A Comparison of the Memory Management sub-systems in FreeBSD and Linux

Rohit Dube
Institute for Advanced Computer Studies and Department of Computer Science
University of Maryland
College Park, MD 20742
Current Email: rohitd@dnrc.bell-labs.com

CS-TR-3929, UMIACS-TR-98-45

September 25, 1998

Abstract

In this article we seek to compare the memory management sub-systems of two popular and freely available operating systems - FreeBSD and Linux. First a framework is developed, spelling out the components of a generic and modern memory management system. The framework is then used in a design level comparison of memory management in the two operating systems.

1 Introduction

Given that FreeBSD ([6]) and Linux ([7]) are two of the most popular (free) Unix derivatives, and that the memory management sub-system is an important constituent of any operating system, one would expect technical material comparing memory management in the two OSs. Unfortunately no such material exists. This article is designed to fill the gap and provide a design level comparison of memory management between the two systems.

Note that we restrict our attention to design issues – no performance evaluation is attempted. Further the focus of the article is on core memory management for uni-processor systems. Issues concerning multiprocessor systems and interaction with various types of I/O devices are omitted. (Symmetric Multi-processing and disk and network performance enhancements are being actively worked on in both communities. A discussion of their impact on memory management is out of the scope of this article.) The discussion is kept hardware independent where ever possible; where it is important to discuss hardware dependent concepts, an Intel x86 ([2], [10]) architecture is assumed.

We start by tracing a brief history of memory management in the two OSs in section 2. This is followed by the description of a generic memory management system in section 3 which serves as a framework for the comparison which follows in sections 4, 5 and 6. We close with some some notes in section 7.

2 History and Influences

FreeBSD, like NetBSD ([8]) and OpenBSD ([9]) is a 4.4BSD ([14]) derivative. Unlike its cousins, FreeBSD is solely targeted towards Intel x86 hardware[1]. Like the other BSDs though, FreeBSD derives the basic memory management code from 4.4BSD which in turn was based on a Mach [16] like design and experience gained from implementing and running 4.3BSD ([13]). In recent (2.x and 3.x) releases, the FreeBSD team has extensively modified the 4.4BSD code, adding features and making the implementation more efficient.

The influences on the Linux memory management are a lot less clear. To the best of the authors knowledge, early Linux versions had a memory management system which derived from System V release 2 as described in [3] (implementation details can be found in [17] and [4]). In recent (2.x) kernels, the memory management system has been completely revamped. Unlike FreeBSD, Linux runs on multiple hardware architectures including Intel x86.

3 A Generic Memory Management Framework

To make application programming easier, modern OSs allow user processes to address memory in access of that actually available through the popular concept of *virtual memory (VM).* With VM, the available physical memory can be thought of as a cache for data and code fetched from secondary memory. This requires hardware support to allow loading of data and instructions from secondary storage into main memory on the fly. The memory management hardware may export a segmented[2] and or a paged[3] view of memory to software. In addition the hardware implements the logic to translate a virtual memory address presented to it, to a physical memory address and report an exception if the translation is not possible. To handle the idiosyncrasies of different memory management hardware, OSs implement a *hardware address translation (HAT)* layer ([12]) which abstracts away the memory management hardware and exports a hardware independent interface to the rest of the OS. Further they implement a *page level allocator* which sits on top of the HAT and exports a page interface to the *kernel memory allocator* and the *paging system* ([18]). The kernel memory allocator handles dynamic memory allocation for kernel modules and allocates memory to the kernel in addition to that reserved for it at boot time. The paging system is part of the VM system and handles memory allocation for user processes. In addition the paging system may handle memory requests for the (file system) buffer cache. This layered approach to memory management ([1], [18]) is depicted pictorially in figure 1.

A few words about the Intel x86 memory management unit ([10]) are in order before we go on to discussing the other layers. Intel x86 supports both segmentation and paging. The segmentation and paging units are independent of each other allowing for four different organizations. When the memory management hardware is presented with a virtual memory address (VMA), it goes through 2 stages of translations - the first from the VMA to a linear memory address (LMA) obtained by taking the offset from the selected segments base register, and then from the LMA to the physical memory address (PMA) using 2 level page tables. Most Unix implementations run with paging enabled for user processes and disabled for kernel modules. On the other hand, segmentation is enabled for both the kernel and user processes

[1] In recent times, work has started on an Alpha port.

[2] A segment is a variable length chunk of logically related words.

[3] A page is a fixed-length block of words.

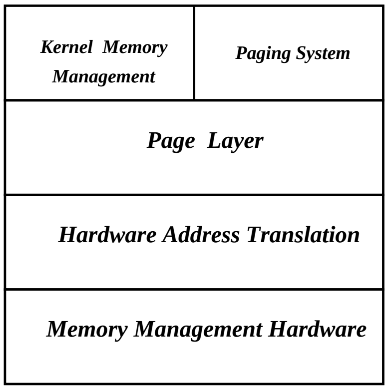

Figure 1: A generic layered Memory Management design

and is used to provide memory protection, kernel entry and context switching. User processes don't see segments, but only a single flat address space with varying protection, as the base address of all the segments is set to 0.

The HAT layer provides an interface using which the higher layers can load and unload translations from an address space to a set of pages and change the protection on these pages, unload translations to a set of pages from a set of address spaces and manipulate the referenced and modified status of a page. This HAT interface is used to enforce various higher level memory management policies independent of the hardware. The HAT itself consists of a machine-independent portion which implements the interface to the page layer and a machine-dependent portion which rolls various machine architectures into a unified and virtual machine model using either a generic data structure or a generic set of functions which are set to machine dependent implementations at compile time.

The page level allocator hands out pages to the kernel memory allocator and the paging system on demand. Since a full page is often an over-kill for temporary storage that a kernel module may need, a kernel memory allocator which implements a policy like the buddy-system or power-of-two allocation ([18]) to handout smaller chunks of memory is included in the kernel. The paging system implements the major VM policies like demand paging and swapping. Detailed descriptions of these concepts are available in literature elsewhere so we'll stick to the differences between the FreeBSD and the Linux implementations which are discussed in section 6.

4 Hardware Address Translation

FreeBSD currently runs only on the x86 architecture. Portability requirements are thus minimal and as such the related HAT functionality can be dispensed with. But since the FreeBSD memory management system derives from Mach ([14], [16]), it inherits the x86 specific part of the *Pmap module*. The Pmap module provides a common abstracted interface across multiple hardware platforms, by requiring a hardware specific implementation of a set of functions. As long as this set of functions is implemented, the rest of the memory management system is independent of the hardware. On the other hand, Linux HAT ([4], [17]) works by rolling in the architecture specific data structures (typically some page table or extended TLB organization) into 3-level kernel page tables at compile time. All operations are then carried out uniformly over this data structure.

5 The Page Layer and Kernel Memory Allocation

Both FreeBSD and Linux use the *buddy-system* in their page level allocators. Older versions of both these OSs used just this for kernel memory allocation too. The buddy-system is not particularly suited for kernel memory allocation as the kernel requires short living memory chunks in odd sizes which causes the buddy-system based allocator to coalesce blocks frequently, slowing down the allocation process ([18]). Due to this, a number of OSs, including FreeBSD and Linux, have moved away from buddy-system based kernel memory allocators.

FreeBSD adopted the *zone* allocator, which runs on top of the buddy-system based allocator in the page layer. Kernel modules register with the zone allocator, specifying the range of memory sizes (i.e. zones) they are likely to ask for at run time. The zone allocator in turn grabs pages from the page layer (allocator) and carves them into smaller ready-to-go chunks for each zone. At run time, most memory requests are satisfied by these preallocated chunks. A memory free request returns the chunk to the free list for the zone the chunk was allocated from. A background garbage collector looks for free pages to return to the page layer. Not all modules in the FreeBSD kernel use the zone allocator – modules which predate the zone allocator, continue to use the buddy-system.

Linux adopted the *slab* allocator ([5]) which is a modified and object-oriented version of the zone algorithm. Kernel modules provide the allocator with a constructor a deconstructor along with the object size. The allocator maintains a cache per object. The constructor is used on the object only when it is brought into the cache and the deconstructor is used when the object is drained out from the cache. Between these two events, the construction/deconstruction overhead is avoided. The slab allocator does not do any garbage collection by itself, but relies on the page layer to reclaim free pages when needed.

Detailed information about all these methods can be found in [5], [14], and [18] and the references therein.

6 The Paging System

Both FreeBSD and Linux implement a VM sub-system based on demand paging and swapping. Most of this functionality lies in the paging system which keeps track of the contents and usage of pages obtained from the page layer. Both OSs maintain a pool of replaceable pages, which is added to by the page replacement module and drained by requests for memory from user processes. Dirty pages are swapped to secondary storage whenever the pool of pages fall below a low-water-mark.

The following paragraphs discuss the subtle differences between the techniques employed by the two OSs:

Swapping On running out of physical memory, FreeBSD swaps out entire idle processes. FreeBSD is able to use any device for swapping. Hence it can swap across NFS, into a Memory File system or into files by using them as devices. Linux on the other hand just freezes some processes when physical memory is in short supply and relies on page replacement to get rid of the dirty pages to a swap partition or into swap files.

Page Replacement FreeBSD uses Global-LRU across all the user pages. FreeBSD also employs page-coloring ([11]) to pick the page to be replaced from the pool of pages available for replacement in order to evenly distribute pages colliding in the cache. Linux uses an approximation to Global Least Regularly Used, biasing page replacement towards pages which have a lower page fault rate.

Contention with the File System As [15] points out, older OSs partitioned physical memory statically between the file system and VM. This often led to suboptimal use of the memory available. Both FreeBSD and Linux have a unified file system buffer cache and VM, with the file system implemented on top of the VM. Some additional memory is needed by the file system to hold its meta-data. This resides in a dynamic cache on both systems.

7 Closing Notes

During the process of writing this article, it became very clear to the authors that comprehensive, reliable and up-to-date documentation on the design and implementation of both FreeBSD and Linux is lacking. But perhaps this is to be expected as the developer community in both the camps is geographically diverse and the projects themselves are largely unfunded and therefore run informally. Given this and the *moving target* nature of the subject of this article, nailing down the salient similarities and differences between the two systems, proved to be very tedious.

Surprisingly, both camps have implemented very similar features in their memory management sub-systems. Again, perhaps this is to be expected – the two communities keep tabs on each other (at some level) and are able to find the time and resources to implement a feature which has proven its worth in the other camp or in the research community.

Acknowledgments

We would like to thank Shamik Sharma and Anurag Acharya for discussions on general VM concepts, David Greenman, John Dyson and Thomas Graichen for help with FreeBSD specific issues and Douglas Jardine and Michael Johnson for help with Linux specifics. We also thank the Department of Computer Science at the University of Maryland, College Park for letting us use their computing facilities to prepare this article.

Disclaimers

Any opinions in this article are the authors alone and do not represent the opinions of any of the authors previous or present employers. Further this article does not reflect the views of any of the net-projects discussed. This article was written in September 1997 but published a year later.

References

[1] V. Abrossimov, M. Rozier, and M. Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *SOSP*. ACM, 1989.

[2] D. Anderson and T. Shanley. *Pentium Processor System Architecture*. Addison-Wesley, second edition, 1995.

[3] M.J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[4] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 1996.

[5] J. Bonwick. An Object-Caching Memory Allocator. In *Summer Technical Conference*. USENIX, 1994.

[6] Various Contibutors. The FreeBSD Project. Sources and Information available from http://www.freebsd.org.

[7] Various Contibutors. The Linux Project. Sources and Information available from http://www.linux.org.

[8] Various Contibutors. The NetBSD Project. Sources and Information available from http://www.netbsd.org.

[9] Various Contibutors. The OpenBSD Project. Sources and Information available from http://www.openbsd.org.

[10] J.P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, second edition, 1988.

[11] R.E. Kessler and M.D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM TOCS*, 10(4), 1992.

[12] Y.A. Khalidi, V.P. Joshi, and D. Williams. A Study of the Structure and Performance of MMU Handling Software. Technical Report SMLI TR-94-28, SUN Microsystems Laboratories, Inc., 1994. Available at http://www.sunlabs.com/technical-reports/1994/smli_tr-94-28.ps.

[13] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.R. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1988.

[14] M.K. McKusick, K. Bostic, M.J. Karels, and J.R. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996. The chapter on Memory Management is co-authored by M. Hibler.

[15] M.N. Nelson. Virtual Memory vs The File System. Technical Report 4, DEC Western Research Laboratory, 1990. Available at ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-90.4.ps.

[16] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *ASPLOS*. ACM, 1987.

[17] D.A. Rushling. The Linux Kernel. Draft Version 0.1-10(30) dated April 1997. Available at http://sunsite.unc.edu/mdw/LDP/tlk.

[18] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.