Commenting the Virtual Memory Management Kernel Source Code 2.6.31 for Educational Purpos

Archana S. Sumant , Pramila M.Chawan

Abstract- While doing a study of Operating System one can easily learn theoretical concepts but if anyone wants to study an actual operating system code there is only one way that is the documentation which comes with an operating system source code. As you can easily download a Linux source code, you will found that very less document is available for doing a study of Virtual Memory manager .Practically if any one wants to study the techniques used in designing an Operating System then it will take a long time to understand the implementation. So what I have done in my project is that the detail documentation of kernel code from virtual memory management point of view is presented to those who wants to study an VM part of an operating system Linux kernel 2.6.31.

This project deals with the study of Virtual Memory Manager in Linux kernel 2.6.31 and some comparative conclusions between 2.4 and 2.6 kernel features.

Index Terms*—page allocator, anonymous memory ,page cache ,*memory allocators , 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. Virtual memory is implemented in Linux with secondary storage disks as extension so that the memory size can be increased according to program need though system have physical RAM size less. 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.

Memory management is one of the most complex and at the same time most important parts of the kernel. It is characterized by the strong need for cooperation between the processor and the kernel because the tasks to be performed require them to collaborate very closely [9].

Figure 1 gives a conceptual overview on how basic Linux memory management works. Central to all memory management is the *page allocator*. The page allocator can hand out pieces of memory in chunks of *page size* bytes. The page size is fixed in hardware at 4 KBytes for i386, x64 and many other architectures. The page size is configurable on several platforms. On Itanium the page size is usually configured to be 16k. All other memory allocators are in one way or another based on the page allocator and take pages out of the pool of pages managed by the page allocator. The page allocator may provide pages that are mapped into a processes virtual address space. There are two ways that pages mapped into user space are used.The first type of pages is used for *anonymous memory*. These are pages for temporary use while a process is running. They are not associated with any file and are typically used for variables, the heap and the stack. Anonymous memory is light weight and can be managed in a more efficient way than file backed pages because no mappings to disk (which require serialization to access) have to be maintained.

Anonymous memory is private to a process (hence the name) and will be freed when a process terminates. Anonymous memory may be temporarily moved to disk (swapping) if memory becomes very tight. However, at that point an anonymous page acquires a reference to swap space and therefore a mapping to secondary

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

storage, which adds overhead to the future processing of this page.

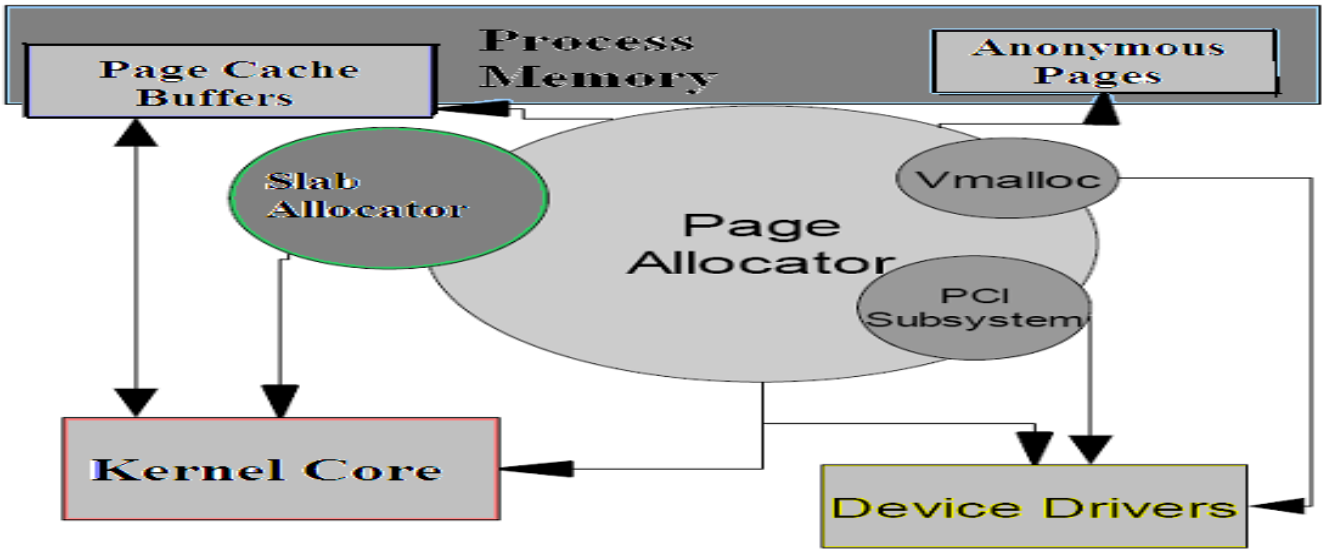

Figure 1 : Linux Memory Subsystems

The *page cache* or *buffers* are pages that have an associated page on a secondary storage medium such as a disk. Page cache pages can be removed if memory becomes tight because their content can be restored by reading the page from disk. Most important is that the page cache contains the executable code for a process. A process may map additional files into its address space via the **mmap()** system call. Both the executable code and the mapped files are directly addressable via virtual addresses from the user process. The operating system may also maintain buffers in the page cache that are not mapped into the address space of a process. This frequently occurs if a process manipulates files through system calls like **sys_write()**, **sys_read()** that read and modify the contents of files. The unmapped pages may be also thought as belonging in some loose form to a process. However, all page cache pages may be mapped or accessed by multiple processes and therefore the ownership of these pages cannot be clearly established.

The *kernel core* itself may need pages in order to store meta data. For example file systems may use buffers to track the location of sections of a file on disk, pages may be used to establish the virtual to physical address mappings (page tables) and so on. The kernel also needs to allocate memory for structures of varying sizes that are not in units of the page size in use on the system. For that purpose the *slab allocator* is used. The slab allocator retrieves individual pages or contiguous ranges of pages from the page allocator but then uses its own control structures to be able to hand out memory chunks of varying sizes as requested by the kernel or drivers. The slab allocator employs a variety of caching techniques that result in high allocation performance for small objects. Slab allocations are used to build up structures that maintain the current system state. This includes information about open files, recently used filenames and a variety of other state objects.

The *device drivers* utilize both the page allocator and the slab allocator to allocate memory to manage devices. There are a couple of additional variations on page sized allocations for device drivers. First there is the *vmalloc* subsystem. Vmalloc allows the allocation of larger chunks of memory that appear to be virtually contiguous within kernel context but the actual pages constituting this allocation may not be physically contiguous. Therefore vmalloc can generate a virtually contiguous memory for large chunks of memory even if the page allocator cannot satisfy request for large contiguous chunks of memory anymore because memory has become fragmented. Accesses to memory obtained via the vmalloc allocator must use a page table to translate the virtual addresses to physical addresses and may be not as efficient as using a direct physical address as handed out from the page allocator. Vmalloc memory may not be mapped into user space. Finally, the PCI subsystem itself may can be used by a device driver to request memory that is suitable for DMA transfers for a given device via **dma_alloc_coherent()**. The way of obtaining that type of memory varies with the type of underlying hardware and therefore the allocation technique varies for each platform supported by Linux.

Table 1 Basic Memory Allocators under Linux

Allocation Function	***Allocator***	***Action***
alloc_pages(flags, order)	Page allocator	Allocates 2^order contiguous pages
kmalloc(size, flags)	Slab allocator	Allocate *size* bytes of memory
kmem_cache_alloc(cache, flags)	Slab allocator	Allocate an entry for the indicated slab
vmalloc(size)	vmalloc subsystem	Allocate a virtual contiguous memory area with a minimum of *size* bytes.
dma_alloc_coherent(device, size, &addr, flags)	PCI subsystem/ architecture specific support	Allocate DMA capable memory for the indicated device.

Table 1 gives an overview of the basic memory allocators under Linux:

II. COMMENTORY ON KERNEL CODE

To get a comprehensive view on how the kernel works, one is required to read through the source code line by line. This project focus on giving detail documentation of kernel code 2.6.31 so that the time to understand the kernel functions will be measured in weeks and not months. For managing such huge source code I have used a LXR tool which can be downloaded from *http://lxr.linux.no/.*

The code commentary will be done according following flow.

1 Boot Memory Allocator
- 1.1 Representing the Boot Map
- 1.2 Initializing the Boot Memory Allocator
- 1.3 Allocating Memory
- 1.4 Freeing Memory

2 Physical Page Management
- 2.1 Allocating Pages
- 2.2 Free Pages
- 2.3 Page Allocate Helper Functions
- 2.4 Page Free Helper Functions

3 Non-Contiguous Memory Allocation
- 3.1 Allocating A Non-Contiguous Area
- 3.2 Freeing A Non-Contiguous Area

4 Slab Allocator
- 4.1 Introduction
- 4.2 Slabs
- 4.3 Objects
- 4.4 Sizes Cache
- 4.5 Per-CPU Object Cache
- 4.6 Slab Allocator Initialization
- 4.7 Interfacing with the Buddy Allocator

5 Process Address Space

5.1 Managing the Address Space

5.2 Process Memory Descriptors
- 5.2.1 Allocating a Descriptor
- 5.2.2 Initializing a Descriptor
- 5.2.3 Destroying a Descriptor

5.3 Memory Regions .
- 5.3.1 Creating A Memory Region
- 5.3.2 Finding a Mapped Memory Region
- 5.3.3 Finding a Free Memory Region
- 5.3.4 Inserting a memory region
- 5.3.5 Merging contiguous region
- 5.3.6 Remapping and moving a memory region
- 5.3.7 Locking a Memory Region
- 5.3.8 Unlocking the region
- 5.3.9 Fixing up regions after locking/unlocking
- 5.3.10 Deleting a memory region
- 5.3.11 Deleting all memory regions

5.4 Page Fault Handler
- 5.4.1 Handling the Page Fault
- 5.4.2 Demand Allocation
- 5.4.3 Demand Paging
- 5.4.4 Copy On Write (COW) Pages

III. APPLICATIONS

1.This will reduce the amount of time a developer or researcher needs to understand Linux Virtual memory manager.

2.Further this study and documentation can be used to improve some aspects of Virtual memory management.

3.Developer can even change particular part of Virtual memory manager code for particular applications .

IV. CONCLUSION

Very little help or code documentation available for practically understanding an operating system one may require an extra time for doing so. My project gives

detail study of kernel code 2.6.31 from point of view of virtual memory manager (architecture independent features) and will help to those who wants to swim in operating system code .In future enhancement one can modify the kernel code and recompile it to make new version.

REFERENCES

[1] http://lwn.net/ Linux info from the source
[2]Gorman Mel. "*Understanding the Linux Virtual Memory Manager "* Prentice Hall Professional Technical Reference 2004
[3]http://kernel.org/doc
[4]http://www.linuxhq.com/kernel/v2.4/index.html
[5]http://www.linuxhq.com/kernel/v2.6/index.html
[6]Neil Horman Understanding Virtual Memory In Red Hat Enterprise Linux 4 Version 0.1 –
[7]http://www.perens.com/Book (Mel Gorman book site)
[8]The Linux Kernel Source Tree. Version 2.6.31 http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.31.5.tar.bz2
[9]Wolfgang Mauerer "Professional Linux® Kernel Architecture "Wiley Publishing,

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".