Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Shared Page Tables Redux

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

Abstract

When a large memory region is shared each process currently maps it using its own page tables. When several processes map the same region the overhead for these page tables is significant. Shared page tables allows the processes to all use the same set of page tables for the shared region. This results in significant memory savings and performance gains.

In this paper I will discuss how page tables are shared, how the decision to share is made, the issues it introduces in the memory management subsystem, and what applications can benefit from it.

1 Introduction

The Linux memory subsystem goes to great lengths to share (minimize the duplication of) data pages in the system. There is almost always at most one copy of a given data page in memory at any time unless a process has made private modifications to it.

Linux does not, however, currently make any attempt to share the infrastructure needed to map those pages, even though for large mappings large parts of that infrastructure may be identical. The shared page table project is an attempt to add that sharing, with its concurrent reduction in memory overhead, with an associated improvement in performance. It also enables optimizations on some architectures that result in additional performance improvements.

2 A Brief History of Shared Page Tables

Sharing page tables is far from a new idea. In 2002 during 2.5 development the addition of *pte* chains for reverse mapping made *fork*, *exec*, and *exit* very slow. Sharing page tables and, in particular, doing a copy-on-write of even normally unshareable page tables looked like it might dramatically improve that performance. Daniel Phillips coded up a sample implementation that showed some promise.

In Fall of 2002 I started from Daniel's premise and wrote a complete implementation of page table sharing, including the copy-on-write behavior. What I discovered, however, was that the vast majority of programs only had three page table pages, all of which faulted and had to be copied immediately after fork. This resulted in a measureable performance penalty for all but large applications. This implementation was ultimately rejected for inclusion.

In 2003 and early 2004 the reverse mapping problem was revisited for 2.6, resulting in elimination of *pte* chains and the slow *fork*/*exec*/*exit*

problem. There was still an issue with the ballooning overhead of page tables with large applications that ran multiple processes all accessing large shared memory regions (primarily a characteristic of database applications).

In 2005 I once again addressed the issue of shared page tables. I dropped the concept of copy-on-write and concentrated on only sharing page tables for memory areas that are truly shareable. This eliminated a major source of complexity in the implementation.

3 Relevant VM Data Structures

There are several data structures that define the virtual memory subsystem. It is important to understand these structures and how they interact to also understand how page table sharing works.

3.1 The *mm_struct* Structure

The primary structure in the memory subsystem is the *mm_struct*, commonly called *mm*. There is one of these structures per virtual address space, *i.e.* one per process or collection of threads. The *mm* is the anchor for describing all memory connected to a process.

3.2 The *vm_area_struct* Structure

The next level of data structure is the *vm_area_struct*, commonly referenced to as the *vma*. The there is one *vma* for each mapped data area in the virtual memory. The collection of *vma*s is anchored to the *mm*.

Each *vma* includes the virtual starting and ending virtual address, a set of flags that describe the characteristics of the mapping, and a pointer to the backing file if there is one.

3.3 The Page Table

In parallel to the set of *vma*s is the page table, also anchored in the *mm*. The page table is a tree of arrays, each one physical page in size. Each element of the array points to the page containing an array for the level below, with the bottom level entries pointing to the data pages for the process.

There are four levels to the page table array, though some levels are dummied out on some architectures. The four levels are *pgd*, *pud*, *pmd*, and *pte*. On a three level architecture the *pud* is not present. On a two level architecture the *pmd* is also not present.

3.4 The *address_space* Structure

Another critical part of the memory subsystem is the *address_space* structure, commonly referred to as the *mapping*. This is not to be confused with the process's address space. There is one *mapping* for each open file. It contains a chain of all *vma*s that map the file plus a cache of all data pages in memory that come from the file. The *vma*s also contain a pointer back to the *mapping*.

3.5 The *page* Structure

The last important relevant data structure is the *page*. There is one *page* for each physical page in the system. It contains the major information on how the phsical page is being used as well as a pointer to the *mapping* it is a part of.

4 Types of Mapped Memory

When memory is mapped into a process it falls into one of several types. The key parameters

are read-only or read-write, shared or private, and file-backed or anonymous. Only some of these types are eligible to be shared. Anonymous mappings can never be shared. For file-backed memory all read-only mappings can be shared. for read-write memory only that marked shared can be shared.

4.1 File-backed *vs* Anonymous

The restriction against sharing anonymous memory is less restrictive than it sounds. When memory is mapped by an explicit mapping call from the process it is always file-backed, even that which the process marks anonymous. The memory subsystem uses a special file system to create a dummy file for such mappings. The only truly anonymous memory areas are the *bss* and the stack, created when the process is started. Both of these are inherently writeable and private, and thus are not shareable.

5 What is Shared

The specific parts of the infrastructure that can be shared are the lower levels of the page table. The granularity of sharing is the array in each physical page. When the mapped areas in two or more processes span an entire physical page in the page table, this page is identical in each process. Page table sharing uses just one copy of that page, and sets the higher level pointer in each process to point to it.

On 32 bit architectures it only makes sense to share the lowest level table (the *pte* level). For most 64 bit architectures it can be useful to share the next level as well (the *pmd* level). These levels generally map 2MiB and 1GiB respectively.

6 The *powerpc* architecture

The *powerpc* architecture is different in that it does not use hardware page tables. It instead uses hash tables to store its page table entries in hardware. Additionally, the virtual address space is divided into segments of 256MiB in size. Each segment has an identifier that need not be unique to the process, but could be shared between all processes which map that segment if all data in that segment is to be shared between those processes.

Currently the memory management implementation for *powerpc* does not take advantage of this sharing capability. It assigns a segment identifier that is based on the process which is mapping it. Segments that are otherwise identical in different processes are given separate identifiers.

One of the goals of implementing shared page tables was to enable use of shared segment identifiers on *powerpc*. In the shared page table implementation the page table levels are divided so that a single *pmd* page maps 256MiB. This page is then used to generate a segment identifier that can be shared among all processes mapping that segment. Due to the way the hardware hash table interacts with the software page table, no sharing can be done at the *pte* level. This means that page table sharing can only be done for areas 256MiB or larger.

7 How Sharing is Done

When memory is mapped into a process a *vma* is created and linked into the appropriate *mm* and *address_space*, but no page table is allocated. All page table pages are allocated as necessary during page fault handling to correctly map the faulting address.

When a page table page is allocated at a level where sharing is enabled, the *vma* is checked whether it can be shared. If it can, it follows the *mapping* pointer to find the *address_space*. All the *vma*s connected to the *address_space* are checked. If the *vma* maps the same offset in the file to the same virtual address as the faulting *vma*, it looks for a corresponding page table page. If one is found, its share count is incremented and it is returned as the page table page to be installed. The *vma* linkage in *address_space* is a *prio* tree based on the starting and ending virtual addresses so lookup is fast.

8 Unmapping and Unsharing

There are several places where a shared page table needs to be unshared. The first and most common place is when the memory region is unmapped. Unshared page tables are deleted when they are unused. Shared page tables need to be disconnected from the tree and the share count decremented, but can not be deleted since they are still in use by other processes.

Additionally, there are several memory operations that change the shareability of a memory area. In particular these are *mprotect*, *mremap*, and *fremap*. When any of these are called on a memory range, that range is scanned for shared page table pages. If any are found, they are unshared.

Unsharing is actually very simple. Since all memory mapped by shared page tables is backed by files it is sufficient to simply unlink the shared page table page and clear the reference to it. When the process attempts to access memory in that area it simply faults the pages back in. If the page table is still shareable it will re-share. If not, it will allocate a new unshared page table page.

9 Locking

The memory subsystem has several locks that control concurrent access to its data structures. The *mm* contains the *mmap_sem* semaphore which protects the *vma*s associated with that process. The *mmap_sem* is a read/write semaphore. It is taken for write for calls that map, unmap, or change memory mappings, and is taken for read during page faults.

The *mm* also contains the *page_table_lock* spinlock, which protects the page table. A recent optimization is a new lock, the *pt_lock*, which is in the *struct page* associated with the *pte* page table page. This lock is held once the *pte* is found and allows greater concurrency between faulting threads in a process.

Another critical lock is the *i_mmap_lock* in the *address_space*. This protects the *vma* linkage in the *address_space*.

9.1 Locking Modifications for Shared Page Tables

To properly implement shared page tables the *pt_lock* concept was extended to apply to all levels of page table that could be shared. The *page_table_lock* was no longer adequate since a page table page could now be part of more than one page table. Under the shared page table model, the *pt_lock* is taken when entries for that page need to be modified. This allows multiple processes to safely take faults in the same region and on the same page.

Another extended use of a lock is the *i_mmap_lock*. This lock is held while searching the *vma*s in the *address_space* for a page table page to share.

9.2 The Unshare Race

There is a race condition when unsharing page table pages due to *mremap*, *mprotect*, or *fremap*. The call to unshare the page tables needs to be made while the original *vma* is still present and linked to the *address_space*. This means there is a window of time after the page table has been unshared when another process could come in and begin a new share.

The solution is to define a flag in the *vma* called VM_TRANSITION. This flag is set on entry to the functions that will change the *vma*. It remains set until all changes have been made, then is unset before the call completes. The page table sharing code will then refuse to look at any *vma* marked as VM_TRANSITION. While this may result in an occasional missed opportunity to share page tables, it eliminates any chance that page tables will be erroneously shared.

10 *Hugetlb* Interaction

The *hugetlb* code creates a pool of large pages that can be requested by an application when it maps memory. This is particularly useful because many architectures allow data pages to be directly mapped using the *pmd* level of the page table. The *hugetlb* pages are sized to be mapped in this fashion.

While *hugetlb* in many ways provides a similar tool to sharing page tables, it is much more limited in its function. It requires a system-wide dedicated pool of larger pages and requires that applications be recoded to use it. Page table sharing is entirely transparent to applications and will happen whenever the shareable memory region is large enough.

An additional feature of shared page tables is that for architectures that support sharing at the *pmd* level and that also support *hugetlb*, even memory areas that are using *hugetlb* will benefit from shared page tables.

11 Performance

The first step in testing performance was to measure applications that do not share large mapped areas and thus do not benefit from sharing page tables. Tests run with these applications (the primary test being *kernbench*) showed no performance difference at all. This indicates that the overhead of looking for page table sharing has no measurable cost.

The next step was to test using large applications that do massive sharing. An obvious candidate here was large database applications. Performance improvement for applications that do not use *hugetlb* for their shared areas was in the range of 35% to 40%. Applications that do use *hugetlb* still showed a benefit in the 3% to 5% range, which is considered significant by those who do database benchmarking.

12 Future Enhancements

In the current implementation only those memory areas which are mapped at the same address and span a shareable page table page can be shared. No attempt is made to idenfity page table pages that, while they are not fully filled by a shareable region, are otherwise empty. It should be possible to identify those areas and share them, with a concurrent call to unshare them if the empty space is subsequently filled with a different memory area.

In conjunction with checking for empty space, it should also be possible to modify the allocation strategy used when mapping memory to assign shareable memory areas a section of virtual memory that has no other memory mapped in it, therefore making it more likely that the page table could also be shared.

Another current limitation of the code is that the areas must be mapped at the same virtual address in each process. This means that the memory must either be allocated in a parent, then the children forked, or the application must use a known address to map the memory to. In practice this is common enough in large applications that memory can often be shared. It should be possible, however, to allow sharing as long as the mapped memory areas share a common alignment with respect to the page table pages, even though they are mapped at different addresses. This alignment could be ensured at mapping time.

13 Legalese