Comparison of Memory Management Systems of BSD, Windows, and Linux

Gaurang Khetan

Graduate Student,
Department of Computer Science,
University of Southern California,
Los Angeles, CA.
`gkhetan@usc.edu`

December 16, 2002

Abstract

This paper is a study of memory management systems of an operating system. We begin with a brief introduction to memory management systems and then we compare the memory management systems of real-life operating systems - BSD4.4, Windows 2000 and Linux 2.4

1 Introduction

In this paper, we will be comparing the Memory Management (MM) Sub-Systems of these operating systems - BSD 4.4, Linux 2.4 and Windows 2000. BSD 4.4 was chosen since it is a representative Unix version including important operating system design principles, and today many operating systems like FreeBSD [3], NetBSD [5] and OpenBSD [6] are based on it. Moreover, it is very well documented in [12]. Windows 2000 was chosen since it is a very popular operating system for use as a desktop especially with beginners, and has now evolved into a mature operating system. Linux [4] 2.4 was chosen because it is growing more and more popular by the day, and seems to have an important place in the future. We will not be much interested in the performance characteristics of these systems in this paper, instead our focus will be on their design and architecture.

2 Memory Management Systems

The Memory Management System is one of the important core parts of an operating system. Its basic function is to manage the memory hierarchy of RAM and hard disks available on a machine. Its important tasks include allocation and deallocation of memory to processes taking care of logistics,and implementation of Virtual Memory by utilizing hard disk as extra RAM. The Memory system should be optimized as far as possible, since its performance greatly affects the overall performance and speed of the system.

2.1 Virtual Memory

An important concept in the context of MM Systems is Virtual Memory. Back in the early days of computing, researchers sensed the ever growing memory requirements of application programs, so the idea of Virtual Memory was born. The idea is to provide an application program the illusion of the presence of a very large amount of memory available for its use. The kernel will provide such a facility by making use of secondary storage - the hard disk - to fulfill the extra space requirements. [13]

For the virtual memory system to work, we require some mapping function which will perform address translation, converting the *virtual address* to the *physical address*. The virtual address is the address which the application uses to refer to a memory location, and the physical address is the actual memory location passed out to the local memory bus. This function is generally one of Paging, or Segmentation, or both - depending on the kernel, processor architecture and its state.

2.2 Paging

In Paging, the address space (both virtual and real) is divided into fixed-sized (though they can be multiple-sized [14]) pages. The pages can be individually manipulated, and placed at different places in the physical memory and the hard disk. The address translation is actually done by the *Memory Management Unit* (MMU) of the processor by the use of a *Page Table*, as shown in Figure 1. Page tables specify the mapping between virtual pages and physical pages - i.e. which virtual memory page currently occupies which physical page. The MMU converts the virtual memory address to a physical address which consists of a page frame number and an offset within that page. Protection can be applied on an page by page basis.

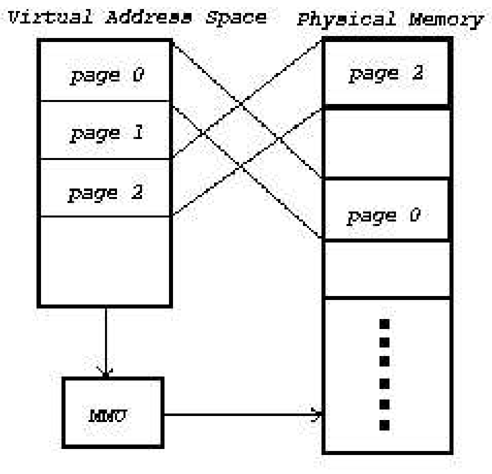

Figure 1: Page table

Since the virtual address space is huge compared to the physical memory, we must use the hard disk to store pages which cannot be stored in the physical memory. Associated with each virtual page in the page table is a bit to denote whether the page is present in the physical memory or not. If the page is not present in the physical memory, the hardware generates a *page fault exception*. This exception is handled in software, which places the required page back from the hard disk to the physical memory, or if it is invalid, generates an error.

Coffman and Denning [2] characterize paging systems by three important policies:

1. When the system loads pages into memory - the *fetch* policy.

2. Where the system places pages into memory - the *placement* policy

3. How the system selects pages to be removed from main memory when pages are unavailable for a placement request - *the page replacement policy*.

The placement policy is of importance only for optimizing certain behavior [16]. So, practically, the behavior of a paging system is dependent only on the fetch and placement policy. In most modern systems, for the fetch policy a demand paging system is used in which the system brings a page to memory only when it is required, however sometimes prepaging certain pages that are *expected* to be required. With regard to the page replacement policy, many algorithms have been developed over the years. An account can be found in [19]. Comparisons of performance of page replacement algorithms can be found in many papers, such as [15].

3 Comparison

Now we shall concentrate on the MM systems of Windows 2000, Linux 2.4 and BSD 4.4.

The BSD4.4 VM system is based on Mach 2.0,2.5 and 3.0 VM code. The Windows 2000 was developed in a long series of operating systems since MSDOS. The Linux 2.4 has been developed by hackers originally founded by Linux Torvalds.

Other than the resources cited elsewhere in this article, more information on the topic can also be obtained from [18, 17, 21].

Instead of describing each of the system's MM system in detail, which will be a very long exercise, we compare here some of their significant parts.

All the three systems have modern MM systems, and have surprisingly a lot in common. The data structures are quite similar, and the features of each are also quite similar. Some similarities of these systems are enumerated below -

- Hardware Abstraction Layer: All OSes have a layer called the hardware abstraction layer (HAL) which does the system-dependent work, and thus enables the rest of the kernel to be coded in platform independent fashion. This eases porting it to other platforms.

- Copy-on-write: When a page is to be shared, the system uses only one page with both processes sharing that same copy of the page. However, when one of the process does a write onto the page, a private copy is made for that process, which it can then manipulate individually. This gives a lot better efficiency.

- Shadow paging: A shadow object is created for an original object such that the shadow object has some of its pages modified from the original object, but shares the rest of the pages with the original object. They are formed as a result of Copy-On-Write action.

- A Background daemon: There exists a background daemon which is invoked periodically and performs tasks like page flushing, freeing unused memory, etc.

- Memory mapped Files: A file can be mapped onto memory, which then can be used with simple memory read/write instructions.

- Inter-Process Communication: The memory mapped files are allowed to be then shared between processes forming a method for inter-process communication.

In the following subsections, we will compare these systems on certain aspects.

3.1 Data Structures to describe a process space

Now we will study the data structure the systems use to maintain and keep track of the virtual memory.

3.1.1 BSD4.4

The BSD4.4's data structures are shown in the figure 2. This structure is repeated for every process, since each process essentially has its own flat virtual address space.

The main structures are -

- vmspace
- vm_map
- vm_map_entry
- object
- shadow object
- vm_page

The vm_pmap is a hardware dependent layer. It takes care of memory management at the lowest level, generally taking care of the different methods different processors have for virtual memory programming, etc. Putting the hardware dependent code into just one module makes the rest of the VM Code

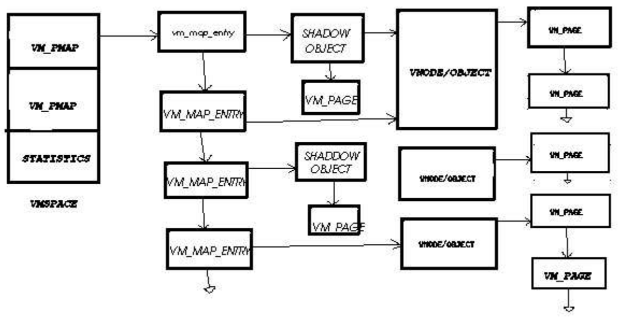

Figure 2: BSD4.4 Data Structures for managing Process Virtual Memory

hardware independent, which makes for a modular design and renders it relatively easier to port the code to different architectures. The vm_map structure contains a pointer to vm_pmap and pointer to a vm_map_entry chain. One vm_map_entry is used for each contiguous region of the virtual memory that has the same protection rights and inheritance. This then points to a chain of vm_object objects. The last one in the list is the actual object (file, etc), and the others are *shadow objects*. Shadow objects will be dealt with in another aspect of comparison. The object has a pointer to a list of vm_page objects, which represent the actually physical memory pages. These pages in the main memory are considered as cache of the hard disk, a virtual memory concept. The vm_object also consists pointers to functions which will perform op-

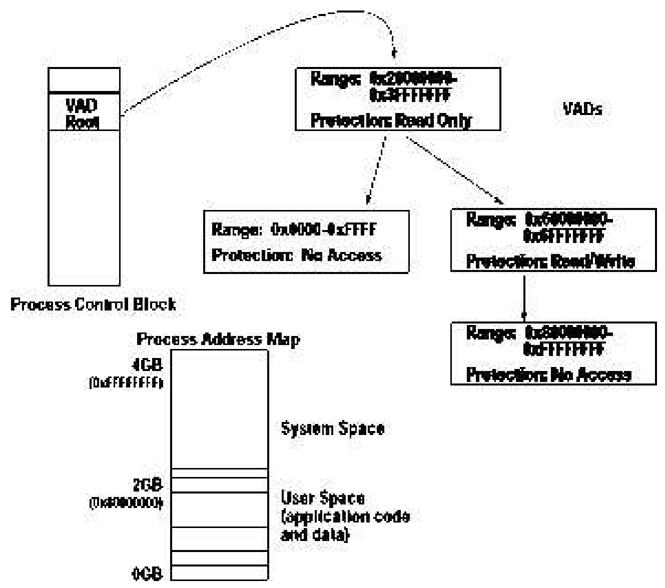

Figure 3: Windows NT 4.0 Data Structures for managing Process Virtual Memory

erations on it.

3.1.2 Windows

The data structures used by Windows NT are as shown in Figure 3.

Instead of a linked list, the Windows NT System keeps it in a tree form. Each node of the tree is called *Virtual Address Descriptors*(VAD). Each VAD denotes a range of address which has the same protection parameters and commit state information. The tree is also a *balanced*, which means that depth of the tree is kept at a minimum. This then implies that the search time, say when finding the node containing a location, will be relatively low. The VAD marks each node as either *committed*, *free*, or *reserved*. Committed are the ones which have been used i.e. code or data has been mapped onto it. Nodes that are marked free are yet unused, and those marked reserved are the ones which are not available for being mapped until the reservation is explicitly removed. Reservation are used in special cases, for example, a node can be reserved for a thread's stack when a thread is created. The link to the root of the tree is kept in the Process Control Block.

3.1.3 Linux

Linux implements the virtual memory data structure in a similar manner to UNIX. It maintains a linked list of vm_area_structs. These are structures which represent continuous memory areas which have the same protection parameters etc. This list is searched whenever a page is to be found that consists a particular location. The structure also records the range of address it is mapping onto, protection mode, whether it is pinned in memory(not page-able), and the direction (up/down) it will grow in. It also records whether the area is public or private. If the number of entries grows greater than a particular number, usually 32, then the linked list is converted into a tree. This is a quite good approach which uses the best structure in the best situations.

3.2 Distribution of Process Address Space

All the three systems distribute the process virtual address space in a similar manner. Higher part of it is used by the kernel, while the process can use the lower part. The kernel part of the space of all process usually point to the same kernel code. So while switching a process, we need to switch the page table entries of the lower part, while the upper part can remain the same. In Linux and BSD, usually 3GB is kept for the process and 1 GB given to the kernel, while in Windows, 2GB are kept for each.

3.3 Page Replacement

Page Replacement is an important part of any MM System. Basically, page replacement concerns with choosing which page to page-out - i.e. swap out from memory, whenever the need for more free memory arises.

The Ideal Page Replacement Algorithm is to remove the page which will be required for access in the most distant future. Doing this will cause the least number of page faults, and thus least wastage of time doing swapping, in turn improving system performance and throughput. But since it is not possible to know what pages will be accessed in the future, the ideal page replacement algorithm is impossible to implement.

Let us see how each of the system works for page replacement.

3.3.1 BSD4.4

The system uses *demand paging system*(with some prepaging) for its fetch policy, and an approximation of *global Least Recently Used* algorithm.

Demand Paging means that pages will be brought to memory only when they are required. In practical circumstances, however, the pages that are expected to be used are also brought to memory initially itself.

For paging out, the system operates with a *global* replacement algorithm. Global means that the system chooses the page to be removed irrespective of the process using that page, which means pages of all processes are considered equally and some other parameter is used for selection.

The system divides the main memory into four lists:-

1. Wired: These pages are *locked*. They cannot be move be swapped out. These pages are usually used by the kernel.
2. Active: The pages that are (supposed to be) actively being used are put in this list.
3. Inactive: The inactive pages, which have known content, but no active use for some time.
4. Free: The pages which have no known content, and hence are immediately usable.

A Page daemon is used for maintaining some amount of free memory in the system. The page daemon is a process that is started in the beginning in the kernel mode, and remains until the computer is shut off. The goal of the page daemon is to maintain a minimum amount of pages in the free list, specifically stored in *free_min* (usually 5 percent of memory). There is another variable *free_target*, which is usually 7 percent of memory - the daemon stops working when it has achieved its target of *free_target* amount of free memory. The amount of pages in the inactive list is also to be maintained at at least *inactive_target*(which is usually 33 percent of memory). However the value of *inactive_target* is tuned automatically by the system with time. So whenever the free memory falls below *free_min* the daemon is invoked.

The daemon starts scanning the inactive list from the oldest to the youngest and does the following for each page:-

- If the page is clean and unreferenced, move it to the free list.
- If the page has been referenced by an active process, move it from the inactive list to the active list.
- If the page is dirty and is being written to the swap file currently, skip it for now.

- If the page is not dirty and is not being actively used, then it is written back to the disk.

After scanning, it checks whether the inactive list is smaller than *inactive_target*, then, it starts scanning the active list to bring some pages back to the inactive list.

There is also a concept of swapping in BSD. When it cant keep up with the page faults even while performing its function, or if any process has been inactive for more than 20 seconds, the page-out daemon goes into swapping mode. In the swapping mode, it takes the process which has been running for the longest time, and swaps it *completely* back to the hard disk, or secondary store.

3.3.2 Windows

The system used by Windows in this case it too sophisticated and complicated.

Windows uses *clustered demand paging* for fetching pages, and *the clock* algorithm for the page replacement.

In Clustered demand paging, the pages are only brought to memory when they are required. Also, instead of bring 1, Windows, often brings a cluster of them of 1-8 pages, depending on the current state of the system.

The kernel receives 5 kinds of page faults -

- The page referenced is not committed.
- A protection violation has occurred.
- A shared page has been written.
- The stack needs to grow.
- The page referenced is committed but not currently mapped in.

The first two are irrecoverable errors. The third indicates an attempt to write to read-only page. Copy that page somewhere else and make the new one read/write. This is how copy-on-write works. The fourth needs to be responded by finding an extra page.

The most important point about the Windows Paging Systems that it makes heavy use of the *working set* concept. The working set is defined as the amount of main memory currently assigned to the process, so the working set consists of its pages that are present in the main memory. The size of the working set is, however, not constant. So the disadvantages that come with working sets are heavily reduced.

The clock algorithm used by Windows is *local*. When a page fault occurs, and faulting process's working set is below a minimum threshold, then the page is simply added to the working set. On the other hand, if the working set is higher than one another threshold, then it reduces the size of working set. Hence the algorithm can be called global. But the system does do some global optimizations too. For example, it increases the working set of processes that are causing a large number of page faults, and decreasing the working set for those who do not require enough memory.

Instead of just working when there is a page fault, just like Unix, Windows has a daemon thread working too, but called in this case as *Balance Set Manager*. This is invoked every 1 second, and it checks whether there is enough free memory. If there is not, then it invokes the *working set manager*. The working set manager maintains to keep the free memory above threshold. It checks the working sets of process from old and big to the young and small. And depending on how many page faults they have generated, it increases or decreases them. If a page's reference bit is clear, then counter associated with the page is incremented. If the reference bit is set, the counter

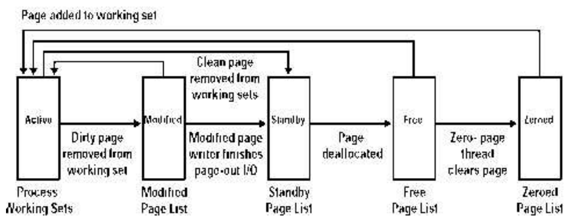
Page added to working set
Clean page removed from working sets
Active
Dirty page removed from working set
Modified
Modified page writer finishes page-out I/O
Standby
Page deallocated
Free
Zero- page thread clears page
Zeroed
Process Working Sets
Modified Page List
Standby Page List
Free Page List
Zeroed Page List

Figure 4: Windows NT 4.0 Data Structures for managing Process Virtual Memory

is set to zero. After the scan, pages with the highest counter are removed from the working set. Thus, the global aspect of this clock algorithm is given by this working set manager.

Windows divides the list of pages into four lists:-

1. Modified Page List
2. Stand-bye Page list
3. Free Page list
4. Zeroed Page List

These are shown in figure 4

The first is list of dirty pages, stand-bye is a list of clean pages, are currently associated with a process. Whereas Free Pages are those clean pages which are not even associated with some process. The Zeroed list is the list of zeroed out pages, if needed.

The transitions between these lists is handled by working set manager and some other daemon threads such as - swapper thread, mapped page write and modified page writer.

3.3.3 Linux

Up to Linux 2.2, the Linux VM had focused on simplicity and low overhead. Hence it was rather quite primitive and had many problems, especially under heavy load. It was influenced by System V.

However Riel [20] has worked a lot on the Linux VM in the past couple of years, and improved it a lot for the Linux 2.4 release.(his discussion with Matthew Dillon [1], a FreeBSD VM Hacker, is interesting and informative.)

Linux uses a demand paged system with no prepaging. [19]

Until kernel version 2.2, Linux used NRU algorithm for page replacement, but due to the various shortcomings of the algorithm, they have changed it and implemented an approximate Least Recently Used in 2.4.

The aging to effect LRU is brought about by increasing the age (a counter associated with a page) of a page by a constant when the page is found to be referenced during a scan, and, decreased exponentially (divided by 2) when found not to have been referenced. This method approximates LRU fairly well.

Linux 2.4 divides the virtual pages into 4 lists [9]-

1. Active list
2. Inactive-dirty list
3. Inactive-clean list
4. Free list

To separate the pages which were chosen for eviction, the inactive-dirty list was made. Normally, the active pages are on the list 1. But as time passes, if some of the pages are not active, then their age decreases and goes down to 0, which indicates it is a candidate for eviction. Such pages are moved from list 1 to list 2.

The inactive list in BSD4.4 (and FreeBSD) has a target of 33%, which the daemon manages to keep it at that level. However, for linux 2.4, the inactive list size was made dynamic. Now the system itself will decide how many inactive pages it should keep in the memory given the particular situation.

The unification of the buffer cache and page cache has been completed in 2.4, as had been implemented in FreeBSD.

Another optimization present in the Linux Kernel, is that they now recognize continuous I/O, i.e. they now decrease the priority of the page “behind” and so that page becomes a candidate for eviction sooner.

The page daemon in Linux is *kswapd* which awakens once a second, and frees memory if enough is not available.

And the flushing is done by another daemon *bdflush*, which periodically awakes to flush dirty pages back to the disk. The page flushing that takes place from the inactive list, does not happen in an ad-hoc fashion, but the system waits for the right time, when clustering could be used, and disk read-writes could be minimized, thus optimizing the flushing.

4 FreeBSD, NetBSD, OpenBSD and UVM

The discussion given above pertains to BSD4.4 which is now outdated and is no longer in use. However, there are many successors to BSD4.4 like FreeBSD, NetBSD, etc which are based on its code. Though they have many similarities, the newer versions have undergone some significant development since BSD4.4.

Regarding the VM, the FreeBSD VM has been developed and optimized a lot through the work of John Dyson, David Greenman, and Matthew Dillon. [10].

And NetBSD and OpenBSD now have evolved to using UVM [8], which was developed as a PhD thesis [7]. The authors of UVM claim that UVM design is much better than both BSD4.4 and FreeBSD.

The designs of the VMs of FreeBSD and Linux have been compared in this paper [11]. But the author of this article could not find sufficient evidence to give verdict on the comparison of *performance* of the VMs of Linux and the BSDs, which is still debated.

5 Comments and Conclusion

All the three systems have originated in different backgrounds - BSD4.4 in Academia, Windows in Commercial Settings, and Linux in Hackers settings.

All the three are pretty modern and have sound theoretical concepts, and are all suitable for production environments.

They have a lot in common, and few differences, technically speaking.

Windows, being developed with strong monetary motivation, has gone through more effort and thought in its design and development. And one must say that the design decisions made at various levels are tended to be conducive to better performance.

In the case of Unix-based systems like BSD4.4 and Linux, the decision was taken often favoring simplicity against performance.

Thus Windows has developed into sophisticated, complex code whereas Unix is simple and elegant but still modern.

The result of which is that Windows has more features but is difficult to maintain and improve from the developers’ view, while Unix has less features but is easier to maintain and develop.

However, for the end-user, Windows is likely to give better performance while occasionally crashing.

Still more research and development is required for the Open Source Systems, and there is good scope for it.

It is quite clear that documentation on Open Source operating systems like FreeBSD and Linux is lacking, especially ones which are comprehensive and up-to-date. It seems that as soon as some documentation is completed, the fast development of these operating systems render them out of date.

The rate of development of these Open Source Operating systems, which are maintained by hundreds and thousands of hackers around the world, is staggering. It may well be expected that in the future, these operating systems become at par or better than the commercial offerings.

References

[1] Discussion between Matthew Dillon and Rik van Riel. The linux-mm mailing list, 2000. Available on http://mail.nl.linux.org/linux-mm/2000-05/msg00419.html.

[2] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice Hall, 1973.

[3] Various Contributors. The FreeBSD Project. Further information can be found on http://www.freebsd.org.

[4] Various Contributors. The Linux Project. Further information can be found on http://www.linux.org.

[5] Various Contributors. The NetBSD Project. Further information can be found on http://www.netbsd.org.

[6] Various Contributors. The OpenBSD Project. Further information can be found on http://www.openbsd.org.

[7] C. Cranor. Design and implementation of the uvm virtual memory system, 1998.

[8] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the Usenix 1999 Annual Technical Conference*, pages 117–130, 1999.

[9] Brazil Dept of Computer Science, Univesity of Sao Paulo. Linux 2.4 vm overview. Available on http://linuxcompressed.sourceforge.net/vm24/.

[10] Matther Dillon. Design elements of the FreeBSD VM system. *DaemonNews*, January 2000.

[11] Rohit Dube. A comparison of the memory management sub-systems in freeBSD and linux. Technical Report CS-TR-3929, 1998.

[12] M.K. McKusick et al. *The Design and Implementation of 4.4BSD Operating System*. Addison-Wesley, 1996.

[13] T. Kilburn et al. One level storage system. *IRE Transactions*, EC-11(2):223–235, 1962.

[14] Yousef A. Khalidi, Madhusudhan Talluri, Michael N. Nelson, and Dock Williams. Virtual memory support for multiple page sizes. In *Workshop on Workstation Operating Systems*, pages 104–109, 1993.

[15] W.F. King. Analysis of demand paging algorithms. In *In International Federation for Information Processing Conference Proceedings*, pages 485–490, 1972.

[16] T. Romer. Using virtual memory to improve cache and TLB performance. Technical Report TR-98-05-03, 1998.

[17] Mark Russinovich. Inside memory mangagement. *Windows and .NET magazine*, June June 1998.

[18] David A. Solomon and Mark E. Russinovich. *Inside Windows 2000*. Microsoft Press, third edition, 2000.

[19] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[20] Rik van Riel. Page replacement in linux 2.4 memory management. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[21] Paul R. Wilson. The GNU/Linux 2.2 virtual memory system, January 1999.