the page table layout is unchanged. They use one PTE for each base page of a large page and create a set of PTEs that correspond to all addresses falling withing a large page. As expected, for the large page PTEs, the page frame numbers are contiguous.

To support multiple page sizes, the TLB miss handler needs to set a page mask register in the processor on each TLB miss. To ensure that programs that do not use large pages do no incur unnecessary runtime overhead, a TLB handler is configured per process. The allocation policy is specified on a command line (on a per segment basis) before starting an application. Hence, applications do not need to be modified to take advantage of large pages, and applications that do not use large pages are not put at disadvantage.

The advantage of this design is that it allows different processes to map the same large page with different page sizes. The disadvantages are (i) this approach does not reduce the size of page tables for applications that use large pages and (ii) the information stored in PTEs that cover a large page needs to be kept consistent.

They demonstrated that applications from SPEC95 and NAS parallel suite do benefit from large pages. For these applications, they registered 80% to 99% reduction in TLB misses and 10% to 20% performance improvement. A business application like the TPC-C benchmark (which is known to have poor locality and large working set) was also shown to benefit from large pages. The authors report 70% to 90% reduction in TLB misses and 6% to 9% performance improvement for this application.

Subramanian et al. [17] describe their implementation of multiple page size support in the HP-UX operating system for the HP-9000 Series 800 system which uses the PA-8000 microprocessor.

In their design the VM data structures such as the page table entry, virtual and physical page frame descriptors are based on the smallest page size supported by the processor. A large page is defined as a set of contiguous small base size pages. Hence, this design is conceptually similar to that of Ganapathy and Schimmel [6].

The authors note that an important advantage of this approach is that it does not require changes to many parts of the OS. However, it neither reduces the sizes of data structures for applications that use large pages. In addition, locking, access, and updates of data structures for large pages are somewhat inefficient. In spite of the benefits of space efficiency and the efficiency of updates, they choose not to use variable page size based data structures because, as the authors indicate, such an approach would lead to more changes in the OS and would have negative performance implications (e.g., a high page-fault latency in certain cases).

In their scheme, applications do not need to be recompiled to take advantage of large pages. The hints specifying large page sizes are region-based and are used at page fault time. In some cases, such as for performance reasons, the OS can ignore these page size hints and fall back to mapping small pages.

They implemented their design in the HP-UX operating system and studied the impact of large pages on several VM benchmarks, SPEC95 applications, and one commercial application. The reported performance improvements range from 15% to 55%.

7 Conclusions and Further Work

Many modern processors support pages of various sizes ranging from a few kilobytes to several megabytes. The Linux OS uses large pages internally for its kernel (to reduce the overhead of TLB misses) but does not expose large pages to applications. Growing memory latencies and large working sets of applications make it important to provide support for large pages to the user-level code as well.

In this paper, we discussed the design and implementation of multiple page size support in the Linux kernel. We validated our implementation on a simple microbenchmark. We also demonstrated that realistic applications can take advantage of large pages to achieve significant performance improvements.

This work opens up a number of interesting direction. In the future, we plan to modify kernel's memory allocator to further support large pages. We would also like to evaluate the impact of large pages on database and web workloads. These types of workloads are known to have large working sets and poor locality. Achieving high performance on commercial workloads is crucial for continuing success of Linux.

The latency of fetching a large 4M page from a disk (as a result of a page fault) can be significant. We consider implementing the "early restart" feature that would fetch and map the critical chunk of data first and complete fetching the remaining data chunks later, thereby reducing pauses experienced by applications.

Some architectures support a number of different page sizes (e.g., 16K, 256K, 4M, and 64M). We would be interested in evaluating the performance of applications on systems that have this architectural support.

Acknowledgements

We would like to thank Pratap Pattnaik and Manish Gupta for supporting this work. We also like to thank Chris Howson for all of his helpful advice.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), 2000.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño - a compiler-supported Java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 1999)*, Atlanta, GA, May 1999.

[3] C. Attansio, D. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Proc. of Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug. 2001.

[4] H.-J. Boehm. Reducing garbage collector cache misses. In *Proc. of ISMM 2000*, Oct. 2000.

[5] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Summer 1994 USENIX Conference*, pages 87–98, 1994.

[6] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proc. of the 1998 USENIX Technical Conference*, New Orleans, USA, June 1998.

[7] J. Gosling, B. Joy, and G. Steele. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley, 1996.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1995.

[9] The Java Hotspot Performance Engine Architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[10] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[11] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proc. of SIGMETRICS 2000*, June 2000.

[12] C. Navarro, A. Ramirez, J.-L. Larriba-Pey, and M. Valero. Fetch engines and databases. In *Proc. of Third Workshop On Computer Architecture Evaluation Using Commercial Workloads*, Toulouse, France, 2000.

[13] V. Oleson, K. Schwan, G. Eisenhaur, B. Plale, C. Pu, and D. Aminv. Operational information systems - an example from the airline industry. In *First USENIX Workshop on Industrial Experiences with Systems Software (WIESS)*, San Diego, California, October 2000.

[14] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proc. of SIGMETRICS 2001*, June 2001.

[15] Standard Performance Evaluation Council. *SPEC JVM98 Benchmarks*, 1998. `http://www.spec.org/osg/jvm98/`.

[16] Standard Performance Evaluation Council. *SPEC CPU2000 Benchmarks*, 2000. `http://www.spec.org/osg/cpu2000/`.

[17] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proc. of the 1998 USENIX Technical Conference*, New Orleans, USA, June 1998.

[18] R. van Riel. Rik van Riel's Linux kernel patches. `http://www.surriel.com/patches/`.

Embedding Linux

David Woodhouse
Red Hat, Inc.
dwmw2@cambridge.redhat.com

Abstract

Linux is becoming widely accepted in the embedded systems arena. This paper will give a brief overview of the applications for which it is currently being used and new applications for which development is in progress, and will discuss the requirements and problems which are unique to such embedded applications.

Some discussion will also be given to situations in which Linux is *not* the most appropriate tool for the task at hand, and in which a smaller, more application-specific operating system such as eCos may be more useful. *[eCos]*

1 Introduction

Linux was developed as a general-purpose operating system. A single kernel is intended to scale usefully from handheld devices such as the Compaq iPAQ and Sharp Zaurus to "Big Iron" such as the IBM zSeries mainframes.

For many years, Linux has commonly been used on PC machines as a router. More recently, Linux has been used by many companies in embedded "black box" products intended for applications such as network routing and firewalling, file and print serving, web serving, and in one well-known case for recording to hard disk and playback of television programmes. These applications typically use PC-class or similarly powerful hardware, and make no particular requirements on the kernel that traditional desktop and server applications do not.

Linux is also becoming widespread on smaller hardware such as Personal Digital Assistants (PDAs), especially the Compaq iPAQ and now the Sharp Zaurus, which is one of the first PDAs to be shipped with Linux *instead* of Windows CE, rather shipping with Windows CE but having Linux available for installation. These devices have limited battery capacity, very limited amounts of flash memory available for storage, and small displays with touch screens. Therefore, the use of Linux on such devices has motivated much development in the areas of power management, flash storage and code size reduction, and user interfaces targetted at such displays — all of which are discussed later.

2 Scaling down

A significant criterion affecting decisions regarding embedded applications is the cost per unit. Significant up-front costs and development time will be borne in order to reduce the per-unit cost of hardware and software by pennies. This is partly why Linux in the embedded space is so attractive to many developers of such products — the per-unit licensing cost of Linux and most Linux applications is zero.

The importance of per-unit cost means that hardware resources are often strictly limited.

The cost of extra RAM and storage space which may be required is equally important when comparing Linux against alternative operating systems; any wastefulness of resources cannot be tolerated in mass-production. Therefore, there is significant effort required to ensure that Linux remains "lean and mean," without gratuitous increases in the demands made from hardware.

Although Linus is fairly good at guarding against the introduction of gratuitous bloat, this is still not a particularly easy task. Figure 1 shows the size of the Linux source tarball since the first releases[1]. Since Linux v0.01 with a gzipped tarball size of 73091 bytes, the Linux kernel has grown exponentially over time to reach roughly 32 MiB at the time of writing, with the 2.5.12 kernel.

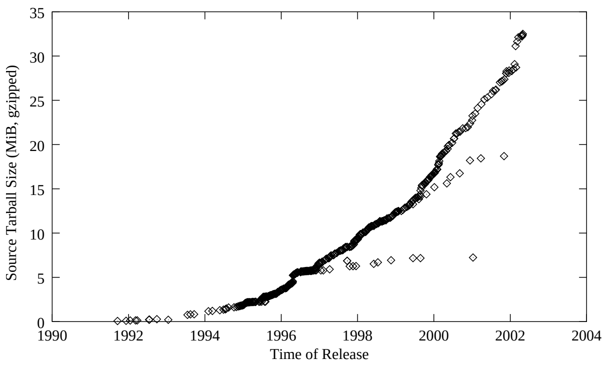


Figure 1: Growth of the Linux kernel.

Thankfully, this exponential increase in source code size does not translate linearly to an increase in object code size. Much of the increase in the source code size is optional features and new architectures and subsystems.

Nevertheless, the kernel *is* growing steadily. The transition of the μClinux code base from the 2.0 series of kernels to 2.4 was delayed significantly when it was realised that the resulting kernel images would *double* in size. This is largely due to expansion in code which has not been made conditional. Especially in the networking parts of the kernel, new features are often added unconditionally, without much consideration for situations where they will be unrequired.

[1]Graph thanks to James Smaby. `http://virgo.umeche.maine.edu/misc/kernel_size/`

There has been a great deal of work recently on "scalability" of Linux, with a lot of publicity and large companies getting involved. Mostly, attention has been concentrated on scaling *up* to large multi-processor and NUMA boxes; reducing lock contention and bouncing of shared cache lines, dealing efficiently with large amounts of memory and storage space, etc.

However, there has also been less visible work on scaling Linux *down* — reducing the storage and RAM footprint of both userspace and the kernel, and to a certain extent keeping the other developers honest by ensuring that optimisations made for larger boxes are not pessimisations for the embedded targets.

Inevitably, there have been some trade-offs required to accommodate the vast range of target hardware supported by Linux, but the most important such choices have been made configurable by the user at compilation time, so features can be included or omitted at will.

One area which is currently receiving attention is the support for block devices. Many embedded boxes have no block devices of any kind, and do not require any of the support for block I/O which is currently built into each kernel image. Significant size savings could be made by stripping this code out. However, the task is complicated; the block I/O subsystem has always been present in the Linux kernel and a large amount of code is written with the assumption that this will remain true. Large amounts of file system core (VFS) code need attention, but mostly to separate the block-related functions from generic file system code to allow conditional compilation. More complex is the virtual memory subsystem, which is littered with assumptions about the presence of block I/O, which is required for paging.

3 μClinux

μClinux, mentioned above, is a port of Linux to microcontrollers without memory management units. In embedded single-user and single-application boxes, a memory management unit serves little purpose. If the whole point of the box is to run a single application, it matters little whether a crash of that application can scribble over kernel memory and kill the kernel too, or whether the application will take a page fault and be killed — if the application dies, the game is over already. Hardware watchdogs to reset the board in the event of a malfunction are just as effective as a script to restart an individual application if it crashes; and of course neither are a suitable substitute for ensuring that the application doesn't crash in the first place.

The MMU, therefore, is a prime candidate for removal when counting the pennies which are being spent for each unit shipped.

Surprisingly, the Linux kernel code is not particularly intrusive. Most driver, networking and file system code does not need modification to accomodate the lack of MMU support. The memory management code in the `mm/` directory is replaced with equivalent routines in a parallel `mmnommu/` directory, and the architecture-specific code is also replaced with a new version.

Of course, most kernel code is designed to run in a single address space with no protection on memory accesses anyway. In userspace, the distinction is far more important. Because applications must *also* share a single address space, a drastic rearrangement of how user space programs are loaded was required. μClinux uses a new type of binary, known as 'flat' format. μClinux executables contain Position Independent Code (PIC) and hence the text segment containing the program instructions can be loaded anywhere inside the available address space, as can the data segment.

Although μClinux is not merged with the mainstream Linux kernel, the unintrusive nature of the port means it remains a possibility, and key kernel developers have repeatedly expressed a desire to do so.

μClinux supports a wide range of platforms and CPUs including Motorola m68k-based CPUs, ARM, Axis' ETRAX and the Intel i960. It is used in a number of products using μClinux including MP3 players, voice-over-IP telephones, Network Cameras, routers, etc.

4 Low-fat libraries and utilities

After the kernel, the next obvious large object in a Linux system is usually the C libraries. The GNU C library, glibc, is a multi-megabyte monster which can account for a large proportion of the available storage and RAM space on a small device. The maintainer of GNU libc, Ulrich Drepper, has clearly stated that GNU libc is not targetted at embedded systems: "*...glibc is not the right thing for [an embedded OS]. It is designed as a native library (as opposed to embedded). Many functions (e.g., printf) contain functionality which is not wanted in embedded systems.*" [Drepper]

Thankfully, there are alternatives to glibc. The older Linux libc5, the endpoint of the branch which was originally taken from GNU libc 1.07.4 to add Linux support, is still maintained and is significantly smaller than glibc. Also, there at least two C library implementations specifically designed for a small footprint.

As mentioned above, the μClinux kernel required drastic changes to userspace libraries. A new C library, μClibc, was developed for use with μClinux. After it became apparent that there was a need for a bloat-free C library in full MMU-capable Linux systems too, support for such systems was added to μClibc. μClibc supports most target architectures on which embedded Linux is found, including ARM, MIPS, PowerPC and Hitachi Super-H. μClibc is licensed under the GNU Lesser General Public License, and is available at `http://www.uclibc.org/`.

There is also diet libc, which claims to achieve even better code size reduction than μClibc by rewriting far more routines rather than copying them intact from other sources. The diet libc is licensed under the GNU General Public License, not the LGPL, and is available at `http://www.fefe.de/dietlibc/`.

Both μClibc and diet libc have multifunction binaries associated with them which can replace a large number of standard utilities such as `cat`, `mv`, `cp`, `ln`, etc. By using a single multifunction binary such as these to replace a multitude of overly feature-laden GNU utilities, further dramatic improvements in required space can be achieved.

BusyBox works with both μClibc and glibc, and is available at `http://www.busybox.net/`.

The set of utilities which works with diet libc is called "embutils" and is available at `http://www.fefe.de/embutils/`.

5 Power management

Another area which has received much attention in Linux, and still requires further development, is power management. Traditionally, Linux would power up and initialise devices at boot time or when the driver module was loaded, and would keep them powered at all times thereafter, often not even powering them down when a driver module was unloaded. This is extremely wasteful of power, which is extremely important on battery-powered computers such as laptops and handheld devices.

Obvious improvements are achieved by modifying drivers to remove power from unused circuits while devices are inactive. Often, this precise control over the application of power is very platform-specific, but hooks are required in generic code such as UART drivers so that the platform-specific code can be called at appropriate times when the port becomes active or inactive.

A great deal of work has therefore been done on extending the device driver APIs to accommodate power management facilities.

Suspend modes

Many battery-powered systems support a mode where all circuits except the RAM can be disabled and even the CPU can be placed into a low-power state until woken by an interrupt.

In order to enter this state and correctly return from it, it is necessary to maintain information about bus connectivity so that devices can be powered down before the busses which connect them, and the resumption of power can be performed in the opposite order.

Once all devices have been powered down, the entire CPU state can be stored in memory, the RAM can be switched into a self-refresh mode and even the CPU can be placed into an extremely low-powered state, to be woken only by a specially-configured interrupt. Often, only a single 32-bit register is retained over such a sleep state, and the CPU will start to execute the boot loader from the reset vector when it wakes just as it would after a normal power up cycle. The boot loader must then check the contents of the register and behave appropriately if it detects that it's waking from a sleep state, not a hard reset. Usually, the value in the register is a return address, and the boot loader will switch the RAM back to its normal state and jump back to the kernel code at the specified address.

This mechanism is used on PDA devices to implement the "instant-on" power mode which is reached by pressing the power button. A complete reset and reboot is rare, and usually requires pushing or switching a recessed reset or battery disconnect switch.

Frequency scaling

A CPU will consume less power when running at slower speeds, and many current CPUs can dynamically scale their clock speed under software control.

In addition to removing power to individual devices and circuits and shutting down the CPU completely, it is also possible to achieve power savings by utilising this facility to reduce the speed at which the CPU runs to match the current requirements of the running system.

Scaling CPU speed dynamically requires careful changes to timing-related functions and CPU-external bus timing. Basic support for management of CPU clock scaling is being developed and is present in the 2.4 version of the kernel for the ARM architecture. CPUs which are supported include ARM Integrator, SA1100 and SA1110. The various Intel IA32 clone manufacturers each have their own method of clock scaling, and CPUFreq contains support for AMD PowerNOW and VIA Cyrix Longhaul technologies.

Support for the Intel SpeedStep method of clock scaling is lagging far behind the rest, because Intel have so far refused to give sufficient documentation; preferring to push ACPI as their preferred method of accessing such functionality. Essentially, ACPI provides control methods in a form of interpreted bytecode similar in concept to Java, which must be trusted by the Linux kernel and run in privileged mode. This is no substitute for true GPL'd Linux drivers for the hardware in question.

Another power-saving feature which is not yet implemented but which is planned is the possibility of removing the system timer interrupt. Currently, Linux systems have a fixed-frequency interrupt, often at a frequency of 100 Hz, which is used for keeping system time and for running timers. If the CPU is entering a low-power state during idle periods, it must wake up and run the interrupt service routine every 10 ms — usually to find it has nothing to do but go immediately back to sleep again. This causes a significant power drain which should be unnecessary. Therefore, it is planned to develop code which allows Linux to abolish the fixed-frequency timer interrupt and instead use a one-shot timer to set a wake-up time each time the low-power idle state is entered. The CPU will be woken either by the first pending timed event or by interrupts from other sources such as I/O devices.

This improvement will be useful not only for embedded devices where power consumption is paramount, but also at the opposite end of the spectrum; on mainframe hardware where many hundreds of Linux kernels may run inside virtualised machines, and the overhead of a timer interrupt on *every* virtual Linux machine each few milliseconds quickly starts to take a significant proportion of the available CPU time.

6 Hotplug capabilities

Linux has for a long time supported PCMCIA and CardBus peripherals; 16-bit PCMCIA being significantly more common on handheld devices than CardBus. The Linux PCMCIA code is based heavily on the architecture laid out in the PCMCIA specification, which seems to be overly complicated and designed for legacy drivers and MS-DOS. This level of complexity appears to be overkill for Linux, and work has started on a rewrite of the PCMCIA support based solely on the reality of PCMCIA hardware rather than the intricacies of the PCMCIA specification. This work has yet to reach a state in which it can be announced to the public for further development.

In networking and control applications, Linux is also often required to support hot-swapping of PCI and CompactPCI peripherals. Basic support for dynamic addition and removal of PCI devices is present in the Linux kernel — each device driver must be individually upgraded to the new PCI driver API to be capable of supporting hot-swappable devices, and this has not yet happened for all drivers.

Support for physical insertion and removal of devices, probing of new devices and notification of drivers is implemented. Recently, some support for correct handling of the CompactPCI procedures for device insertion and removal, involving notification of the opening of the removal handle, lighting of the appropriate LED to signal that the system is ready for device removal etc.

One severe problem currently faced by the existing PCI hotswap code is the assignment of address space resources to newly-inserted cards. There is a limited amount of physical address space which may be assigned to BARs of PCI devices, and this space is further subdivided by having to configure ranges for each PCI bridge in the system, with each bus getting a single range of each type of address space. The current approach is to reserve some address space for each PCI bus which may accept hot-swap cards, in the hope that it will be enough. Yet if multiple PCI busses are present, then by repeatedly inserting and removing cards on different busses it is possible to fragment the allocation of resources to the extent that a newly-inserted card cannot be assigned a range of address space on the bus into which it has been introduced. Therefore, it is being proposed that another addition to the Linux PCI driver API be considered, which allows supporting drivers to have the BARs of their devices moved by the core PCI code to make room for other devices in the address space.

In many cases, it should be sufficient for the driver to momentarily quiesce the card, to prevent interrupts from occurring during the relocation, change the BARs to the new address range given by the PCI code, and reenable the device. Virtual mappings of memory BARs will need to either torn down and set up again for the new location, hence the need to quiesce the hardware rather than simply disabling interrupts while performing the move.

If accepted and implemented, this enhancement will allow for more reliable management of resources, assuming that the drivers for all hot-swapped cards provide support for this method of relocation.

7 Storage

The storage requirements of embedded devices differ significantly from traditional Linux installations. Often, the only storage available will be flash memory. Flash is a form of solid-state storage which provides persistant storage with low power requirements and relatively low cost.

The most common form of flash is NOR flash. This can be connected directly to the CPU's address and data busses and for reading is treated as ROM. As with ROM, each bit of storage can be in one of two states — either it contains a zero or a one.

Each bit of storage in NOR flash chip will start containing a one, and by a predefined sequence of writes of magic numbers to magic addresses, the contents of each bit can be individually changed to zeroes.

However, bits which have been cleared cannot be individually reset to contain ones again. Bits can only be reset to ones, or "erased," in large blocks of typically 64 or 128 KiB in size, known as "erase blocks." Furthermore, the lifetime of a flash chip is measured in erase cycles; typically each block can be erased 100,000 times before it is expected to fail.

It is important to note that the lifetime of flash chips is measured in erase cycles *per block*, not total erase cycles. Individual erase blocks can be erased to the point of destruction without affecting other erase blocks in the chip.

Therefore, by repeatedly erasing a few blocks it is possible to destroy them while the remainder of the chip is still usable — however, even with appropriate detection of bad blocks this reduces the storage capacity of the device, and as the storage available is unlikely to have exceeded the *required* amount by any significant margin, would quickly lead to the device being unusable.

Therefore, it is necessary to perform "wear levelling" on flash devices, to ensure that the block erases are evenly distributed over the entire range of the chips rather than concentrated in particular areas. This is particularly important because the normal use of permanent storage will be precisely the opposite of what is required — typically a device with 16 MiB of available flash would have 14 MiB of static data, programs and libraries, 1 MiB of dynamic data and 1 MiB of space; without wear levelling the 14 MiB of static data would never be moved and the remaining 2 MiB of the chip would be destroyed very quickly.

In addition to the need for wear levelling, the large block size of flash means that traditional file systems cannot easily be used, as they rely on being able to replace data blocks in-place, which is not possible on flash without also erasing and replacing the surrounding data in the rest of the same block. There is an extremely naïve driver available for Linux which does present a flash device to file systems as a block device with 512-byte sectors, then on writes will read the whole erase block, modify the contents as desired and then write back the new version. This is obviously extremely unsafe, but can be useful for setting up file systems which are going to be read-only in production.

The traditional approach to using flash has been to use a form of pseudo-filesystem on the raw flash to emulate a normal block device with smaller sectors. This solution evolved in the days of DOS, where providing an INT 13h disk service interrupt was sufficient.

In practice, this is very suboptimal. To ensure reliability, the pseudo-filesystem used must be a journalling one - it must be able to revert to a consistent state if power is lost or a crash occurs during a write. Furthermore, the traditional file system used on the emulated block device must *also* be a journalling file system, for precisely the same reasons. The result is a journalling file system running atop another journalling file system, which is inefficient in terms of both speed and wear on the flash devices.

A better approach is that taken by the Journalling Flash File Systems, which are designed to operate directly on the underlying flash device rather than through an intermediate emulation layer. *[JFFS]*

These file systems are log-structured, writing packets of data to the flash describing each *change* to the file system, and requiring a complete playback of those logs on remounting of the file system to recreate the current contents of the file system. As the log progresses, older log entries (or "nodes") will be obsoleted by newer entries which overwrite the old data, delete files, etc.

When the medium becomes close to full, the system must perform garbage collection to reclaim the space taken by such obsoleted nodes. An erase block is selected for garbage collection and the nodes which are still relevant are copied into the remaining empty space, before the victim block is erased. More details of the operation of these file systems are given in the referenced paper.

Since its development in the first quarter of 2001, JFFS2 has rapidly become extremely common in the deployment of embedded Linux devices with flash storage.

In addition to the common NOR flash, support has recently been added to JFFS2 for NAND flash. NAND differs from NOR flash in that it is not directly accessible as if it were ROM; instead data, addresses and commands are exchanged a byte at a time over a single 8-bit bus. NAND flash is smaller erase block sizes than NOR, typically around 8 KiB, and is further subdivided into "pages" of typ. 512 bytes, each of which is associated with a further range of "out-of-band" data, used for ECC and metadata. NAND flash chips are cheaper than NOR flash, and tend to have higher production tolerances, leading to higher incidence of bit errors and bad erase blocks.

Execute in place

A feature which is not implemented in Linux is "execute in place" (XIP). This refers to the arrangement where data are not copied from the flash medium into RAM, but are used directly by entering pages of the flash chip directly into the page tables of user space processes.

In many situations, XIP is not desirable. For obvious reasons, XIP and compression are mutually exclusive — if data are compressed, they cannot simply be used in-place. In terms of cost per byte, flash is generally more expensive than DRAM, hence the cost savings from using compression and reducing flash requirements are more than the cost savings from using XIP and reducing RAM.

However, XIP becomes a more sensible option in situations where low power consumption is an extremely important criterion. In this situation, static RAM may be used instead of DRAM, and this is normally more expensive than flash.

The implementation of XIP presents some interesting problems which have yet to be properly solved. Obviously, it can only work with NOR flash technology, as NAND flash cannot be directly accessed. The problem with NOR flash is that the write and erase commands are performed by writing magic numbers to magic addresses within the chip and then reading status words back from the chip. When the flash chip is in a command mode, the values returned on read accesses are not necessarily valid data. The flash drivers handle this by keeping a state machine and ensuring that the chip is always in the correct mode by sending the appropriate commands before performing any operation, including any reads from the device.

However, if pages of the flash chip are simply mapped into user space processes using the MMU, it is not possible to ensure that the proper sequences are followed; either scheduling must be disabled during the entirety of each period for which the flash chip is placed in a mode other than read mode, or every mapping of a flash page to user space must be found and torn down before each such access. As erase and write operations may take an extremely long time, the former option is not particularly feasible. Until recently, the latter was not possible either — only with the advent of the memory managment code based on reverse mappings of physical addresses to virtual was it possible to find all the mappings of a given page without scanning the entire address space of every process in the system.

Now that the rmap VM has made XIP at least technically feasible, there are plans to adapt the flash drivers to accommodate this form of mapping. However, there remains the problem of designing a file system which can make use of this facility. In order for XIP to be used, each page of file data must be page-aligned in the flash chip, because no common MMU hardware allows for the remapping of arbitrary byte ranges. This effectively means that the JFFS2 mode of operation, writing a node header followed immediately by a payload, is extremely suboptimal. An ideal file system designed for XIP would separate metadata from pages of data, yet still perform in a broadly similar manner to JFFS2. However, designs for such a file system have yet to come any closer to completion than the above.

Removable storage

There are a multitude of forms of removable solid state storage. The best supported by Linux is CompactFlash, which presents itself to the system as an IDE device. It uses a translation layer as described above to emulate a block device, but this is performed internally to the device, and the host computer treats it exactly as if it were a normal IDE drive. Some CompactFlash devices perform wear levelling internally, but some do not. It is often not easy to tell whether a particular device performs wear levelling or not.

Another common form of removal storage is SmartMedia. SmartMedia is effectively just a NAND flash device, and the host computer must implement a similar translation layer in software. Linux does not currently support the SmartMedia translation layer, although there are drivers under development. However, there exist USB devices which perform the necessary transformations in their own firmware, presenting an interface to the host computer which is a simple USB storage device.

8 User Interface

With the advent of handheld devices running Linux, the user interface has become extremely important. Once the initial excitement of reaching a shell prompt over the serial console has passed, it rapidly becomes apparent that traditional Linux graphical user interfaces based on X Windows are not ideally suited for use on a 320x200 pixel display.

9 Alternatives

10 Conclusion

Linux has come a long way since Linus first played with multitasking by making a kernel which would interleave his two processes printing ‘AAAAAA’ and ‘BBBBBB’. Linux is becoming widely accepted in the embedded market, and there has been a great deal of good work on improving its applicability to these targets.

Still more development is currently under way and being planned, and it is clear from the discussions above that there remains a lot more work to be done in these areas in the future; certainly there's plenty to keep us from getting bored.

References

[eCos] Red Hat, Inc., *eCos — Embedded Configurable Operating System.* `http://sources.redhat.com/ecos/`

[Drepper] Ulrich Drepper `<drepper@cygnus.com>`, posting to `bug-glibc@gnu.org` mailing list, 24 May 1999. `http://sources.redhat.com/ml/bug-glibc/1999-05/msg00039.html`

[JFFS] David Woodhouse, Red Hat, Inc. *JFFS: The Journalling Flash File System*, May 2001. `http://sources.redhat.com/jffs2/jffs2.pdf`

Linux Security Module Framework

Chris Wright and Crispin Cowan [*]
WireX Communications, Inc.
chris@wirex.com, crispin@wirex.com

James Morris
Intercode Pty Ltd
jmorris@intercode.com.au

Stephen Smalley [†]
NAI Labs, Network Associates, Inc.
sds@tislabs.com

Greg Kroah-Hartman [‡]
IBM Linux Technology Center
gregkh@us.ibm.com

Abstract

Computer security is a chronic and growing problem, even for Linux, as evidenced by the seemingly endless stream of software security vulnerabilities. Security research has produced numerous access control mechanisms that help improve system security; however, there is little consensus on the best solution. Many powerful security systems have been implemented as research prototypes or highly specialized products, leaving systems operators with a difficult challenge: how to utilize these advanced features, without having to throw away their existing systems?

The Linux Security Modules (LSM) project addresses this problem by providing the Linux kernel with a general purpose framework for access control. LSM enables loading enhanced security policies as kernel modules. By providing Linux with a standard API for policy enforcement modules, the LSM project hopes to enable widespread deployment of security hardened systems. This paper presents the design and implementation of the LSM framework, a discussion of performance and security impact on the kernel, and a brief overview of existing security modules.

1 Introduction

Security is a chronic and growing problem: as more systems (and more money) go on line, the motivation to attack rises. Linux is not immune to this threat: the “many eyes make shallow bugs" argument [25] not withstanding, Linux systems do experience a large number of software vulnerabilities.

An important way to mitigate software vulnerabilities is through effective use of access controls. Discretionary access controls (`root`, user-IDs and mode bits) are adequate for user management of their own privacy, but are not sufficient to protect systems from attack. Extensive research in non-discretionary access control models has been done for over thirty years [2, 26, 18, 10, 16, 5, 20] but there has been no real consensus on which is the *one true* access control model. Because of this lack of consensus, there are many *patches* to the Linux kernel that provide enhanced access controls [7, 11, 12, 14, 17, 19, 24, 20, 32] but none of them are a *standard* part of the Linux kernel.

[*]This work supported in part by DARPA Contract N66001-00-C-8032 (Autonomix)

[†]This work supported by NSA Contract MDA904-01-C-0926 (SELinux)

[‡]This work represents the view of the authors and does not necessarily represent the view of IBM. But that sentence did.

The Linux Security Modules (LSM) [30, 27, 31] project seeks to solve this Tower of Babel [1] quandary by providing a general-purpose framework for security policy modules. This allows many different access control models to be implemented as loadable kernel modules, enabling multiple threads of security policy engine development to proceed independently of the main Linux kernel. A number of existing enhanced access control implementations, including POSIX.1e capabilities [29], SELinux, Domain and Type Enforcement (DTE) [14] and Linux Intrusion Detection System (LIDS) [17] have already been adapted to use the LSM framework.

The remainder of this paper is organized as follows. Section 2 presents the LSM design and implementation. Section 3 gives a detailed look at the LSM interface. Section 4 describes the impact LSM has on performance and security, including a look at some projects that have been ported to LSM so far. Section 5 presents our conclusions.

2 Design and Implementation

At the 2001 Linux Kernel Summit, the NSA presented their work on Security-Enhanced Linux (SELinux) [19], an implementation of a flexible access control architecture in the Linux kernel. Linus Torvalds appeared to accept that a general access control framework for the Linux kernel is needed. However, given the many Linux kernel security projects, and Linus' lack of expertise in sophisticated security policy, he preferred an approach that allowed security models to be implemented as loadable kernel modules. In fact, Linus' response provided the seeds of the LSM design. The LSM framework must be:

- truly generic, where using a different security model is merely a matter of loading a different kernel module;
- conceptually simple, minimally invasive, and efficient; and
- able to support the existing POSIX.1e capabilities logic as an optional security module.

To achieve these goals while remaining agnostic with respect to styles of access control mediation, LSM takes the approach of mediating access to the kernel's internal objects: tasks, inodes, open files, etc., as shown in Figure 1. User processes execute system calls, which first traverse the Linux kernel's existing logic for finding and allocating resources, performing error checking, and passing the classical UNIX discretionary access controls. Just before the kernel *attempts to* access the internal object, an LSM **hook** makes an out-call to the module posing the question, "Is this access ok with you?" The module processes this policy question and returns either "yes" or "no."

One might ask why LSM chose this approach rather than *system call interposition* (mediating system calls as they enter the kernel) or *device mediation* (mediating at access to physical devices).[1] The reason is that information critical to sound security policy decisions is not available at those points. At the system call interface, userspace data, such as a path name, has yet to be translated to the kernel object it represents, such as an inode. Thus, system call interposition is both inefficient and prone to time-of-check-to-time-of-use (TOCTTOU) races [28, 6]. At the device interface, some other critical information (such as the path name of the file to be accessed) has been thrown away. In between is where the full context of an access request can be seen, and

[1]The glib answer is that the Linux kernel already provides those features and there would be nothing for us to do :-)

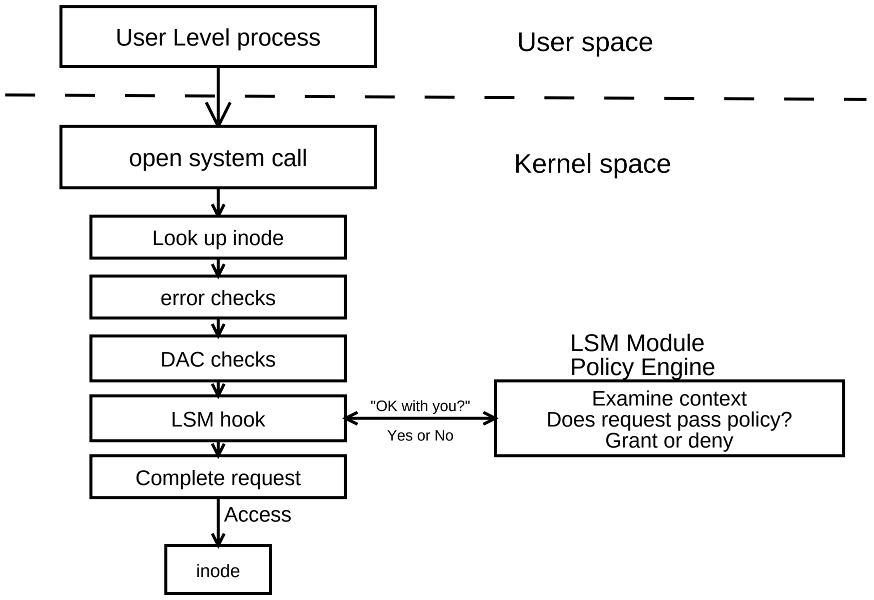


Figure 1: LSM Hook Architecture

where a fully informed access control decision can be made.

A subtle implication of the LSM architecture is that access control decisions are *restrictive*[2]: the module can really only say "no" [31]. Functional errors and classical security checks can result in an access request being denied before it is ever presented to the LSM module. This is the opposite of the way mandatory access control systems are normally implemented. This design choice limits the flexibility of the LSM framework, but substantially simplifies the impact of the LSM on the Linux kernel. To do otherwise would have required implementing many instances of the *same* hook throughout the kernel, to ensure that the module is consulted at *every* place where a system call could "error out."

Composition of LSM modules is another problematic issue. On the one hand, security policies do not compose in the *general case* because some policies may explicitly conflict [13]. On the other hand, it is clearly desirable to compose some combinations of security policies. Here, LSM effectively punts to the module writer: to be able to "stack" modules, the first module loaded must also export an LSM interface to subsequent LSM modules to be loaded. The first module is then responsible for composing the access control decisions that it gets back from secondary modules.

3 LSM Interface

Having discussed the high-level design philosophies of LSM in Section 2, we now turn to the implementation of the LSM interface. At the core, the LSM interface is a large table of functions, which by default are populated with calls that implement the traditional superuser DAC policy. The module writers are then responsible for providing implementations of the functions that they care about. This section provides a detailed

[2]Caveat: the `capable()` hook, which is needed to support POSIX.1e capabilities, can override DAC checks, see Section 3.8.

analysis of those functions.[3] Section 3.1 shows how to register a security module. Sections 3.2 through 3.8 are organized by kernel object and discuss the LSM interface available to mediate access to each object.

3.1 Policy Registration

The LSM interface is implemented as a structure of callback methods, `security_ops`. A security module is responsible for implementing the callbacks according to the security policy it is enforcing. At boot time the `security_ops` structure is initialized with default callbacks, which implement traditional superuser semantics.

The security module can be built as a dynamically loadable module or statically linked into the kernel. It is initialized either at module load time for dynamically loaded modules or during `do_initcalls()` for statically linked modules. During this initialization, the security module must register its callbacks with the LSM framework by calling `register_security()`. A module should call `unregister_security()` when it is unloaded to return the `security_ops` structure to its default superuser policy.

The LSM framework is aware of only one primary security policy at any time. Once a security policy is registered with the LSM framework, subsequent attempts to register new security policies will fail. In some cases it is appropriate to compose security policies, as noted in Section 2. LSM allows modules to stack with each other, however, the framework remains aware of only a single `security_ops` structure. In order to register additional security policies, the subsequent modules register with the primary module using `mod_reg_security()`. This allows the LSM framework to remain simple, pushing the policy which defines composition into the primary security module.

[3]However, it is not a programmer's guide.

3.2 Task Hooks

The `task_struct` structure is the kernel object representing kernel schedulable tasks. It contains basic task information such as user and group ID, resource limits, and scheduling policies and priorities. LSM provides a group of task hooks, `task_security_ops`, that mediate a task's access to this basic task information. Interprocess signalling is mediated by the LSM task hooks to monitor tasks' abilities to send and receive signals. LSM adds a security field to the `task_struct` to allow security policies to label a task with a policy specific security label.

The LSM task hooks have full task life-cycle coverage. The `create()` task hook is called, verifying that a task can spawn children. If this is successful, a new task is created and the `alloc_security()` task hook is used to manage the new task's security field. When a task exits, the `kill()` task hook is consulted to verify that the task can signal its parent. Similarly, the `wait()` task hook is called in the parent task context, verifying the parent task can receive the child's signal. And finally, the task's security field is released by the `free_security()` task hook.

During the life of a task it may attempt to change some of its basic task information. For example a task may call `setuid(2)`. This is, of course, managed by LSM with a corresponding `setuid()` task hook. If this is successful the kernel updates the task's user identity and then notifies the policy module via the `post_setuid()` task hook. The notification allows the module to update state and, for example, update the task's security field.

To avoid leaking potentially sensitive task information, LSM mediates the ability to query another task's state. So, for example, a query for the process group ID or the scheduler policy of an arbitrary task is protected by the `getpgid()` or `getscheduler()` task hooks respectively.

3.3 Program Loading Hooks

The `linux_binprm` structure represents a new program being loaded during an `execve(2)`. LSM provides a set of program loading hooks, `binprm_security_ops`, to manage the process of loading new programs. Many security models, including Linux capabilities, require the ability to change privileges when a new program is executed. Consequently, these LSM hooks are called at critical points during program loading to verify a task's ability to load a new program and update the task's security field.

LSM adds a security field to the `linux_binprm` structure. At the beginning of an `execve(2)` after the new program file is opened, the `alloc_security()` program loading hook is called to allocate the security field. The `set_security()` hook is used to save security information in the `linux_binprm` security field. This hook may be called multiple times during a single `execve(2)` to accommodate interpreters. Either of these program loading hooks can be used to deny program execution.

In the final stages of program loading, the `compute_creds()` program loading hook is called to set the new security attributes of a task being transformed by `execve(2)`. Typically, this hook will calculate the task's new credentials based on both its old credentials and the security information stored in the `linux_binprm` security field. Once the new program is loaded, the kernel releases the `linux_binprm` security field by calling the `free_security()` program loading hook.

3.4 File System Hooks

The VFS layer defines three primary objects which encapsulate the interface that low level filesystems are developed against: the `super_block`, the `inode` and the `file`. Each of these objects contains a set of operations that define the interface between the VFS and the actual filesystem. This interface is a perfect place for LSM to mediate filesystem access. The LSM filesystem hooks are described in Sections 3.4.1 through 3.4.3.

3.4.1 Super Block Hooks

The kernel's `super_block` structure represents a filesystem. This structure is used when mounting and unmounting a filesystem or obtaining filesystem statistics, for example. The super block hooks, `super_block_security_ops`, mediate the various actions that can be taken on a `super_block`. As a simple example, the `statfs()` super block hook checks permission when a task attempts to obtain a file system's statistics.

When mounting a filesystem, the kernel first validates the request by calling the `mount()` super block hook. Assuming success, a new `super_block` is created[4] regardless of whether it is backed by a block device or by an anonymous device. The kernel then allocates space for a security field in the new `super_block` by calling the `alloc_security()` super block hook. Next, when the `super_block` is to be added to the global tree, the `check_sb()` super block hook is called to verify that the filesys-

[4]In some cases, `super_blocks` are recycled.

tem can indeed be mounted at the point in the tree that is being requested. If this is successful, a `post_addmount()` hook is invoked to synchronize the security module's state.

The super block hook `umount()` is called to check permission when unmounting a filesystem. If successful, the `umount_close()` hook is used to synchronize state and, for example, close any files in the filesystem that are held open by the security module. Once the `super_block` is no longer referenced, it will be deleted, and the `free_security()` hook will free the security field.

3.4.2 Inode Hooks

The kernel's `inode` structure represents a basic filesystem object, e.g., a file, directory, or symlink. The LSM inode hooks mediate access to this fundamental kernel structure. A well defined set of operations, `inode_operations`, describe the actions that can be taken on an inode — `create()`, `unlink()`, `lookup()`, `mknod()`, `rename()`, and so on. This encapsulation defines a nice interface for LSM to mediate access to the `inode` object. In addition, LSM adds a security field to the `inode` structure and corresponding inode hooks to manage security labelling.

The kernel's `inode` cache is populated by either file lookup operations or filesystem object-creation operations. When a new `inode` is created, the security module allocates space for the `inode` security field with the `alloc_security()` inode hook. Either post-lookup or post-creation, the newly created objects are labelled. The label may be cleared by the `delete()` inode hook when an inode's link count reaches zero. And finally, when an `inode` is destroyed, the `free_security()` inode hook is called to release the space allocated for the security field.

In many cases, the LSM inode hooks are identical to the `inode_operations`. For all `inode_operations` that can create new filesystem objects a "post" inode hook is defined for coherent security labelling. For example, when a task creates a new symlink, the `symlink()` inode hook is called to check permission to create the symlink. Then if the symlink creation is successful, the `post_symlink()` hook is called to set the security label on the newly created symlink.

Whenever possible, LSM leverages the existing Linux kernel security infrastructure. The kernel's standard UNIX DAC checks compare the uids, guids, and mode bits when checking for permission to access filesystem objects. The VFS layer already has a `permission()` function which is a wrapper for the `permission()` `inode_operation`. LSM uses this pre-existing infrastructure and adds its `permission()` inode hook to the VFS wrapper.

3.4.3 File Hooks

The kernel's `file` structure represents an open filesystem object. It contains the `file_operations` structure, which describes the operations that can be done to a `file`. For example, a `file` can be read from and written to, seeked through, mapped into memory, and so on. Similar to the inode hooks, LSM provides file hooks to mediate access to `files`, many of which mirror the `file_operations`. A security field has been added to the `file` structure for labelling.

When a file is opened, a new `file` object is created. At this time, the alloc_security() file hook is called to allocate a security field and label the file. This label persists until the file is closed, when the free_security() file hook is called to free the security field.

The permission() file hook can be used to revalidate read and write permissions at each file read or write. This is not effective against reading and writing of memory mapped files, and the changes required to support this page level revalidation are considered too invasive. Actually mapping a file is, however, protected with the mmap() file hook. And changing the protection bits on mapped file regions must pass the mprotect() file hook.

When using file locks to synchronize multiple readers or writers, a task must pass the lock() file hook permission check before performing any locking operation on a file.

If the O_ASYNC flag is set on a file, asynchronous I/O ready signals are delivered to the file owner when the file is ready for input or output. The ability to specify the task that will receive the I/O ready signals is protected by the set_fowner() file hook. Also, the actual signal delivery is mediated by the send_sigiotask() file hook.

Miscellaneous file operations that come through the ioctl(2) and fcntl(2) interfaces are protected by the ioctl() and fcntl() file hooks respectively. Another miscellaneous action protected by the file hooks is the ability to receive an open file descriptor through a socket control message. This action is protected by the receive() file hook.

3.5 IPC Hooks

The Linux kernel provides the standard SysV IPC mechanisms: shared memory, semaphores, and message queues. LSM defines a set of IPC hooks which mediate access to the kernel's IPC objects. Given the design of the kernel's IPC data structures, LSM defines one common set of IPC hooks, ipc_security_ops, as well as sets of object specific IPC hooks: shm_security_ops, sem_security_ops, msg_queue_security_ops, and msg_msg_security_ops.

3.5.1 Common IPC Hooks

The kernel's IPC object data structures share a common credential structure, kern_ipc_perm. This structure is used by the kernel's ipcperms() function when checking IPC permissions. LSM adds a security field to this structure and an ipc_security_ops hook, permission(), to ipcperms() to give the security module access to these existing mediation points. LSM also defines an ipc_security_ops hook, getinfo(), to mediate info requests for any of the IPC objects.

3.5.2 Object Specific IPC Hooks

The LSM IPC object specific hooks define the alloc_security() and free_security() functions to manage the security field in each object's kern_ipc_perm data structure. An IPC object is created with an initial "get" request, which triggers the object specific alloc_security. If the "get" request finds an already existing object, the associate() hook is called to check permissions before returning the object.

IPC object control commands, shmctl(2), semctl(2), and msgctl(2) are mediated by object specific "ctl" hooks. For example, when a SHM_LOCK request is issued, the shm_security_ops shmctl() hook is checked for permission prior to completing the request.

Any attempt to change a semaphore count is protected by the sem_security_ops semop() hook. Attaching to a shared memory segment is protected by the shm_security_ops shmat() hook. Sending and receiving messages on a message queue are protected by the msg_queue_security_ops msgsnd() and msgrcv() hooks. The individual messages are considered as well as the queue when verifying permission. When a new message is created, the msg_msg_security_ops alloc_security() hook allocates the security field stored in the actual message data structure. Upon receipt, the msgrcv() hook can verify the security field on both the queue and the message.

3.6 Module Hooks

The LSM interface would surely be incomplete if it didn't mediate loading and unloading kernel modules. The LSM module loading hooks, module_security_ops, add permission checks protecting the creation and initialization of loadable kernel modules as well as module removal.

3.7 Network Hooks

The Linux kernel features an extensive suite of network protocols and supporting components. As networking is an important aspect of Linux, LSM extends the concept of a generalized security framework to this area of the kernel.

A key implementation challenge was to determine the initial requirements for the network hooks. The existing SELinux implementation was utilized as a model, as SELinux is itself a highly generalized security infrastructure which was to be ported to LSM. Other Linux security projects were reviewed, although none relevant to the version 2.5 kernel series were found with networking requirements in excess of SELinux. Potential requirements for IPSec and traditional labeled networking systems were also taken into account.

As the Linux network stack utilizes the Berkeley sockets model [21], LSM is able to provide coarse coverage for all socket-based protocols via the use of hooks within the socket layer.

Additional finer-grained hooks have been implemented for the IPv4, UNIX domain, and Netlink protocols, which were considered essential for the implementation of a minimally useful system. Similar hooks for other protocols may be implemented at a later stage.

Coverage of low level network support components such as routing tables and traffic classifiers is somewhat limited due to the invasiveness of the code which would be required to implement consistent fine-grained hooks. Accesses to these objects can be interposed at higher levels (e.g., via system calls such as ioctl(2)), although granularity may be reduced by TOCTTOU issues. The existing kernel code does however impose a CAP_NET_ADMIN capability requirement for tasks which attempt to write to important network support components.

The details of the network hooks are described in Sections 3.7.1 through 3.7.6.

3.7.1 Sockets and Application Layer

Application layer access to networking is mediated via a series of socket-related hooks, socket_security_ops. When an application attempts to create a socket with the socket(2) system call, the create()

hook allows for mediation prior to the actual creation of the socket. Following successful creation, the `post_create()` hook may be used to update the security state of the inode associated with the socket.

Since active user sockets have an associated `inode` structure, a separate security field was not added to the `socket` structure or to the lower-level `sock` structure. However, it is possible for sockets to temporarily exist in a state where they have no `socket` or `inode` structure. Hence, the networking hook functions must take care in extracting the security information for sockets.

Mediation hooks are also provided for all of the socket system calls:

```
    bind(2)
    connect(2)
    listen(2)
    accept(2)
    sendmsg(2)
    recvmsg(2)
    getsockname(2)
    getpeername(2)
    getsockopt(2)
    setsockopt(2)
    shutdown(2)
```

Protocol-specific information is available via the `socket` structure passed as a parameter to all of these hooks (except for `create()`, as the socket does not yet exist at this hook). This facilitates mediation based on transport layer attributes such as TCP connection state, and seems to obviate the need for explicit transport layer hooks.

The `sock_rcv_skb()` hook is called when an incoming packet is first associated with a socket. This allows for mediation based upon the security state of receiving application and security state propagated from lower layers of the network stack via the `sk_buff` security field (see section 3.7.2).

Additional socket hooks are provided for UNIX domain communication within the abstract namespace, as binding and connecting to UNIX domain sockets in the abstract namespace is not mediated by filesystem permissions. The `unix_stream_connect()` hook allows mediation of stream connections, while datagram based communications may be mediated on a per-message basis via the `unix_may_send()` hook.

3.7.2 Packets

Network data traverses the network stack in packets encapsulated by a structure called an `sk_buff` (socket buffer). The `sk_buff` structure provides storage for packet data and related state information, and is considered to be owned by the current layer of the network stack.

LSM adds an opaque security field to the `sk_buff` structure, so that security state may be managed across network layers on a per-packet basis.

A set of `sk_buff` hooks is provided for lifecycle management of the security field. For LSM, the critical lifecycle events for an `sk_buff` are:

- Allocation
- Copying
- Cloning
- Setting ownership to sending socket
- Datagram reception
- Destruction

Hooks are provided for each of these events, although they are only intended to be used for maintaining the security field data. Encoding, decoding and interpretation of the security field data is performed by layer-specific hooks such as the socket and network layer hooks.

Generally, the `sk_buff` hooks and security field only need to be used when the security state of a packet must be managed between layers of the network stack. Examples of such cases include labeled networking via IP options and management of nested IPSec Security Associations [15].

3.7.3 Transport Layer (IPv4)

Explicit hooks are not required for the transport layer, as sufficient protocol state information for LSM is available at the socket and network layer hooks (discussed in section 3.7.1).

3.7.4 Network Layer (IPv4)

Hooks are provided at the network layer for IPv4 to facilitate:

- Integrated packet filtering
- IP options decoding for labeled networking
- Management of fragmented datagrams
- Network layer encapsulation (e.g., secure IP tunnels)

Existing Netfilter [23] hooks are used to provide access to IP datagrams in pre-routing, local input, forwarding, local output and post-routing phases. Through these hooks, LSM intercepts packets before and after the standard iptables-based access control and translation mechanisms. Note that the Netfilter hooks used by LSM do not increase the code footprint imposed by LSM on the standard kernel.

3.7.5 Network Devices

Within the Linux network stack, hardware and software network devices are encapsulated by a `net_device` structure. LSM adds an security field to this structure so that security state information can be maintained on a per-device basis.

The security field for the `net_device` structure may be allocated during first-use initialization. A security field management hook is called when the device is being destroyed, allowing any allocated resources associated with the associated security field to be freed.

3.7.6 Netlink

Netlink sockets are a Linux-specific mechanism for kernel-userspace communication. They are similar to BSD route sockets, although more generalized.

As Netlink communications are connectionless and asynchronously processed, security state associated with an application layer origin needs to be stored with Netlink packets, then checked during delivery to the destination kernel module. The `netlink_send()` hook is used to store the application layer security state. The `netlink_recv()` hook is used to retrieve the stored security state as the packet is received by the destination kernel module and mediate final delivery.

3.8 Other System Hooks

LSM defines a miscellaneous set of hooks to protect the remaining security sensitive actions that are not covered by the hooks discussed above. These hooks typically mediate system-level actions such as setting the system's host name or domain name, rebooting the system, and accessing I/O ports. The existing capability checks already protect these actions; however, the LSM hooks provide more finely grained access control.

The LSM interface leverages the pre-existing POSIX.1e capabilities infrastructure in the Linux kernel. The capability checks can often override standard DAC checks (akin to root). The checks are limited to a 32 bit vector describing the required capability, e.g., `CAP_DAC_OVERRIDE`, and thus give the module limited context when making access control decisions. The system-level `capable()` hook is placed in the existing `capable()` function which gives LSM easy compatibility with POSIX.1e capabilities as well as a moderate ability to override DAC checks.

The LSM framework adds a security system call, which is a thin wrapper around the `sys_security()` hook in the LSM interface. This system call is a simple multiplexor which allows a module to define a set of policy specific system calls. The LSM security system call interface is modeled after the standard Linux socket system call multiplexor, `sys_socketcall(2)`.

4 Testing and Functionality

The true impact of LSM will be felt if and when LSM is accepted as a standard part of the Linux kernel, and end-users can adopt security modules as readily as they adopt other applications for Linux. To be accepted into Linux, LSM must be highly cost-effective. Section 4.1 summarizes the performance cost of the LSM infrastructure. Section 4.2 presents the security impact of LSM, in the form of modules that have already been implemented or ported to LSM.

4.1 Performance Impact

The performance cost of the LSM framework is critical to its acceptance; in fact, performance cost was a major part of the debate at the Linux 2.5 developer's summit that spawned LSM. To rigorously document the performance costs of LSM, we performed both microbenchmarks and macrobenchmarks that compared a stock Linux kernel to one modified with the LSM patch, but with no modules loaded.[5]

For microbenchmarks, we used the LMBench [22] tool. LMBench was developed specifically to measure the performance of core kernel system calls and facilities, such as file access, context switching, and memory movement. LMBench has been particularly effective at establishing and maintaining excellent performance in these core facilities in the Linux kernel.

LMBench outputs prodigious results. The worst case overhead was 6.2% for `stat()`, 6.6% for `open/close`, and 7.2% for file delete. These results are to be expected, because of the relatively small amount of work done in each call compared to the work of checking for LSM mediation. The common case was much better, often 0% overhead, ranging up to 2% overhead.

For macrobenchmarking, we used the common approach of building the Linux kernel from source. The results here were even better: no measurable performance impact.[6] More detailed performance data can be found in [31].

[5]The performance costs of each module are the responsibility of the module's authors.

[6]In fact, the LSM case was actually faster, but we regard that as an experimental anomaly, and do not claim that LSM is a performance optimization :-)

4.2 Security Impact

Another key factor in the acceptance of the LSM framework is that it provide some real security value. This can be viewed in two ways. First, LSM must not create new security holes and needs to be thorough and consistent in its coverage. Second, the LSM framework must be general enough to support a variety of access control models.

Proving the correctness of the LSM framework has not been handled by the LSM project directly. However, a project from IBM [9] has developed tools to do both static and dynamic analysis of the LSM framework. These tools have, in fact, helped improve the LSM interface, and can help with ongoing maintenance.

The real value of LSM is delivering effective security modules. Porting access control models to the LSM framework proves that it is functional as a general purpose access control framework. As the name suggests, LSM does not impact system security without security modules. Presently, LSM supports the following security modules:

- **SELinux** A Linux implementation of the Flask [28] flexible access control architecture and an example security server that supports Type Enforcement, Role-Based Access Control, and optionally Multi-Level Security. SELinux was originally implemented as a kernel patch [19] and was then reimplemented as a security module that uses LSM. SELinux can be used to confine processes to least privilege, to protect the integrity and confidentiality of processes and data, and to support application security needs. The generality and comprehensiveness of SELinux helped to drive the requirements for LSM.

- **DTE Linux** An implementation of Domain and Type Enforcement [3, 4] developed for Linux [14]. Like SELinux, DTE Linux was originally implemented as a kernel patch and was then adapted to LSM. With this module loaded, types can be assigned to objects and domains to processes. The DTE policy restricts access between domains and from domains to types. The DTE Linux project also provided useful input into the design and implementation of LSM.

- **LSM port of Openwall kernel patch** The Openwall kernel patch [8] provides a collection of security features to protect a system from common attacks, e.g., buffer overflows and temp file races. A module is under development that supports a subset of the Openwall patch. For example, with this module loaded a victim program will not be allowed to follow malicious symlinks.

- **POSIX.1e capabilities** The POSIX.1e capabilities [29] logic was already present in the Linux kernel, but the LSM kernel patch cleanly separates this logic into a security module. This change allows users who do not need this functionality to omit it from their kernels and it allows the development of the capabilities logic to proceed with greater independence from the main kernel.

- **LIDS (Linux Intrusion Detection System)** started out as an intrusion detection system, and then migrated towards intrusion prevention in the form of an access control system similar to SubDomain [7] that manages access by describing what files a given *program* may access.

5 Conclusions

Linux is a shared playroom, and thus needs to make most players reasonably happy. LSM thus needs to meet two criteria: be relatively painless for people who don't want it, and be useful and effective for people who do want it.

We feel that LSM meets these criteria. The patch is relatively small, and the performance data in Section 4 shows that the LSM patch imposes nearly zero overhead. The broad suite of security products from around the world that have been implemented for LSM shows that the LSM API is useful and effective for developing Linux security enhancements.

6 Acknowledgements

Thanks to the LSM mailing list for engaging in the sometimes tedious and heated discussions that helped shape LSM. Special thanks to the SELinux project that helped kickstart LSM with the original presentation at the 2001 Kernel Summit.

7 Availability

LSM is available as a kernel patch for both the 2.4 and 2.5 Linux kernels. The patches are available from `http://lsm.immunix.org`.

References

[1] The Holy Bible: Genesis 11:1-8.

[2] J. Anderson. Computer Security Technology Planning Study. Report Technical Report ESD-TR-73-51, Air Force Elect. Systems Div., October 1972.

[3] L. Badger, D.F. Sterne, and et al. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.

[4] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the USENIX Security Conference*, 1995.

[5] D. Baker. Fortresses built upon sand. In *Proceedings of the New Security Paradigms Workshop*, 1996.

[6] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at `http://olympus.cs.ucdavis.edu/~bishop/scriv/index.html`.

[7] Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.

[8] Solar Designer. Non-Executable User Stack. `http://www.openwall.com/linux/`.

[9] Antony Edwards and Xiaolan Zhang. Using CQUAL for Static Analysis of Authorization Hook Placement. In *USENIX Security Symposium*, San Francisco, CA, August 2002.

[10] M. Abrams et al. *Information Security: An Integrated Collection of Essays*. IEEE Comp., 1995.

[11] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[12] Timothy Fraser. LOMAC: MAC You Can Live With. In *Proceedings of the FREENIX Track, USENIX Annual Technical Conference*, Boston, MA, June 2001.

[13] Virgil D. Gligor, Serban I Gavrila, and David Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[14] Serge Hallyn and Phil Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[15] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[16] Jay Lepreau, Bryan Ford, and Mike Hibler. The persistent relevance of the local operating system to global applications. In *Proceedings of the ACM SIGOPS European Workshop*, pages 133–140, September 1996.

[17] Linux Intrusion Detection System. World-wide web page available at `http://www.lids.org`.

[18] T. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4), December 1976.

[19] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[20] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, October 1998.

[21] M. K. McKusick, M. J. Karels, S. J. Leffler, W. N. Joy, and R. S. Faber. *Berkeley Software Architecture Manual, 4.4BSD Edition*. University of California, Berkeley, Berkeley, CA, 1994.

[22] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, 1996. `http://www.bitmover.com/lmbench/`.

[23] Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4, 1999. `http://www.netfilter.org/`.

[24] Amon Ott. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In *Proceedings of the 8th International Linux Kongress*, November 2001.

[25] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, 1999. `http://www.oreilly.com/catalog/cb/`.

[26] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[27] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux Security Modules: General Security Hooks for Linux. `http://lsm.immunix.org/`, September 2001.

[28] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.

[29] Winfried Trumper. Summary about POSIX.1e. `http://wt.xpilot.org/publications/posix.1e`, July 1999.

[30] WireX Communications. Linux Security Module. `http://lsm.immunix.org/`, April 2001.

[31] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium*, San Francisco, CA, August 2002.

[32] Marek Zelem and Milan Pikula. ZP Security Framework. `http://medusa.fornax.sk/English/medusa-paper.ps`.

Mandatory Access Control for Linux Clustered Servers

Miroslaw Zakrzewski
Open Systems Lab
Ericsson Research
8400 Decarie Blvd
Town of Mont Royal, Quebec
Canada H4P 2N2
Miroslaw.Zakrzewski@ericsson.ca

Abstract

In today's world, the use of computers and networks is growing and the vision of a single infrastructure for voice and data is becoming a reality. However, with different technologies and services using the same networking infrastructure, the realization of this vision requires higher levels of security to be implemented in computer systems. Current security solutions do not address all of the security challenges facing today's computer systems, including clustered platforms, in one comprehensive and coherent fashion.

This paper presents the previous work done in the area of access control and then focus on new mechanisms for clustered Linux servers as part of the research project at the Ericsson Open Systems Lab. In this paper, we address the design and implementation of a framework for the mandatory access control in the distributed security infrastructure (DSI). The ongoing work is mainly based on the Flask architecture and the Linux Security Module (LSM) framework with a focus on Linux clustered servers. The paper also addresses the effects of the cluster security on the performance of the distributed system, since enforcing security may introduce degradation in the performance, an increase in administration, and some annoyance for the user.

We are implementing cluster-aware access control mechanisms in the Linux kernel. We expect that our work will help position Linux as a secure operating system for clustered servers.

1 Introduction

The security of computing systems could be enforced on different levels of the computing environment such as hardware, operating system, application and network level. The primary subject from the security prospective is the operating system level, being a fundamental piece of the security of every computer system, and a critical point of failure for the entire system. Currently implemented security mechanisms of operating systems are based on user privileges and are inadequate to protect against the various kinds of attacks in today's complex environments. To address these problems, security in operating systems has long been a well-researched topic, which formulated various security models and policies.

Various research results have shown that mandatory security provided by the operating system is essential for the security of the whole system [3]; furthermore, they proved that mandatory access control mechanisms are very efficient in supporting complex relationships between different entities in the computing environment.

Several attempts were made to reach a very secure platform. For instance, The FLASK architecture [5,12] (on which SE Linux [17] is based) was created as an attempt to serve as a generic architecture for the mandatory access control. An important design goal was to provide flexible support for security policies. The FLASK architecture achieved the goal by separating the security policy from the enforcement mechanism and by having security checks transparent to the applications. Another attempt from SE Linux was to prototype the access control in the Linux kernel.

However, the existing solutions, including Flask and SE Linux, do not address the access control in distributed environments. One such environment is computer cluster. In our context, a cluster is defined as a collection of interconnected stand-alone computers working together to solve a problem as a single computer. The cluster can appear as a single system to users and applications. Since from the logical point of view we can see a cluster as a single entity, we should apply this definition to the cluster security as well and treat the subjects and resources as if they were located on one virtual machine.

Even though new security approaches, such as FLASK, address the problem of mandatory access control between subjects and resources belonging to the same processing node, they are still missing the mandatory, finer-grained security checks between the subjects and resources belonging to different nodes.

There exist many security solutions for clustered servers ranging from external solutions, such as firewalls, to internal solutions such as integrity checking software. However, there is no solution dedicated for clusters. The most commonly used security approach is to package several existing solutions. Nevertheless, the integration and management of these different packages is very complex, and often results in the absence of interoperability between different security mechanisms. Additional difficulties are also raised when integrating these many packages, such as the ease of system maintenance and upgrade, and the difficulty of keeping up with numerous security patches and upgrades.

Carrier class clusters have very tight restrictions on performance and response time. Therefore, much pressure is put on the system designer while designing security solutions. In fact, many security solutions cannot be used due to their high resource consumption.

In a distributed environment, subjects and resources can be located anywhere on the network so the relations between them are more complex.

In this paper, we present the preliminary results developing the Linux security module (LSM) that links all the nodes of the cluster in a transparent fashion; the Linux security module is also referred to as the distributed security module. The security module enforces the security checking on a node between subjects and resources belonging to the same or different nodes of the cluster. The distributed security module is a part of the distributed security infrastructure (DSI) and cannot be used without it. The DSI decides about the security policy and defines mechanisms that control the module. In the next section, a brief description of the distribution security infrastructure is introduced.

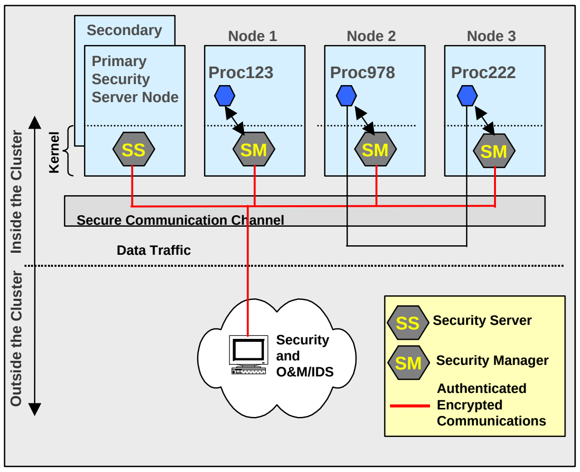


Figure 1: Distributed Architecture of DSI

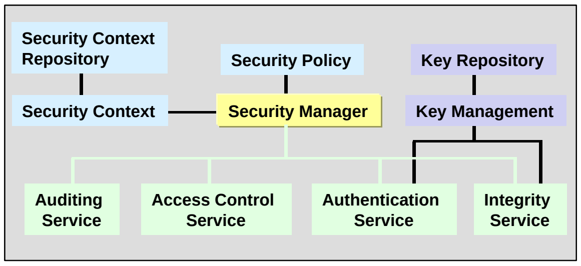


Figure 2: DSI Services

2 Distributed Security Infrastructure

2.1 DSI Characteristics

As part of a carrier class Linux cluster, DSI [6] must comply with carrier class requirements such as reliability, scalability, and high availability. Furthermore, DSI supports the following requirements: coherent framework, process level approach, pre-emptive security, dynamic security policy, transparent key management, and minimal impact on performance.

2.2 DSI Architecture

DSI has two types of components: the management components and service components. DSI management components define a thin layer of components that includes a security server, security managers, and a security communication channel (Figure1). The service components define a flexible layer, which can be modified or updated by adding, replacing, or removing services according to the needs.

The security server is the central point of management in DSI, the entry point for secure operation and management, and intrusion detection systems from outside the cluster. It is the central security authority for all the security components in the system. It is responsible for the distributed security policy. It also defines the dynamic security environment of the whole cluster by broadcasting changes in the distributed policy to all security managers.

Security managers enforce security at each node of the cluster. They are responsible for locally enforcing changes in the security environment. Security managers only exchange security information with the security server.

The secure communication channel provides encrypted and authenticated communications between the security agents. All communications between the security server and the outside of the cluster take place through the secure communication channel.

The DSI architecture at each node is based on a set of loosely coupled services (Figure 2). Each service, upon its creation, sends a presence announcement to the local security manager, which registers these services and provides their access mechanisms to the internal modules.

There are two types of services: security services (access control, authentication, integration, auditing) and security service providers (for example secure key management) that run at user level and provide services to security managers.

3 Cluster Access Control

3.1 General Discussion

In general, the Access Control Service (ACS) can be seen as a layer (software, hardware) that enforces the security policy as a two-parameter function. It relies on the notions of subject (or access request initiator), resource (or target), environment, decision, and enforcement.

A subject could be a program or process and a resource can be a file or a communication resource. The same process can be a subject in one access control operation and a resource in another.

An access control could be interpreted as a matrix where one axis is the list of all possible subjects and the other is the list of all possible resources. The entries in the matrix define the permissions. Even for reasonable-sized systems the matrix gets complicated, very fast so there is a need to reduce its complexity. In order to do this, the term class is introduced. Class groups the subjects and resources, which have the same permission and create only one entry for them in the matrix.

When a Subject tries to access a Resource (Figure 3), the access request is intercepted by the access control layer and based on the subject's rights, the access either is granted or not. The access control of an operation system is usually added in the system call layer (Linux). This is ideal for the operating system because it makes the access control transparent for the applications, and more secure because it's located in one of the lowest software layers, in addition it's fast because it's embedded into the operation system.

The ACS assumes that the subjects have been properly authenticated. One of the important characteristics of the access control for clusters is that it allows verifying the access control privileges even when subjects and resources are located on different nodes in the cluster.

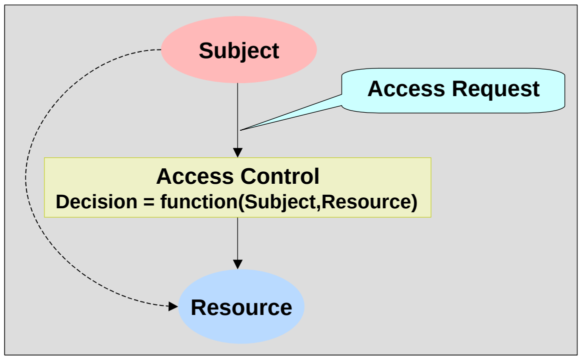


Figure 3: Access Control

The ACS that runs on the cluster processors is comprised of two parts:

1. A kernel-space part: This part is responsible for implementing both the enforcement and the decision-making tasks of access control. These two responsibilities are separated. The kernel-space part maintains an internal representation of the information upon which it bases its decisions. This information (security policy) is supplied by the security server and stored in the local memory for fast access (hash table). On Linux, the kernel-space part is implemented as a Linux Security Module (LSM).

2. A user-space part: This part has many responsibilities. It takes the information from the Distributed Security Policy and from the Security Context Repository, combines them together, and feeds them to the kernel space part in an easily usable form. It also takes care of propagating back alarms from the kernel space part to the security manger, which will feed

them to the Auditing and Logging Service and if necessary propagate to the security server through SCC.

Both parts, kernel-space and user-space, are started and monitored by the local Security Manager (SM) on each node. The SM also introduces them to other services and subsystems of the DSI infrastructure with which they need to interact.

The ACS aims to provide fine-grained access control (at the system call level). It respects the minimization principles of least privilege to limit the propagation and damage caused by eventual security breaches. As such, it provides defense in depth.

The ACS that is running on a processor must make as little assumptions as possible about other processors, including whether they have been compromised. For that reason, an ACS instance is always the one making access decisions about resources that are local to its processor. For the initial design of the ACS, only grant/deny decision will be considered. Other more involved decisions would involve rate limiting and total usage limiting. Actions other than access control decision, such as interposition and active reactions, are not implemented either.

3.2 Cluster Access Types

The distributed environments allow that the actors of ACS (subject and resource) can be located anywhere in the cyber space.

Based on their mutual location in the cluster (Figure 4), and to reduce the complexity when analyzing access control, we can distinguish the following types of the access control:

- Cluster Local Access: Both subject and resource are located on the same node in the cluster

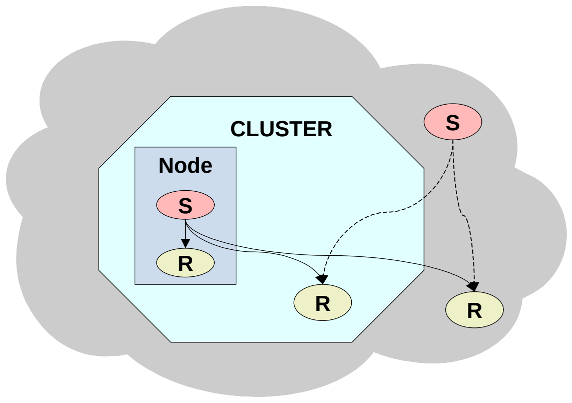


Figure 4: Cluster Access Control

- Cluster Remote Access: Both subject and resource are located on different nodes inside the same cluster
- Cluster Outside Access: Subject is located on a node inside cluster and resource is outside the cluster or Subject is located outside the cluster and resource on a node inside the cluster.
- No Cluster Access: Both subject and resource are outside the cluster

The above classification allows us to reduce the complexity of the cluster access control by classifying the various access approaches. First, we analyze the Cluster Local Access and next we will move to the Cluster Remote Access. The Cluster Outside Access and No Cluster Access are out of the scope of this paper.

3.3 Distributed Access Control Architecture

Finding an efficient solution to the cluster mandatory access control is a complex task. There are many factors involved in defining the access rights because the subjects and resources can be located on different nodes in the

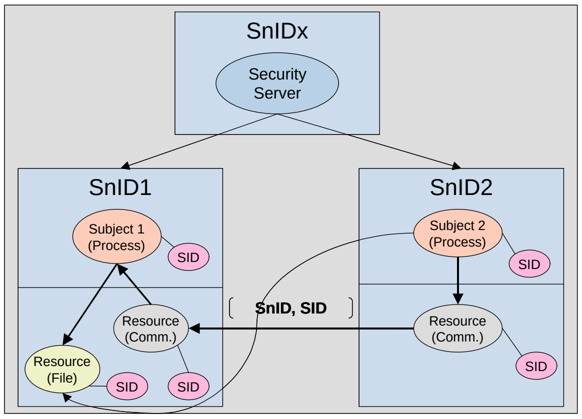


Figure 5: Distributed Access Control

cluster. To simplify the relationships, we can handle the access control in two levels:

1. Local when subject and resource are located on the same node, and
2. Remote when subject and resource are located on different nodes.

For local access control, the access rights are the functions of the security IDs of the subject (SSID) and the resource (TSID). This is based on the FLASK architecture:

```
Access = Function (SSID, TSID)
```

The FLASK architecture can serve as a solution for the single node processing. When the nodes are presented as a cluster, security solutions become more complicated. In this case, we extend the FLASK architecture to the cluster remote access model. One of the new parameters is the security node ID (SnID) (Figure 5), which defines the node in terms of the security. Access rights are no more just the function of the subject and target security ID's, but as well, the function of the security node ID.

```
Access = Function
(SSnID,SSID,TSnID,TSID)
```

An important part of the distributed system is the network, which spans the nodes of the cluster. To apply the access control functions in the cluster, there must be a way to pass the security parameters between the nodes in a transparent fashion. In our research, we try to find the appropriate architecture for this problem as well.

Our prototype is based on a cluster of Linux machines and the implementation of the mandatory access control will by exercised in the Linux kernel. By implementing the mandatory access control inside the kernel, we can achieve security transparency in the system.

Another functionality of the access control is to be able to generate alarms in case of intrusion detection. When the security module detects the intrusion, an alarm notification will be passed to the security manager and later to the security server. Based on the severity of the alarm, the security server will take an action. An example of the action will be a change of the security node ID, loading a new security policy, or declaring a node compromised and disconnecting it from the cluster. In the most severe case, the security server may ask ACS to block all accesses except the security path of the security manager.

4 DSI Security Module

Our security enforcement software for Linux is built as a Linux module and works in the kernel space. We based our development on the Linux Security Module (LSM) infrastructure (security hooks) introduced in the Linux kernel.

LSM framework does not supply any additional security in the Linux kernel. It only provides the infrastructure to support the security development as Linux modules. The LSM kernel patch adds security fields to kernel data structures and inserts calls (called hooks) at special points in the kernel code to perform a module specific access control. LSM adds methods for registering and un-registering security modules, and a general security system call that allows the communication between user programs and the LSM for security aware applications. Each LSM hook is a function pointer in a global structure called `security_ops`. Because the hooks are embedded in the kernel and are called even before a security module is installed, this structure is initialized to a set of functions provided by a dummy security module. These functions are just placeholders for more useful security mechanisms that can be loaded as a Linux module. A `register_security` method is introduced to allow a security module to set its own security functions (to overlay the dummy functions). An `unregister_security` method is used to return to the dummy functions.

The LSM methods are organized into two categories:

- Hooks to handle the security fields
- Hooks to perform access control

We started the development with the kernel 2.4.17 [13] and the appropriate security patch (`lsm-full-2002_01_15-2.4.17.patch` [15]). The DSM module cannot act alone and rely on the services supplied by DSI. DSM only enforces the access control but the policy is decided by the DSI security server. The security server is responsible for giving the security policy to the security module. The security server (SS) is responsible to supply the security node ID to each node of the cluster as well. Sending the security node ID to the node of the cluster means that the node is part of the cluster from the security point of view and it can start the security operation. Before the security node ID is sent to the cluster node, all the security checks are disabled on this node.

DSM takes security decisions based on the security policy decided by the security server and the security identifiers (SID) assigned to each subject and resource. Security Identifiers are non-persistent and are meaningful only on the local node. The security server provides functionality for converting a SID to its corresponding security context. All the entities for which security is being enforced are divided into security classes. A security class is a distinct type of resource with a distinct set of legal operations, for example, a process, a file, etc.

When a security decision must be taken, the security IDs of a subject and a resource are extracted from their kernel representations and will be used for the security access decision. For efficiency, the security policy is represented in the kernel memory.

4.1 Labels

As already mentioned, all the subjects and resources must be labeled. Since the security module can be loaded run-time, we distinguish two modes of subjects labeling. Before the module is loaded there are no labels attached to any subject or resource in the system. At the module initialization time, all the running tasks are scanned and the labels are attached to them. When a new process is created after the security module is loaded, the security hooks are used to do the labeling.

Because Linux stores the process descriptor and the Kernel Mode process stack in a single 8KB memory area, we can use this fact and avoid allocating memory for labeling the subjects (Figure 6). The other labels are attached to the resources run-time, which implies that the module checks if the label is there. If the

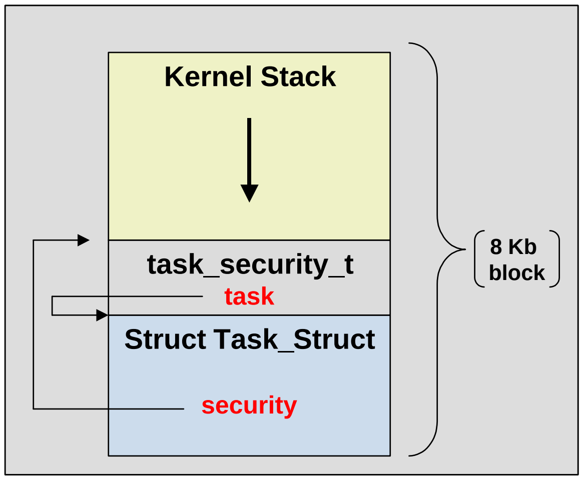


Figure 6: Task Label

label is not attached, a new label will be created.

4.2 Network Labels

Because the access in the cluster can be performed from a subject located on one node to the resource located on another, there is a need to control such accesses as well.

When a process on one node makes an access to a resource on another node, first the local access to the communications resources (socket, network interface) is checked. When the local access is granted then the message can be sent to the remote location. In order to identify the sending subject, the Security Node ID (security node identifier) and the Security ID of the subject (security subject identifier) are added to the message. For the purpose of this exercise, we use the IP protocol for the security information transfer. A new option is added after the IP header based on the hooks in the IP protocol stack. On the receiving side, these two information (Security Node ID and Security SID) are extracted (based on the hooks in the IP stack) and are used to build the network security ID (NSID).

```
NSID = Function (SnID, SID)
```

This function is specified by the security server in form of the conversion table. The receiving side looks up into the table by specifying SnID and SID and extract the Security Network ID. Now the security network ID can be used as a local label to all the access controls.

For instance, a client process tries to access a server's process. The client node does not know what is the security of the server node, so it can only perform access control checks based on the security attributes of the communications resources (sockets, network interface). The server node can perform access control checks based on the security attributes of the client process, the source node, and the server process. When a process attempts to accept a connection or receive a packet, if the policy prohibits the server process from receiving data from the client process, the connection or the packet is dropped and alarm is generated.

4.3 Implementation status

We are in the process of building the working prototype of the cluster security infrastructure. The kernel module has been implemented where the subjects (tasks) have been labeled. The local access to the communication objects (sockets) has been implemented and we are currently working on the remote access implementation.

In the current implementation, the security information is added to the IP message after the IP header as an option. There is no implementation of the interface to other parts of the distributed security architecture. The actions of the security server are simulated by the user mode programs (load policy, load security node ID). The alarms generated by the distributed security module are sent to the special user mode program as well. The current implementation is not optimized for the performance and it is built in order to check the overall logic of the cluster security.

5 Performance Challenges

Enforcing security does not come free; there is always a performance price to pay. At the same time, an over secured system is almost unusable; therefore the security introduced to the system must be properly balanced. This section discusses the impact of the security implementation on the overall performance of the system.

We performed testing for three different kernel configurations: the first testing was done with kernel 2.4.17; the second was done with the same kernel and our security module loaded plus the IP packet modification; the third was done with the same kernel and the security module but without IP packet modification. These tests were executed on a Pentium III 650 MHz Dell laptop with 265 MB RAM.

5.1 Test Types

We performed three types of testing: process creation with fork, UDP local access, and UDP remote access. The purpose of the testing was to get a preliminary performance evaluation of the security module, to answer the question of how much performance we lose when adding extra security features. The UDP tests were performed with and without IP packet modification in order to see how much performance was lost during IP packet modification. In the following subsection, we explain the testing procedure per testing type.

`Process Creation Testing`

This test measures the time a process can fork a child that immediately exits. The parent process loops 100,000 performing fork and wait calls. The test was performed 5 times and the average was calculated. Later the average time of the single loop (fork, wait) was calculated.

`UDP Local Access Testing`

The UDP Local Access test measures the time needed by a process to send a UDP message. This test sends 500,000 UDP messages in a loop. The test was performed 5 times and the average was calculated. Later the average time of the single loop (send) was calculated. The sending process does not check if the message was sent outside the node; in addition, it does not wait for the confirmation. In this case, it is not important whether the server has DSM installed or not.

`UDP Remote Access Testing`

The UDP Remote Access test measures the time needed by a process to send a UDP messages and receive a UDP response from a server. The client process will send a new message after receiving the confirmation from the server. It is important, in this case, that the server runs the DSM software for the permission to be checked on the receiving side. In this test, the second server is a Pentium II 300 MHz desktop with 128 MB RAM. This test sends and receives 100,000 UDP messages in a loop. The test was performed 5 times and the average was calculated. Later the average time of the single loop (send, recv) was calculated.

5.2 Test Results and Interpretation

Based on the testing performed, we present the results in Table1 and 2. All numbers are in microseconds.

`Process Creation Testing Results`

The average fork test with kernel 2.4.17 and the DSM module was completed in 167 microsec-

	Linux 2.4.17	Linux 2.4.17 with DSM	Overhead %
Fork	167	169	+1.20%
UDP Local Access (Send Message)	16.388	19.7	+20%
UDP Remote Access (Loopback)	133.44	173.88	+30%

Table 1: Performance Analysis with IP packet modification

	Linux 2.4.17	Linux 2.4.17 with DSM	Overhead %
UDP Local Access (Send Message)	16.388	17.084	+4.2%
UDP Remote Access (Loopback)	133.44	140.64	+5.4%

Table 2: Performance Analysis without IP packet modification

onds, compared to 169 microseconds with kernel 2.4.17 without the DSM module. As a result, we have a 1.2% increase as overhead. This is because the system had to perform a permission check on the fork operation and to spend some extra time on labeling of the child process.

UDP Local Access Testing Results

In this case, the average overhead for the setting with DSM module against the setting without the DSM module is 20%. This overhead consists of performing permission check on the socket send message and `sk_buff` label attachment for each message sent plus the labeling of IP messages. When the IP packet modification is disabled (Table 2) the overhead drops to 4.2%. As we can see most of the overhead is related to IP packet modification. Only a small fraction of the overhead is caused by the security module.

UDP Remote Access Testing Results

In this case, the average overhead for the setting with DSM module against the setting without the DSM module is 30%. The overhead consists of the following:

- Performing a permission check on the send socket side,
- Attaching a label to `sk_buff`,
- Attaching the security information to the IP message,
- Retrieving the security information on the receive side,
- Attaching the network security ID to `sk_buff`,
- Performing the permission checking on `sk_buff`,

- Performing the security checking on the socket, and,

- Repeating all the above operations on the return message.

When the IP packet modification is disabled (Table 2), the overhead drops to 5.4%. As we can see most of the overhead is related to IP packet modification. Only a small fraction of the overhead is caused by the security module.

5.3 Discussion

One of the most frequently asked questions is how adding security mechanisms will affect the performance of the system. Based on the testing results (Table 1), the percentage overhead for some operations, such as the UDP remote access, is considerable. The simple test, like fork, has relatively small overhead because there is only one security check. Nevertheless, some more complicated tests, like loopback, have high overhead because the security is checked in many points on the way of the traveling message. As it is shown in Table 2, the most of the overhead is added by the IP packet modification.

These results must be regarded as an upper case of the performance because no single security operation has been optimized. Nevertheless, the results demonstrated the challenges facing the development of efficient distributed security.

We believe that after optimizing the implementation, we will decrease the percentage overhead significantly.

6 Conclusion

6.1 Lessons learned

One of our objectives was to prototype a distributed security module for Linux clusters. During the process, we acquired a lot of competence in the area of Linux kernel internals, which allowed us to set up the task security structure without memory allocation.

It is always important to divide complex problems into smaller parts in order to simplify the solution. In our case, we approached the problem of distributed access control in the way that we tried to answer three important questions:

1. How to perform the local access control?
2. How to perform the remote access control?
3. How to transfer the security information from one node to another in a transparent way?

While building the first prototype, we managed to crash the kernel many times. We realized that the swapper task (task 0) is not on the `for_each_task` list and has to be handled separately.

One of the lessons was that the system could not be over secured because it becomes unusable. By loading a very strict policy, we were not able to interact with the operating system up to the point where we had to reboot the system.

6.2 Final remarks

We were able to achieve our first goal of building the framework of the mandatory access control for Linux cluster. The security checks can be performed on the subjects and resources located on the same (local access) and different nodes (remote access) of the cluster.

We tested the framework with buffer overflow attacks and it proved that the current solution could guard against these types of attacks.

We continue to work implementing the new functionality in DSM for Linux clusters. In addition, we are in the process on building a benchmarking environment (Security Evaluation Lab) that is capable of testing the performance and the resistance of the system against various possible attacks such as denial of service attacks.

The distributed security module (DSM) is an integral part of the distributed security infrastructure (DSI). It relies on the services of DSI and provides access control services to DSI. The development of DSM and DSI are ongoing at full speed.

In the short term, we plan to implement interfaces between some services of DSI and DSM. One of the examples could be the interface between the security manager on a node and the DSM. This interface will be used to load a new policy and to pass a new node security node ID downloaded by the security manage to a node. In addition, we plan to introduce the mechanisms to pass alarms from DSM to SM and later to SS.

7 Future Work

We are in the early stage of the prototyping and in the first stage of building the mandatory access control for Linux clusters. Our first goal is to prototype the framework of the distributed access control to check the logic of the distributed access.

Based on the limited functionality (socket level network access), we plan to exercise the server security as a function of the received connection (traffic) from the clients with different security ID's. When a server accesses resources on the local node the access control does not know whether the access is a local access or is performed on behalf of a remote client. In this case, there must be a change of the server access rights based on the clients connected to it.

In the current implementation, the security information sent on the network is not protected; they can be sniffed and used in possible attacks. Our next objective is to securely transmit this information without any performance degradation.

The security information is attached as options to the IP packet. Because the IP protocol is relatively high level, there is a need to implement this feature on lower levels of the network stack.

One of our next steps is to investigate the relationships when a subject or a resource is outside the cluster. Since we are at an early prototype phase, the performance optimization is not done yet. Therefore, improving the performance of the secure system is the next challenge.

Finally, we plan to test our security mechanisms built into the servers of the cluster through generating different types of attacks to verify how the new security mechanisms can improve the overall system security.

Acknowledgments

Ibrahim Haddad, Ericsson Research Canada, for commenting and reviewing this paper. David Gordon, Ericsson Research Canada, for contributing to the IP options implementation and buffer overflow exercise.

References

[1] A. Chitturi "Implementing Mandatory Network Security in a Policy-Flexible System," Masters Thesis, University of Utah, June 1998.

[2] P. Loscocco, S. Smallay "Integrating Flexible Support for Security Policies into Linux Operating System" Technical Report, NSA and NAI Labs, Oct 2000.

[3] P. Loscocco, S. Smallay, P.A. Muckelbauer, R.C. Taylor, S.J. Turner, J.F. Farrell "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments" In *Proceeding of the 21st National Information Systems Security Conference*, Oct 1998.

[4] P. Loscocco, S. Smallay "Meeting Critical Security Objectives with Security Enhanced Linux" Technical Report, NSA and NAI Labs, Oct 2000.

[5] R. Spencer, P. Loscocco, S. Smallay, M. Hibler, D. Andersen, J. Lepreau "The Flask Architecture: System Support for Diverse Security Policies," NSA, SCC, University of Utah.

[6] M. Dagenais, I. Haddad, C. Levert, M. Pourzandi, M Zakrzewski "A New Architecture for Security in Carrier Class Clusters," Apr. 2002.

[7] G. Nutt *Kernel Projects for Linux*, Addison Wesley Longman, 2001.

[8] A. Rubini, J. Corbet *Linux Device Drivers*, O'Reilly, 2001, Second Edition.

[9] M. Beck, H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner *Linux Kernel Internals*, Addison Wesley Longman, 1998, Second Edition

[10] D.P. Bovet, M. Cesati *Understanding the Linux Kernel*, O'Reilly, 2001, First Edition.

[11] Buffer Overflow, `http://www.insecure.org/stf/smashstack.txt`

[12] Flask Architecture, `http://www.cs.utah.edu/flux/fluke/html/flask.html`

[13] Linux Kernel, `http://www.kernel.org`

[14] Linux Kernel Module Programming, `http://metalab.unc.edu/mdw/LDP/lkmpg/mpg.html`

[15] LSM Patches to Kernel, `http://lsm.immunix.org`

[16] Network Patch (selopt), `http://www.intercode.com.au/jmorris/selopt/old/`

[17] SELinux, `http://www.nsa.org/selinux`

Glossary

ACS	Access Control Service
DSI	Distributed Security Architecture
DSM	Distributed Security Module
LSM	Linux Security Module
NSID	Network Security ID
SCC	Secure Communication Channel
SM	Security Manager
SnID	Security Node ID
SS	Security Server
SSID	Source Security ID

SSnID	Source Security Node ID
TSID	Target Security ID
TSnID	Target Security Node ID