Sharing Page Tables in the Linux Kernel

Dave McCracken
IBM Linux Technology Center
Austin, TX
dmccr@us.ibm.com

Abstract

An ongoing barrier to scalability has been the amount of memory taken by page tables, especially when large numbers of tasks are mapping the same shared region. A solution for this problem is for those tasks to share a common set of page tables for those regions.

An additional benefit to implementing shared page tables is the ability to share all the page tables during *fork* in a copy-on-write fashion. This sharing speeds up *fork* immensely for large processes, especially given the increased overhead introduced by rmap.

This paper discusses my implementation of shared page tables. It covers the areas that are improved by sharing as well as its limitations. I will also cover the highlights of how shared page tables was implemented and discuss some of the remaining issues around it.

This implementation is based on an initial design and implementation done by Daniel Phillips last year and posted to the kernel mailing list. [DP]

1 Introduction

The Linux® memory management (MM) subsystem has excellent mechanisms for minimizing the duplication of identical data pages by sharing them between address spaces whenever possible. The MM subsystem currently does not, however, make any attempt to share the lowest layer of the page tables, even though these may also be identical between address spaces.

Detecting when these page tables may be shared, and setting up sharing for them is fairly straightforward. In the following sections the current MM data structures are described, followed by an explanation of how and when the page tables would be shared. Finally some issues in the implementation and some performance results are described.

2 Major Data Structures

To understand the issues addressed by shared page tables it is helpful to understand the structures that make up the Linux MM.

2.1 The *mm_struct* Structure

The anchor for the memory subsystem is the *mm_struct* structure. There is one *mm_struct* for each active user address space. All memory-related information for a task is found in the *mm_struct* or in structures it points to.

2.2 The *vm_area_struct* Structure

The address layout is contained in two parallel sets of structures. The logical layout is described by a set of structures called

*vm_area_struct*s, or *vma*s. There is one *vma* for each region of memory with the same file backing, permissions, and sharing characteristics. The *vma*s are split or merged as necessary in response to changes in the address space. There is a single chain of *vma*s attached to the *mm_struct* that describes the entire address space, sorted by address. This chain also can be collected into a tree for faster lookup.

Each *vma* contains information about a virtual address range with the same characteristics. This information includes the starting and ending virtual addresses of the range, and the file and offset into it if it is file-backed. The *vma* also includes a set of flags, which indicate whether the range is shared or private and whether it is writeable.

2.3 The Page Table

The address space page layout is described by the page table. The page table is a tree with three levels, the page directory (pgd), the page middle directory (pmd), and the page table entries (pte). Each level is an array of entries contained within a single physical page. The entry at each level contains the physical page number of the next level. The pte entries contain the physical page number of the data page.

For the architectures that use hardware page tables, the page table doubles as the hardware page table. This use of the page table puts constraints on the size and contents of the page table entries when they are in active use by the hardware.

2.4 The *address_space* Structure

In addition to the structures for each address space, there is a structure for each file-backed object called struct *address_space* (not to be confused with an address space, which is represented by an *mm_struct*). The *address_space* structure is the anchor for all mappings related to the file object. It contains links to every *vma* mapping this file, as well as links to every physical page containing data from the file.

All shared memory uses temporary files as backing store, so each shared memory *vma* is linked to an *address_space* structure. The only *vma*s not attached to an *address_space* are the ones describing the stack and the ones describing the bss space for the application and each shared library. These are called the 'anonymous' *vma*s and are backed directly by swap space.

2.5 The *page* Structure

All physical pages in the system have a structure that contains all the metadata associated with the page. This is called a "struct *page*." It contains flags that indicate the current usage of the page and pointers that are used to link the page into various lists. If the page contains data from a file, it also has a pointer to the *address_space* structure for that file and an index showing where in the file the data came from.

3 How Memory Gets Mapped

Memory areas are created, modified, and destroyed via one of the mapping calls. These calls can map new memory areas, change the protections on existing memory areas, and unmap memory areas.

3.1 Creating A Memory Mapping

New memory regions are created using the calls *shmget/shmmap* or *mmap*. Both calls create a mapped memory region, either backed by an open file passed as an argument or by a temporary file created for the purpose. The newly mapped memory region is represented by a new or modified *vma* that is attached to

the taskś *mm_struct* and to the *address_space* structure for the file. The page table is not touched during the mapping call.

Various flags are passed in when a page is mapped. These flags specify the characteristics of the mapped memory. Two of the key characteristics are whether the area is shared or private and whether the area is writeable or read-only. For read-only or shared areas, the file data is read from and written back to the file. Private writeable areas are treated specially in that while the data is read from the file, modified data is not written back to the file. The data is saved to swap space as an anonymous page if it needs to be paged out.

Pages are actually mapped when a task attempts to access a virtual address in the mapped area. The page fault code first finds the *vma* describing the area, then finds the *pte* entry that maps a data page for that address. A page is then found based on the information in the *vma*.

3.2 Locks and Locking

The MM subsystem primarily relies on three locks. Two locks are in the *mm_struct*. First is the *mmap_sem*, a read/write semaphore. This semaphore controls access to the *vma* chain within the *mm_struct*.

The second lock in the *mm_struct* is the *page_table_lock*. This spinlock controls access to the various levels of the page table.

The third lock is in the *address_space* structure. This lock controls the chain of *vma*s that map the file associated with that *address_space*. In 2.4, it is a spinlock and is called *i_shared_lock*. In 2.5, it was changed to a semaphore and is called *i_shared_sem*.

During a page fault the *mmap_sem* semaphore is taken for read at the beginning of the fault and is held until the fault is complete. The *page_table_lock* is taken early in the fault, but is released as necessary when the fault needs to block. Holding the *mmap_sem* semaphore for read allows other tasks with the same *mm_struct* to take page faults but not change their mappings.

4 Sharing the PTE Page

Normally the overhead taken up by page tables is small. On 32 bit architectures, there is typically a maximum of one pgd page, three pmd pages, and one pte page for every 512 or 1024 data pages. This ratio may be somewhat higher for sparsely populated address spaces, but virtual addresses are typically allocated in order so pte pages tend to be filled in fairly densely.

This ratio changes for shared memory areas. A shared memory region that covers an entire pte page may be shared among many address spaces. This sharing means there will be 1 pte page for each address space for every 512 or 1024 mapped data pages. Note that the pte pages, once allocated, are not freed even if the data pages have been paged out, so the pte page overhead is fixed even under memory pressure.

Shared memory is a common method for many applications to communicate among their processes. It is possible for a massively shared application to use over half of physical memory for its page tables.

Each address space in this scenario has an identical set of pte pages for its shared areas, with all its entries pointing to the same physical data pages. The premise behind shared page tables is to only allocate a single set of pte pages and set the pmd entry for each address space to point to it.

A beneficial side effect of sharing the pte page is once a data page has been faulted in by one

task, the page appears in the address space of all other tasks mapping that area. Without sharing, each task would have to take its own page fault to get access to that page.

There are clearly some constraints on when pte pages can be shared. The shared area must span the virtual memory mapped by the entire pte page. All address spaces must map the area at the same virtual address. While in theory the mapped areas only need to be aligned per pte page, the current implementation requires that the virtual addresses be the same.

4.1 Finding PTE Pages to Share

For sharing to work, it is necessary to find an existing pte page if one exists. Finding this page is accomplished at page fault time when there is not already a pte page for the faulting address.

First, the current *vma* is checked to see if it is eligible for sharing. It needs to be either shareable or read-only, and needs to span the entire address range for that pte page.

Next, the code searches for an existing pte page that can be shared. This search is done by going to the *address_space* for the current memory area, then walking its *vma* chain. Each *vma* is checked for compatibility. The *vma* needs to be at the same file offset and virtual address as the faulting *vma* and needs to also span the entire pte page. For each matching *vma*, its corresponding page table is checked for a pte page. If a pte page is found, it is installed in the pmd entry and its use count is incremented. While in theory it should be possible to share pte pages between any *vma*s whose mappings have the same pte page alignment, the current rmap implementation limits sharing to those that map to the same virtual address.

4.2 Copy On Write

On some architectures there is a second use for shared page tables. During *fork*, every pte entry is copied to the new page table. Data pages that can't be fully shared are marked as "copy on write." Marking a page as copy on write involves setting both the parent and the child pte entry to point to the same data page, but with write disabled. When either task tries to write to that page, the page is then duplicated and write access is enabled.

Some architectures (including x86) support disabling write access in the pmd entry, and interpret this to mean disabling write to all the data pages mapped by that entry. Disabling write allows the copy on write concept to be extended to shared page tables. Instead of copying each pte entry at *fork* time, each pmd entry is set to point to the same pte page and write access is disabled. When a write fault occurs, a new pte page is allocated and all the pte entries are copied in the same fashion as during *fork* in the non-shared version.

4.3 Locking Changes for Shared PTE Pages

When page tables have shared pte pages, the existing locking scheme becomes inadequate. The *page_table_lock* in the *mm_struct* can no longer protect the entire page table, since pte pages may be shared with other page tables.

There is a spinlock in the page struct that is normally used for *pte_chain*s in data pages. Since pte pages have no *pte_chain*s, the lock in the pte page's page struct can be used to control access to it. For pte pages this lock becomes the *pte_page_lock*. This change means the *page_table_lock* protects the pgd and pmd levels and the *pte_page_lock* protects the pte page.

Using this lock changes the locking protocol in

the page fault path. The *page_table_lock* must still be taken to until the pte page is found and selected. The *pte_page_lock* is taken for that page, at which point the *page_table_lock* can be released. The rest of fault resolution is done under the *pte_page_lock*.

5 Complications

While making pte pages shared seems like a simple task in theory, there are several things that complicated the task.

Part of *pte_chain*-based reverse mapping is a pointer in the pte page's *struct page* that points to the *mm_struct* the page belongs to. Sharing the pte page means it can belong to several *mm_struct*s. It was necessary to add an *mm_chain* structure which points to a chain of *mm_structs* that use the pte page.

There are various memory management-related system calls that can modify existing mappings. These include *mremap*, *mprotect*, *remap_file_pages*, and *mmap* itself. These calls can all change the mappings for an address space such that the pte page can no longer be shared. Each of those system calls was modified to identify shared pte pages and unshare them as necessary.

6 Performance

Shared page tables are primarily intended to reduce the space overhead of page tables, but there are some performance benefits, as well.

The primary performance gain is during *fork* because of copy-on-write sharing. Instead of duplicating a reference to each data page *fork* only needs to duplicate a reference to each pte page. This can improve *fork* performace by up to a factor of 10.

Fork speedup is balanced, however, by the cost of unsharing each pte page when one of its data pages is written to. Typically, three pte pages are unshared after every *fork* due to the user space layout. Since small programs generally only have three pte pages, only larger programs benefit from the improvement. In fact, the cost of sharing the pte, then unsharing it on page fault, has a small but measureable cost compared to copying. The simple solution to this is to copy the pte pages on *fork* if the address space only has 3 pte pages.

There is a corresponding performance improvement for *exit* and *exec* when they tear down the address space. Any pte pages that are shared can be detached simply by decrementing their reference count. For each pte page that is not shared, the code must examine each entry to determine what to do with its data page or swap page.

7 Conclusion

Shared page tables achieves its primary objective of dramatically improving the scalability of massively shared applications, as well as also improving the *fork* and *exit* performance of large tasks. While there is some cost in added complexity, the benefits far outweigh the cost.

Legal Statement

This paper represents the views of the author, and not the IBM Corporation.

IBM® is a registered trademark of International Business Machines Corporation.

Linux® is a registered trademark of Linus Torvalds.

Other company, product or service names may be the trademarks or service marks of others.

References

[DP] Daniel Phillips.
`http://nl.linux.org/phillips/page.table.sharing`

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.