NPTL Optimization for Lightweight Embedded Devices

2011 Linux Symposium

Geunsik Lim
Samsung Electronics
geunsik.lim@samsung.com

Hyun-Jin Choi
Samsung Electronics
hj89.choi@samsung.com

Sang-Bum Suh
Samsung Electronics
sbuk.suh@samsung.com

Abstract

One of the main changes included in the current Linux kernel is that, Linux thread model is transferred from LinuxThread to NPTL[10] for scalability and high performance. Each thread of user-space allocates one thread (1:1 mapping model) as a kernel for each thread's fast creation and termination. The management and scheduling of each thread within a single process is to take advantage of a multiple processor hardware. The direct management by the kernel thread can be scheduled by each thread. Each thread in a multi-processor system will be able to run simultaneously on a different CPU. In addition, the system service while blocked will not be delayed. In other words, even if one thread calls blocking a system call, another thread is not blocked.

However, NPTL made features on Linux 2.6 to optimize a server and a desktop against Linux 2.4 dramatically. However, embedded systems are extremely limited on physical resources of the CPU and Memory such as DTV, Mobile phone. Some absences of effective and suitable features for embedded environments needs to be improved to NPTL. For example, the thread's stack size, enforced / arbitrary thread priority manipulation in non-preemptive kernel, thread naming to interpret their essential role, and so on.

In this paper, a lightweight NPTL (Native POSIX Threads Library) that runs effectively on embedded systems, for the purpose of a way to optimize is described.

1 Introduction

Generally speaking, most existing embedded environments have been designed and developed for specific purposes, such as Mobile phone, Camcorder, DTV. However, in order to satisfy customer's diverse needs, recent embedded products are required to become smarter.

Many customers want to efficiently use embedded products in their life. To fulfill these needs, most embedded products offer their own app-stores. Customers download from these app-stores and install programs they need in their daily life. With the appearance of these app-stores, the number of applications for embedded products is dramatically increasing.

In mobile environments, where CPU and memory[12] resources are limited compared to desktop or server environments, the increase of applications make it more important to design and develop an efficient system without any resource expansion.

The higher performance hardware incurs higher cost in manufacturing; therefore, most companies prefer lower cost hardware solutions if they can achieve safe and reasonably fast run-time execution environment through efficient supports from underlying platform itself. Companies equipped with these solutions will have a big advantage in business especially in terms of cost competitiveness.

The performance and degree of integration of hardware improves every year. However, designing and developing optimized software especially for lightweight embedded environments is still a hard goal to achieve. Recently, the number of processes or threads running in user-space of Linux-based embedded products has reached at least 200 and sometimes exceeds 700.

For comparison, Microsoft Windows 7 running in desktop environment with a high performance CPU and high capacity memory, on average has more than 700 threads created and managed. These processes and threads have process condition cycles such as running and waiting depending on the conditions given.

Considering this, we may guess that the number of processes or threads running in embedded products with lower hardware specification is non-negligible, i.e. large enough to require an efficient solution using minimum memory footprint and achieving optimized performance.

The NPTL 1:1 thread mapping model, which appeared in Linux Kernel 2.6, dramatically improved the limits of scalability and performance as compatered with the existing LinuxThread. NPTL[1] [3] was originally provided as an alternative option for LinuxThread for compatibility, but with the popularity of NPTL model, recent Linux distribution releases only support NPTL, which obsoletes thepierce augments on the useless of NPTL model.

The NPTL model adapted in released versions of Debian/Ubuntu, Fedora/RHEL, and openSUSE need improvements in order to support lightweight development environments. This paper tackles this issue and proposes many ways of achieving the improvements.

2 Thread stack size for embedded system

In implementing multi-programs through user-space threads, the stack size of a single thread is directly proportional to the maximum number of threads which a developer can produce. When a thread is created, stacks in shared memory region are allocated to the thread.

In general, the basic stack size in latest Linux distributions is 10 MB per thread. The stack size of a thread has been increased from 2MB to 8MB and now 10MB to avoid stack overflow problems at various Linux distributions such as Ubuntu, Fedora, openSUSE.

Due to low power management and cost competitiveness, most mobile embedded systems provide limited physical resources. Moreover, the swap device, which is normally used to overcome physical memory shortage, is not supported in most embedded environments.

Therefore, in order to implement efficient applications, developers should obey good thread programming styles and find an optimal stack size for threaded applications. In this way, we can implement an efficient and economic system without additional hardware support.

Generally speaking, two main reasons of system panic caused by user-space thread libraries are 1) OS bugs, and 2) page segment fault error by abnormal program operations. In application developers' point of view, OS bugs are difficult to control, but page segment fault error by abnormal program can be avoided through the adjustment of stack size to the right size.

Other than memory expansion and swap space support, there are three software approaches to solve system problems effectively.

- Increase or decrease the available stack size with ulimit

  Check and adjust the stack size through ulimit command in running shell. Adjustments made with the ulimit command only affect the shell where the ulimit command is given and its children. Therefore, in order to alter the setting globally in the system, the ulimit command should be executed at the end of booting cycle (eg. `/etc/rc.sysinit`).

- Define individual stack size of a user-space thread (pthread_attr_)

  Thread attribute value that application developer defined must be initialized by pthread_attr_init(), when developer want to define a stack size using POSIX standards for multi-thread programming. Developers call pthread_attr_getstacksize() library function to determine the stack size, and they can dynamically control the stack size of a thread with pthread_attr_setstacksize() library function. Stack size may also be set when creating threads using the pthread_create() library function.

- Establish appropriate stack size policy for all threads in NPTL layer

  Many application developers use pthread_attr_getstacksize() with t_attr.__stacksize variable which is defined differently for each architecture internally. In the NPTL library, t_attr.__stacksize variable takes the value assigned to the thread's stack size. If this value is not set, the system set by default stack size ulimit command brings.

Figure 1 shows the internal operation flow of user-space thread created through any process under NPTL environment. We have to decide the default stack size of embedded system that want to select as a default stack size value about all threads that are created in specified

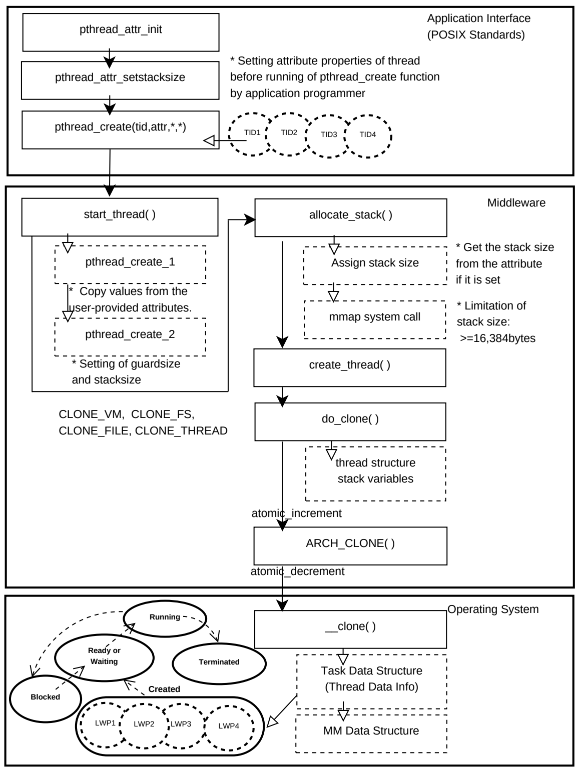
Application Interface
(POSIX Standards)
pthread_attr_init
pthread_attr_setstacksize
* Setting attribute properties of thread before running of pthread_create function by application programmer
pthread_create(tid,attr,*,*)
TID1
TID2
TID3
TID4
Middleware
start_thread( )
allocate_stack( )
pthread_create_1
* Copy values from the user-provided attributes.
pthread_create_2
* Setting of guardsize and stacksize
Assign stack size
* Get the stack size from the attribute if it is set
mmap system call
* Limitation of stack size: >=16,384bytes
create_thread( )
CLONE_VM, CLONE_FS, CLONE_FILE, CLONE_THREAD
do_clone( )
thread structure stack variables
atomic_increment
ARCH_CLONE( )
atomic_decrement
Operating System
Running
Ready or Waiting
Terminated
Blocked
Created
LWP1
LWP2
LWP3
LWP4
__clone( )
Task Data Structure (Thread Data Info)
MM Data Structure

Figure 1: Internal operation flow of thread

embedded system. Through it, we can manage consistent policy that adjust default stack size of thread at the middleware level automatically.

Since most threads created under embedded environment run light operations, it is typical to use stack sizes much less than the 10MB default used on servers. Thread stack size of embedded application is less than 1MB in general. Considering the minimum memory space of data structure for the creation of threads, Linux-based embedded system needs the stack size of more than 16,384 bytes per thread essentially. The stack size of 16,384 bytes includes guard size of 4,096 bytes during memory mapping practically. In other words, to prevent stack overflow, deduction 4,096 bytes from stack size by using pthread_attr_setstacksize() function is allocated to memory space for guard size as below.

```
4003b000 1127K r-x-- /lib/libc-2.12.so
40139000    8K ----- /lib/libc-2.12.so
4013b000   20K rw--- /lib/libc-2.12.so
40140000    8K r---- /lib/libc-2.12.so
40142000    8K rw--- /lib/libc-2.12.so
40144000    8K rw---   [ anon ]
40146000    4K -----   [ anon ]
40147000   12K rwx--   [ anon ]
402bf000    4K -----   [ anon ]
402c0000   12K rwx--   [ anon ]
402c3000    4K -----   [ anon ]
402c4000   12K rwx--   [ anon ]
 ..... remainder omitted ......
```

To decide default stack size in embedded system, we have to profile the distribution map of stack size that all threads are utilizing in embedded system. Otherwise, *Run-time error* may occur when application developer try to use stack size at run-time after reduce stack size of thread remarkably more than default stack size for normal operation.

When application developers compile threaded application source, they can compile entire source without any error problems. However, a segmentation fault error can occur when stack usage exceeds the size defined during running ELF binary code compiled like above.

In contrast, if the application allocates too large stack size for new thread using pthread_attr_setstacksize() function, then new thread creation will sometimes fail. As a result, system will return error code (number is 12) for the call to run pthread_create() function, and thereafter will not be able to create more threads. Therefore, in spite of Linux system different from uCLinux can support improved Virtual memory Management, it needs attention to allow a large stack size thoughtlessly under embedded environment.

3 Thread naming

It is very hard to identify the purpose of each thread when there hundreds of them in an embedded system. Optimization of user-space functions and SQA(Software Quality Assurance) is essential process before mass production for product competitiveness and reliability after complete entire development process using limited hardware resources.

If it is possible to monitor the purpose of individual thread for hundreds of threads run in embedded system at this time, we can improve development productivity rapidly by finding the part of thread's abnormal running and debugging and optimizing it. It is possible to distinguish main role of relevant child threads using the name of thread function set as the 3rd parameter during pthread_create() function call. Alternatively, the application developer may obtain a unique value for each thread with Thread Local Storage (TLS)[11] that is supported by CPU and cross-compiler. This value is used for the purpose of identifying which thread runs a specified function at some point.

However it is difficult to know the unique purpose of each thread executed by this method, when there are hundreds of threads calling the same function by using pthread_create() function. If we expand and support the "Thread Naming" interface at middleware layer, or user-space thread model like NPTL, it would be easy to understand the operational purpose of all threads in the platform.

In a large scale project, many teams participate and cowork to develop, and often cause bad effect by creating another task to check and understand the operational purpose of threads created by other teams. When several developers in each team produce thread using additional support like pthread_set_naming_np() and pthread_get_naming_np() for embedded system, the team to improve performance can analyze the detail information of all threads produced obviously by defining the role of additional thread by using pthread_set_naming_np() library call.

Table 1 below shows the sample output including the role of threads. When threaded applications using GDB (GNU Debugger) use named threads, system programmers can distinguish each other. Readability can be improved by understanding the detail flow of thread to analyze the purpose of creating, sleeping and finishing of each thread.

4 Thread profiling

We can find the optimal time point for booting the system by profiling the time interval between ahead thread and back thread and CPU share of all threads created before the initial GUI screen of embedded platform appears.

For example, when the share of CPU for specific region is low during the system booting, we can maximize the CPU usage and shorten the system boot time by separating independent functions as threads, and by moving some threads. If the generation of thread occurred long after specific thread's execution, Analyze CPU usage of thread executed for long time in detail. If CPU usage were not high, the research for system optimization would be possible. Despite high CPU usage during embedded system booting, if relevant scheduling work could be possible after initial screen appears, it would be effective to run those threads after initial screen appearance.

Table 2 shows information like Stack Size, Guard Size, Priority Value, Start Time, Time Gap which can be used to optimize system boot time. Performance optimization is possible by debugging internal operation information of user-space functions and identifying naming information of relevant threads because thread number 168 is executed almost for 1 second in the table below.

5 Extension of pthread_{set|get}_priority_np interface

When Linux kernel based on 2.6 version uses system call and library call, it consists of total 140 priorities with normal priority level using nice value from -20 to 19 and real-time priority level from 0 $\sim$ 99. Low number have high priority in Linux kernel-space.

Normal priority is defined in the file of `kernel/sched.c` and Linux kernel schedule this tasks with O(1) scheduler or CFS scheduler (since 2.6.23)[4]. Depending on Linux kernel version, after allocate one normal priority between bitmap 100 and bitmap 139 about a nice value between -20 and 19 by user-space application developers.

User-space real-time support and a few challenges for 100% POSIX compliance was written in the section "8. Remaining Challenges" in *Native POSIX Threads Library for Linux*[10] paper by Ulrich Drepper.

Infrastructure for POSIX compatible real-time support for user-space was improved by adding features such as Priority Queuing, Robust Mutex (`RT_MUTEX`)[8] and Priority Inheritance[5] [7] [9] to Linux and Glibc. This means application developer can realize real-time threaded programming in user-space. Table 3 shows the system call and library call for setting scheduling priority against the process/thread with normal priority and the process/thread with real-time priority.

The scheduling priority of an already-running normal priority thread can be changed by a system call like setpriority(), nice().

In case of tasks having real-time priority value, there are possible values for scheduling policy like `SCHED_RR`

STACK-ADDR (hex)	*PID* (process)	*TID* (thread)	*Thread Naming* (thread's role)	*Stack Size* (kbytes)	*Memory(RSS)* (kbytes)
[0x40a1b460]	[339]	[342]	Files Copy Extension	64	910
[0x40a2b460]	[339]	[343]	LifeCycle Controller	64	4,211
[0x40a3b460]	[339]	[344]	Micom Task	64	200
[0x40a4b460]	[339]	[345]	Event Dispatcher	64	355
[0x40a5b460]	[339]	[346]	Device Node Manager	64	305
[0x40a6b460]	[339]	[347]	Micom Task Extension #1	64	200
[0x40aeb460]	[339]	[348]	Media API	512	5,442
[0x412a8460]	[339]	[350]	Message Event Handler	64	34
[0x41328460]	[339]	[351]	Drawing Process	512	204
[0x41338460]	[339]	[352]	FlashTimerTask	64	382
[0x41ac5460]	[339]	[353]	UI Manager	512	566
[0x41ad5460]	[339]	[354]	Multi IPI Recovery	64	705
[0x41267460]	[339]	[355]	System Process	64	9,376
[0x41cbc460]	[339]	[356]	MediaCaptureComponent	64	202
[0x41ccc460]	[339]	[357]	MP4 MediaPlayer	512	7,109
[0x41cdc460]	[339]	[358]	JPEG Component	64	153
[0x41cec460]	[339]	[359]	Media Component	64	776
[0x41cfc460]	[339]	[360]	Async I/O	64	371
[0x41cdc460]	[339]	[361]	Video Output Component	64	2,221
[0x41cec460]	[339]	[362]	Service Manager	64	460
-	-	...	below omission	...	-

Table 1: Thread naming information of each user-space thread

(real-time round-robin policy), `SCHED_FIFO` (real-time FIFO policy), `SCHED_OTHER` (for regular non-real-time scheduling) and so on. The scheduling priority in user-space can be set from 1 to 99 for real-time scheduling policy. Therefore, the priority of normal non-real-time threads is counted as 0.

Considered real-time property under embedded environment `SCHED_RR` seems ideal, `SCHED_FIFO` is more useful to take advantage of performance practically because simple policy is good for performance and effective for management.

```
struct sched_param {
  . . . . . .
  int sched_priority;
  . . . . . .
};
```

Table 4 shows number of gettid() system call according to architecture.

Because the use of gettid() is CPU-architecture dependent in Linux kernel 2.6, system call number varies different among CPU architectures. You may utilize gettid() system call with the following definition because of non-implementation of gettid() in Linux system.

```
/* Using gettid syscall in user-space */
#define gettid() syscall(__NR_gettid)
```

We use gettid() instead of getpid() in NPTL thread model to find out the unique number of thread executed in the related function region to apply normal priority to threads are created as nice value. The gettid() function has to be made using syscall(__NR_gettid). And then, the use of gettid() function is available to utilize gettid() function by syscall(__NR_gettid) in the function of relevant thread.

Above _syscall() function returns kernel-space thread id that mapped about user-space thread id that is running by calling `include/asm-arm/unistd.h` header file. The gettid() system call is defined as follows in the file `kernel/timer.c`.

PID (process)	*TID* (thread)	*StackSZ* (byte)	*GuardSZ* (byte)	*Priority* (nice)	*StartTime*	*Time Gap* (msec)
160	162	262,144	4,096	5	1792015420	195
160	163	262,144	4,096	0	1792015423	3
160	164	262,144	4,096	0	1792015551	128
160	165	262,144	4,096	5	1792015666	115
160	166	262,144	4,096	5	1792015668	2
160	167	262,144	4,096	5	1792015670	2
160	168	262,144	4,096	5	1792016634	977
160	169	262,144	4,096	10	1792016637	3
160	170	262,144	4,096	5	1792016781	144
160	171	262,144	4,096	5	1792016783	2
160	172	262,144	4,096	5	1792016786	3
160	173	262,144	4,096	5	1792016788	2
160	174	262,144	4,096	-5	1792016873	85
160	175	262,144	4,096	-5	1792016874	1
160	176	262,144	4,096	5	1792016876	2
160	177	262,144	4,096	5	1792016878	2
160	178	262,144	4,096	5	1792016880	2
160	179	262,144	4,096	5	1792016917	37
160	180	262,144	4,096	5	1792016920	3
160	181	262,144	4,096	5	1792016922	2
160	182	262,144	4,096	-15	1792016925	3
160	183	262,144	4,096	5	1792017258	333
160	184	262,144	4,096	5	1792017260	2
160	185	262,144	4,096	5	1792017262	2
160	186	262,144	4,096	0	1792017264	2
160	187	262,144	4,096	0	1792017266	2
160	188	262,144	4,096	5	1792017284	18
-	-	...	below omission	...	-	-

Table 2: Thread profiling result

```
/* gettid syscall details in Linux */
asmlinkage long sys_gettid(void){
  return current->pid;
}
```

When you try to utilize gettid() system call using above method, it is recommended to add thread library function including system calls after considering the impact of embedded system's performance because of the cost of system calls. It is very useful to measure execution time, calls and errors for system calls of thread library function to be added to know the cost of CPU usage. The file `arch/arm/kernel/call.S` of the ARM Architecture defines sys_set_thread_aread() as sys_ni_syscall (224) and sys_get_thread_area as sys_ni_syscall (225).

Maintenance of source code of large scale project can be simplified by avoiding having many different functions preferred by developer in embedded platform. Instead a uniform common interface can be obtained by extending thread function of pthread_set_priority_np() or pthread_get_priority_np() additionally for the application developer to get ID value of a thread easily.

POSIX compatibility is very important in the view of standardization, but considering the characteristics of embedded platform, when we need additional thread API, application developers are able to know the extended thread API by making the name of function with the format of _np() to express "Non Portable" meaning

Scheduling Priority	*PID/TID*	*Function Name* (API)	*Call Interface* (classification)	*LinuxThread* (interface)	*NPTL* (interface)
Normal priority (from -20 to 19)	Process	setpriority() nice()	System call	getpid()	gettid()
	Thread	setpriority() nice()	System call	getpid()	gettid()
real-time priority (from 1 to 99)	Process	sched_setscheduler() sched_setparam()	System call	getpid()	gettid()
	Thread	pthread_setschedprio() pthread_setschedparam()	Library call	getpid()	gettid()

Table 3: LinuxThread VS. NPTL scheduling system call comparison

Architecture	*File name*	*gettid() syscall number*
i386	./arch/i386/kernel/entry.S ./arch/x86/kernel/syscall_table_32.S	224
ARM	./arch/arm/kernel/call.S	224
MIPS	./arch/mips/kernel/scall32-o32.S ./arch/mips/kernel/scall64-932.S	4,222
PPC	./arch/ppc/kernel/misc.S	207
SH	./arch/sh/kernel/entry.S	224

Table 4: The number of gettid() per CPU architecture

in the end of function.

In addition, the extension of these additional common interface and unified programming specification maintain consistently application interface management of embedded system that is extended the scale of platform more and more. We can continue software update easily and rapidly for new features while preserving without modifying existing code.

6 Controlling CPU scheduling of {self|another} thread

By increasing the speed of user's application under embedded system environment at specific time, users often want to get shorten application's waiting time. The support of these mechanisms raise the flexibility of scheduling priority for CPU usage when threads need higher CPU usage at specific time. Effective throughput of applications are possible by grouping thread applications based on the importance of processing speed and response speed in embedded system having limited CPU performance.

When new threads according to *Task scheduling importance hierarchy* are created, we need the thread dealing mechanism to realize the way to give suitable scheduling priority value of thread. Table 5 shows explanation about task classification and task meaning according to the Task scheduling importance hierarchy table.

We can minimize user's waiting time for embedded devices by self-adjusting a thread's scheduling priority at specific time, or by changing another thread's normal priority at run-time dynamically with pthread_setschedparam() library call in Linux 2.6 based NPTL environment.

Improvement is important, keeping POSIX compatibility with additional thread APIs to give different priority to many threads produced in one process. This means that reuse of the existing source is possible continually.

The pseudo code below shows the implementation of NPTL Library to control scheduling priority arbitrarily or by force for user-space threads based on normal priority that are created on non-preemptive Linux kernel 2.6.

Hierarchy of scheduling priority	Description
Busy Task (Urgent)	Busy task means the threads in the top of screen which interact with user or which occupy CPU usage under processing CPU.
Foreground Task (Normal)	Foreground task is thread that appear in the screen of user's embedded device but doesn't have activity to be processed immediately.
Service Task (Support)	Service task is middleware level component which supply important functions for processing of application and thread that occupies service of system.
Background Task (Hidden)	Background task is thread that occupies activity not visible to user.
Idle Task (Unlimited)	Idle task is thread that doesn't occupy component of any active application in embedded system.

Table 5: Task scheduling importance hierarchy

```
int __pthread_setschedparam(tid,
    policy, param)
  pthread_t tid;
  int policy;
  const struct sched_param *param;
 {
/* To support priority,if use SCHED_FIFO,
 * SCHED_RR,display notification message
 * ( @/usr/include/linux/sched.h ) */

/* Default value is a normal priority */
struct pthread *pd=(struct pthread *)tid;

if (policy == SCHED_OTHER){
/* Scheduling priority of thread */
int which = PRIO_PROCESS ;

/* Handling of SCHED_OTHER priority */
if ( param->sched_priority < -20 &&
     param->sched_priority > 19 ){
   printf("ERR! Nice range:-20~19\n");
   return errno;
}
/* Getting LWP(thread id of kernel) to
 * change scheduling priority about
 * assigned thread id.
 */
if (setpriority(which,unique_kernel_tid(),
    param->sched_priority) ){
   perror("setpriority() Error.\n");
   result = errno;
}
```

Mentioned above, after improving scheduling-related thread function of NPTL library, the way described below can control thread application's scheduling actively to apply different scheduling priority to many threads which are created in one process in embedded system.

```
/*
 * @Description: arbitrarily & by force
 * thread scheduling for urgent threads
 * @thread variables:(pthread_t thread[max])
 * If you want to affect priority about each
 * thread in a process in Linux 2.6 + NPTL,
 * We recommend that you use SCHED_OTHER
 * policy based on priority scheduling.
 * Or,If you need time critical performance
 * about threads, use real-time SCHED_RR
 * using pthread_setschedparam( ) syscall.
 */

struct sched_param schedp;
/* priority number of between -20~19. */
int priority = -20 ;
memset(&schedp, 0, sizeof(schedp));
schedp.sched_priority = priority;

/* for controlling self thread */
pthread_setschedparam(pthread_self(),
SCHED_OTHER, &schedp)

/* for controlling another thread */
pthread_setschedparam(thread[i],
SCHED_OTHER, &schedp)
```

7 Optimization: configurability and building with -Os

Glibc library[6] including NPTL has a lot of features. However embedded system do not need all features of glibc library. For this reason, it takes compilation time of more than 40 minutes to build the entire glibc source on average at the high-end Linux development computer (e.g: Intel Core2 Quad 9400, RAM 2GB).

We can consider how to compile entire glibc source with same configuration structure and build process like the compilation of Linux kernel. By doing so, we can compile only the necessary software components among a lot of components of glibc source for embedded system. How to compile this method can be modular as a functional unit reduce a long compilation time at software development step. Through this method, we do not select a unnecessary shared object libraries (e.g: NSS, NIS, DNS, CIDN, Locales, Segfault, Crypt, NSS, Resolv, etc) for lightweight rootFS. As a result, hard disk space and memory footprint can be minimized through configurable build method.

If you try to compile glibc sources with -Os option without -O2 through gcc compiler's optimization option interface[2] at the configurable build system menu of NPTL library, shared object binary files can be minimized to fit in the embedded system environment.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

- -Os: Optimize for size. -Os enables all -O2 optimization that do not typically increase code size. It also performs further optimization designed to reduce code size.
- -O0: Reduce compilation time and make debugging produce the expected results. This is the default.
- -O1: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With -O, the compiler tries to reduce code size and execution time, without performing any optimization that take a great deal of compilation time.
- -O2: Optimize even more. GCC performs nearly all supported optimization that do not involve a space-speed trade-off. As compared to -O, this option increases both compilation time and the performance of the generated code.
- -O3: Optimize yet more. -O3 turns on all optimization specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize and -fipa-cp-clone options.

8 Further work

Two issues of the current approach need to be addressed for future research direction. First, the performance gain issues of the CFS scheduler-based embedded Linux system, and second, the performance trade off issues when there are tasks with real-time priorities.

At latest version of the Linux kernel, the existing O(1) scheduler was replaced by the CFS scheduler[4] in order to maximize fairness of running tasks. This replacement was applied after 2.6.23 version. Our proposed approach shows some performance issues in evaluating application responsiveness at the CFS scheduler-based embedded Linux system.

For example, the time slot gap generated by our approach in the CFS scheduler is usually smaller than that of O(1) scheduler. This results not much performance gain in the CFS scheduler. CFS scheduler merged in Linux 2.6.23 by Ingo Molnar is that nice value entered by the application developer was replaced from time slice table to weight table. This is possible through vruntime (virtual run-time) which schedules tasks fairly. If two nice values produce a small gap like 0, instead of a large gap like 10, it is not easy to produce any performance gain. Therefore, we need a new approach for task scheduling policy which produces a wide gap for better performance gain even in the CFS scheduler-based embedded Linux system.

Second, the proposed approach doesn't consider cases when some tasks have real-time priorities. Any task with real-time priority share CPU time with others, so we need to find a preemptive way to replace these tasks with more emergent tasks.

9 Conclusions

We have shown through examples that current NPTL thread model, which was introduced in Linux 2.6 for improvements of performance and scalability, has several limitations. These limitations sometimes cause users to wait several seconds tediously after they launch an application downloaded from app-stores.

In general, the nature of embedded system environment has physical conditions with limited CPU and memory. Therefore, the existing embedded systems using NPTL are needed to improve by operating lightly and speedily with the best technical methods.

We introduced several approaches of improving the current NPTL thread model: a suitable thread stack size for embedded environments, thread naming interface expansion for optimization, supports of thread profiling and debugging components to minimize the boot time of embedded platform, a thread priority management method according to scheduling importance of thread application, an arbitrary or enforced thread scheduling control policy to speed up user application processing, selective source compile methods using modular configuration structure for minimizing memory footprint, etc.

We also have shown that the improved NPTL thread model has better performance and fewer memory footprint compared to the current model.

Our results confirm that the existing NPTL thread model in Linux can be utilized in embedded system through the several improvement features. This provides cost effective development opportunity for embedded developers to start developing thread models for embedded systems through existing open source like Linux.

10 Acknowledgments

We would like to thank Wonsuk Lee for his valuable reviews and comments. We would like to thank my Samsung colleagues KyungIm Jung and Jae-Min Ryu for some suggestions and review of the article. We'd also like to thank all those members of the Linux kernel development community who have contributed patches.

References

[1] Sebastien DECUGIS. Nptl stabilization project(nptl tests and trace). In *Ottawa Linux Symposium*, 2005.

[2] Free Software Foundation(FSF). GCC Online Manual(Options that control optimization). http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options.

[3] Steven J. Hill. Native posix threads library (nptl) support for uclibc. In *Ottawa Linux Symposium*, 2006.

[4] Ingo Molnar. CFS Scheduler Design. http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.

[5] Ingo Molnar. PI-futex. http://lwn.net/Articles/102216/.

[6] Roland McGrath. GNU C Library(Glibc). http://www.gnu.org/software/libc/.

[7] Rusty Russell. Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[8] Steven Rostedt. RT-mutex subsystem with PI support. Linux kernel documentation: kernel/Documentation/{rt-mutex.txt|rt-mutex-design.txt}.

[9] Ulrich Drepper. Futexes Are Tricky. http://www.akkadia.org/drepper/futex.pdf.

[10] Ulrich Drepper. Native Posix Thread Library for Linux. http://people.redhat.com/drepper/nptl-design.pdf.

[11] Ulrich Drepper. [TLS]ELF Handler For Thread-Local Storage. http://people.redhat.com/drepper/tls.pdf.

[12] Ulrich Drepper. What Every Programmer Should Know About Memory. http://www.akkadia.org/drepper/cpumemory.pdf.