IACSIT International Journal of Engineering and Technology, Vol.2, No.2, April 2010
ISSN: 1793-8236

Virtual Memory Management Techniques in 2.6 Kernel and Challenges

Archana S. Sumant , Pramila M.Chawan

***Abstract*— The 2.6 Linux kernel employs a number of techniques to improve the use of large amounts of memory, making Linux more enterprise-ready than ever before. This article outlines a few of the more important changes, including reverse mapping, how reverse mapping is used for page reclaim, the use of larger memory pages, storage of page-table entries in high memory,kernel shared memory increases efficiency and flexibility and greater stability of the memory manager.**

Index Terms*—page-table entries *(PTEs)*, *page-direct approach, page reclaiming,* reverse mapping(*RMAP*), *translation lookaside buffer (TLB), Copy on Write (CoW).

I. Introduction

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it. Linux supports virtual memory, that is, using a disk as an extension of RAM so that the effective size of usable memory grows correspondingly. The kernel will write the contents of a currently unused block of memory to the hard disk so that the memory can be used for another purpose. When the original contents are needed again, they are read back into memory. This is all made completely transparent to the user; programs running under Linux only see the larger amount of memory available and don't notice that parts of them reside on the disk from time to time[3].

As the Linux kernel has grown and matured, more users are looking to Linux for running very large systems that handle scientific analysis applications or even enormous databases. These enterprise-class applications often demand large amounts of memory in order to perform well. The 2.4 Linux kernel had facilities to understand fairly large amounts of memory, but many changes were made to the 2.6 kernel to make it able to handle larger amounts of memory in a more efficient manner.

Manuscript received February 22, 2010.

Archana S. Sumant is with the Veermata Jijabai Technological Institute ,Matunga , Mumbai 400019 ,India (phone: +91 9503666033, email: archana.s.vaidya@gmail.com).

Pramila M.Chawan, is with the Veermata Jijabai Technological Institute , Matunga,Mumbai 400019 ,India (phone: +91 9869074620, email: pmchawan@vjti.org.in).

A. Reverse mapping

In the Linux memory manager, page tables keep track of the physical pages of memory that are used by a process, and they map the virtual pages to the physical pages. Some of these pages might not be used for long periods of time, making them good candidates for swapping out. However, before they can be swapped out, every single process mapping that page must be found so that the page-table entry for the page in that process can be updated. In the Linux 2.4 kernel, this can be a daunting task as the page tables for every process must be traversed in order to determine whether or not the page is mapped by that process. As the number of processes running on the system grows, so does the work involved in swapping out one of these pages [6].

Reverse mapping, or *RMAP*, was implemented in the 2.5 kernel to solve this problem. Reverse mapping provides a mechanism for discovering which processes are using a given physical page of memory. Instead of traversing the page tables for every process, the memory manager now has, for each physical page, a linked list containing pointers to the page-table entries (PTEs) of every process currently mapping that page. This linked list is called a *PTE chain*[5]. The PTE chain greatly increases the speed of finding those processes that are mapping a page, as shown in Figure 1.

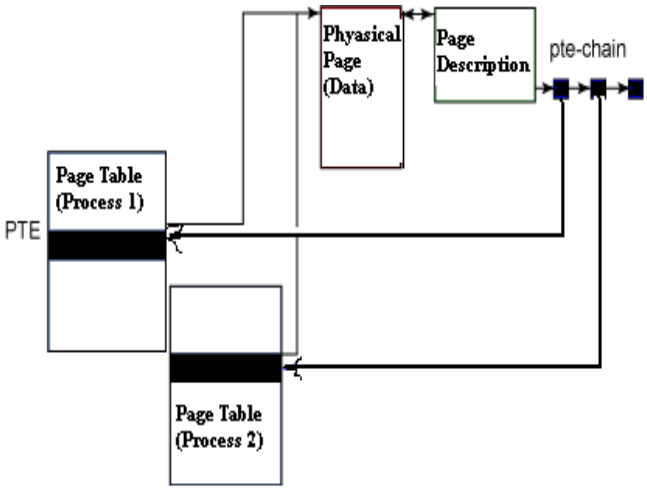

Figure 1 : Reverse-mapping in kernel 2.6

The performance gains obtained by using reverse mappings come at a price. The most notable and obvious cost of reverse mapping is that it incurs some memory overhead. Some memory has to be used to keep track of all those reverse mappings. Each entry in the PTE chain uses 4 bytes to store a pointer to the page-table entry and an additional 4 bytes to store the pointer to the next entry on the chain. This memory must also come from low memory, which on 32-bit

hardware is somewhat limited. Sometimes this can be optimized down to a single entry instead of using a linked list. This method is called the *page-direct approach*. If there is only a single mapping to the page, then a single pointer called "direct" can be used instead of a linked list. It is only possible to use this optimization if that page is mapped by only one process. If the page is later mapped by another process, the page will have to be converted to a PTE chain. A flag is set to tell the memory manager when this optimization is in effect for a given page.

There are also a few other complexities brought about by reverse mappings. Whenever pages are mapped by a process, reverse mappings must be established for all of those pages. Likewise, when a process unmaps pages, the corresponding reverse mappings must also be removed. This is especially common at exit time. All of these operations must be performed under locks. For applications that perform a lot of forks and exits, this can be very expensive and add a lot of overhead. Reverse mappings have proven to be a valuable modification to the Linux memory manager. A serious bottleneck with locating processes that map a page is minimized to a simple operation using this approach. Reverse mappings help the system continue to perform and scale well when large applications are placing huge memory demands on the kernel and multiple processes are sharing memory. There are also more enhancements for reverse mapping currently being researched for possible inclusion in future versions of the Linux kernel.

1) Reverse mapping used in page reclaiming

Page reclaim is the keystone to one of the kernel's fundamental decisions with respect to caching. In the swapping subsystem — the *swap policy* adopted to determine which pages can be swapped out of RAM memory without seriously degrading the kernel's performance is a major aspect. Since page frames are freed by this and new memory is available for urgent needs, the technique is also called *page reclaim*. The techniques employed by the kernel to write pages from memory into a swap area and to read pages from the swap area back into memory uses swap cache .The swap cache is an agent between the page selection policy and the mechanism for transferring data between memory and swap areas. These two parts interact via the swap cache. Input in one part triggers corresponding actions in the other.

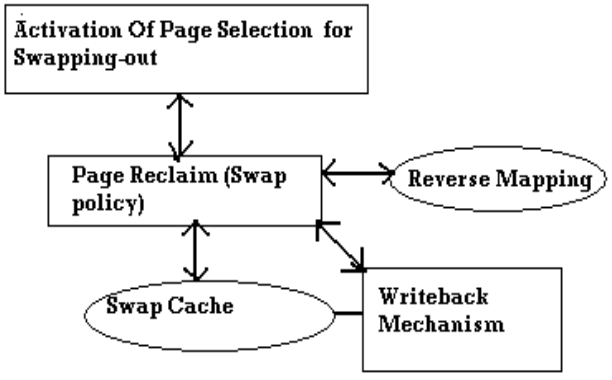

Figure 2 : Reverse mapping used in swapping subsystem.

Figure 2 shows how the swap cache interacts with the other components of the swap subsystem. Reverse mapping mechanism finds all processes that share a page. In earlier versions, shared memory pages could only be swapped out using the swap cache. Once the page had been removed from the page tables of a *single* process, the kernel had to wait until the page had also been removed from the page tables of all other processes before it could remove the data from memory; this required the systematic scanning of all system page tables. The pages were kept in the swap cache in the meantime. When a shared page is swapped out, RMAP finds all processes that reference the data in the page. Consequently, the relevant page table entries of *all* processes that reference the page can be updated to indicate the new position in the swap area. This means that the page data can be swapped out immediately without having to retain in the swap cache for a lengthy period.

Reverse mapping has following advantages:

1) Unmap a page from all processes using it, without needing to search the virtual memory of all processes.
2) Unmap only those pages it really wants to evict, instead of scanning the virtual memory of all processes and unmapping more pages than it wants to evict in order to be on the safe side. This could reduce the number of minor page faults.
3) Evict pages in a certain physical address range, which is useful since Linux divides physical memory in various zones.
4) Scan only the virtual mappings of known inactive pages, which means the pageout code has a smaller search space in virtual [2].

B. Huge Pages

Typically, the memory manager deals with memory in 4 KB pages on x86 systems. The actual page size is architecture dependent. For most uses, pages of this size are the most efficient way for the memory manager to deal with memory. Some applications, however, make use of extremely large amounts of memory. Large databases are a common example of this. For every page mapped by each process, page-table entries must also be created to map the virtual address to the physical address. If you have a process that maps 1 GB of memory with 4 KB pages, it would take 262,144 page-table entries to keep track of those pages. If each page-table entry consumes 8 bytes, then that would be 2 MB of overhead for every 1 GB of memory mapped. This is quite a bit of overhead by itself, but the problem becomes even worse if you have multiple processes sharing that memory. In such a situation, every process mapping that same 1 GB of memory would consume its own 2 MB worth of page-table entries. With enough processes, the memory wasted on overhead might exceed the amount of memory the application requested for use[1].

Typically, the total number of translations required by a program during its lifetime will require that the page tables are stored in main memory. Due to translation, a virtual memory reference necessitates multiple accesses to physical memory, multiplying the cost of an ordinary memory reference by a factor depending on the page table format. To cut the costs associated with translation, VM implementations take advantage of the principal of locality by storing recent translations in a cache called the Translation Lookaside Buffer (TLB) The amount of memory that can be translated by this cache is referred to as the "TLB

reach" and depends on the size of the page and the number of TLB entries. Inevitably, a percentage of a program's execution time is spent accessing the TLB and servicing TLB misses.

One way to help alleviate this situation is to use a larger page size. Most modern processors support at least a small and a large page size, and some support even more than that. On x86, the size of a large page is 4 MB, or 2MB on systems with physical address extension (PAE) turned on. Assuming a large page size of 4 MB is used in the same example from above, that same 1 GB of memory could be mapped with only 256 page-table entries instead of 262,144. This translates to only 2,048 bytes of overhead instead of 2 MB.

The use of large pages can also improve performance by reducing the number of *translation lookaside buffer (TLB)* misses. The TLB is a sort of cache for the page tables that allows virtual to physical address translation to be performed more quickly for pages that are listed in the table. Of course, the TLB can only hold a limited number of translations. Large pages can accommodate more memory in fewer actual pages, so as more large pages are used, more memory can be referenced through the TLB than with smaller page sizes.

Many modern operating systems, including Linux, support huge pages in a more explicit fashion, although this does not necessarily mandate application change. Linux has had support for huge pages since around 2003 where it was mainly used for large shared memory segments in database servers such as Oracle and DB2.

The huge page support is built on top of multiple page size support that is provided by most modern architectures. For example, i386 architecture supports 4K and 4M (2M in PAE mode) page sizes, ia64 architecture supports multiple page sizes 4K, 8K, 64K, 256K, 1M, 4M, 16M,256M and ppc64 supports 4K and 16M[4] .A TLB is a cache of virtual-to-physical translations. Typically this is a very scarce resource on processor. Operating systems try to make best use of limited number of TLB resources. This optimization is more critical now as bigger and bigger physical memories (several GBs) are more readily available. Users can use the huge page support in Linux kernel by either using the mmap system call or standard SYSv shared memory system calls (shmget, shmat).

C. Storing page-table entries in high memory

Page-tables can normally be stored only in low memory on 32-bit machines. This low memory is limited to the first 896 MB of physical memory and required for use by most of the rest of the kernel as well. In a situation where applications use a large number of processes and map a lot of memory, low memory can quickly become scarce. A configuration option, called *Highmem PTE* in the 2.6 kernel now allows the page-table entries to be placed in high memory, freeing more of the low memory area for other kernel data structures that do have to be placed there. In exchange, the process of using these page-table entries is somewhat slower. However, for systems in which a large number of processes are running, storing page tables in high memory can be enabled to squeeze more memory out of the low memory area. Figure 3 shows mapping of various memory regions.

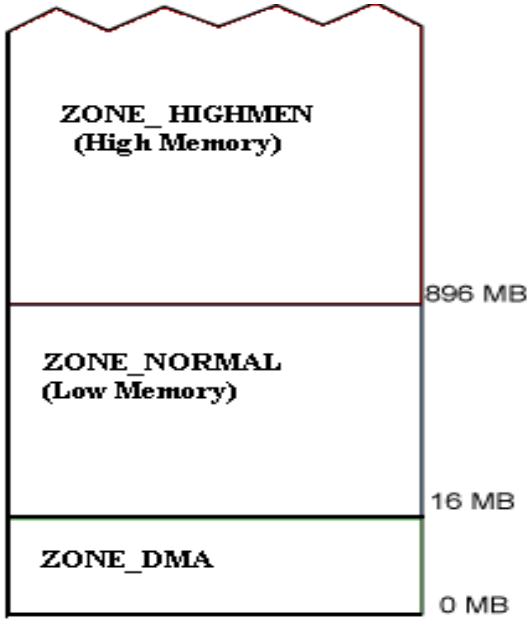

Figure 3. Memory regions

D. Kernel Shared Memory

Linux as a hypervisor includes a number of innovations, and one of the more interesting changes in the 2.6.32 kernel is Kernel Shared Memory (KSM). KSM allows the hypervisor to increase the number of concurrent virtual machines by consolidating identical memory pages. This feature is also known as *Kernel Samepage Merging.* Although memory sharing in Linux is advantageous in virtualized environments (KSM was originally designed for use with the Kernel-based Virtual Machine [KVM]), it's also useful in non-virtualized environments. In fact, KSM was found to be beneficial even in embedded Linux systems, indicating the flexibility of the approach.

KSM exists as a daemon in the kernel (called ksmd) that periodically performs page scans to identify duplicate pages and merges duplicates to free pages for other uses. It does this in a way that's transparent to the user. For example, duplicate pages are merged (and subsequently marked read only), but if one of the users of the page changes it for any reason, that user will receive his or her own copy (in a Copy on Write fashion). You can find the full implementation of the KSM kernel module within the kernel source at ./mm/ksm.c [7].

KSM relies on a higher-level application to provide instruction on which memory regions are candidates for merging. Although it would be possible for KSM to simply scan all anonymous pages in the system, it would be wasteful of CPU and memory (given the space necessary to manage the page-merging process). Therefore, applications can register the virtual areas that are likely to contain duplicate pages.

When KSM is enabled, it searches for identical pages, keeping one page in a write-protected Copy On Write fashion and freeing one for other uses. In the KSM, pages are managed by two red-black trees, one of which is temporary. The first tree, call the *unstable tree,* is used to store new pages that are not yet understood to be stable. In other words, pages that are candidates for merging (unchanged for some period of time) are stored in the unstable tree. Pages in the unstable tree are not write protected. The second tree, called

the *stable tree,* stores those pages that have been found to be stable and merged by KSM. To identify whether a page is volatile or non-volatile, KSM uses a simple 32-bit checksum. When a page is scanned, its checksum is calculated and stored with the page. On a subsequent scan, if a newly computed checksum is different from the previously generated checksum, the page is changing and is therefore not a good candidate for merging.

The first step in handling a single page using KSM's process is to see whether the page can be found in the stable tree. The process of searching the tree is interesting in that each page is treated as a very large number (the contents of the page). A memcmp (memory compare) operation is performed on the page and node's page of consideration. If the memcmp returns **0**, the pages are equal and a match is found. Otherwise, memcmp can return **-1** (which means that the candidate page is less than the current node's page) or **1** (the candidate page is greater than the current node's page). In each case, the operation leads the search down the red-black tree (to the left if less than and to the right if greater than). Although comparing 4KB pages seems fairly heavyweight, in most cases, the memcmp will end prematurely once a difference is found. Therefore, the process itself is fast and efficient.

When the scan is complete (performed through ksm.c/`ksm_do_scan()`)[7], the stable tree is kept, but the unstable tree is removed and rebuilt at the time of the next scan. Because all pages in the stable tree are write protected, a page fault is generated when a page tries to be written, allowing the CoW process to un-merge the page for the writer. Orphaned pages in the stable tree are subsequently removed (unless two or more users of the page exist, indicating that the page is still shared).

E. Stability

Better stability is another important improvement of the 2.6 memory manager. When the 2.4 kernel was released, users started having memory management-related stability problems almost immediately. Given the system wide impact of memory management, stability is of utmost importance. The problems were mostly resolved, but the solution entailed essentially gutting the memory manager and replacing it with a much simpler rewrite. This left a lot of room for Linux distributors to improve on the memory manager for their own particular distribution of Linux. The other side of those improvements, however, is that memory management features in 2.4 can be quite different depending on which distribution is used. In order to prevent such a situation from happening again, memory management was one of the most scrutinized areas of kernel development in 2.6. The new memory management code has been tested and optimized on everything from very low end desktop systems to large, enterprise-class, multi-processor systems.

II. CONCLUSION

The memory management improvements in the Linux 2.6 kernel go far beyond the features mentioned in this article. Many of the changes are slight but equally important. These changes all work together to produce a memory manager in the 2.6 kernel designed for better performance, efficiency, and stability. Some changes, like Highmem PTE and Large pages, work to reduce the overhead caused by memory management. Other changes, like reverse mappings, speed up performance in certain critical areas. How reverse mapping speed up performance in page reclaim in swapping subsystem is explained in detail. How Kernel Shared Memory is advantageous in virtual as well as non-virtualized systems is explained with it's working.

REFERENCES

[1] [1] Martin Bligh and David Hansen ," Linux Memory Management on Larger Machines " in 2003 Linux Symposium . Available: http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Bligh-OLS2003.pdf
[2] Rik van Riel ," Towards an O(1) VM" Available : http://www.surriel.com/lectures/ols2003/
[3] Mel.Gorman,"*Understanding the Linux Virtual Memory Manager* " Published By Prentice Hall ISBN 0-13-145348-3 pp :1-20.
[4] http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO-7.html
[5] Dave McCracken" Object-based reverse-mapping VM"
[6] Paul Larson ,"Improvements to the process of kernel development for the 2.6 kernel "Available : http://www.ibm.com/developerworks/linux/library/l-mem26/index.html
[7] "2.6.32 kernel release notes " Available : http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb

Archana S. Sumant is currently doing her M.Tech at "Veermata Jijabai Technological Institute ,Matunga , Mumbai (INDIA) and received Bachelors' Degree in Computer science and Engineering from "Walchand College Of Engineering ",Sangli (INDIA) in 2002. Her areas of interest are Operating System and Database management System. She is life member of ISTE (Indian Society Of Technical Education).She has authored 4 National and One International papers in Conferences.

Pramila M.Chawan is currently working as an Assistant Professor in the Computer Technology Department of "Veermata Jijabai Technological Institute (V. J. T. I.), Matunga, Mumbai (INDIA)". She received her Masters' Degree in Computer Engineering from V. J. T. I., Mumbai University (INDIA) in 1997 & Bachelors' Degree in Computer Engineering from V. J. T. I., Mumbai University (INDIA) in 1991 .She has an academic experience of 18 years (since 1992). She has taught Computer related subjects at both the (undergraduate & post graduate) levels. Her areas of interest are Software Engineering, Computer Architecture & Operating Systems. She has published 14 papers in National Conferences & 4 papers in International Conferences & Journals. She has guides 25 M. Tech. projects & 75 B. Tech. projects. Currently she is guiding Ms. Archana Sumant's M. Tech. project named "Virtual Memory Management in Linux kernel 2.6".