SEVER INSTITUTE OF TECHNOLOGY

DOCTOR OF SCIENCE DEGREE

DISSERTATION ACCEPTANCE

(To be the first page of each copy of the dissertation)

DATE: July 24, 1998

STUDENT'S NAME: Charles D. Cranor

This student's dissertation, entitled Design and Implementation of the UVM Virtual Memory System has been examined by the undersigned committee of five faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Doctor of Science.

APPROVAL: ______________________________ Chairman

____

____

____

____

Short Title: Design and Implementation of UVM Cranor, D.Sc. 1998

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DESIGN AND IMPLEMENTATION OF THE UVM VIRTUAL MEMORY SYSTEM

by

Charles D. Cranor, M.S.

Prepared under the direction of Professor Gurudatta M. Parulkar

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 1998

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

DESIGN AND IMPLEMENTATION OF THE UVM
VIRTUAL MEMORY SYSTEM

by Charles D. Cranor

ADVISOR: Professor Gurudatta M. Parulkar

August, 1998
Saint Louis, Missouri

We introduce UVM, a new virtual memory subsystem for 4.4BSD that makes better use of existing hardware memory management features to reduce overhead and improve performance. Our novel approach focuses on allowing processes to pass memory to and from other processes and the kernel, and to share memory. This approach reduces or eliminates the need to copy data thus reducing the time spent within the kernel and freeing up cycles for application processing. Unlike the approaches that focus exclusively on the networking and inter-process communications (IPC) subsystems, our approach provides a general framework for solutions that can improve efficiency of the entire I/O subsystem.

Our primary objective in creating UVM was to produce a virtual memory system that provides a Unix-like operating system kernel's I/O and IPC subsystems with efficient VM-based data movement facilities that have less overhead than a traditional data copy. Our work seeks to:

- allow a process to safely let a shared copy-on-write copy of its memory be used either by other processes, the I/O system, or the IPC system;

- allow pages of memory from the I/O system, the IPC system, or from other processes to be inserted easily into a process' address space; and
- allow processes and the kernel to exchange large chunks of their virtual address spaces using the VM system's higher-level memory mapping data structures.

UVM allows processes to exchange and share memory through three innovative new mechanisms: page loanout, page transfer, and map entry passing. We present test results that show that our new VM-based data movement mechanisms are more efficient than data copying.

UVM is implemented entirely within the framework of BSD and thus maintains all the features and standard parts of the traditional Unix environment that programmers have come to expect. The first release of UVM in NetBSD runs on several platforms including I386-PC, DEC Alpha, Sun Sparc, Motorola m68k, and DEC VAX systems. It is already being used on systems around the world.

to my family, *especially* Lorrie

Contents

List of Tables

List of Figures

Acknowledgments

Designing, implementing, and testing a software system as large and complex as UVM was quite a challenge. I would never have been able to complete such a task without support from my colleagues, friends, and family. Thus, I am pleased to have the opportunity to thank them in my dissertation.

I would first like to thank my advisor Dr. Gurudatta Parulkar for encouraging me to attend Washington University and for providing funding throughout my years in St. Louis (even during the “hard times”). Having an advisor who is accessible, easy to communicate with, and supportive certainly has made my life easier.

I would also like to thank the rest of my committee for taking the time to learn about my work and providing me with valuable feedback on how it could be improved. In addition to providing feedback on my research, my committee has also been a source of inspiration. Dr. George Varghese’s constant enthusiasm for research and my work helped me keep my spirits up throughout this project. Dr. Douglas Schmidt’s boundless energy and views on how to sell systems research helped me improve the presentation of my work by teaching me to look at it from new points of view. Dr. Ron Cytron’s natural curiosity and willingness to examine research outside of his primary area of interest showed me the value of being open minded. Finally, Dr. Ronald Minnich taught me the value of patience and perseverance when working on large software projects.

Several BSD developers helped in the latter stages of this project. I would like to thank Matthew Green for helping to write and debug UVM’s swap management code. Matt also coordinated the integration of UVM into the NetBSD source tree. Thanks to Chuck Silvers for writing the aobj pager for me and helping with debugging. Finally, thanks to Jason Thorpe for helping to port UVM to new platforms.

There are a number of people who contributed indirectly to the success of the UVM project. I would have never gone to graduate school if it was not for my involvement in undergraduate research at University of Delaware. I credit Professor David Farber for allowing me to join his research group “F-Troup.” My positive experiences as a member of F-Troup convinced me that I wanted to continue my education beyond my undergraduate degree. I would like to thank all my friends from University of Delaware.

Washington University's Computer Science Department, Computer and Communications Research Center (CCRC), and Applied Research Lab (ARL) have been a wonderful environment in which to work. The three key ingredients to this environment are the faculty, staff, and students. The faculty has contributed to this environment by obtaining research funding for graduate students, teaching interesting courses, and mentoring their students. I would particularly like to thank Dr. Turner, Dr. Cox, Dr. Franklin, and Dr. Chamberlain for their roles in forming and maintaining the CCRC and ARL research labs.

The CS, CCRC, and ARL office staff have helped me maintain my sense of humor while I have made my way through the administrative red tape that comes with a university bureaucracy. I would like to thank Jean, Sharon, Myrna, Peggy, Paula, Diana, Vykky, and the rest of the staff for their assistance during my years at Washington University.

I would like to thank my fellow students for their friendship during my stay in St. Louis. First, I would like to thank the members of my research group. Thanks to Milind Buddhikot, Zubin Dittia, R. Gopalakrishnan (Gopal), and Christos Papadopoulos. Milind kept me entertained through his tales of woe and wonder and also by leaving his keys and wallet at random locations throughout CCRC (despite my best efforts to break him of this habit). Zubin taught me the value of "great fun" by being willing to drop all his work at *any* time on short notice so that we could go watch a movie. Gopal shared both his wit and his technique for giving technical presentations (the "freight-train approach"). Christos went from spending a year car-less to becoming our research group's top auto mechanic, and I thank him for all his service. I would also like to thank Anshul Kantawala for coining the often-used phrase "you fool!" and Hari Adiseshu for letting me resolve some of his technical doubts. Thanks to the rest of the group (James, Tony, Sanjay, Jim, Vikas, Deepak, Larry, Hossam, Marcel, Dan, Jane, Woody, Sherlia, Daphne, Yuhang, and Ed) for their support.

Thanks to Sarang Joshi and Gaurav Garg for helping me get through my first year in St. Louis. Thanks to Greg Peterson for helping me to arrange the "phool" movie nights. Thanks to Brad Noble for the great conversations, keeping Rob in control, and introducing Elvis to the back hallway. Thanks to Giuseppe Bianchi (Geppo) for pulling off the best practical jokes and not making me a target. Thanks to Rex Hill for introducing me to my wife and for helping Zubin move the APIC a little closer to reality. Thanks to Shree for teaching me about the "junta." And thanks to Maurizio Casoni, Girish Chandranmenon, Apostolos Dailianas, Andy Fingerhut, Rob Jackson, Steve von Worley, and Ellen Zegura for making the back hall a more interesting place.

Thanks to Ken Wong for letting me help manage the CCRC systems and for being easy and fun to work with. Thanks to John deHart and Fred Kuhns for helping with the ARL equipment that I used in my research.

I would like to thank AT&T-Research for allowing me to use their summer student space during the winter. In particular, I would like to thank Paul Resnick and Brian

LaMacchia for allowing me to tag along with their group when I was not in the best of moods (having had to move to New Jersey without completing UVM). I would also like to briefly thank the following people for their support: Kalpana Parulkar, Nikita Parulkar, Betsy Cytron, Jessica Cytron, Melanie Cytron, Maya Gokhale, Austin Minnich, Amanda Minnich, Luke Beegle, Chris Handley, Pam Perkins, Erik Perkins, Scott Hassan, Penny Noble, Theo de Raadt, AGES, Joe Edwards, Dr. Hurley, Elvis Presley, H.P. Lovecraft, and the authors of the BSD games in `/usr/games`.

Finally, I would like to thank my family to whom this dissertation is dedicated. My wife Lorrie has been a constant source of support throughout the development of UVM. Her feedback on both the written and oral presentation of UVM helped me present complex ideas in a way that is clear, concise, and readable. I would like to thank my parents, Morris and Connie Cranor, for raising and supporting me. I would not be where I am today without their support and I am proud to honor them with my accomplishments. I would like to thank Lorrie's parents, Drs. Michael and Judy Ackerman, for accepting me into their family and for their encouragement and understanding during my career as a doctorial student. Finally, I would like to thank the rest of my family, including (but not limited to) my sister Ginny, my brother-in-law Jeremy, and all eight of my grandparents for their support.

Charles D. Cranor

Washington University in Saint Louis
August 1998

Chapter 1

Introduction

New computer applications in areas such as multimedia, imaging, and distributed computing demand high levels of performance from the entire computer system. For example, multimedia applications often require bandwidth on the order gigabits per second. Although hardware speeds have improved, traditional operating systems have had a hard time keeping up [32, 50]. One major problem with traditional operating systems is that they require unnecessary data copies. Data copies are expensive for a number of reasons. First, the bandwidth of main memory is limited. Each copy made of the data is effected by this. Second, to copy data, the CPU must move it word-by-word from the source buffer to the destination. While the CPU is busy moving data it is unavailable to the user application. Finally, data copying often flushes useful information out of the cache. since the CPU accesses main memory through the cache, the process of copying data fills the cache with the data being copied — displacing potentially useful data that was resident in the cache before the copy.

Applications that transmit data from disk storage over a network are an example of a class of applications that suffer from unnecessary data copying. Applications in this class include video, file, and web servers. Figure 1.1 shows the path the data in this class of application takes in a traditional system. First the server issues a read request. To satisfy this request, the kernel first has the disk device store the data into a kernel buffer in main memory. To complete the read request the kernel copies the data from the first kernel buffer into the buffer the user specified. Note that this data copy traverses the MMU and cache. Once the data has been read, the server application then immediately writes the data to the network socket (without even accessing the data). This causes the kernel to copy the data from the user buffer into another kernel buffer that is associated with the networking system. The kernel can then program the network interface to read the data directly from

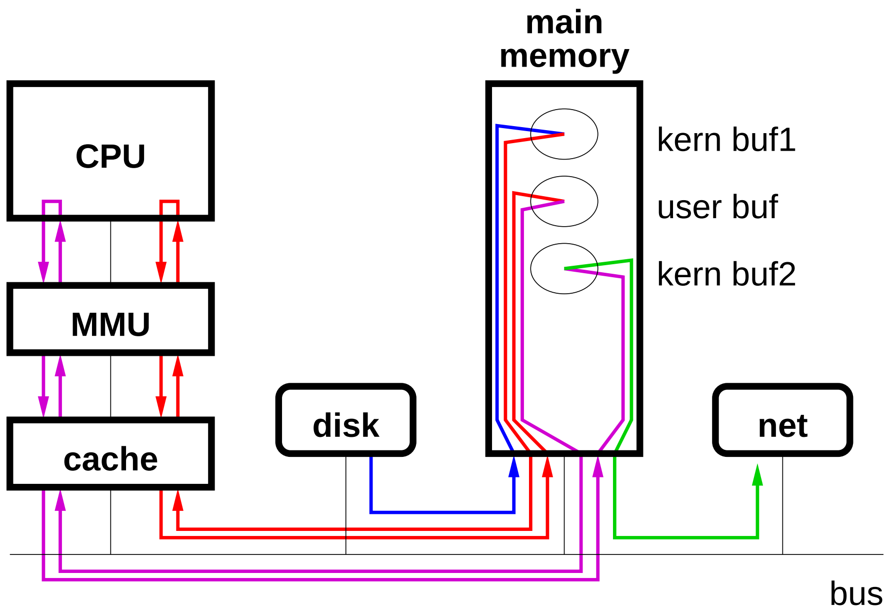


Figure 1.1: Transmitting disk data over a network

the second kernel buffer and then transmit it. This operation requires two processor data copies; however if the operating system used only one buffer instead of three, then no processor data copies would be required to read and transmit the data.

A variety of efforts have sought to improve system performance by increasing the efficiency of data flow through the operating system. Most of these efforts have been centered on adapting the I/O and networking subsystems of the operating system to reduce data copying and protocol overhead [4, 24, 27, 53]. These approaches tend to involve superficial modifications of the virtual memory subsystem that do not involve changing the underlying design of the system. This is understandable because the virtual memory subsystem is typically much larger and more complex than the networking subsystem. However, substantial gains can be made by changing the design of the virtual memory subsystem to exploit the ability of the hardware to map a physical address to one or more virtual addresses.

This research seeks to modify a traditional Unix-like operating system's virtual memory subsystem to allow applications to better take advantage of the features of the memory management hardware of the computer. Our novel approach focuses on allowing processes to pass memory to and from other processes and the kernel, and to share memory. The memory passed or shared may be a block of pages, or it may be a chunk of virtual memory space that may not be entirely mapped. The memory may or may not be pageable. This approach reduces or eliminates the need to copy data thus reducing the time spent within the kernel and freeing up cycles for application processing. Unlike the approaches that focus exclusively on the networking and inter-process communications (IPC) subsystems, our approach provides a general framework for solutions that can improve efficiency of the entire I/O subsystem.

Modifying the kernel's virtual memory system is more of a challenge than working with other parts of the kernel (e.g. writing a device driver) for several reasons. First, the virtual memory system's data structures are shared between the kernel and the processes running on the system and thus must be properly protected for concurrent access with locking. Second, the VM system must be able to handle both synchronous and asynchronous I/O operations. Managing asynchronous operations is more difficult than synchronous operations. Third, errors in the VM system can be harder to diagnose than in other kernel subsystems because the kernel itself depends on the VM system to manage its memory (thus if an error occurs it may not be possible to get the kernel into a debugable state) and also because the VM system is typically in the process of handling many service requests at once. Fourth, the BSD VM system is poorly documented when compared to other kernel subsystems (for example, the networking stack has several books describing its operation while the BSD VM system is only documented in a few book chapters and Mach papers). In short, the virtual memory system is both large and complicated and has vital components that all other parts of the kernel depend on to function properly.

We selected NetBSD [54] as the platform for this work. NetBSD is a descendant of the well known 4.4BSD-lite operating system from University of California, Berkeley [39]. NetBSD is a multi-platform operating system that runs on systems such as i386 PCs, DEC Alphas, and Sun Sparc systems. It is currently being used by numerous institutions and individuals for advanced research as well as recreational computing. NetBSD is an ideal platform for university research because the source code is freely available on the Internet and well documented in books and papers, and the TCP/IP stack is the standard BSD TCP/IP that is used throughout the world. NetBSD is also closely related to other free BSD operating systems such as FreeBSD [25] and OpenBSD [55], thus allowing applications to easily migrate between these systems. When we began work on this project the NetBSD virtual memory subsystem was largely unchanged from its roots in 4.4BSD.

The primary purpose of a virtual memory system is to manage the virtual address space of both the kernel and user processes [18, 19]. A virtual address space is defined by a map structure. There is a map for each process (and the kernel) on the system, consisting of entries that describe the locations of memory objects mapped into the virtual address space. In addition, the virtual memory system manages the mappings of individual virtual memory pages to the corresponding physical page that belongs to each mapped memory object. The virtual memory system is responsible for managing these mappings so that the data integrity of the operating system is maintained.

In this dissertation we introduce UVM [1], a new virtual memory system based on the 4.4BSD VM system. UVM allows processes to exchange and share memory through three innovative new mechanisms: page loanout, page transfer, and map entry passing.

- In page loanout, a process can loan its memory to other processes or to non-VM kernel subsystems. This is achieved with a reference counter and special handling of kernel memory mappings. Special care is taken to handle events such as write faults and pageouts that could break the loan. Applications that transfer data over a network can take advantage of this feature to loan data pages from the user's address space to the kernel's networking subsystem. In our tests, single-page loanouts to the networking subsystem took 26% less time than copying data. Tests involving multi-page loanouts show that page loaning can reduce the processing time further, for example a 256 page loanout took 78% less time than copying data.

- In page transfer, a process can receive pages of memory from other processes or the kernel. Once a page has been transfered into a process' address space it is treated like any other page of a process' memory. Page transfer is achieved without the need for complex memory object operations that occur in BSD VM. Our tests show that single page transfers do not show an improvement over data copying, but a two-page transfer shows a 22% reduction in processing time. The processing time decreases as larger chunks of pages are transfered, for example a eight-page page transfer reduces processing time by 57%.

- In map entry passing, a process exports a range of its address space to other processes or the kernel by allowing a chunk of its map entry structures to be copied. This map entry copying can be done in such a way that the exporting process can retain, share, or surrender control of the mappings. Map entry passing can be used to quickly transfer memory between processes in ways that were not possible with the BSD VM system. Single-page map entry passing tests do not show any improvement over data copying, however two-page map entry passing shows a 30% reduction in processing time. Processing time decreases further when larger blocks of data are

[1] Following Larry Wall [70] we provide no definitive explanation for what UVM stands for. Expanding this acronym is left as an exercise for the reader.

passed, for example an eight-page transfer of memory decreases processing time by 69%.

UVM also includes well-documented changes to the BSD VM system that improve the accessibility of the code and the overall performance and stability of the operating system:

- The old anonymous memory handling system that used "shadow" and "copy object chains" is completely replaced by a simpler two-layer system. Anonymous memory is memory that is freed as soon as it is no longer referenced.

- The new anonymous memory pageout clustering code can improve anonymous memory pageout by a factor of six.

- The new pager interface gives the pager more control over its pages and eliminates the need for awkward "fictitious" device pages.

- The new memory object handling code eliminates the need for the redundant "object-cache" and allows vnode-based objects to persist as long as their backing vnodes are valid.

- Kernel map fragmentation has been reduced by revising the handling of wired memory.

- The code to handle contiguous and non-contiguous memory has been merged into a single interface thus reducing the complexity of the source code.

- Redundant virtual memory calls have been eliminated in the fork-exec-exit path, thus reducing the processing time of a fork operation. For example, the processing time of a fork operation on a small process has been reduced by 13% under UVM.

- A new system call allows unprivileged process to get the virtual memory system status. UVM is distributed with an X window program that uses this system call to display virtual memory system status graphically.

- The new i386 pmap eliminates "ptdi" panics and deadlock conditions that plagued the old pmap in some cases.

UVM has been designed to completely replace the BSD VM system in NetBSD. It was incorporated into the master NetBSD source tree in February of 1998 and has proven stable in NetBSD's Alpha, ARM32, Atari, HP300, I386, MAC68k, MVME68k, NewsMips, PC532, PMAX, Sparc, and VAX ports. It will appear in NetBSD release 1.4.

The rest of this dissertation is organized as follows: In Chapter 2 we present background on the VM subsystem. In Chapter 3 we detail the overall UVM design and describe related work. We describe anonymous memory handling, our new page fault handler, our new pager interface, and the page/map loanout interface in Chapters 4, 5, 6, and 7 respectively. We describe secondary design elements in Chapter 8. In Chapter 9 we introduce our implementation methods. Finally we present our results in Chapter 10 and our conclusions and suggestions for future work in Chapter 11. Definitions of key terms can be found in the glossary.

Chapter 2

Background

This chapter describes the traditional role of the virtual memory subsystem. It explains the history and evolution of the BSD VM and presents an overview of the current BSD VM system as background for understanding the UVM system.

2.1 The Role of the VM System

The operating system kernel manages hardware resources — including memory, devices, and peripherals — and schedules when processes will run. The kernel has a variety of subsystems including I/O, networking, file systems, process management, and virtual memory. The job of a kernel's virtual memory sub-system is to manage access to the physical memory chips where data is stored while in use.

When a process is run, the VM subsystem first sets up the process' virtual address space. As the process executes, each instruction must be fetched from virtual memory. This requires the memory management unit (MMU) — a hardware component — to translate a virtual address into a physical memory location where the instruction is actually stored. Sometimes the MMU is unable to complete the translation because the virtual address has not been mapped. In this case the MMU signals a fault and the virtual memory sub-system is called again. The VM either establishes a mapping or signals a "segmentation violation." The VM sub-system is also called as part of some systems calls.

The virtual memory subsystem allows processes to use more memory than physically available by using a computer's disk to store data that does not fit in memory. Because accessing data on disk is slower than accessing data in physical memory, the VM system selects data that has not been recently accessed for disk storage. The disk space used for storing memory data is known as "backing store." Entire programs can reside in backing store; the VM system will load needed parts of the program into physical memory as the program executes.

Another function of the VM system is to allow each process to have its own virtual address space. As a result, programs can be loaded at a fixed virtual address without having to check whether that address is in use by another process. Also, since each process' virtual address space is private, the VM system allows processes to access their entire range of allocated memory without regard for other processes memory usage.

Since all processes requires the use of physical memory, managing memory access is a complex task. Key operations that the virtual memory system performs include:

- Allocation of a computer's physical memory. This is done by keeping track of which pages of physical memory are allocated and which are free, and allocating free pages as they are needed by the kernel and other programs.

- Allocation of each process' virtual address space. This is done by keeping a list of all allocated regions in each virtual address space and allocating free regions as needed by the process.

- Mapping physical pages into virtual address space. This is done by programming the computer's MMU to map physical pages into one or more virtual address spaces with the appropriate protection.

- Handling invalid memory accesses (i.e. page faults). This is done through the VM system's page fault handler. Page faults can happen when unmapped memory is referenced, or when memory is referenced in a way that is inconsistent with the current protection on the page.

- Moving data between physical memory and backing store. This is done by making requests to the I/O system to read and write data into physical pages of memory.

- Managing memory shortages on the system. This is done by freeing unused and inactive pages of physical memory (saving to backing store if necessary).

- Duplicating the address space of a process during a fork. This can include changing protection of a virtual memory region to allow for "copy-on-write." In copy-on-write the VM subsystem write-protects the pages so that they will be duplicated when modified [9].

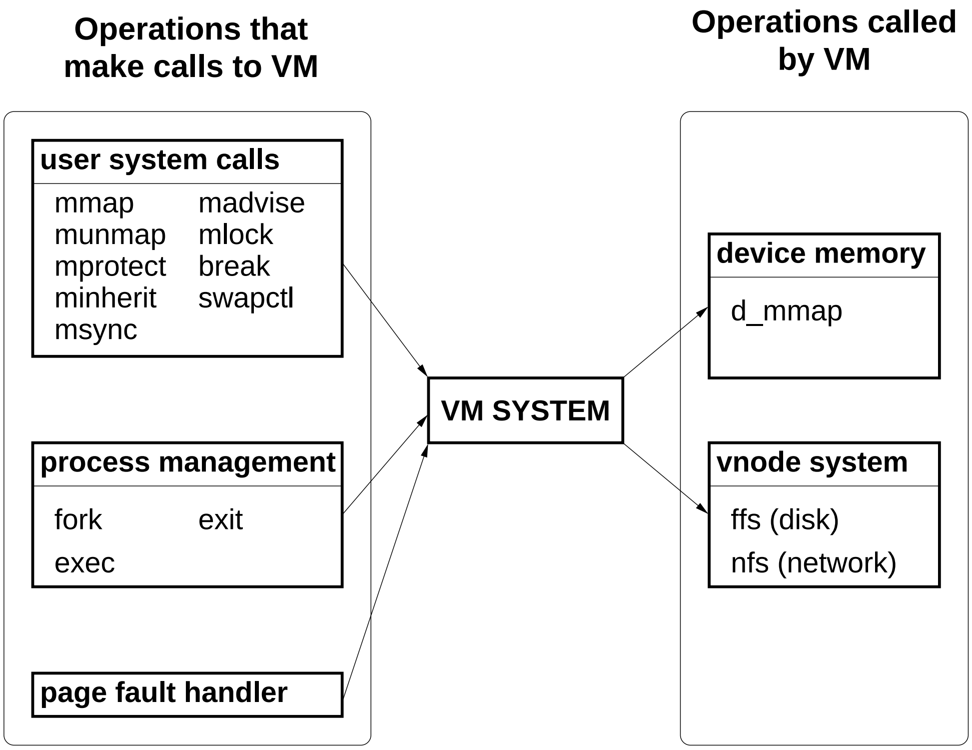


Figure 2.1: The role of the VM system

The role of the VM system is shown in Figure 2.1. As shown on the left side of the figure, the services of the VM system are requested in three ways. First, system calls such as `mmap` and `mprotect` may be called by a process to change the configuration of its virtual memory address space. Second, virtual memory process management services are called by the kernel when a process is in the midst of a `fork`, `exec`, or `exit` operation. Finally, the VM system is called when a page fault occurs to resolve the fault either by providing memory for the process to use or by signaling an error with a "segmentation violation."

In the course of its operation the VM system may need to request the services of other parts of the kernel in the form of an I/O request. The VM system makes two types of I/O requests, as show in the right half of Figure 2.1. For devices such as frame buffers that allow their device memory to be memory mapped (with the `mmap` system call), the device's `d_mmap` function is called to convert an offset into the device into a physical address that can be mapped in. For normal files the vnode system is called to perform I/O between the VM system's buffers and the underlying file. As shown in the figure, the underlying file may reside on a local filesystem or it may reside on a remote computer (and is thus accessed via the network file system). Note, because it is possible for these I/O systems to make requests back into the VM system, care must be taken to avoid potential deadlock.

2.2 The VM System and Process Life Cycle

Understanding the intimate details of the operation of the VM system is a difficult task. However, after getting an understanding of the role of the VM system, a good next step to understanding the overall structure of the VM system is to examine the role of the VM system through the life cycle of a process. A good process to start with is the "init" process.

2.2.1 Process Start-up

The init process is the first user-level process created by the kernel after being bootstrapped. When starting init, the kernel creates a new user virtual address space for init that contains no mappings (i.e. it is completely unmapped). Before the kernel can let init run it must map data into the address space. Data is mapped from the file `/sbin/init`. The rest of the init process' memory is left unmapped or zeroed. The file `/sbin/init` contains the following items:

text: the compiled code for the init program. The text starts off with a special header data structure[1] that indicates the size of each segment of the program and the starting address of the first machine instruction of the program.

data: the pre-initialized variable data for the program. For example, if init declared a global variable "x" like this: "`int x = 55;`" then the value 55 would be stored in the data area of the `/sbin/init` file.

symbols: the symbols stored in the program are used for debugging and are often stripped out of system programs such as init. They are ignored by the kernel.

Zeroed memory is a form of "anonymous memory" that is initialized or "filled" with zeros when it is first referenced. Anonymous memory is memory that is not part of a file, and thus has no filename associated with it. The modified pages of a copy-on-write mapped region are also anonymous. The init program's zeroed memory is divided into two parts: the uninitialized data ("bss"[2]) and the stack. The uninitialized data area is where

[1]The format of the header data structure varies depending on what system you are on. Two popular formats for this header are the "a.out" format and the ELF format.

[2]"bss" is short for "block started by symbol" — a term no longer commonly used.

Table 2.1: The init program's virtual address space attributes

Region	Protection	Copy-on-write	backing object
text	r/o	yes	`/sbin/init`
data	r/w	yes	`/sbin/init`
bss/heap	r/w	—	<zero-fill>
stack	r/w	—	<zero-fill>

global data variables that have not been assigned a value are stored. Also, if the program does dynamic memory allocation with `malloc` the bss area is often expanded to provide memory for `malloc` to manage[3]. This expanded area is called the "heap." The stack area is where the program's stack, including local variables, is stored.

When the kernel prepares the init process for execution it must use the VM system to establish the virtual memory mappings of the text, data, bss, and stack areas. This is part of the "exec" process of a process' life cycle. In an exec system call, the kernel opens the file to be run and reads the header to determine the size of the text, data, and bss areas. It then uses the VM system to establish virtual memory mappings for these areas, as shown in Figure 2.2. Each of the four mappings has three main attributes as shown in Table 2.1: the backing object, the copy-on-write status, and the protection. The backing object indicates what object is mapped into the space. This is usually a file or "zero-fill" memory which means that the area should be filled with zeroed memory as it is accessed. The copy-on-write status indicates whether changes made to the memory should be reflected to the backing object (if it is not zero-fill). If part of a file is memory mapped without copy-on-write ("shared mappings") then changes to that portion of memory will cause the file itself to be changed. For zero-fill memory, the copy-on-write flag doesn't matter. Finally, the protection is usually one of read-only or read-write, but it can also be set to "none." The MMU is used to map pages in the allocated ranges of virtual address space to their corresponding physical address.

When the kernel does an exec call on `/sbin/init` the following mappings are set up:

text: The text memory area is a read-only mapping of the text part of the `/sbin/init` file. The mapping is marked copy-on-write in case the program is run under a debugger (in which case the debugger may change the protection of the text to read-write

[3]Some newer versions of `malloc` allocate memory using `mmap` rather than expanding the bss area.

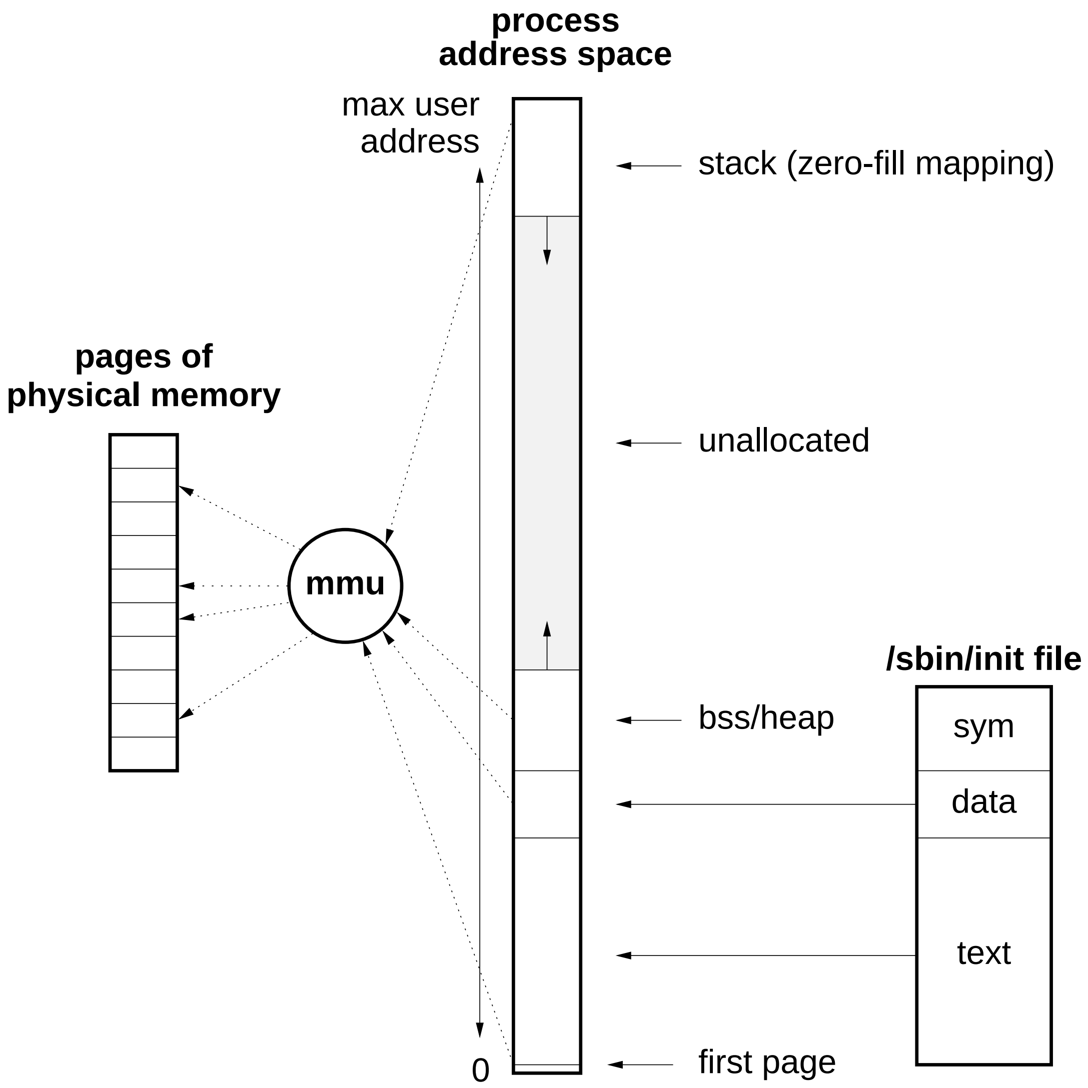


Figure 2.2: The init program's virtual memory layout. The stack and bss/heap mappings are zero-fill, while the text and data mappings are taken from the `/sbin/init` file. The MMU maps addresses in the virtual address space to their corresponding physical address.

to insert and remove break-points). Under normal operation a process never writes to the text area of memory.

data: The data memory area is a read-write copy-on-write mapping of the data part of the `/sbin/init` program. This means that the initial values of the data area are loaded from the file, but as the data is changed the changes are private to the process. Thus, if other users run the program, they will get the correct initial values in their data segment.

bss: The bss memory area is mapped as a zero-fill area after the data memory area.

stack: The stack memory area is mapped as a zero-fill area at the end of the user's address space.

Note that there is a very large gap of free virtual address space between the end of the bss area and the beginning of the stack that can be used by the process or the operating system to map in other files and data as needed. For programs that use shared libraries this area is used by the dynamic linker ("`ld.so`") at run time to `mmap` in the shared library files. In the case of init we assume it is statically linked, and thus does not need to map in any extra files.

2.2.2 Running Processes and Page Faults

Once the kernel has established the mappings for init, it begins the init program execution at the starting address specified in the file's header. The processor hardware will try and execute the first instruction of init only to discover that no physical page of memory is mapped in at the starting virtual address. This will cause a page fault to occur and the VM system's page fault handler will be called. The steps in the page fault process are as follows:

1. The process accesses a memory address that is not mapped (or improperly mapped) in to its virtual address space. The processor's memory management until generates a page fault trap. At this point the process is frozen unit the page fault is resolved.

2. The processor-specific part of the kernel catches the page fault and uses the MMU to determine the virtual address that triggered the page fault. It also determines the access type of the fault by checking the hardware to see if the process was reading from or writing to the faulting address.

3. The VM system's page fault routine is called with information on the faulting process and the address and access type of the fault.

4. The VM system looks in the process' mappings to see what data should be mapped at the faulting address (for example, the fault address could be mapped to the first page of text of the `/sbin/init` program).

5. If the fault routine finds that the process' address space doesn't allow access to the faulting address, then it returns an error code, which under Unix-like operating systems causes the process to get a "segmentation violation" error.

6. On the other hand, if the fault routine finds that the memory access was valid, it loads the data needed by the process into a page of physical memory and then has the MMU map that page of memory into the process' address space, as shown in Figure 2.2. This is referred to as faulting in data. The fault routine then returns "success" and the process is unfrozen and can resume running with the needed memory mapped into its address space.

The init program starts by faulting in the first page of text from the `/sbin/init` file. It then continues running, faulting in more pages of text, data, bss, and stack. As shown in Figure 2.3, regions mapped copy-on-write require special handling by the fault routine to ensure that the underlying executable file's memory is not modified by a write to copy-on-write mapping of it. Copy-on-write pages start off in the "pages un-written" state. If a process has a read-fault on a copy-on-write page, then that page is fetched from the backing file and mapped in read-only (thus protecting the object from being changed). If the process has a write-fault on the page (either because it was not mapped at all, or because it was mapped read-only due to a previous read-fault), then a copy of the backing object's page is made. The new copy of the page is then mapped in read-write, replacing any previous read-only mappings of the data. At that point the page is in the "written" state. Once the page is written, all future read and write faults cause the page to be mapped in read-write. Such faults can be caused by the pagedaemon paging out the "written" page. Pages in the written state stay in that state until they are freed or they are part of a copy-on-write fork operation (described below).

2.2.3 Forking and Exiting

When the system is booted, init is the only process on the system. In order to allow others to use the system, init needs to create more processes by using the `fork` system call. The fork

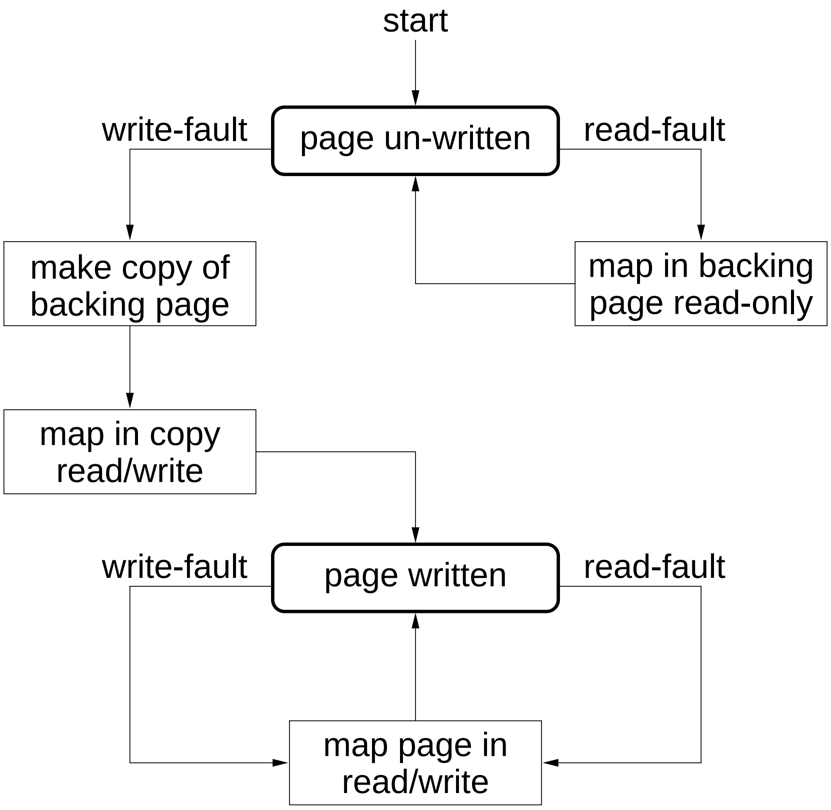


Figure 2.3: Handling copy-on-write pages when read and write faults occur

system call takes a process and creates a child process that has a copy of the parent process' virtual address space. The VM system handles the duplication of the parent's address space. If the VM system actually copied every page in the parent's address space to a new page for the child process, a fork operation would be very expensive (because it takes a long time to move that much data). The VM system avoids this by using the copy-on-write feature again. For example, when the init program forks, all the pages that are copied to the child process are put into the "pages un-written" state, as shown in Figure 2.3.

Typically, after doing a fork operation, the child process will do another exec operation on a different file (for example, `/bin/sh`). In preparing for the exec operation, the kernel will first ask the VM system to unmap all active regions in the process doing the exec. This will cause the process to return to a completely unmapped state. Then the kernel will map the program into the address space, as previously described above.

Finally, when the process exits, the VM system removes all mappings from the process' address space. Handling all the cases that can occur with copy-on-write and shared memory, and recovering from error conditions add to the complexity of the VM system.

2.2.4 VM Operations

Other VM operations that the process may perform while running include the following system calls:

`break`: changes the size of a process' heap area.

`mmap`: memory maps a file or anonymous memory.

`munmap`: removes the mapping of a file or anonymous memory.

`mprotect`: changes the current protection of a mapping. Each mapped region has a "maximum allowed protection" value controlled by the kernel that keeps a process from gaining unauthorized access to a block of memory.

`minherit`: changes the inheritance of a mapped region. The inheritance attribute is used during a fork to control the child process' access to the parents memory. It can be either "copy," "share," or "none."

`msync`: flushes modified memory back to backing store. This is used when a file is mapped directly into a process' address space and the process wants to ensure that the changes it has made to the file's memory are saved to disk.

`madvise`: changes the current usage pattern of the memory region. The advice is a hint from a process to the VM system as what access pattern will be used to access the memory. Access patterns include: normal (the default), random, sequential, "will need," and "won't need."

`mlock`: locks data into physical memory. This ensures timely access to memory by forcing the data to stay resident.

`swapctl`: configures the swap area. The system manager of a system uses `swapctl` to add and remove swap space from the system.

In normal operation the majority of the VM system's time is spent resolving page faults and adding, removing, and copying mappings in process' virtual address space. As the system runs it is likely that more and more of the pages of physical memory will be allocated for use by the operating system. When the number of free pages drops below a threshold, then the VM system causes the "pagedaemon" to run. The pagedaemon is a special system process that looks for pages of memory that are allocated but haven't been used for a while and adds them back to the free list. If a page contains modified data, it has to be flushed out before it can be freed. The pagedaemon manages this flushing process.

2.3 The Evolution of the VM System In BSD

The VM system has changed quite a bit throughout the evolution of the BSD operating system. This evolution is shown in Figure 2.4. Early BSD VM started with a VAX VM system [36]. This VM system was tightly bound to the memory management features provided by the DEC VAX processor. Thus, when porting BSD to non-VAX processors, it was important to make the processor's VM hardware emulate the VAX as closely as possible. For example, SunOS 3.x which ran on Motorola processors did this. Making a non-VAX system emulate a VAX is quite cumbersome and inefficient. In addition to being non-portable, the BSD VAX VM did not support memory mapped files, thus complicating the implementation of shared libraries. So, Sun threw out the early BSD VAX VM system and designed a new one from scratch. This VM system appeared in SunOS 4 and its descendant is currently used in Solaris [28, 46]. In the mean time a new operating system project called Mach was started at Carnegie-Mellon University (CMU) [72, 57, 71, 64]. The virtual memory system of Mach had a clean separation between its machine-dependent and machine-independent features. This separation is important because it allows a virtual memory system to easily support new architectures. The Mach 2.5 VM system was ported to BSD thus creating the BSD VM system [39, 69]. This VM system appeared in 4.3BSD Net2. Eventually the CMU Mach project came to a close and 4.4BSD-Lite was released in source code form. A number of operating system projects are based on the 4.4BSD-Lite code, and thus use the BSD VM system. As time has gone on the BSD VM system has diverged into three branches: the BSDI branch [33], the FreeBSD branch [25], and the NetBSD/OpenBSD branch [54, 55]. The BSDI branch and the FreeBSD branch have stuck with the basic BSD VM concepts imported from Mach, however they have worked to improve the performance of the BSD VM in several areas. The NetBSD/OpenBSD branch of the VM system is not as much changed from the original 4.4BSD VM code as the other branches.

UVM's roots lie partly in the BSD VM system from the NetBSD/OpenBSD and the FreeBSD branches, and also with the SunOS4 VM system. UVM's basic structure is based on the NetBSD/OpenBSD branch of the BSD VM system. UVM's new i386 machine-dependent layer includes several ideas from the FreeBSD branch of BSD VM. UVM's new anonymous memory system is based on the anonymous memory system found in SunOS4. UVM has recently been added to NetBSD and will eventually replace the BSD VM. There are also plans to add UVM to OpenBSD as well.

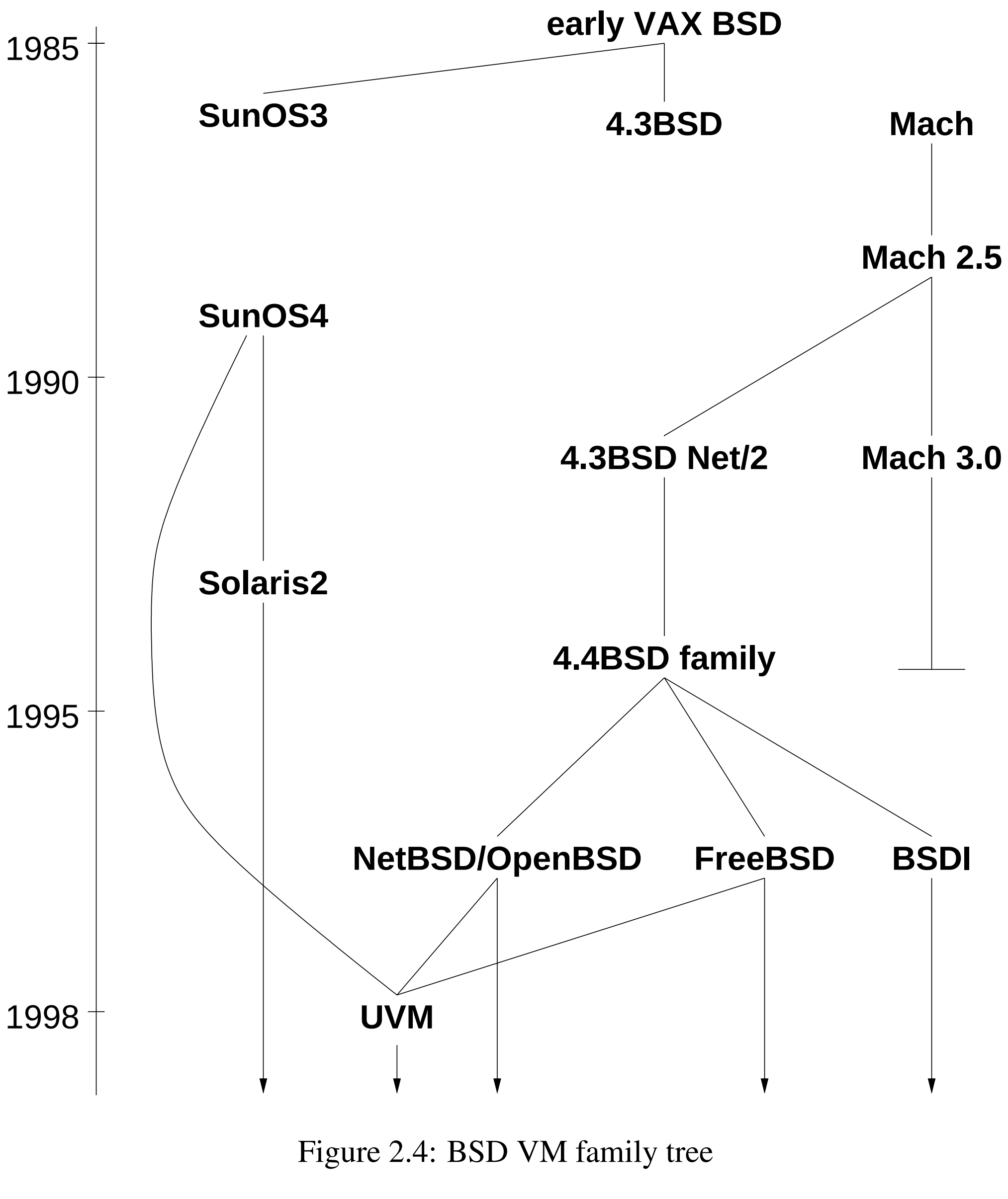


Figure 2.4: BSD VM family tree

Machine-Dependent Layer	Machine-Independent Layer
processor specific	used on all supported processors
maps pages: "map page A to virtual address B"	maps chunks of memory objects: "map object X at offset Y into the virtual address space starting at address Z"
programs MMU	programs pmap layer

Table 2.2: VM MD/MI layering

2.4 Design Overview of the BSD VM System

This section contains a design overview of the BSD VM system that was imported from the Mach operating system for the Net/2 release of BSD. Understanding the BSD VM is critical to understanding the design of UVM. It is also useful for understanding new VM features introduced in UVM.

2.4.1 Layering in the BSD VM System

The BSD VM system is divided into two layers: a large machine-independent ("MI") layer, and a smaller machine-dependent ("MD") layer. The machine-independent code is shared by all BSD-supported processors and contains the code that performs the high level functions of the VM system. The machine-dependent code is called the "pmap" (for physical map) layer, and it handles the lower level details of programming a processor's MMU. Each architecture supported by the operating system must have its own pmap module. Each layer of the VM system does its own level of mapping. The machine-independent layer maps "memory objects" into the virtual address space of a process or the kernel. A memory object is any kernel data structure that represents data that can be mapped into a virtual address space. For example, a file can be a memory object. The machine-dependent layer does not know about higher level concepts such as memory objects; it only knows how to map physical pages of memory into a virtual address space. Table 2.2 summarizes the difference between the machine-independent and machine-dependent layers.

2.4.2 The Machine-Dependent Layer

The pmap layer's job is to be the machine-independent code's interface to a processor's MMU. At an abstract level, the pmap layer can be viewed as a big array of mapping entries that are indexed by virtual address to produce a physical address and attributes. These attributes describe the page's current protection, whether the page has been referenced or modified, and other characteristics. Since each process has its own virtual address space, each process must have its own pmap data structure to describe the low-level mappings of that virtual address space. Mapping entries contained in the pmap layer are often called "page table entries" (PTEs) due to the way many MMUs work. The actual way the array of PTEs is stored varies from system to system depending on the MMU hardware. For example, on an i386 PC the array is broken up into two levels. One level contains 1024 fixed-sized chunks of PTEs called "page tables," the other level contains a directory of page tables called the "page directory." Other processors have a similar structure, but with three levels of tables rather than two. Regardless of how the page mappings are stored, the interface provided by the pmap layer to the machine-independent code is consistent across all platforms.

The most common pmap operations and examples of their usage are described below:

`pmap_enter`: adds a virtual-to-physical address mapping to the specified pmap at the specified protection. The `pmap_enter` function is called by the fault routine to establish a mapping for the page being faulted in.

`pmap_remove`: removes a range of virtual-to-physical address mappings from a pmap. The `pmap_remove` function is called during an unmap operation to remove low-level machine-dependent mappings.

`pmap_protect`: changes the protection of all mappings in a specified range of a pmap. The `pmap_protect` function is called during a copy-on-write operation to write protect copy-on-write memory.

`pmap_page_protect`: changes the protection of all mappings of a single page in every pmap that references it. The `pmap_page_protect` function is called before a pageout operation to ensure that all pmap references to a page are removed.

`pmap_is_referenced`, `pmap_is_modified`: tests the referenced and modified attributes for a page. This is calculated over all mappings of a page. These functions are called by the pagedaemon when looking for pages to free.

`pmap_clear_reference`, `pmap_clear_modify`: clears the reference and modify attributes on all mappings of a page. These functions are called by the pagedaemon to help it identify pages that are no longer in demand.

pmap_copy: copies mappings from one pmap to another. The `pmap_copy` function is called during a `fork` operation to give the child process an initial set of low-level mappings.

Note that pmap operations such as `pmap_page_protect` may require the pmap module to keep a list of all pmaps that reference a page.

2.4.3 The Machine-Independent Layer

The high-level functions of the VM system are handled by the machine-independent layer of the BSD VM system. Such functions include managing a process' file mappings, requesting data from backing store, paging out memory when it becomes scarce, managing the allocation of physical memory, and managing copy-on-write memory. The activities of the machine-independent layer are centered around five main data structures:

vmspace: describes a virtual address space of a process. The `vmspace` structure contains pointers a process' `vm_map` and `pmap` structures, and contains statistics on the process' memory usage.

vm_map: describes the virtual address space of a process or the kernel. It contains a list of valid mappings in the virtual address space and those mapping's attributes.

vm_object: describes a file, a zero-fill memory area, or a device that can be mapped into a virtual address space. The `vm_object` contains a list of `vm_page` structures that contain data from that object.

vm_pager: describes how backing store can be accessed. Each `vm_object` on the system has a `vm_pager` structure. This structure contains a pointer to a list of functions used by the object to fetch and store pages between the memory pointed to by `vm_page` structures and backing store.

vm_page: describes a page of physical memory[4]. When the system is booted a `vm_page` structure is allocated for each page of physical memory that can be used by the VM system.

[4]On a few systems, that have small hardware page sizes, such as the VAX, the VM system has a VM page structure manage two or more hardware pages. This is all handled at the pmap layer and thus is transparent to the machine-independent VM code.

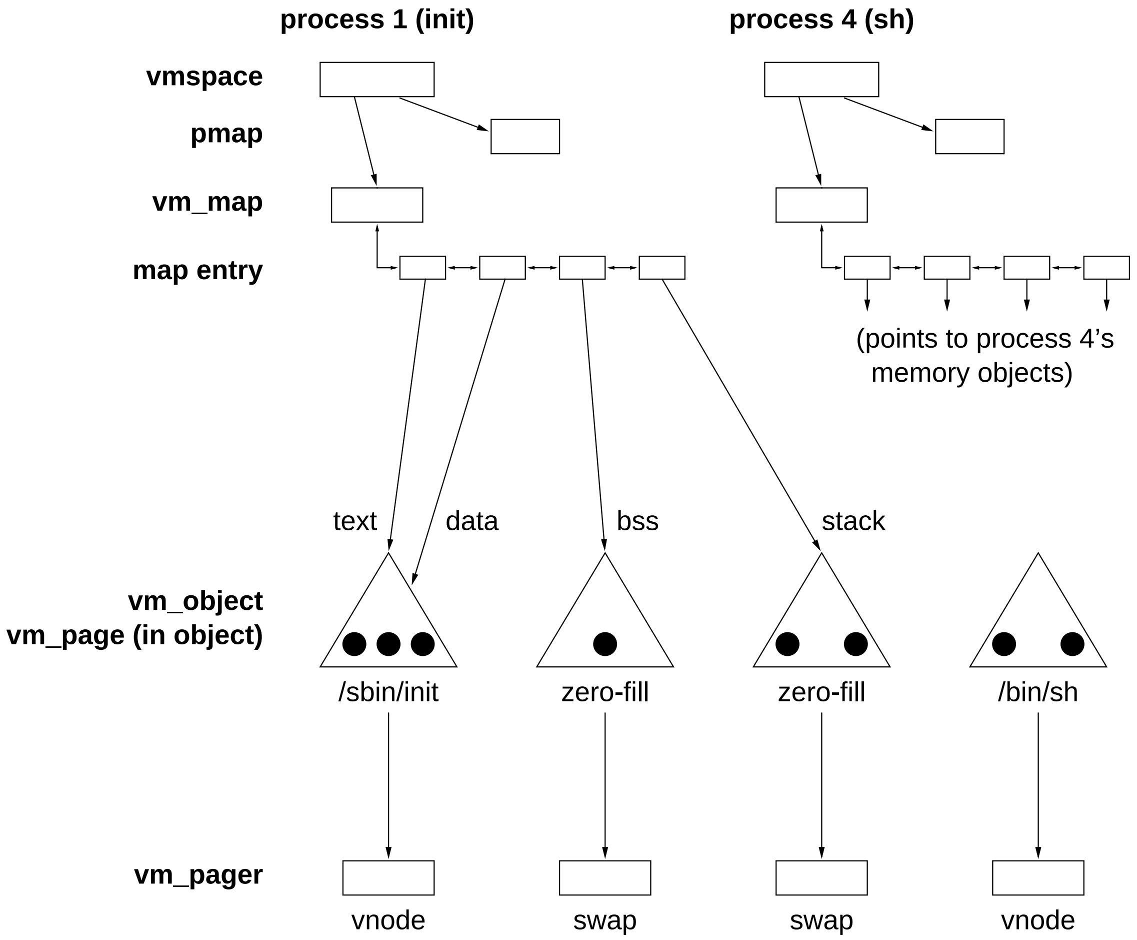


Figure 2.5: The five main machine-independent data structures: `vmspace`, `vm_map`, `vm_page`, `vm_object`, `vm_pager`. The triangles represent `vm_objects`, and the dots within them represent `vm_pages`. A `vm_object` can contain any number of pages. Note that the text and data areas of a file are different parts of a single object.

As shown in Figure 2.5, VM map structures map VM objects into an address space. VM objects store their data in VM pages. Data in VM pages is copied to and from backing store by VM pagers. Note that each `vm_map` structure has an associated pmap structure to contain the lower level mapping information for the virtual address space mapped by the `vm_map`. The `vm_map` and the pmap together are referred to by a `vmspace` structure (not shown).

In order to find which `vm_page` should be mapped in at a virtual address (for example, during a page fault), the VM system must look in the `vm_map` for the mapping of the virtual address. It then must check the backing `vm_object` for the needed page. If the page is resident (often due to having been recently accessed by some other process), then

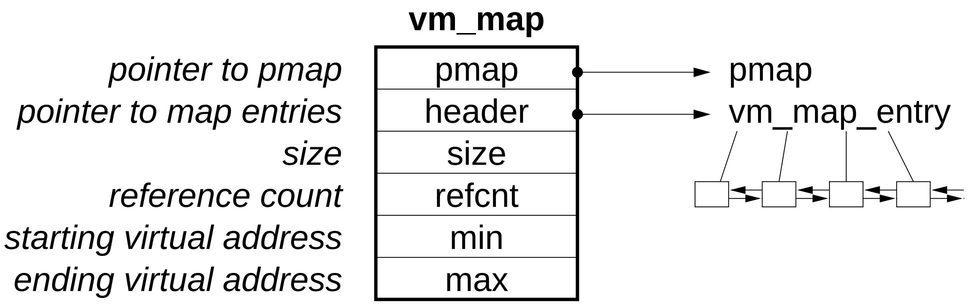


Figure 2.6: The VM map data structure

the search is finished. However, if the page is not resident, then the VM system must issue a request to the `vm_object`'s `vm_pager` to fetch the data from backing store.

We will now examine each of the machine-independent data structures in more detail.

VM Maps

A VM map (`vm_map`) structure maps memory objects into regions of a virtual address space. Each VM map structure on the system contains a sorted doubly-linked list of "map entry" structures. Each map entry structure contains a record of a mapping in the VM map's virtual address space. For example, in Figure 2.5 the map for the init process would have four map entry structures: text, data, bss, and stack. The kernel and each process on the system have their own VM map structures to handle the allocations of their virtual address space.

The `vm_map` data structure is shown in Figure 2.6. This structure contains the starting and ending virtual addresses of the managed region of virtual memory, a reference count, and the size of the region. It also contains two pointers: one to a linked list of map entries that describe the valid areas of virtual memory in the map, and one to the machine-dependent pmap data structure that contains the lower-level mapping information for that map. The map entry structure is shown in Figure 2.7. This structure contains pointers to the next and previous map entry in the map entry list. It also contains a starting and ending virtual address, a pointer to the backing object (if any), and attributes such as protection, and inheritance code.

A map entry usually points to the VM object it is mapping. However, in certain cases a map entry can also point to another map. There are two types of maps that can be pointed to by a map entry: submaps and share maps.

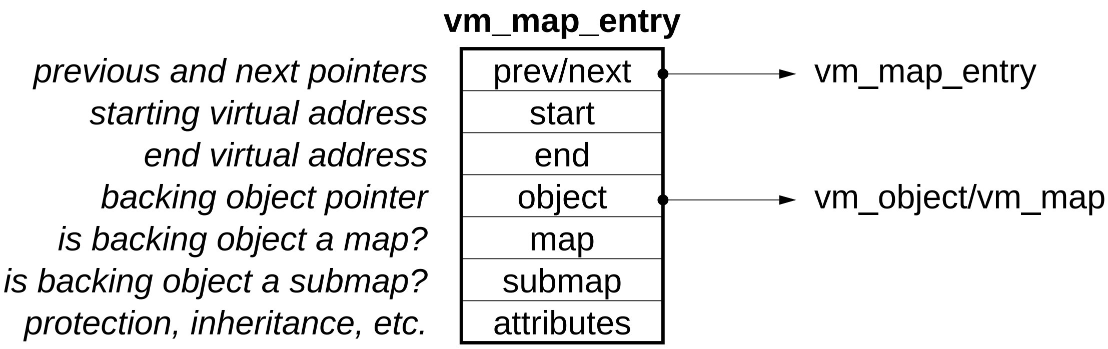


Figure 2.7: The VM map entry structure

- Submaps can only be used by the kernel. The main purpose of submaps is to break up the the kernel virtual address space into smaller units. The lock on the main kernel map locks the entire kernel virtual address space except for regions that are mapped by submaps. Those regions (and their maps) are locked by the lock on the submap.

- A share map allows two or more processes to share a range of virtual address space and thus all the mappings within it. When one process changes the mappings in a share map, all other processes accessing the share map see the change.

A whole set of functions perform operations on VM maps. There are functions to create, free, add a reference from, and drop a reference to a VM map. There are functions to find free space in a map, to allocate and free ranges of virtual addresses in a map, and to copy a map for a fork operation. There are also functions to change the attributes of a mapping. Attributes include protection, memory usage pattern (advice), and wire-count. It should be noted that an attribute applies to the whole mapping defined by the map entry. If the kernel wants to change the attributes of part of a map entry, then that map entry must be broken into two or three parts. For example, if there is a map entry mapping the virtual address range 0x2000 to 0x5000 read/write and the VM system is asked to change the protection of 0x3000 to 0x4000 to read-only, then the map entry will get broken into three parts: 0x2000 to 0x3000 at read-write, 0x3000 to 0x4000 at read-only, and 0x4000 to 0x5000 at read/write.

The kernel uses a VM map data structure to describe its own address space, including the area used by the kernel dynamic memory allocator (`malloc`). VM map entry structures are normally allocated with the kernel `malloc`. However, the kernel cannot allocate its own VM map's entries with `malloc` because it might loop while allocating map

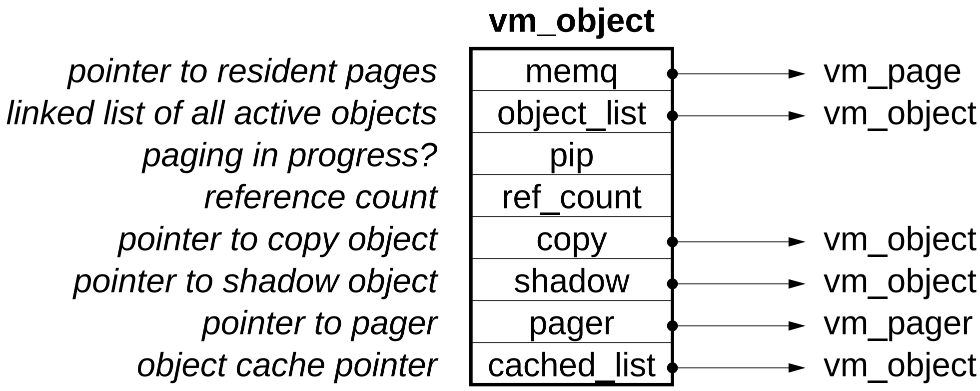


Figure 2.8: The VM object structure

entries. Thus, the kernel allocates its VM map entries from a private pool of map entry structures rather than with `malloc`.

VM Objects

A VM map contains a list of map entries that define allocated regions in a map's address space. Each map entry points to a memory object that is mapped into that region. Objects may have more than one map entry pointing to them. In the BSD VM all memory objects are defined by the VM object structure. The VM object structure is shown in Figure 2.8. A VM object contains a list of all pages of memory that contain data belonging to that object (resident pages). Pages in a VM object are identified by their offset in the object. Pages are often added to VM objects in response to a page fault. There is also a global linked list of all VM object allocated by the system. The VM object contains a reference counter and a flag to indicate if data is currently being transfered between one of the VM object's pages and backing store (this is called "paging"). The VM object contains a shadow object and a copy object pointer. These pointers are used for copy-on-write operations, described in detail in Section 2.4.4. The VM object contains a pointer to a VM pager structure. The pager is used to read and write data from backing store. Finally the VM object contains pointers for the object cache.

The object cache is a list of VM objects that are currently not referenced (i.e. their reference count is zero). Objects can be removed from the cache by having their reference count increased. Objects in the object cache are said to be "persisting." The goal of having persisting objects is to allow memory objects that are often reused to be reclaimed rather than reallocated and read in from backing store. For example, the program `/bin/ls` is

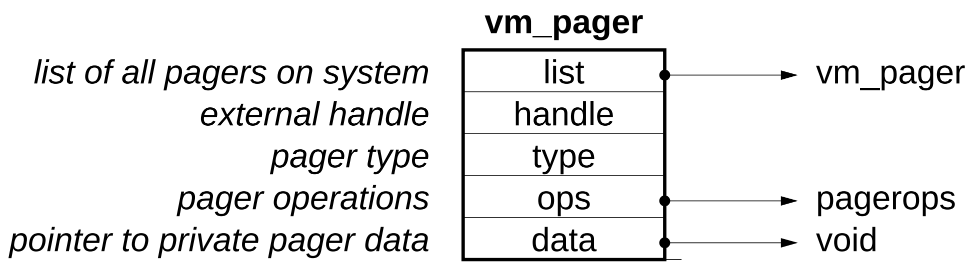


Figure 2.9: The VM pager data structure

run frequently, but only for a short time. Allowing the VM object that represents this file to persist in memory saves the VM system the trouble of having to read the `/bin/ls` file from disk every time someone runs it.

VM Pagers

VM objects read and write data from backing store into a page of memory with a pager, which is defined by a VM pager structure. Each VM object that accesses backing store has its own VM pager data structure. There are three types of VM pagers: device, swap, and vnode. A device pager is used for device files that allow their memory to be `mmap`'d (e.g. files in `/dev`). A swap pager is used for anonymous memory objects that are paged out to the swap area of the disk. A vnode pager is used for normal files that are memory mapped. The kernel maintains a linked list of all the VM pager structures on the system.

The VM pager structure is shown in Figure 2.9. The structure contains pointers to maintain the global linked list of pager structures. It contains a "handle" that is used as an identification tag for the pager. It contains a type field that indicates which of the three types of pagers the pager is. The pager also has a private data pointer that can be used by pager-specific code to store pager-specific data. Finally, the pager structure has a pointer to a set of pager operations. Supported operations include: allocating a new pager structure, freeing a pager structure, reading pages in from backing store, and saving pages back to backing store.

VM Pages

The physical memory of a system is divided into pages. The size of a page is fixed by the memory management hardware of the computer. The VM system manages these hardware pages with the VM page structure. On most systems there is a VM page structure for every page of memory that is available for the VM system to use.

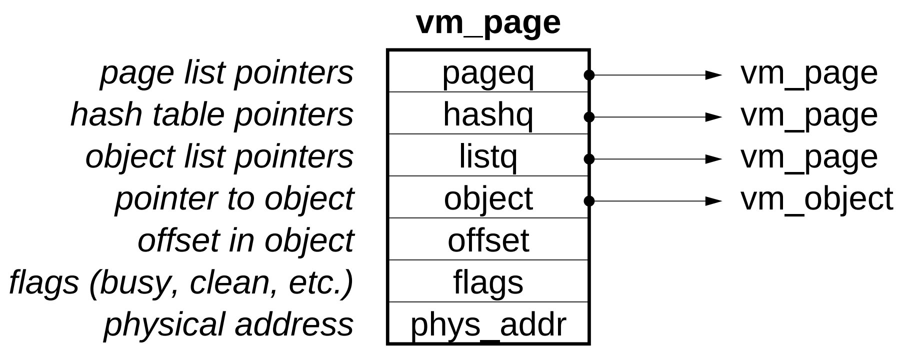


Figure 2.10: The VM page structure

The VM page structure is shown in Figure 2.10. A page structure can independently appear on three different list of pages. These lists are:

pageq: active, inactive, and free pages. Active pages are currently in use. Inactive pages are allocated and contain valid data but are currently not being used (they are being saved for possible reuse). Free pages are not being used at all and contain no valid data.

hashq: used to attach a page to a global hash table that maps a VM object and an offset to a VM page structure. This allows a page to be quickly looked up by its object and offset.

listq: a list of pages that belong to a VM object. This allows an object to easily access pages that it owns.

Each page structure contains a pointer to the object that owns it, and its offset in that object. Each page structure also contains the physical address of the page to which it refers. Finally, there are a number of flags that are used by the VM system to store the state of the page. Such flags include “busy,” “clean,” and “wanted.”

The VM system provides functions to allocate and free pages, to add and remove pages from the hash table, and to zero or copy pages.

2.4.4 Copy-on-write and Object Chaining

One important aspect of the VM system is how it handles memory objects that are mapped copy-on-write. In a copy-on-write mapping, changes made to an object’s mapped pages are not shared — they are private to the process that made the changes. The BSD VM system manages copy-on-write mappings of VM objects by using "shadow objects." A shadow object is an anonymous memory object that contains the modified pages of a copy-on-write mapped VM object. When searching for pages in a copy-on-write mapping, the shadow object is searched before the underlying object is searched.

There are two forms of copy-on-write mappings: private mappings and copy mappings.

- In a **private** copy-on-write mapping, changes made by the process with the mapping are private to that process, but changes made to the underlying object that are not shadowed by a page in the shadow object are seen by the mapping process. This is the standard form of copy-on-write that most Unix-like operating systems use, and it is the behavior specified for the `mmap` system call by the Unix standards [30].
- In a **copy** copy-on-write mapping the mapping process gets a complete snapshot of the object being mapped at the time of the mapping. The mapping process will *not* see changes other processes make to the unshadowed areas of the underlying object. This form of copy-on-write requires the use of copy objects — anonymous memory objects that contain the unmodified copy of a modified page — and is not supported on most Unix-like platforms.

Private Copy-on-write

Consider a file "`test`" that has just been mapped private copy-on-write. The map entry in the VM map structure that maps `test` will point to the VM object that corresponds to `test`, but it will have both the "copy-on-write" and "needs copy" attribute set. The copy-on-write attribute in a map entry indicates that the mapped object is mapped copy-on-write. The needs copy attribute indicates that a shadow object to hold changed pages will need to be allocated. When the process firsts writes to the mapped area, a shadow object containing the changed page will be allocated and inserted between the map entry and the underlying `test` object. This is shown in Figure 2.11.

Now consider what happens if the process mapping `test` forks off a child process. In that case the child will want its own copy of the copy-on-write region. In the BSD VM this causes another layer of copying to be invoked. The original shadow object is treated like a backing object, and thus both the parent's and the child's mapping enter the "needs copy" state again, as shown in Figure 2.12(a). Now when either process attempts to write to the memory mapped by the shadow object the VM system will catch it and insert a new shadow object between the map entry and the original shadow object and clear "needs

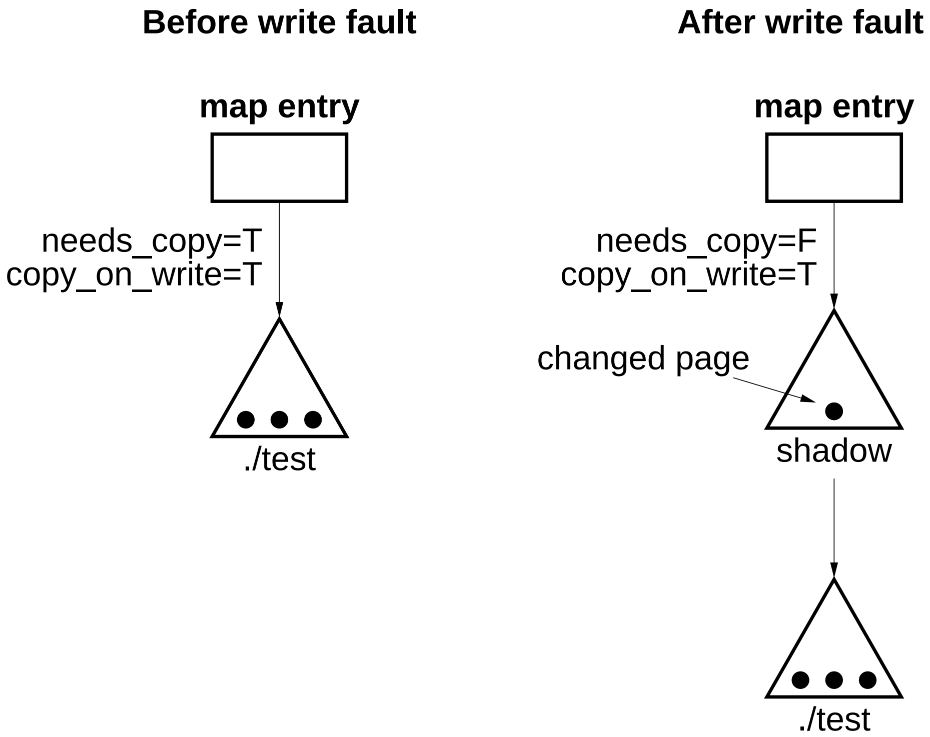
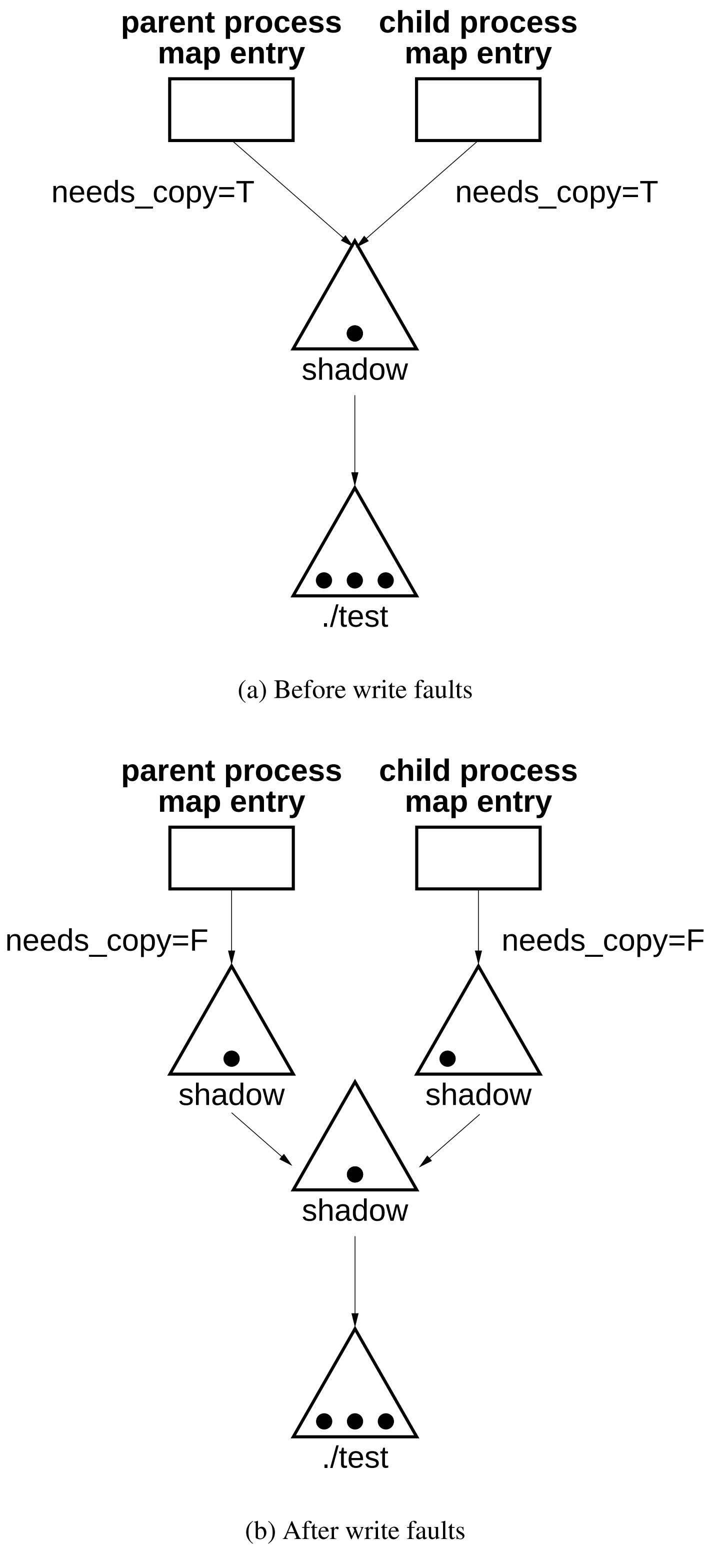


Figure 2.11: The copy-on-write mapping of a file. Note the pages in an object mapped copy-on-write can not be changed.

copy." This is referred to as "shadow object chaining." After both processes perform writes, the objects will be as shown in Figure 2.12(b).

Note that in Figure 2.12(b) the parent process has modified the center page in the first shadow object and thus it has its own copy of it in its most recent shadow object (on the left). The child process has not modified the center page, and thus if it was to read that page it would read the version stored in the original shadow object (that is shared by both the parent and child process). Now consider what would happen if the child process was to exit. This is shown in Figure 2.13. Note that the center page appears in both remaining shadow objects. However the parent process only needs access to the copy of the page in the most recent shadow object. The center page in the other shadow object is inaccessible and not needed. There is no longer a need for two shadow objects; they should be collapsed together and the inaccessible memory freed. However, the BSD VM has no way to realize this, and thus the inaccessible memory resources in the other shadow object remain allocated even though they are not in use.

This is referred to as the "object collapse problem." 4.4BSD VM did not properly collapse all objects, and thus inaccessible memory could get stranded in chains of shadow



(a) Before write faults

(b) After write faults

Figure 2.12: The copy-on-write mapping after a fork

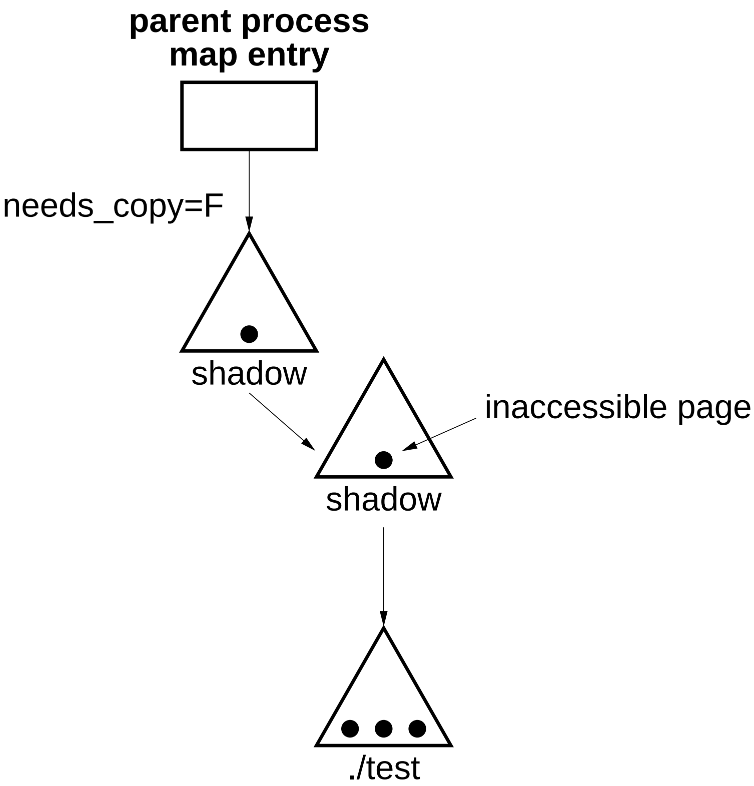


Figure 2.13: The copy-on-write mapping after the child exits

objects. Eventually the inaccessible memory would get paged out to the swap area so the physical page could be reused. However, the VM system would eventually run out of swap space and the system would deadlock. Thus this problem was also referred to as the "swap memory leak" bug because the only way to free this memory is to cause all processes referencing the shadow object chain to exit.

The object collapse problem has been partly addressed in most versions of BSD. However the object collapse code used to address the problem is rather complicated and has triggered many hours of debugging. UVM handles copy-on-write in a completely different way that avoids the object collapse problem.

Copy Copy-on-write

Consider two processes mapping the file `test`, as shown in Figure 2.14. Process "A" has a shared mapping of `test`, and thus when process A changes its memory the changes get reflected back to the backing file. Process "B" has a *private* copy-on-write mapping of `test`. Note that process B has only written to the the center page of the object. If process

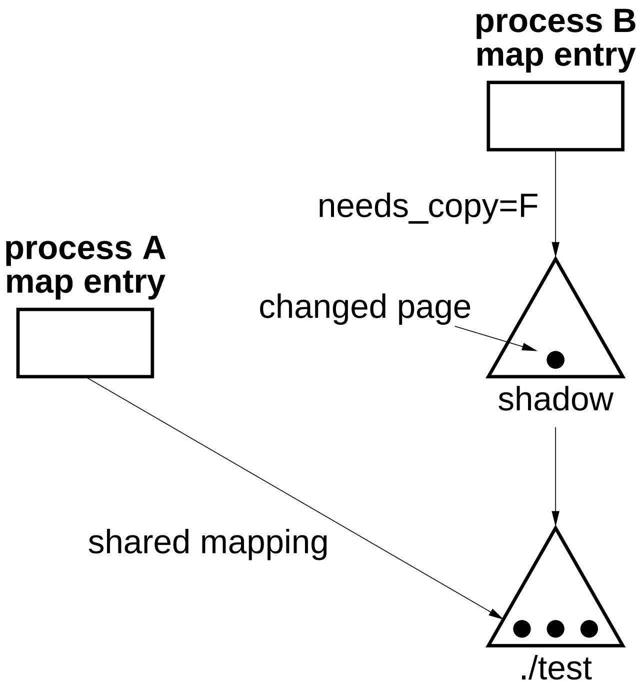


Figure 2.14: Private copy-on-write mapping

A writes to the left-hand page of `test` before process B does, then process B will see the changes made to that page by A up to that point. However, once the page gets added to B's shadow object then B will no longer see changes made by A.

Now consider a setup in which process A has a shared mapping of `test` and process B has a *copy* copy-on-write mapping as shown in Figure 2.15. Before process A changes the left page, it must create a copy object so that process B can access the original version of the left page. When B changes the middle page it creates a shadow object. The shadow object now shadows the copy object rather than `test`'s object. A list of objects formed by the copy object pointer is called a "copy object chain."

Copy objects are needed to support the non-standard copy copy-on-write semantics. Copy objects add another layer of complexity to the VM code and make the object collapse problem more difficult to address. Maintaining copy copy-on-write semantics is more also expensive than private copy-on-write semantics. There are more kernel data structure to allocate and track. There is an extra data copy of a page that the private mapping semantics doesn't have to do. Processes with shared mappings of files must have their mappings write protected whenever a copy object is made so that write accesses to the object can be caught

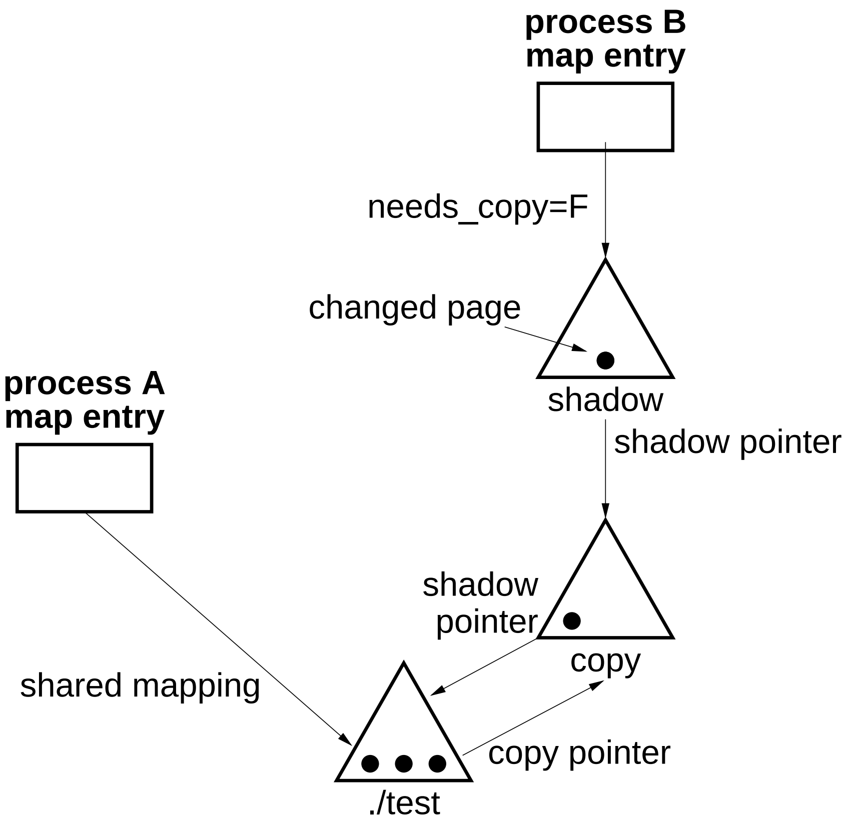


Figure 2.15: Copy copy-on-write mapping

by a page fault. Because of this, several branches of the BSD family have removed copy objects and copy copy-on-write mappings from the kernel.

2.4.5 Page Fault Handling

Now that the VM object structure has been explained the general procedure used to handle page faults can be explained. When memory is referenced by a process one of two things can happen:

- The referenced address has a **valid mapping** and the data is accessed from the mapped physical page.
- The referenced address has an **invalid mapping** and the MMU causes a page fault to occur. The low level machine-dependent code catches the page fault and calls the VM page fault routine. If the page fault routine cannot handle the fault then it returns a failure code that causes the program to receive a "segmentation violation."

The page fault handler is called with the VM map of the process that caused the fault, the virtual address of the fault, and whether the faulting process was reading or writing memory when the fault occurred. A simplified version of the fault routine's operation is as follows:

1. The list of entries in the VM map is searched for the map entry whose virtual address range the faulting address falls in. If there is no such map entry, then the fault routine returns an error code and the process gets a segmentation violation.

2. Once the proper map entry is found, then the fault routine starts at the VM object it points to and searches for the needed page. If the page is not found in the first object, the fault routine traverses the shadow object chain pointers until it either finds the page or runs out of objects to try. If the fault routine runs out of objects to try it may either cause the fault to fail or allocate a zero fill page (depending on the mapping). The fault routine leaves a busy page in the top level object to prevent other processes from changing its object while it is processing the fault.

3. Once the proper page is found the object's VM pager must retrieve from backing store if it is not resident. Then the fault routine checks to see if this is a write-fault and if the object that owns that page has a copy object. If it does, then a copy of the page is made and the old version of the page is placed in the object pointed to by the copy object chain pointer.

4. If the fault routine had a VM pager do I/O to get the needed page, then the pager unlocked all the data structures before starting the I/O. In this case the fault routine must reverify the lookup in the VM map to ensure that the process' mappings have not been changed since the start of the fault operation.

5. Next the fault routine asks the pmap layer to map the page being faulted. The fault routine must determine the appropriate protection to use depending on the copy-on-write status and the protection of the map entry.

6. Finally, the fault routine returns a success code so that the process can resume running.

2.4.6 Memory Sharing

We have discussed the various way memory can be copied in the BSD VM system. Memory sharing is much simpler than memory copying and does not require object chains. Memory can also be shared in a number of ways in the BSD VM system including:

- In a multithreaded environment, the VM map and pmap structures of a process can be shared by its threads.
- Virtual memory space can be shared using share maps.
- Memory objects such a files can be mapped into multiple virtual address spaces by being referenced by multiple VM map entry structures. This allows those memory objects to be shared among processes. For example, there is only one `/bin/ls` memory object and its text and data pages are shared by all processes on the system.
- Objects that are copied with copy-on-write use shared read-only pages to defer the copy until the pages are actually written.

2.5 Summary

In this chapter we have reviewed the overall architecture and operation of the BSD VM system. We introduced the major data structures and functions, and explained in detail the copy-on-write operation. In the next chapter we will give an overview of the UVM system and explain how it differs from the BSD VM system.

Chapter 3

UVM Overview and Related Work

A computer's virtual memory management hardware provides an operating system with a powerful tool for moving and sharing data among processes. However, most Unix-like operating systems do not take full advantage of this tool. This is due to the evolution and complexity of memory management in Unix-like operating system [39]. Early versions of Unix ran on computers that did not have hardware support for virtual memory. Thus Unix's I/O application programmer interface (API) was not designed with virtual memory in mind and did not include features like memory mapped files and page remapping. Although Unix was ported to computers with hardware support for virtual memory in 1979, the `mmap` system call was not commonly available until the late 1980s with the release of SunOS 4. In BSD, `mmap` was first scheduled to appear in 4.2BSD but did not actually appear until the 1993 4.4BSD release when the aging BSD VAX VM system was replaced with the system from Mach [39, 69]. While the new Mach-based VM system brought some memory features such as `mmap` to BSD, it still did not take full advantage of the flexibility that a modern MMU can offer.

We introduce UVM, a new virtual memory subsystem for BSD that makes better use of existing hardware memory management features to reduce overhead and improve performance. UVM provides new features that are either not possible, or difficult or expensive to achieve with the old BSD system. UVM is implemented entirely within the framework of BSD and thus maintains all the features and standard parts of the traditional Unix environment that programmers have come to expect. The first release of UVM in NetBSD runs on several platforms including I386-PC, DEC Alpha, Sun Sparc, Motorola m68k, and DEC VAX systems. It is already being used on systems around the world.

In this chapter we present a high-level overview of UVM and discuss UVM in the context of related work. We begin by presenting UVM design goals in Section 3.1. In

Section 3.2 we examine the new and improved features of UVM. In Section 3.3 we present a design overview that parallels the overview of BSD VM from Chapter 2. In Section 3.4 we discuss related work. We conclude the chapter with two tables that summarize the UVM work. Table 3.1 contains a complete list of functional components of UVM, corresponding to UVM's major source files. Table 3.2 summarizes the main design points and "issues faced" while designing and implementing UVM.

3.1 Goals

Our primary objective in creating UVM is to produce a virtual memory system that provides a Unix-like operating system kernel's I/O and IPC subsystems with efficient VM-based data movement facilities that have less overhead than a traditional data copy. While traditional VM research often focuses on working set size and page replacement policy, our research is focused on efficient VM-based data movement. Unlike many other VM research projects, our work has been implemented as part of an operating system that is in wide-spread use. Thus, we have designed our new virtual memory features so that their presence in the kernel does not disrupt other kernel subsystems. This allows experimental changes to be introduced into the I/O and IPC subsystem gradually, thus easing the adoption of these features. Our work centers around five major goals:

Allow a process to safely let a shared copy-on-write copy of its memory be used either by other processes, the I/O system, or the IPC system. The mechanism used to do this should allow copied memory to come from a memory mapped file, anonymous memory, or a combination of the two. It should provide copied memory either as wired pages for the kernel's I/O or IPC subsystems, or as pageable anonymous memory for transfer to another process. It should gracefully preserve copy-on-write in the presence of page faults, pageouts, and memory flushes. Finally, it should operate in such a way that it provides access to memory at page-level granularity without fragmenting or disrupting the VM system's higher-level memory mapping data structures. Section 7.1 describes how UVM meets this goal through the page loanout mechanism.

Allow pages of memory from the I/O system, the IPC system, or from other processes to be inserted easily into a process' address space. Once the pages are inserted into the process they should become anonymous memory. Such anonymous memory should be indistinguishable from anonymous memory allocated by traditional means. The mechanism used to do this should be able to handle pages that have been copied from another process' address space using the previous mechanism (page loanout). Also, if the operating system is allowed to choose the virtual address where the inserted pages are placed, then it should be able to insert them without fragmenting or disrupting the VM system's higher-level memory mapping data structures. Section 7.2 describes how UVM meets this goal through the page transfer mechanism.

Allow processes and the kernel to exchange large chunks of their virtual address spaces using the VM system's higher-level memory mapping data structures. Such a mechanism should be able to copy, move, or share any range of a virtual address space. This can be a problem for some VM systems because it introduces the possibility of allowing a copy-on-write area of memory to become shared with another process. The per-page cost for this mechanism should be minimized. Section 7.3 describes how UVM meets this goals through the map entry passing mechanism.

Optimize the parts of the VM system that effect the performance and complexity of our new VM operations. We wish to take advantage of lessons we have learned from observing the Mach based BSD VM system (and other VM systems); we hope to make use of BSD VM's positive aspects while avoiding its pitfalls. UVM addresses four major pitfalls of the BSD VM as follows:

- UVM eliminates object chaining and replaces it with a simple two-level memory scheme, thus removing the inefficiencies and code complexity associated with maintaining and traversing shadow object chains of arbitrary length.

- UVM eliminates swap memory leaks associated with partially unmapping anonymous memory objects by providing efficient per-page reference counters that are invoked when memory is partially unmapped.

- UVM eliminates unnecessary map entry fragmentation associated with the wiring and unwiring of memory.

- UVM introduces a new i386 pmap module that takes advantage of all the VM hardware features of the i386 (e.g. single page TLB flush, global TLB entries) and reduces the probability of deadlock and system crashes.

Improve the secondary elements of the VM system based on our observations of UVM and other VM systems. We hope to reduce unnecessary complexity and improve the kernel's overall performance and design. UVM features a large number of secondary design improvements that meet this goal. For example, aggressively clustering anonymous memory for pageout allows UVM to always form the largest possible cluster when paging out to swap, and clustered pagein of resident pages reduces application page faults.

In addition UVM unifies the handling of contiguous and non-contiguous physical memory. It also improves memory object handling by reducing the number of software layers associated with object management. Chapter 8 details these and other secondary design improvements.

3.2 UVM Features

In this section we describe the new features introduced in UVM as well as the existing BSD features that have been significantly improved in UVM.

3.2.1 New Features

UVM introduces four significant new features not provided by the BSD VM system. Together these features reduce the overhead associated with moving data, thus improving system performance. The implementation of these features was made possible through the improved design and organization of UVM.

Page Loanout. UVM allows pages that are managed by the virtual memory system to be used by other kernel subsystems as well as other processes. Loaned out pages are shared read-only. Should a process attempt to change data in a loaned out page UVM copies the data to a new non-loaned page before allowing the change. Any managed page in the VM system can be loaned out, whether it is part of a memory mapped vnode or part of a block of anonymous memory allocated with `malloc`. Pagers need only have minimal knowledge of a page's loanout status to operate. The BSD VM system has no support for page loanout.

As an example of page loanout's usefulness, consider the process of transmitting a disk file over the network. Before `mmap` was available, a user process would have to read the file into its anonymous memory and then write it to a socket. This process results in two data copies: one into the user's address space and one from the user's address space into the kernel networking subsystem's mbuf data area. With the advent of `mmap`, BSD VM users could reduce the overhead of this operation by one copy if they simply memory mapped the file into their address space rather than read it into anonymous memory. In UVM, page loanout allows for further improvements: the process can now `mmap` the file and then cause its pages to be loaned to the networking subsystem, there by eliminating the copy from the memory mapped area to the mbuf data area. When the networking subsystem is finished with the pages it will terminate the loan. The same loanout scheme can be applied to devices as well. For example, when writing to an audio device, the user's buffer can be loaned out to the audio hardware rather than copied into an internal kernel buffer, there by eliminating one data copy.

Page Transfer. UVM allows other kernel subsystems to transfer pages under their control to a process' normal virtual memory. Page transfer is the opposite of page loanout. In page loanout a process loans its pages out to the kernel or another process. In page transfer, a process receives pages from the kernel or another process. Page transfer allows kernel subsystems to read data into pages and then donate those pages to UVM. This causes them to become anonymous memory pages that can be inserted into a process' address space or into the kernel's address space. Thus, the networking subsystem could use a page for a large mbuf and then pass that mbuf data area to a process. Or the audio system, when recording, could store the recorded data in pages that can be directly passed to the user, without a data copy.

The BSD VM system has no interface for page transfers. In order to implement this in the BSD VM one would have to write code to negotiate the shadow and copy object chains associated with anonymous memory and determine the correct object to put the inbound page into while being careful to avoid interfering with objects that are in the process of being collapsed or are currently locked.

Map Entry Passing. UVM allows processes and the kernel to dynamically exchange virtual memory address space. The address space can contain any number of mappings and can be dynamically shared with, or copied or donated to other processes or the kernel. This allows processes to easily pass virtual memory between themselves without copying data. This can be a useful substitute for a Unix pipe when large chunks of data are being exchanged.

The BSD VM system does not support map entry passing. Memory exchange between processes is limited to the copying and sharing that happens when a process forks. Some versions of BSD do, however, support the Unix System V shared memory interface. This interface can be used by a process to preallocate a contiguous block of anonymous memory and then allow other processes to attach to it. While useful, this interface is limited when compared to UVM's map entry passing. System V shared memory only allows processes access to one contiguous block of memory, and processes attaching to it must share it (e.g. they can't get a copy-on-write copy of it).

Map entry passing also allows a process to grant another unrelated process access to only part of a file. In traditional Unix, if a file descriptor is passed to another process using file descriptor passing, then the receiving process gets access to the entire file at what ever permission the file descriptor allows. The only way to pass partial access of a file to another process is to open the file, `mmap` in the part of the file to be accessed, close the file, and then fork a child process that only has access to the part of the file that was memory mapped.

Using UVM's map entry passing, any process can memory map a part of a file and then offer to send that mapping to other processes. This could be useful in allowing processes access to only a page-sized section of a database, for example.

Partial Deallocation. In the BSD VM when a block of anonymous memory is mapped the VM map gets a reference to an anonymous VM object that backs the mapping. If the process deallocates (i.e. unmaps) a part of the mapping then the reference to the backing object is broken up into multiple references, and the reference to the area of memory being released is dropped. Unfortunately, the BSD VM has no way of telling the backing object that part of its memory is no longer in use and should be freed. As a result, the memory remains in the object although it is no longer accessible. This can result in a swap memory leak condition. This is currently not a problem in BSD because most BSD programs use the traditional `malloc` that allocates memory off the heap rather than using anonymous memory mapping for memory allocation. However, due to increased flexibility memory-mapped user-memory allocators are becoming more common so this could become a problem. UVM can handle the partial deallocation of anonymous memory without any swap space leaks.

3.2.2 Improved Features

There are a number of BSD VM features that have been significantly improved in UVM. Some of the more interesting improvements are listed below.

Simplified Copy-on-write. In UVM the management of copy-on-write memory has been greatly simplified. The BSD VM's shadow and copy object chains have been replaced with a simple two-level scheme based on page reference counters. As a result all the BSD VM code needed to manage and collapse shadow and copy object chains is not necessary in UVM. This simplifies the page fault routine and eliminates a large chunk of the object management code.

Clustered anonymous memory pageout. UVM supports the clustering of I/O at all levels. By storing data in a large contiguous area of backing store, multiple I/O paging operations can be "clustered" into a single lager I/O operation. One place where UVM's I/O clustering really pays off is in the pageout of anonymous memory to swap. In the BSD

VM, anonymous memory is statically assigned to an area of swap based on the object in which it lives. Once a section of an anonymous object is assigned to an area of swap it will always be swapped out to that location (as long as the object is still active). Thus memory assigned to different objects will not be contiguous and must be paged out in multiple operations.

UVM takes advantage of the fact that anonymous memory that needs to be paged out can be paged to any part of the swap area and collects anonymous pages together so that it can perform one big pageout I/O operation. As a result when memory is scarce, anonymous memory can be paged out much faster in UVM than BSD VM, thus making the recovery time for scarce memory much shorter.

Improved page wiring. Data is said to be wired if it resides in a page that is set to always be resident in memory. Thus, the access time for wired memory is always small because the VM system never has to wait for the pager to do I/O from backing store since the data is always resident. In both BSD VM and UVM the wired status of memory is stored in two places. Each map entry has a wiring attribute that indicates that all pages mapped by that entry should be wired. Also, each page of memory has a wiring count that if non-zero indicates that some process or the kernel needs that page of memory to remain resident at all times.

Since the attributes — including the wired status — contained in a map entry structure apply to all pages mapped by the entry, changing the wiring of a single page will cause the map entry to be broken up into two or three map entry structures, each with different wiring counts. This can lead to map entry fragmentation, which is inefficient because map entries are stored on a linked list and each time a map entry is fragmented it increases the time it takes to search the list for a mapping. UVM includes improvements that attempt to avoid fragmenting map structures in several common wiring cases, thus reducing mapping fragmentation and the overhead associated with it.

Efficient traversal. Both the BSD VM system and UVM have several structures that can be traversed in more than one way. For example, the pages in a VM object are on both a linked list and in a hash table. BSD VM only traverses data structures in one way. In UVM the code dynamically chooses which style of traversal would have less overhead and uses that style of traversal. For example, when removing a single page from a one-hundred-page object, it makes sense to use the hash table. But when removing all one hundred pages from the object it makes better sense to use the linked list. UVM compares the size of the area being traversed with the total number of pages in the object to decide how to traverse it.

Reduced lock time during unmap. Two operations must be performed to remove mappings from an address space. First, one or more map entries are removed from the map, and second, the memory objects referenced by the map entries are dropped. In the BSD VM, the VM system locks the map for both of these operations. However, this is not really necessary. The map only needs to be locked when the map entry data structures are being changed, not when memory object references are being dropped.

UVM restructures the unmapping functions in such a way that these two functions are separated and the map can be unlocked during the reference drop. This is useful because if the unmap drops the last reference to a memory object it may cause the object's pager to perform I/O (possibly synchronous) as part of a cleanup operation. Since that I/O can take a while, in the BSD VM the map that is being unmapped would be locked throughout the I/O operation. However, in UVM the map is unlocked and can be accessed by other processes or the kernel as needed. The I/O will not block access to the map.

3.3 High-Level Design Overview

This section contains a high-level design overview of UVM. Like the BSD VM system, UVM is divided into two layers: the machine-dependent and machine-independent layers. The machine-dependent layer of UVM is almost identical to the machine-dependent layer of BSD VM. This allows BSD VM pmap modules to be used by UVM with only minor changes.

3.3.1 UVM's Machine-Independent Layer

The high-level functions of the VM system are handled by the machine-independent layer of UVM. The activities of the machine-independent layer are centered around eight major machine-independent data structures[1]. While UVM has more major data structures than BSD VM, UVM's data structures are generally smaller and used in a less complex ways. The UVM data structures are:

`vmspace`: describes a virtual address space of a process. The `vmspace` structure contains pointers to a process' `vm_map` and `pmap` structures, and contains statistics on the process' memory usage.

[1] UVM data structures are called "`vm_`" if they are either new or if they are minor modifications to BSD VM data structures. UVM data structures are called "`uvm_`" if there is a corresponding "`vm_`" data structure in BSD VM that is completely different from the `uvm_` one. Once UVM no longer has to co-exist in the source tree with BSD VM then the data structures will be renamed so that they are more uniform.

vm_map: describes the virtual address space of a process or the kernel. It contains a list of vm_map_entry structures that describe valid mappings and their attributes. The vm_map plays the same role in both BSD VM and UVM.

uvm_object: forms the lower layer of UVM's two-layer mapping scheme. A UVM object describes a file, a zero-fill memory area, or a device that can be mapped into a virtual address space. The uvm_object contains a list of vm_page structures that contain data from that object. Unlike VM objects, UVM objects are not chained for copy-on-write.

vm_amap: forms the upper layer of UVM's two-layer mapping scheme. A vm_amap describes an area of anonymous memory. The area may have "holes" in it that allow references to the lower underlying object layer.

vm_anon: describes a single virtual page of anonymous memory. The page's data may reside in a vm_page structure, or it may be paged out to backing store (i.e. the swap area).

vm_aref: a small data structure that points to a vm_amap and an offset in it. The aref structure is part of the map entry structure that is linked to the vm_map structure.

uvm_pagerops: a set of functions pointed to by a uvm_object that describe how to access backing store. In the BSD VM there is one vm_pager structure per memory object that accesses backing store, and each vm_pager structure points to a set of pager operations. In UVM the vm_pager layer has been removed and uvm_object structures now point directly to a uvm_pagerops structure.

vm_page: describes a page of physical memory. When the system is booted a vm_page structure is allocated for each page of physical memory that can be used by the VM system.

Figure 3.1 illustrates the general layout of the UVM data structures. The kernel and each process on the system have a vm_map structure and a machine-dependent pmap structure that define a vmspace. The vm_map structure contains a list of map entries that define mapped areas in the virtual address space. Each map entry structure defines two levels of memory. The map entry has a vm_aref structure that points to the vm_amap structure that defines the top-level memory for that entry. The vm_amap structure contains a list of vm_anon structures that define the anonymous memory in the top layer. The vm_anon structure identifies the location of its memory with a pointer to a vm_page and a location on the swap area where paged out data may reside. The map entry defines the lower-level memory with a pointer to a uvm_object pointer. The uvm_object structure contains a list of vm_page structures that belong to it, and a pointer to a uvm_pagerops structure that describes how the object can communicate with backing store.

The vm_page structures are kept in two additional sets of data structures: the page queues and the object-offset hash table. UVM has three page queues: the active queue, the inactive queue, and the free queue. Pages that contain valid data and are likely to be in use by a process or the kernel are kept on the active queue. Pages that contain valid data but are currently not being used are kept on the inactive queue. Since inactive pages contain valid data, it is possible to "reclaim" them from the inactive queue without the need to read them from backing store if the data is needed again. Pages on the free queue do not contain valid data and are available for allocation. If physical memory becomes scarce, the VM system will wake the pagedaemon and it will force active pages to become inactive and free inactive pages.

The object-offset hash table maps a page's uvm_object pointer and its offset in that object to a pointer to the page's vm_page structure. This hash table allows pages in a uvm_object to be looked up quickly.

UVM's simplified machine-independent layering makes supporting new UVM features such as page loanout and page transfer easier. For example, UVM's two-level amap-object mapping scheme simplifies the process of locating a page of memory over BSD VM's object chaining mechanism. And UVM's anon-based anonymous memory system allows virtual pages to be exchanged between processes without the need for object chain manipulations.

3.3.2 Data Structure Locking

Data structure locking is an important aspect of a virtual memory system since many processes can be accessing VM data concurrently. Locking can either be done with one large lock that locks all active VM data structures, or with multiple small locks. The use of multiple small locks is known as "fine-grain locking." Fine-grain locking is important on multiprocessor systems where each processor is running its own process; if one big lock was used on such a system only one processor would be allowed to access the VM structures at a time. There are two types of locks commonly used: "sleep locks" and "spin locks." With a sleep lock, if a process is unable to acquire the lock then it releases the

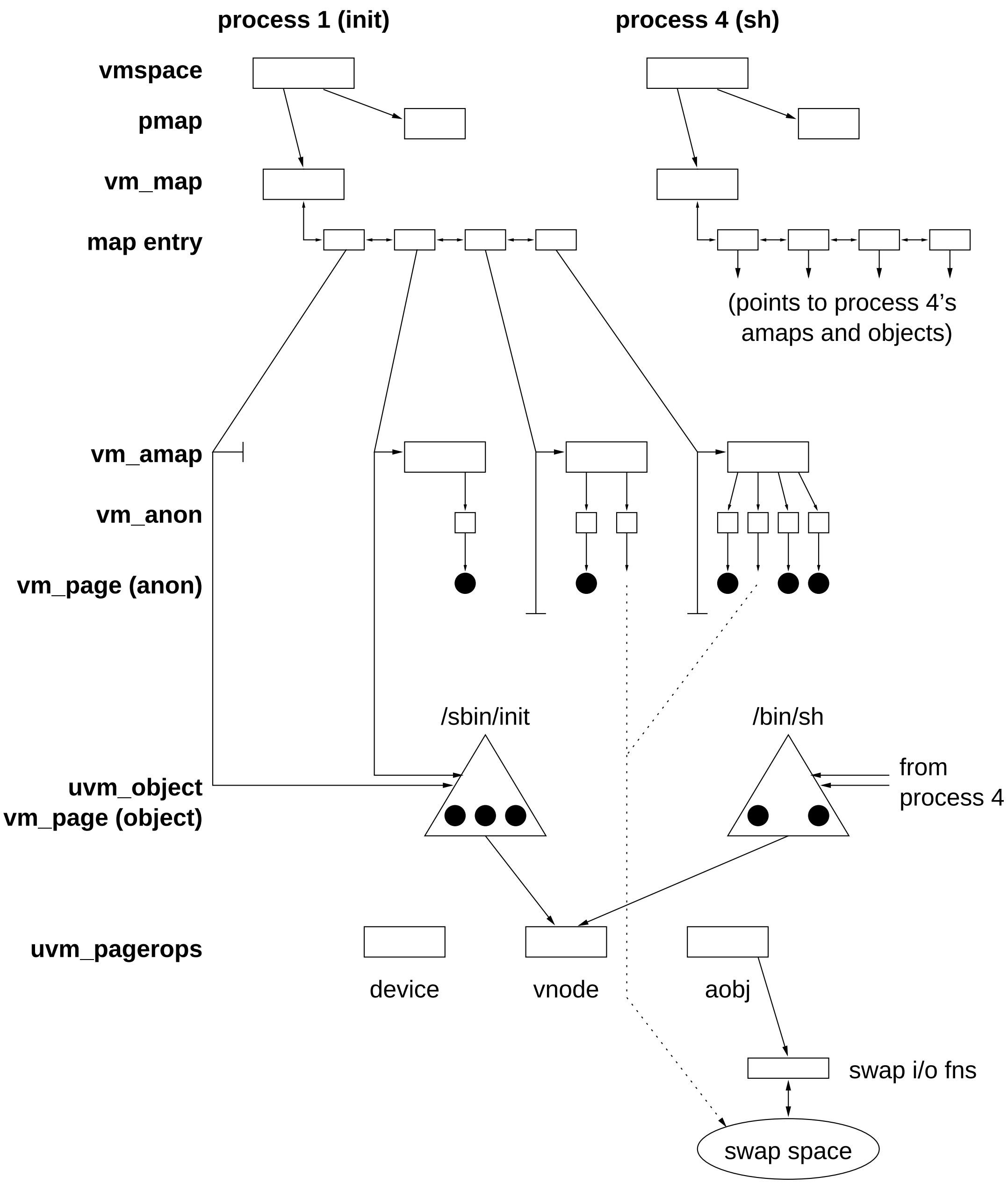


Figure 3.1: UVM data structures at a glance. Note that there is one `vm_aref` data structure within each map entry structure. The page queues and object-offset hash table are not shown.

processor until the lock can be acquired (thus allowing other processes to run). With a spin lock, the process continues to attempt to acquire the lock until it gets it.

The BSD VM system is based on the Mach VM system, which supports fine-grain locking of its data structures for multiprocessor systems. When Mach VM was ported to BSD, the fine-grain locking support was ignored because BSD was being used on single processor systems. Over time, as changes were made to BSD, the locking support decayed. While UVM currently is only being used on single processor systems, BSD is in the process of being ported to multiprocessor systems. Thus, as part of the UVM project, the fine-grain locking code was restored for future use. However, adding amaps and removing shadow/copy object chaining from the VM system fundamentally changes the way data structures are used, and thus the fine-grain locking scheme for UVM had to be redesigned to take these changes into account.

The main challenge with data structure locking is to choose a set of locks and a locking scheme that avoid both data corruption and deadlock. The fact that the BSD VM system's locking is not used, incomplete, and not well documented made repairing the damage and addressing the new data structures in UVM difficult. In addition to figuring out what locking scheme to use, the locking semantics of all the functions of the VM system had to be defined, and used consistently within all VM code. For example, some functions are designed to be called with unlocked maps while other are designed to be called with locked maps. Calling the function with the lock set the incorrect way will either lead to kernel data structure corruption or system deadlock.

A deadlock occurs when a process is waiting to obtain a lock that it is already holding (thus it can never acquire it), or when two or more processes are waiting in a circular pattern for locks that the other processes are holding (e.g. A is waiting for a lock that B is holding, and B is waiting for a lock that A is holding). It is well known that deadlock can be avoided if all processes allocate their locks in the same order (thus no loops can form). UVM uses this scheme to avoid deadlock. UVM also attempts to reduce contention for data structure locks by minimizing the amount of time locks are held. Before starting an I/O operation, UVM will drop all locks being held. Also, UVM does not hold any locks during the normal running of a process. So, when a VM request comes into UVM, it is safe to assume that the requesting process is not holding any locks.

The following data structures have locks in UVM. The data structures are presented in the order in that they must be locked.

map: the lock on a map data structure prevents any process other than the lock holder from changing the sorted list of map entries chained off the map structure. A map can be read or write locked. Read locking indicates that the map is just being read and no mappings will be changed. Write locking the map indicates that mappings will be changed and that the version number of the map should be incremented. Map version numbers are used to tell if a map has been changed while it was unlocked. A write-locked map can only be accessed by the holder of the lock. A read-locked map can be accessed by multiple readers. Attempting to lock a map that is already locked will cause the locking process to be put to sleep until the lock is available.

amap: the amap lock prevents any process from adding or removing anon structures from the locked amap. Note that this does not prevent the anon structures from being added or removed from other amaps.

object: the object lock protects the list of pages associated with the object from being changed. It also protects the "flags" field of the page data structure of all the pages in that object from being changed.

anon: the anon lock protects the page or disk block the anon points to from being changed. It also protects the "flags" field of the page data structure from being changed.

page queues: the page queue lock protects pages from being added or removed from the active and inactive page queues. It also blocks the page daemon from running.

For most of the UVM system this locking order is easy to maintain. However, there is a problem with the page daemon. The page daemon runs when there is a shortage of physical memory. The page daemon operates by locking the page queues and traversing them looking for pages to page out. If it finds a target page it must lock the object or anon that the page belongs to in order to remove it from that object. This is a direct violation of the locking order described above. However, this problem can be worked around. When the page daemon attempts to lock the memory object owning the page it is interested in it should only "try" to lock the object, as shown in Figure 3.2. If the object is already locked, then the page daemon should skip the current page and move on to the next one rather than wait for the memory object to be unlocked. Having the page daemon skip to the next page does not violate UVM's locking protocol and thus has no dire consequences.

3.3.3 VM Maps

In UVM, the `vm_map` structure and its corresponding list of `vm_map_entry` structures is handled in much the same way as in BSD VM. However, there are some differences. In

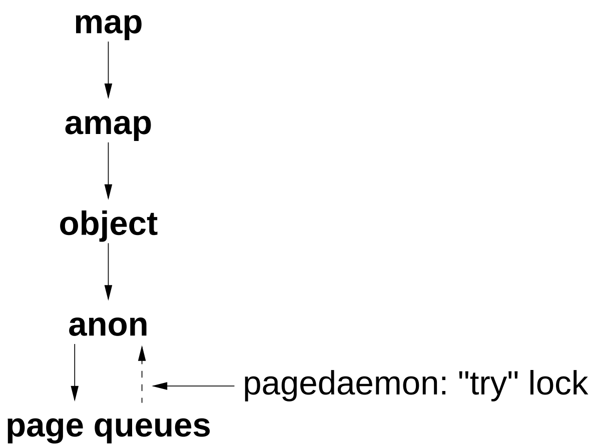


Figure 3.2: The UVM pagedaemon and lock ordering

UVM, the map entry structure has been modified for UVM's two-layer mapping scheme. Not only does it contain a pointer to the backing `uvm_object` structure, but it also contains a `vm_aref` structure. The aref points to any amap associated with the mapped region.

Some of the functions that operate on maps have also been changed. For example, UVM provides a new function called `uvm_map` that establishes a mapping with the specified attributes. The BSD VM system does not have such a function: mappings are always established with default attributes. Thus in the BSD VM, after establishing a mapping further function calls are often required to change the attributes from their default values to the desired values. On a multi-threaded system this can be a problem because the default value for protection is read-write. Thus, when establishing a read-only mapping of a file there is a brief window — between the time the mapping is established (with the default protection) and the time the system write-protects the mapping — where the map is unlocked and could be accessed by another thread, thus allowing it to by-pass system security.

UVM also provides a corresponding `uvm_unmap` call. The unmap call has been restructured so that it holds the lock on a map for a shorter amount of time than the corresponding BSD VM unmap call.

For features such as page loanout and map entry passing, new map related functions had to be written. These functions perform a variety of tasks including:

- reserving a block of virtual memory in a map for later use.
- extracting a list of map entries from a map. The extracted list can either be a copy of the map entries in the source map or the actual map entries from the source map (in which case the mappings are being removed from the source map).

- replacing a map entry that is reserving a block of virtual address space with a new set of map entries.
- extracting pages for loanout. The two-level structure of the amap layer makes it easy to do this. To extract the pages, the loan out routine only needs to look in one of two places for each page being loaned, as compared to the BSD VM where loan out code would have to walk the object chains for each page it wished to loan.

3.3.4 UVM Objects

The role of a `uvm_object` in UVM is somewhat different than the role of a `vm_object` in BSD VM. In BSD VM, the object structure is considered a stand-alone structure under the complete control of the VM system. The BSD VM system has full control over when `vm_object` structures are allocated, when they can be referenced, and how they can be used. In UVM, the `uvm_object` is considered a secondary structure. It is usually embedded within some larger structure in order to provide the VM system a "handle" for memory-mapping the structure. All operations performed on a UVM object are routed through the object's pager operations. The BSD VM system has functions that operate directly on VM objects. For example, in BSD VM there are functions to create new shadow objects, collapse object chains, adjust the object cache, and change the protection of an object's pages. In UVM there are no such functions: they have either been moved to the pager, or in the case of functions relating to object chaining and the object cache, they have been removed all together.

Thus, UVM's `uvm_object` structure shown in Figure 3.3 is simpler than BSD VM's `vm_object` structure. It contains a spin lock, a pointer to its `uvm_pagerops`, a list of pages belonging to it, the number of pages belonging to it, and a reference counter. There are no shadow object pointers, copy object pointers, or object cache pointers in UVM.

3.3.5 Anonymous Memory Structures

UVM has three anonymous memory related data structures: arefs, amaps, and anons. UVM's anonymous memory handling will be examined in detail in Chapter 4, so only a brief discussion of it will be included here.

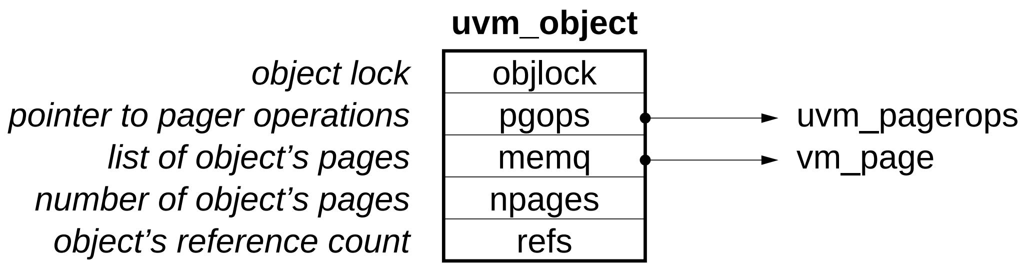


Figure 3.3: The UVM object structure

Map entry structures contain aref structures. Each aref can point to an amap structure. The amap structure forms the top layer of UVM's two-layered virtual memory mapping scheme, and the `uvm_object` forms the backing layer. Each amap can contain one or more anon structures. Each anon represents a page of anonymous virtual memory. An anon's data can either be resident in a page of physical memory (in which case the anon has a pointer to the corresponding `vm_page` structure), or it can be paged out to the swap area of the disk (in which case the anon contains the index into the swap area where the data is located).

Copy-on-write is achieved in UVM using the amap layer. Copy-on-write data is originally mapped in read-only from a backing `uvm_object`. When copy-on-write data is first written, the page fault routine allocates a new anon with a new page, copies the data from the `uvm_object`'s page into the new page, and then installs the anon in the amap for that mapping. When UVM's fault routine copies copy-on-write data from a lower-layer `uvm_object` into an upper-layer anon it is called a "promotion." Once copy-on-write data has been promoted to the amap layer, it becomes anonymous memory. Anonymous memory also can be subject to the copy-on-write operation. An anon can be copy-on-write copied by write-protecting its data and adding a reference to the anon. When the page fault routine detects a write to an anon with a reference count greater than one, it will copy-on-write the anon.

The introduction of the amap layer and the removal of object chaining had a significant effect on the handling of page fault routines, and thus the page fault routine had to be rewritten from scratch. This is because in the BSD VM when a page fault occurs the object chains must be traversed. But in UVM, rather than start looking directly at VM objects (or object chains) the fault routine first looks at the amap layer to see if the needed pages are there. If there is no page in the amap layer, then the underlying object is asked for it. If the underlying object doesn't have the needed data, then the fault routine fails. Thus, in

UVM's fault routine, there are only two places to check for data and no shadow linked lists have to be traversed or managed.

The amap concept was first introduced in the SunOS4 VM system [28, 46]. In UVM, the SunOS amap concept was enhanced, and then adapted for the more Mach-like environment provided by UVM. Amaps and anons may also be used for page loanout and page transfer.

3.3.6 Pager Operations

The `uvm_pagerops` structure defines the set of operations that can be performed on a `uvm_object` structure. Each `uvm_object` has a pointer to its pager operations. This is different from the BSD VM system in which the `vm_object` structure had a pointer to a `vm_pager`, and that `vm_pager` structure had a pointer to a set of pager operations. In UVM the extra `vm_pager` layer has been eliminated and a new set of pager operations have been created. UVM's pager operations include functions such as ones that add and drop references to a `uvm_object` and functions that get and put pages from backing store. Details on UVM's pager operations can be found in Chapter 6.

3.3.7 Pages

In UVM, the `vm_page` structure has been updated for amap-based anonymous memory and for page loanout. In the BSD VM system, an allocated page of memory can be associated with a `vm_object`. That object could be a normal object, or it could be a shadow or copy object that was created as part of a copy-on-write operation. In contrast, in UVM a page can be associated with either a `uvm_object` or an anon structure.

For page loanout, a loan counter has been added to each page structure. A page is loaned out if it has a non-zero loan count. One of the challenges faced when designing the page loanout system was to determine how to handle cases where the VM system wants to do something to a page that is currently on loan. Often this will require that the loan be "broken." If a process or the kernel tries to modify a loaned page, then the loan must be broken. If memory becomes scarce and the VM system wants to pageout a loaned page, then the loan must be broken. If the VM system tries to free or flush out a loaned page then the loan must be broken. All these cases must be handled properly by the page loanout code, or data corruption will result. Thus, the addition of the loan count to the `vm_page` structure was critical in getting loanout to work properly.

Table 3.1: Changes from BSD VM. The "extent of change" values are: *new* (new code), *replaced* (totally rewritten code), *revised* (same basic design as BSD VM, but revised and improved for UVM), *adjusted* (same basic design as BSD VM with minor cleanups), *removed* (BSD layer removed from UVM). Each function corresponds to a UVM source-code file.

Function	Extent of Change	Description
amap	new	anonymous memory map management
aobj	replaced	anonymous memory object pager, replaces the default pager of BSD VM.
device	replaced	device pager that conforms to the new UVM pager interface
fault	replaced	page fault handler that handles the new anonymous memory layer
init	adjusted	VM startup routine
km	revised	kernel memory management, revised for new UVM pager and memory mapping interfaces
loan	new	page loaning — new UVM feature
map	new/adjusted	memory mapping routines — new features include map entry passing and related support functions
map i/o	new	map I/O function (allows reads and writes to a map)
mmap	replaced/adjusted	`mmap` system call replaced, other related system calls adjusted
object	removed	object management code, obsolete due to new UVM pager and anonymous memory handling
page	revised	page management, revised for new non-contiguous physical memory handling
pager	replaced/revised	pager interface — BSD VM's pager structure has been removed and the pager operations revised
pdaemon	revised	the pageout daemon, revised for new anonymous memory handling
swap	adjusted	NetBSD specific swap-space management
vnode	revised	vnode pager, revised for UVM pager interface

Table 3.2: Issues addressed by UVM and the sections of this dissertation that address them. The issues are grouped into eight categories: general, anonymous memory, page fault, pager, page, map, pmap, and tools and implementation methods.

Category	Issue	Section	Description
general	locking	3.3.2	design of a new locking scheme that takes into account UVM data structure changes and restores fine-grain locking dropped in BSD VM
anonymous memory	amap	4	design of a new anonymous memory system based on the amap idea from SunOS 4 with UVM improvements
	inheritance	4.7	a problem faced when trying to get an amap-based anonymous memory system to support Mach-style memory inheritance during a `fork` operation
	stack allocation	8.1	a memory-usage problem faced when trying to use BSD-style stack allocation with an amap-based anonymous memory system
	partial unmap memory leak	4.3.2	design of a new scheme for releasing anonymous memory associated with a map entry when that map entry is only partially unmapped.
page fault	fault	5	design of a new page fault handler that uses amap-based anonymous memory for copy-on-write rather than Mach object chains and also attempts to reduce future faults by taking into account the memory usage of the faulting process
pager	pager	6	design of a new pager interface that removes an unnecessary layer (the Mach VM pager layer) and changes the interface to backing store to match the new amap-based anonymous memory system and to give the pager more power over its pages
	clustered pageout	8.2	a new design that allows the page daemon to dynamically cluster anonymous memory being paged out into large chunks, thus reducing pageout overhead

Table 3.2: (continued)

Category	Issue	Section	Description
	fictitious device pages	8.4	redesign of the device pager to use the new pager fault interface so that it no longer needs to create and pass "fictitious" vm_page structures up to the rest of the VM
	aobj pager	8.3	design of a new pager that supports UVM objects that are backed by anonymous memory; this pager is used to provide the System V shared memory interface, and also to support pageable kernel memory
	vnode pager	8.5	redesign of the vnode pager to allow the kernel's vnode system to control the persistence of vnode VM data rather than have it done both by the VM and the vnode systems
	vnode sync	8.5.3	improved the design of the memory sync code to only consider vnodes that have writable pages rather than all vnodes on the system
	async I/O	6.3.3	redesign of the asynchronous I/O interface to clean up its handling
page	loanout	7.1	a new design that lets virtual memory pages be read-only "loaned" to the kernel or to other processes (in the form of anonymous memory) without disrupting the normal operation of the VM system
	transfer	7.2	a new design that allows other kernel subsystems to donate their pages to the VM system; these pages become anonymous memory and can be mapped into processes
	VM startup	8.12	design of a new VM startup procedure that has a unified way of handling contiguous and non-contiguous physical memory and reduces the number of boot-time statically-allocated data structures
	new page flags	8.11	a design cleanup of the page flags to reduce the number of flags and more clearly define their locking

Table 3.2: (continued)

Category	Issue	Section	Description
map	map entry passing	7.3	design of a map entry passing system that allows virtual memory to easily be passed between processes in a number of ways
	map extraction	7.3.2	design of a map entry extraction function that allows map entry passing and I/O on a map; this is used for the `ptrace` system call and for the process filesystem (procfs)
	mapping	8.6	redesign of the mapping interface to use a single function that provides all necessary features and reduces the need for later VM calls
	unmap	8.7	redesign of the memory unmapping functions so that they only hold the map lock during address space changes, not during reference drops
	kernel memory management	8.8	adjusted kernel memory allocation and management functions to take advantage of the features of the new memory mapping function
	wired memory	8.9	revised the handling of wired memory to reduce map entry fragmentation both in the kernel and in user processes
	buffer map	8.10.1	redesigned the memory allocation of the BSD buffer cache to eliminate the need for `buffer_map`; this reduces the number of statically allocated map entry structures
pmap	new i386 pmap	8.14	rewrite of the i386 MD pmap module to better manage memory and incorporate improvements from other operating systems such as FreeBSD
	new pmap interface	8.13	minor design modifications to the pmap interface to better support page loaning and clean up the interface (PMAP_NEW)
tools and implementation methods	UVM history	9.2	a tool that keeps a running log of functions called during the operation of the VM system

Table 3.2: (continued)

Category	Issue	Section	Description
	VM status system calls	9.4	a system call that allows an application to get the status of the VM system without needing access to `/dev/kmem`; two sample applications are included with UVM: one text based, and one X11 based
	redundant calls	9.2.2	discussion of the use of UVM debugging tools used in the detection of redundant VM calls in the BSD kernel's fork/exec/exit path
	amap overallocate	9.2.3	discussion the use of UVM debugging tools in the detection of the amap overallocation problem and the solution to it
	UVM ddb commands	9.3	discussion of the improvements made to ddb, the BSD kernel debugger, in order to help debug virtual memory problems; this includes new features such as busy-page ownership tracking
	old mmap calls	8.10.2	discussion of the need to detect and remove old MAP_FILE `mmap` system calls

3.4 Related Work

In this section we present an overview of research that is related to UVM. Such work can be divided into two categories:

1. research on the design and implementation of other virtual memory systems.
2. research on I/O and IPC subsystems that take advantage of virtual memory services to improve performance.

We will examine related work in both categories.

3.4.1 Other Virtual Memory Systems

In this section we examine the structure of other virtual memory systems and examine the set of features that they provide. Note that the BSD VM system was described in detail in Chapter 2.

The Mach Virtual Memory System

The Mach virtual memory system is used for memory management and IPC in the Mach micro kernel developed at Carnegie Mellon University [4, 57, 64, 71, 72]. The BSD VM system is a simplified version of the Mach virtual memory system. BSD VM and Mach VM both use the same mapping structure to map processes address space, shadow and copy object chains for copy-on-write memory, and a map-based machine-dependent virtual memory layer. However, Mach VM differs from BSD VM in the areas of pager support and IPC data movement. These differences make virtual memory management under Mach much more complex than managing memory under BSD.

Mach is a microkernel. This means that only the core aspects of the operating system are actually part of the kernel. These services include virtual memory management, interprocess communication, and task scheduling. All other aspects of the operating system such as protocol processing, filesystems, pagers, and system call handling have been pushed out into separate "server" processes. Processes communicate with the kernel and each other by sending messages to each other using Mach message passing IPC. The goal behind this structure is to enforce modularity on the operating system and to ease debugging by allowing the Mach servers to be debugged without the need to reboot the microkernel.

Message passing. Because of the number of server processes needed to provide a useful operating system the performance of Mach is greatly dependent on the performance of its message passing mechanism. Mach message passing operates in one of two ways. If the message is small, then the data is copied into a kernel buffer and then copied out to the recipient. However, if the message is large then the virtual memory system is used to transfer the message's data. Since the data travels separately from the message this is called an "out-of-line" message.

Out-of-line messages are transferred between address spaces using a virtual memory data structure called a *copy map*. In this context a copy map can be thought of as a `vm_map` that is not associated with any process. To send a Mach message, the virtual address space containing the message is copy-on-write copied into the copy map using Mach's shadow and copy objects. The copy map is then passed to the recipient of the message. The recipient then copies the message to its address space. This mechanism is similar to using the copy-on-write feature of UVM's map entry passing mechanism.

The out-of-line message passing facility was found to be too expensive in a networking environment due to the high overhead of Mach virtual memory object operations [4]. To solve this problem, the copy map was modified. Instead of just being able to transfer a list of map entries, the copy object was overloaded to also be able to transfer a list of vm_page structures. This reduces the overhead of sending data over a network. To send a message with this mechanism the data pages are faulted in, marked busy, and put in the page list of the copy map. The copy map is then passed to the networking system which transmits the pages and then clears the busy bit. The pages are marked busy while they are in transit, so they cannot be modified until the network operation is complete. Further optimizations to this approach allow pages to be “pinned” in memory for some cases, but for other cases the data has to be “stolen” (i.e. copied into a freshly allocated page). A copy map may contain only a limited number of pages. If the message contains more pages than will fit in a copy map, then a “continuation” is used. A continuation is a call back that allows the next set of pages to be loaded into a copy map.

External pagers. Another feature of Mach is external pagers. In BSD VM all pagers are compiled into the kernel and are trusted. Mach allows a user process (possibly a Mach “server”) to act as a pager for a vm_object. When Mach’s pagedaemon wants to pageout a page that is managed by an external pager it has to send that pager a message to pageout the page. Since the pager is an untrusted user process it may decide to ignore the pagedaemon’s pageout request, thus creating an unfair situation. To address this problem, the copy map structure was modified to contain a pointer to a copy map copy object. When the pagedaemon wants to page out a page of memory managed by an untrusted external pager it allocates a new copy map and a new vm_object to act as that copy map’s copy object. The pages to be paged out are then inserted into the copy map’s copy object and the copy map is passed to the external pager. The copy map’s copy object is managed by the default pager. The default pager is a trusted pager that is part of the kernel and pages anonymous memory out to the swap area of disk. So, if an external pager ignores a pageout request from the pagedaemon its pages will get paged out by the default pager through the copy map’s copy object. Note that this creates a problem: the pages to be paged out have to belong to both the vm_object that belongs to the external pager and to the copy map’s copy object. Mach solves this problem by allowing pages to be “double mapped.” A double mapped page is a page of physical memory that has two or more vm_page structures that refer to it. In our example, the main vm_page will belong to the externally managed object and the second vm_page belongs to the copy map’s copy object. Note that neither BSD VM nor UVM need or allow double mapped pages. In BSD VM there is a one-to-one correspondence between a vm_page and a page of physical memory.

In current versions of Mach a copy map can contain one of four possible items:

- a list of map entries (like a normal map)

- a copy object
- a list of pages
- a pointer to an area of kernel memory

There are a number of functions that convert copy maps between these four formats (but not all conversions are supported).

Fictitious pages. Mach also makes heavy use of fictitious pages. A fictitious page is a `vm_page` structure that has no physical memory allocated to it. These pages are often used as busy-page place holders to temporarily block access to certain data structures. Mach freely moves physical memory between normal pages and fictitious pages, so normal pages can easily become fictitious and vice-versa.

Mach VM suffers from the same sort of problems that BSD VM does. It has a complex multi-level object-chaining based copy-on-write and mapping mechanism. It does not allow pages to be shared copy-on-write without using this mechanism. Thus, it is difficult to have page-level granularity without the extra overhead of allocating numerous objects. This combined with its page wiring mechanism can lead to map entry fragmentation. It also suffers from the same partial unmap swap-memory leak problem that BSD VM does.

The FreeBSD Virtual Memory System

The FreeBSD virtual memory system is an improved version of the BSD VM system [25]. One of the main emphases of work on the FreeBSD VM system is ensuring that FreeBSD performs well under load. Thus, FreeBSD is a popular operating system for network and file servers.

Work on FreeBSD VM has focused on a number of areas including simplifying data structure management, data caching, and efficient paging algorithms. FreeBSD VM's data structures are similar to BSD VM's data structures, although some structures have been eliminated. For example, FreeBSD VM no longer has share maps or copy object chains. Since neither of these are needed to provide a Unix-like virtual memory environment their elimination reduces the complexity of FreeBSD VM. While FreeBSD retains Mach-style shadow object chaining for copy-on-write, the swap memory leaks associated with BSD VM's poor handling of the object collapse problem have been addressed. FreeBSD has successfully merged the VM data cache with the Unix-style buffer cache thus allowing VM and I/O to take better advantage of available memory. FreeBSD's paging algorithms use clustered I/O when possible and attempt to optimize read-ahead and pageout operations to minimize VM I/O overhead. These paging algorithms contribute to FreeBSD's good performance under load.

FreeBSD's improvements to object chaining produce similar performance improvements as UVM's elimination of object chaining, however the complexities of object chaining remain in FreeBSD. Many of FreeBSD's improvements in the areas of data caching and paging algorithms are applicable to UVM, thus FreeBSD will be a good reference for future work on UVM in these areas. FreeBSD does not have UVM-style features such as page loanout and page transfer. These features could be added to FreeBSD with some difficulty, alternatively object chaining could be eliminated from FreeBSD and UVM features added.

The SunOS/Solaris Virtual Memory System

The SunOS4 virtual memory system is a modern VM system that was designed to replace the 4.3BSD VM system that appeared in SunOS3 [28, 46]. This virtual memory system is also used by Sun's Solaris operating system.

The basic structure of the SunOS virtual memory system is similar to the structure of BSD VM. The kernel and each process on the system have an address space that is described by an `as` structure. The `as` structure contains a list of currently mapped regions called segments. Each segment is described by a `seg` structure. The `seg` structure contains its starting virtual address, its size, a pointer to a "segment driver" and a pointer to the object mapped into that segment. A segment driver is a standard set of functions that perform operations on a memory mapped object. SunOS's `as` and `seg` structure correspond to BSD VM's `vm_map` and `vm_map_entry` structures. The SunOS segment driver corresponds to BSD VM pager operations. The SunOS mapped object corresponds to BSD VM's `vm_object` structure. Once difference between SunOS VM and BSD VM is that in SunOS the format of the memory object structure is private to the object's segment driver. Thus, the pointer to the memory object in the `seg` structure is a void pointer. This forces the pointer to the segment driver function to be in the `seg` structure. BSD VM stores the pointer to its pager operation structure off of its `vm_object` structure (through the `vm_pager`). Both SunOS VM and BSD VM have "page" structures that are allocated for each page of physical memory at VM startup time.

The division of labor between the pager operations and the high-level VM in SunOS VM is different from BSD VM. In SunOS almost all high-level decisions are deligated to the segment driver, while in BSD VM the high-level VM handles most of the work and only references the pager if data transfer is needed between physical memory and backing store. For example, in BSD VM the high-level fault routine calls the pager to fetch a page from backing store. It then goes on to handle copy-on-write and mapping the new page into the faulting address space. In contrast, the high-level SunOS VM fault routine determines which segment the page fault occurred in and calls that segment driver's fault routine to resolve the fault. Issues such as copy-on-write and memory mapping in the faulting page must are handled by the segment driver, not the high-level fault routine.

Both BSD VM and SunOS VM take a similar approach to dividing the machine-dependent and machine-independent aspects of the virtual memory system. SunOS has a machine-dependent layer called the hardware address translation (HAT) layer. The SunOS HAT layer corresponds to the BSD VM pmap layer.

SunOS VM manages copy-on-write and anonymous memory through the use of amaps and anons. In UVM, the SunOS amap concept was enhanced, and then adapted for the more Mach-like environment provided by UVM. In SunOS amaps are not a general purpose VM abstraction as they are in UVM. SunOS amaps can only be used by segment drivers that explicitly allow amaps to be used. Once such segment driver is the vnode segment driver. In SunOS areas of memory that are mapped shared must always remain shared and areas of memory that are mapped copy-on-write must always remain copy-on-write. Thus, SunOS cannot support Mach-style memory inheritance or certain types of map entry passing. UVM has no such restriction. SunOS's amap implementation does not appear to support UVM-style quick traversal or deferred amap allocation during a fork.

One improvement introduced to SunOS VM in Solaris is the introduction of the "virtual swap file system" [13]. In SunOS4 each anon on the system was statically assigned a pageout location in the system's swap area. This forced the size of the swap area to be greater than or equal to the size of main memory. The virtual swap file system allows the backing store of an anon to be dynamically assigned. UVM has dynamically assigned backing store for anonymous memory as well, but a filesystem abstraction is not necessary, and thus is not used. Furthermore, UVM extends this idea to implement aggressively clustered anonymous memory pageout. In this form of pageout a cluster of anonymous memory is dynamically assigned a contiguous block of backing store so that it can be paged out in a single I/O operation.

SunOS VM currently does not provide UVM-like features such as page loanout, page transfer, and map entry passing. Thus, for bulk data transfer under SunOS one would have to use traditional mechanisms such as data copying. However, SunOS's anon-style anonymous memory system and modular design would ease the implementation of UVM-style features.

The Linux Virtual Memory System

Linux is a popular free Unix-like operating system written by Linus Torvalds [66]. We examined the virtual memory system that appears in what is currently the most recent version of Linux (version 2.1.106).

In Linux, the kernel and each process has an address space that is described by a `mm_struct` structure. The `mm_struct` structure contains a pointer to a sorted linked list of `vm_area_struct` structures that describe mapped regions in the address space. Each mapped area of memory has a set of flags, an offset, and a pointer to the file that is mapped into that area. Each page of physical memory has a corresponding `page` structure.

VM Layering. Linux VM's machine-dependent/machine-independent layering is quite different from either SunOS VM or BSD VM. In BSD VM the machine-dependent pmap module provides the machine-independent VM code with a set of functions that add, remove, and change low-level mappings of pages. In Linux, the machine-independent code expects the machine-dependent code to provide a set of page tables for a three-level forward mapped MMU. The machine-independent code reads and writes entries to these page tables using machine-dependent functions (or macros). For machines whose MMU matches the Linux machine-independent model the machine-dependent code can arrange for the machine-independent code to write directly into the real page tables. However, machines whose MMU does not match the Linux three-level MMU machine-independent model must both emulate this style MMU for the machine-independent code and internally translate MMU requests to the native MMU format. An unfortunate result of this arrangement is that hooks for machine-dependent cache and TLB flushing must appear throughout the machine-independent code. In SunOS and BSD VM such machine-dependent details can safely be hidden behind the HAT/pmap layer. Another unfortunate result of this structure is that all machine-independent virtual memory operations that operate on a range of virtual addresses must be prepared to walk all three page table levels of the machine-independent MMU model.

Linux requires that the entire contents of physical memory be contiguously mapped into the kernel's address space. Thus, the kernel's virtual address space must be at least as large as physical memory. By mapping all physical memory into the kernel's address space Linux can access any page of memory without having to map it in. It can also easily translate between `page` structure pointers and physical addresses.

Copy-on-write. Linux handles copy-on-write as follows. First, each page of memory has a reference counter. Each mapping of the page counts as a reference. Also, pages appearing in the buffer cache also have an reference. Thus any page that is mapped in directly from a file will have a reference count of at least two. Any page whose reference count is greater than or equal to two is considered “shared.” Copy-on-write memory is identified by a copy-on-write flag in the `vm_area_struct` structure that maps the region. The first read-fault that occurs on a copy-on-write area of memory will cause the backing file’s page to be mapped in read-only. When a write-fault occurs on a copy-on-write area of region the faulting page’s reference count is checked to see if the page is shared. If so, then a new page is allocated (with a reference count of one), the data from the old page is copied to the new page, and the new page is mapped in. If a process with a copy-on-write region forks, the copy-on-write region is write protected and the reference counter for each mapped page is incremented. This causes future write-faults to perform a copy-on-write.

Copy-on-write memory can be paged out to swap. The swap area is divided into page-sized blocks. Each block has a reference counter that says how many page tables are referencing it so the swap subsystem knows when it is no longer in use. To page a copy-on-write page out to swap the page’s PTE is replaced with an invalid PTE containing the address on swap where the data is located and the page is placed in the “swap” file object. The page’s offset is set to the address of its location on swap. When all PTEs pointing to a copy-on-write page are invalidated the page is removed from the swap file and object and freed for reuse. Note that Linux attempts to place successive swap block allocations in the same area of swap to reduce I/O overhead (but it appears to write to swap page-at-a-time).

When a process references a page who’s PTE has been modified to point to an area of swap a page fault is generated. The page fault routine extracts the location of the page on swap from the PTE and searches the swap file object to see if the page is still resident. If so, the page is mapped in. If the fault is a write fault, the swap block is freed since the data will be modified. If the page is not resident in the swap file object a new page is allocated, added to the swap file object at the appropriate offset, and read in from swap. Then the page can be mapped as before.

Page tables. Note that in a HAT or pmap based VM system, the information stored in the hardware page tables can safely be thrown away because it can be easily reconstructed based on information in the address space or map structure. Systems that use this approach will often call on the HAT or pmap layer to free memory when a process gets swapped out or memory becomes scarce. However, in Linux the page tables are being used to store swap location information, and thus the information contained within them cannot be freed. One way to allow such memory to be freed is to allow certain page tables to be paged out themselves as in Windows-NT [62], however Linux currently does not appear to support this.

Linux, like SunOS, does not support the sharing of copy-on-write memory or memory inheritance. Additionally, since Linux stores copy-on-write state in its page tables, operations such as map entry passing could be expensive since they could require traversing all the page table entries for a mapped region as well as the high-level map in the `mm_struct`. Linux does have some support for remapping memory within a process. The non-standard `mremap` system call is used by some versions of Linux's `malloc` memory allocator to resize its heap. The `mremap` system call takes a range of mapped memory and changes its size. If the new size is smaller than the old, then the extra memory is simply unmapped. If the new size is larger than the old and there is room to grow the allocation in place, then `mremap` does so. However, if there is not enough room, then `mremap` moves the mapped memory to a new location that is large enough for the new size. The current version of `mremap` has two limitations: it will not work across VM area boundaries and it does not handled wired memory properly. Additionally, the `mremap` system call has an interesting side effect. If the region of memory being remapped points to a file rather than a zero-fill area memory and the size of the region is being increased then the amount of the file mapped is increased. This may have security implications because it is giving the user the ability to change a file's mapping without the need for a valid open file descriptor on that file. Since such a feature is not necessary for `malloc` it could easily be removed if it is determined to be a problem.

The Windows-NT Virtual Memory System

Windows-NT is the most advanced of Windows operating system from Microsoft [62]. Although not a Unix-style operating system, NT's virtual memory system shares many aspects of such systems. Under NT, each process has its own private virtual address space that is described by a list of virtual address space descriptors (VADs). Unlike BSD VM's map entry structures, which are organized as a sorted linked list, VADs are arranged in a self balancing binary tree data structure. Each VAD contains a protection and pointer to the *section object* that it maps. A section object is the NT-internal name for a memory mapped file.

Memory allocation. NT allows processes to allocate and deallocate memory in their address spaces in two phases. Virtual memory can be "reserved" by adding an entry to the VAD list. Reserved virtual memory must remain unused until it is "committed" by associating it with a mapped memory object. Processes can "decommit" and "free" memory when it is no longer needed. This two-phase system is used by NT for stack allocation. NT also allows processes to reserve and commit memory with a single system call.

Page table and page management. NT's management of pages tables is similar to the way Linux's VM manages them. But unlike Linux, which assumes a three level MMU structure, NT assumes a two-level MMU structure. If the hardware does not directly support such a page table structure, then the machine-dependent NT hardware abstraction layer (HAL) must translate between two-level page tables and the native MMU format.

NT also maintains a page frame database (PFN database) that contains one database entry for each page of physical memory on the system. An entry in the PFN database contains a share (reference) count, the physical address of the page, and the rest of the page's current state.

Prototype PTEs. NT manages pages that may be shared through a mechanism called "prototype PTEs." Under NT memory can be shared either through shared memory mappings or copy-on-write memory mappings (before the write occurs). Each memory mapped section object has an array of prototype PTEs associated with it. These prototype PTEs are used to determine the location of each page that the section object contains. Such pages could be resident, zero-fill, or on disk. While the format of prototype PTEs is similar to the format used in real PTEs, prototype PTEs are only used to keep track current location of a page of memory and never actually appear in real page tables. For example, when a memory mapped page is first faulted on, NT first checks the page tables of the faulting process to determine the PTE of the page that caused the fault. Since this is the first time the page is being faulted, the PTE contains no useful information, so NT checks the VAD list to determine which section object is mapped into the faulting virtual address. Once the section object is found, the prototype PTE entry for the faulting page is located. If the prototype PTE indicates that the page is resident in physical memory, then the prototype entry is copied into the faulting process' page table with the protection set appropriately. The reference counter for the page in the PFN database is also incremented. If the prototype PTE does not point to physical memory then it will either point to a page-sized block of a swap file or normal file, or it will indicate that the page should be zero-filled.

When a page of physical memory is removed from a process' address space by the NT pagedaemon the PFN database's reference counter for that page is decremented and the process' PTE mapping the page is replaced with a pointer to the prototype PTE for that page. This pointer is treated as an invalid PTE by the hardware. If the PFN database reference counter reaches zero, then page of physical memory is no longer being mapped and it can be recycled. This reference counter is also used during a copy-on-write fault to determine if a copy needs to occur.

NT does not support the sharing of copy-on-write memory or memory inheritance. Additionally, since NT, like Linux, stores copy-on-write state in its page tables, operations such as map entry passing could be expensive since they may require traversing all page table entries for a mapped region as well as the VADs. NT does have an internal-only IPC facility called local procedure calls (LPC) that uses virtual memory features in some cases to transfer data. For LPC messages of less than 256 bytes data copying is used. For LPC messages larger than that, a shared section object is allocated and used to pass data. For very large data that will not fit in a shared section, NT's LPC mechanism allows the server to directly read or write data from the clients address space.

Other Virtual Memory Systems

Sprite was an experimental operating system that was developed at University of California at Berkeley in the late 1980s [51]. Sprite's original virtual memory system provided the basic functionality of the 4.2BSD virtual memory system [48]. Thus, it was lacking support for copy-on-write, `mmap`, and shared libraries. This allowed page table management to be simplified by associating page tables with files rather than processes. These page tables could be shared among all processes. Sprite did support a primitive form of memory inheritance. When a process forked, it had the option of sharing its data and heap segments with its child process. Sprite was later modified to support copy-on-write [49] for data and stack pages. This was done through a segment list structure. Each copy-on-write copy of a segment adds a segment to the segment list. Only one of the segments, the master segment, on the segment list can have a valid mapping of the page at any one time. The PTEs in the rest of the segments contain pointers to the master segment (the hardware treats these pointers as invalid entries). If a copy-on-write fault occurs on the master segment, then a copy is made and another segment is made master. If a copy-on-write fault occurs on a non-master segment, then the data is just copied to a private page.

Chorus is a microkernel based operating system with an object-oriented virtual memory interface developed in the early 1990s at Chorus Systems and INRIA, France [1, 2, 17]. The interface for the virtual memory system is called the "generic memory management interface" or GMI. The GMI defines a set of virtual memory operations that the Chorus microkernel supports. Each process under Chorus has virtual memory "context" that consists of a set of mapped “regions.” This corresponds to Mach’s maps and map entries. Each region points to a local “cache” that maps a “segment.” Data in a segment is accessed through its “segment manager.” A cache corresponds to a Mach memory object, a segment corresponds to backing store, and a segment manager corresponds to a Mach pager. The operations that the GMI defines that can be performed on these objects include adding and removing a mapped region from a context, splitting a mapped region into smaller parts, changing the protection of a mapped region, and locking and unlocking a mapped region in memory. The “paged virtual memory manager” (PVM) is a portable implementation of the GMI interface for paged memory architectures. PVM includes both the hardware-independent and hardware-dependent parts of the virtual memory system. PVM supports copy-on-write through the use of “history objects.” History objects are similar to Mach’s shadow object chains. They form a binary tree that must be searched to resolve a copy-on-write page fault. Chorus supports a per-page form of copy-on-write through the use of “copy-on-write page stubs” which are similar to fictitious pages in Mach. These stubs are kept on a linked list that is rooted off the real page’s descriptor.

Spring is an experimental object-oriented microkernel operating system developed at Sun [35, 31, 44]. Each process under Spring has an address space called a “domain.” Each domain contains a sorted linked list of regions that are mapped into it. Each region points to the memory objects that it maps. Mapped memory object regions can be copy-on-write copied into other address spaces. A cache object is a container for physical memory. A pager object is used to transfer data between physical memory and backing store. Spring, like Mach, supports external pagers. Spring also provide coherent file mappings across a network. Spring supports copy-on-write through a copy-on-write map structure. This structure contains pointers to regions of the source and destination cache, and a bitmap indicating which pages have be copied from the source cache. The caches that the copy-on-write map structures point to can chain to arbitrary levels, much like Mach shadow object. Spring has a bulk data movement facility for IPC that moves data either by moving it or by copy-on-write copying it (using the copy-on-write map mechanism). If the data is being moved rather than copied and the data is resident in memory, then Spring will steal the pages from the source object and place them in the destination. This is important for external filesystem pagers since filesystems often have to move lots of data. Spring reuses the SunOS machine-dependent HAT layer to avoid duplicate work.

3.4.2 I/O and IPC Subsystems That Use VM Features

In this section we examine research on I/O and IPC subsystems that can take advantage of services offered by virtual memory systems to reduce data movement overheads. While this research shares some of the same goals as UVM, our approach is different. In UVM we are interested in creating a virtual memory system whose internal structure provides efficient and flexible support for data movement. In I/O and IPC research, the focus is on using features that the virtual memory is assumed to already provide. Thus, I/O and IPC research addresses some problems that UVM does not address such as buffering schemes and API design.

The Genie I/O Framework

Recent research by Brustoloni and Steenkiste analyzed the effects of data passing semantics on kernel I/O and IPC [10, 11]. Since the traditional Unix API uses "copy" semantics to move data this research focuses on providing optimizations that reduce data transfer overhead while maintaining the illusion of the copy semantic. These optimizations were implemented under Genie, a prototype I/O system that runs under NetBSD 1.1. Since Genie was intended as an experimental testbed to compare various IPC schemes it contains features that may not be suitable for production use. Several techniques that preserve the copy semantic are presented including temporary copy-on-write (TCOW), and input alignment. We briefly describe each of these below.

Genie's TCOW mechanism is similar to UVM's page loanout mechanism. TCOW allows a process to loan its pages out to the Genie I/O system for processing without fear of interference from the pagedaemon or page faults from other processes. TCOW differs from page loanout in three ways. First, TCOW only allows pages to be loaned out to wired kernel pages, while UVM's page loanout allows both object and anonymous pages to be loaned out to pageable anonymous memory. Second, TCOW was implemented on top of the old BSD VM object chaining model, while page loanout is implemented on top of UVM's new anonymous memory layer. Third, in UVM we have demonstrated how page loanout can easily be integrated with BSD's mbuf-based networking subsystem while TCOW was only demonstrated with the Genie I/O framework, completely bypassing the traditional BSD IPC system.

Input alignment is a technique that can be used to preserve an API with copy semantics on data input while using page remapping to actually move the data. In order to do input alignment one of two conditions must hold. Either the input request must be issued *before* the data arrives so Genie can analyze the alignment requirements of the input buffer and configure its buffers appropriately, or the client must query Genie about the alignment of buffers that are currently waiting to be transfered so that it can configure its buffers to match. When remapping pages that are not completely filled with data, Genie uses "reverse copyout" to fill the remaining areas with the appropriate data from the receiving process' address space. Genie also has several techniques for avoiding data fragmentation when the input data contains packet headers from network interfaces.

The Fbuf IPC Facility

The fast buffers (fbufs) kernel subsystem is an operating system facility for IPC buffer management developed at University of Arizona [23, 24, 65]. Fbufs provide fast data transfer across protection domain boundaries. The fbuf system is based on the assumption that IPC buffers are immutable. An immutable buffer is one that is not modified after it has been initialized. An example of an immutable buffer is a buffer that has been queued for transmission on a network interface. Once the data has been loaded into the buffer it will not be changed until the buffer is freed and reused.

Fbufs were implemented as an add-on to the Mach microkernel. The kernel and each process on the system share a fixed sized area of virtual memory set aside as the "fbuf region." All fbuf data pages must be mapped into this shared region. Although the fbuf region of virtual memory is shared among all processes the individual fbufs mapped into the region are protected so that only authorized processes have access to the fbufs they are using. The fbuf region can be thought of as a large `vm_object` that is completely mapped into all address spaces. The fbuf code acts as a pager for this object. This structure allows the fbuf system to operate under Mach without the need to make changes to the core of the Mach virtual memory system.

To send a message using an fbuf, a process first allocates an fbuf using the fbuf system's API. The fbuf allocator will allocate an unused chunk of virtual memory from the fbuf region and mark it for use by the process requesting the allocation. At this point the fbuf system will allow the process read and write access to the allocated part of the fbuf region. The application can then place its data in the fbuf and request that the data be sent. Since fbufs are immutable, the application should not attempt to access the fbuf after it is sent. The fbuf system gives the recipient of the fbuf read access to it so that it may process the data. Data stored in the fbuf object is pageable unless it has been wired for I/O. If a process attempts to read or write an area of the fbuf region that it is not authorized to access then it receives a memory access violation.

The fbuf facility has two useful optimizations: fbuf caching and volatile fbufs. In fbuf caching, the sending process specifies the path the fbuf will take through the system at fbuf allocation time. This allows the fbuf system to reuse the same fbuf for multiple IPC operations. Once an fbuf has been sent, it is returned to the originating process for reuse. This saves the fbuf system from having to allocate and zero-fill new fbufs for each transaction. Volatile fbufs are fbufs that are not write protected in the sending process. This saves memory management operations at the cost allowing a sending process to write an immutable buffer after it has been sent.

While noteworthy, the fbuf facility has several limitations. First, the only way to get data from the filesystem into an fbuf is to copy it. Second, if an application has data that it wants to send using an fbuf but that data is not in the fbuf region, then it must first be copied there. Third, it is not possible to use copy-on-write with an fbuf. It should be possible to combine UVM features such as page loanout and page transfer with the fbuf concept to lift these limitations.

Container Shipping

The Container Shipping I/O system allows a process to transfer data without having direct access to it [53]. Developed at University of California (San Diego), container shipping is based on the container-shipping mechanism used by the cargo-transportation industry. In container-shipping goods are placed on protective pallots which are then loaded into containers. Because the containers are uniform in size they can easily be moved, stacked, and transported.

The container shipping I/O system operates in a similar way. Using a new API, a process can allocate a container. It then can "fill" the container with pointers to data buffers. Once a container has been filled with the necessary data a process can "ship" it to another process or the kernel. Rather than receiving the data directly, the receiving party receives a handle to the container. The receiver then has four options. First, it could "unload" the container, thus causing the data in the container to be mapped into the receiver's address space. Second, it could "free" the container (and the data within). Third, it could "ship" the container off to another process. Fourth, it could "empty" the container, thus freeing data from it. There is also an operation that allows a process to query the status of a container. By using these operations processes can pass data between themselves without mapping it into their address space. A container shipping I/O system could easily take advantage of UVM services such as page loanout and page transfer when loading and unloading containers.

The Peer-to-Peer I/O System

Another I/O system developed at University of California San Diego is the Peer-to-Peer I/O system. The Peer-to-Peer I/O system was designed to efficiently support I/O endpoints without direct user process intervention [26, 27]. This is achieved through the use of the *splice* mechanism. The splice mechanism allows a data source to directly transfer data to a data sink without going through a process' address space.

To use a splice a process must first open the data source for reading and the data sink for writing. The endpoints can either be files or devices. The process then calls the `splice` system call with the source and sink file descriptors as arguments. This causes the kernel to create a splice that associates the source file descriptor with the sink file descriptor. The `splice` system call returns a reference to the splice to the calling process in the form of a a new third file descriptor. Once the splice is established, the process can transfer a chunk of data from the source to sink by writing a control message to the splice's file descriptor. The control message contains the size of the chunk. Such data is transfered through kernel buffers rather than through the process' address space. The process can also delegate control of data transfer to the operating system. In that case, the operating system will continue to transfer chunks of data until the end-of-file for the data source is reached. The process can query the status of a splice by reading a status message from the splice file descriptor. The splice is terminated simply by closing the splice file descriptor.

The splice mechanism also allows for some data processing to occur within the kernel. This is done by allowing software data processing modules to be put between the data source and sink. Such modules can either be compiled into the kernel or dynamically loaded (if the process has suitable privileges). This mechanism is similar to the STREAMS mechanism that is part of System V Unix [56, 58, 68].

Unified Buffering I/O Systems

Rice University's I/O Lite unified buffering and caching system unifies the buffering and caching of all I/O subsystems in the kernel [52]. I/O Lite extends Fbuf's concept of immutable buffers to the entire operating system. These immutable buffers are read-only mapped into the address space of processes that are accessing them. Processes access the immutable buffers through lists of pointers to the immutable buffers called mutable buffer aggregates. A new API is used to pass a mutable list of buffer aggregates between endpoints. When a process wants to modify data stored in an immutable buffer it has three options. First, if the entire buffer is being changed it can be completely replaced with a new immutable buffer. Second, if only part of an immutable buffer is being changed then a new buffer aggregate can be created that directs references to the changed area to a new immutable buffer. Finally, for applications that require write access to the immutable buffer I/O Lite has a special "mmap" mode that provides shared access to the (no longer) immutable buffer.

University of Maryland's Roadrunner I/O system also unifies buffering and caching of all I/O systems in the kernel [41, 42]. All I/O systems in Roadrunner use a "common buffer" to pass data. Roadrunner supports a more generalized version of the splice called a "stream" that uses a kernel thread to provide autonomous I/O between any two endpoints. Roadrunner's caching layer is global. Unlike Unix, Roadrunner uses the same layer and interfaces for all devices.

Washington University's UCM I/O system is an I/O system that was designed to unify the buffering of all I/O subsystems of the kernel [15, 16]. UCM I/O includes a new API that supports both network and file I/O in a general way. It was designed to use page remapping to move data between users and the kernel. The design also includes a splice like facility for moving multimedia data and a mechanism to combine separate system calls into a single system call.

Other Work

Research in the area of remote procedure call has used virtual memory services to help move data. The DEC Firefly RPC system used a permanently mapped unprotected area of memory shared across all processes to pass RPC data [61] and provide low latency. The LRPC and URPC systems copy data into memory statically shared between the client and the server [5, 6]. The DASH IPC mechanism used a reserved area of virtual space in processes to exchange IPC data through page remapping [67]. The Peregrine IPC system used copy-on-write for output, copy for input, and page remapping to move data between IPC endpoints [73]. Finally, the Mach IPC system uses copy maps, as described previously.

Some research has produced special purpose I/O systems to work around a specific problem. For example the MMBUF system is a system that avoids data copying between the filesystem and networking layer by using a special buffer called an MMBUF that acts as both a network mbuf or a filesystem buffer [12]. This system is used to transmit video data from a disk file over an ATM network using real-time upcalls to maintain quality of service [29]. This system could be adapted to use a mechanism like UVM's page loanout to move data without copying it (or mapping it) through a user address space.

New designs in host-network interfaces also look to take advantage of services that a VM system such as UVM can offer. For example, the APIC ATM host-network adaptor can separate protocol headers from data and store inbound data into pages that can be remapped into a user's address space using a mechanism such as UVM's page transfer [20, 21, 22].

Researchers are often reluctant to change traditional APIs for fear of breaking user programs. However, new applications are often taking advantage of communications middleware such as ACE wrappers [60]. ACE wrappers abstract around the differences between operating systems to provide applications with a uniform interface to IPC services. Such techniques could be applied in the I/O and IPC area to hide differences between traditional APIs and newer experimental ones. Applications that use such wrappers would not have to be recompiled to take advantage of new APIs. Instead they could dynamically link with a new middleware layer that provides the appropriate abstractions and understands the new API.

Recent operating systems research has focused on extensible operating systems. For example, the SPIN kernel allows programs written in a type-safe language to be downloaded into the kernel and executed [7, 8]. The Exokernel attempts to provide processes with as much access to the hardware of the system as safely possible so that alternate operating system abstractions can easily be evaluated without the Exokernel getting in the way [34]. The Scout operating system is intended for small information appliances [45, 47]. Scout allows you to build optimized "paths" through the operating system for specific applications. Such work on extensible systems can provide us with insight into servers that a VM system should provide, and in some cases allow applications more direct access to the services that a VM system provides the kernel.

Research using the L3 and L4 microkernels has been used to determine the maximal achievable level of IPC performance the hardware will allow [37, 38]. This was done by hand coding the IPC mechanism in assembly language, ensuring that code and data was positioned in memory in such a way that they all could be co-resident in the cache, and by minimizing the number of TLB flushes used for an IPC operation. The result of this exercise shows that software adds a substantial amount to IPC overhead and provides a "best case" target for software developers to shoot for when developing new IPC mechanisms.

3.5 Summary

In this chapter we have presented an overview of UVM. The overview consisted of the goals of the UVM project, a description of new and improved features that UVM provides, and a description of UVM's main data structures and functions. Table 3.1 and Table 3.2 summarize the changes introduced to BSD by UVM and list the design issues we faced during the implementation of UVM. We also discussed related research in the area of virtual memory systems, I/O, and IPC. In the remaining chapters of this dissertation we will examine the design and implementation of UVM in greater detail.

Chapter 4

Anonymous Memory Handling

This chapter describes the data structures, functions, and operation of UVM's anonymous memory system. Anonymous memory is memory that is freed as soon as it is no longer referenced. This memory is referred to as anonymous because it is not associated with a file and thus does not have a file name. Anonymous memory is paged out to the "swap" area when memory is scarce.

4.1 Anonymous Memory Overview

Anonymous memory is used for a number of purposes in a Unix-like operating system including:

- for zero-fill mappings (e.g. bss and stack).
- to store changed pages in a copy-on-write mapping.
- for shared regions of memory that can be accessed with the System V shared memory system calls.
- for pageable areas of kernel memory.

In the BSD VM system all anonymous memory is managed through objects whose pager is the "swap" pager. Such objects are allocated either for zero-fill memory, or to act as shadow or copy objects for a region of copy-on-write memory.

UVM uses a two-level amap-object memory mapping scheme, as previously described in Chapter 3. In UVM's two-level scheme, anonymous memory can be found at either level. At the lower backing object level, UVM supports anonymous memory `uvm_object` structures ("aobj" objects). These objects use the anonymous memory object pager ("aobj" pager) to access backing store (i.e. the system "swap" area). UVM aobj-objects are used to support the System V shared memory interface and to provide pageable kernel memory.

UVM's upper level of memory consists entirely of anonymous memory. Memory at this level is referenced through data structures called anonymous memory maps ("amaps"). Thus, the upper layer can be referred to as the "amap layer." The amap layer sits on top of the backing layer. When resolving a page fault, UVM must first check to see if the data being faulted is in the amap layer or not. If it is not, then the backing object is checked. Note that unlike the BSD VM, there is only one backing object — there are no chains of objects to traverse. The UVM amap layer serves the same function as a shadow object chain in BSD VM; the amap layer is used for zero-fill mappings and for copy-on-write.

In a typical kernel running UVM, most anonymous memory resides in the amap layer rather than in "aobj" objects. The amap layer can also be used for page loanout and transfer (see Chapter 7). Thus amap-based anonymous memory plays a central role in UVM. The remainder of this chapter will focus on issues relating to amaps. More information on aobj-based anonymous memory can be found in Section 8.3 on the aobj pager.

4.2 Amap Interface

The first time data is written to memory mapped by an anonymous mapping, an amap structure is created. The amap contains a "slot" for each page in the mapping. Initially all the slots are empty. However, when data is written to a page a pointer to an "anon" structure is placed in the corresponding amap slot.

Amaps are referenced from map entry data structures. If a map entry structure that points to an amap is split (e.g. during an unmap operation), the two resulting map entry structures must each point to part of the amap. The original amap could be broken up into two separate amaps, however this would involve expensive data copying of the amap and also cause problems in some cases. For example, consider an amap that is shared between two distinct processes. If one of the processes needs to split its reference to the amap, making a copy of part of the amap would break the sharing of the amap with the other process. One could keep track of all processes sharing an amap, however that would mean chaining a linked list off the amap. To avoid the overhead associated with breaking up an amap, UVM uses an "aref" (a map reference) data structure to point to an amap and the slot offset in which the map entry's mapping starts. The aref structure is shown in Figure 4.1.

Figure 4.1: The aref structure

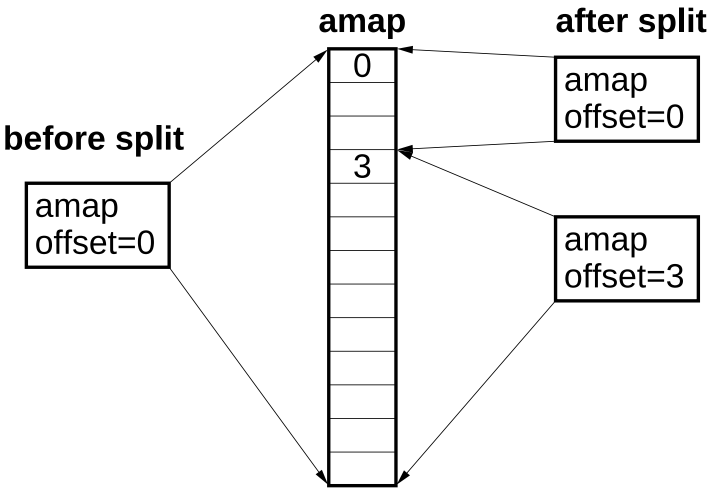


Figure 4.2: Splitting an aref

When a map entry is split, one of the entries can keep the old slot offset, but one of them will need a new slot offset that has been adjusted by the size of the other entry, as shown in Figure 4.2.

The format of an anon structure is shown in Figure 4.3. An anon contains a lock, a reference counter indicating how many amaps have references to the anon, a pointer, and an address in the swap area (the swap slot number). The anon's pointer is a C union that can be used in one of two ways. Unused anon structures use their pointer to form the free list. In-use anon structures use their pointer to point to the `vm_page` structure associated with the anon's data. The swap slot number is non-zero only if the anon's data has been paged out. In that case, the location of the data on backing store is stored in the swap slot number. Note that amaps track anons rather than pages because they need to keep track of the extra information stored in the anon structure (which is not stored in the page structure). Anons that have been paged out may not even have a page of physical memory associated with them. When UVM needs to access data contained in an anon, it first looks for a page structure. If there is none, then it allocates one and arranges for the data to be brought in from backing store.

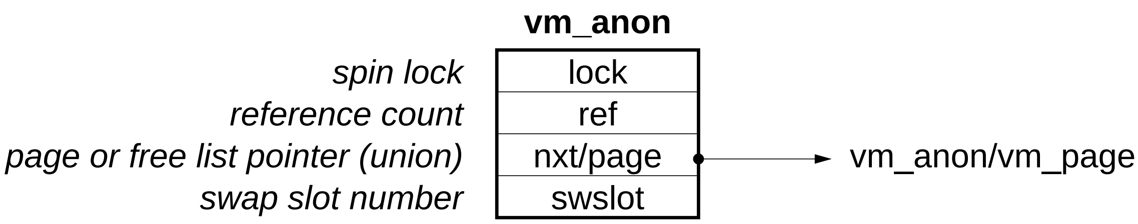


Figure 4.3: The anon structure

Table 4.1: amap API

Function	Description
`amap_alloc`	allocate a new amap of the specified size — the new amap does not contain any anons
`amap_free`	free an amap that is no longer in use
`amap_ref`	add a reference to an amap
`amap_unref`	drop a reference from an amap
`amap_splitref`	split a single reference into two separate references
`amap_extend`	increase the size of an amap
`amap_add`	add an anon to an amap
`amap_unadd`	remove an anon from an amap
`amap_lookup`	look in a slot in an amap for an anon
`amap_lookups`	lookup a range of slots
`amap_copy`	make a copy-on-write copy of an amap
`amap_cow_now`	resolve all copy-on-write faults in an amap now (used for wired memory)
`amap_share_protect`	protect pages in an amap that resides in a share map

UVM defines a set of functions that make up the operations that can be performed on an amap. The amap API is shown in Table 4.1.

4.3 Amap Implementation Options

In UVM, the internal structure of an amap is considered private to the amap implementation. All amap accesses are done through the amap API. This allows multiple implementations of amaps to exist in the source tree at the same time, and a specific implementation can be chosen at kernel compile time. This software structure is useful because there is a time vs. memory space trade-off to be considered when implementing amaps. Machines that typically do not have very much physical memory might be better off using an amap

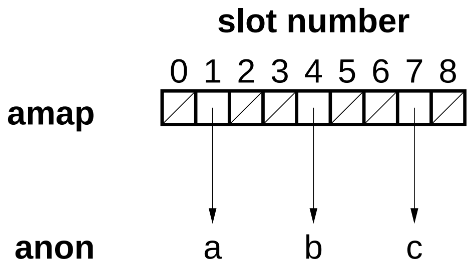


Figure 4.4: A simple array-based amap implementation

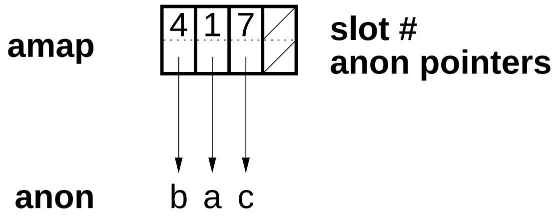


Figure 4.5: A simple list-based amap implementation

implementation that is conservative on its RAM usage, where as machines that have lots of RAM may wish to devote part of it to speeding up amap operations.

There are many ways that an amap can be implemented. For example, one possible implementation of an amap is an array of pointers to anon structures, as shown in Figure 4.4. The pointer array is indexed by the slot number. This implementation makes amap lookup operations fast, however operations that require traversing all active slots in the amap would have to examine every slot in the array to determine if it was in use or not. This is how amaps are implemented in SunOS4 VM.

Another possible amap implementation is shown in Figure 4.5. In this implementation, a dynamically sized array contains a list of anons and their slot number. If the list array fills, a new larger one can be allocated to replace it. This implementation can save physical memory, especially when an amap is mapping a sparsely populated area of virtual memory. This implementation slows down amap lookups (because the list has to be searched for the correct anon for each lookup), however it handles a traversal of the entire amap efficiently.

UVM includes a novel amap implementation that combines the ideas behind these two simple implementations.

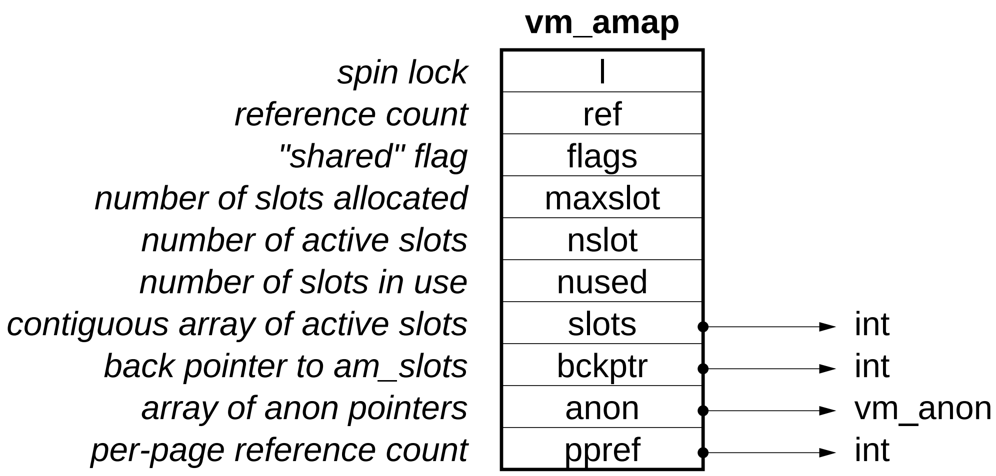


Figure 4.6: UVM reference amap structure

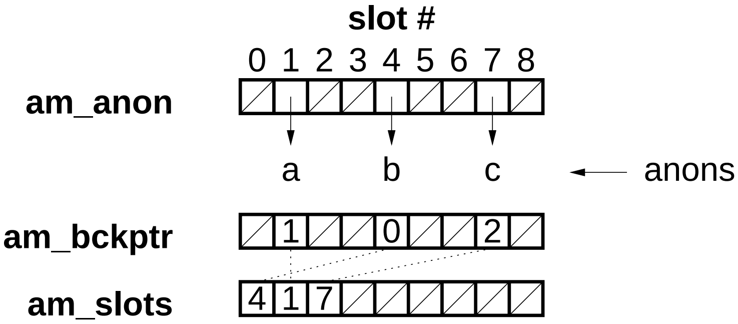


Figure 4.7: UVM reference amap structure's three main arrays

4.3.1 UVM Reference Amap Implementation

UVM comes with a reference amap implementation that was designed to minimize the overhead of all amap operations at the cost of some extra memory. The structure of the reference UVM amap structure is shown in Figure 4.6. The amap structure has a lock, a reference counter that indicates how many maps are referencing it, a flag indicating whether the data in the amap is being shared or not, the number of slots allocated for the amap, the number of those slots currently active, and the number of the active slots currently being used. The amap structure also contains four arrays that describe the anons that the amap maps. These arrays are: `am_slots`, `am_bckptr`, `am_anon`, and `am_ppref`. The `am_ppref` array is optional and will be described in Section 4.3.2. Figure 4.7 shows how the other three arrays are used.

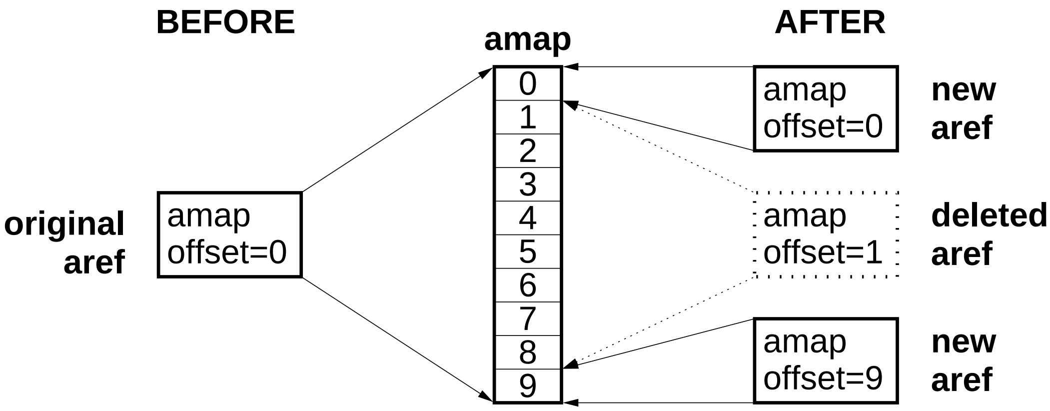


Figure 4.8: A partial unmap of an amap

- The am_anon array consists of a set of pointers to an anon structure. This array is indexed by slot number. The am_anon array allows quick amap lookup operations.
- The am_slots array contains a contiguous list of the slots in the am_anon array that are currently in use. This list is not in any special order. The am_slots array can be used to quickly traverse all anons that are currently pointed to by the amap.
- The am_bckptr array is used to keep the am_anon array and am_slots array in sync. The am_bckptr array maps an active slot number to the index in the am_slots array in which its contiguous entry lies. For example, in Figure 4.7 amap_anon[7] points to anon "c" and its entry in the contiguous am_slots array is at am_slots[2].

4.3.2 Partial Unmap of an Amap

When the final reference to memory mapped by an amap is removed due to an unmap operation, the physical memory and backing store associated with it can be freed because that memory can never be referenced again. However, a problem can occur when only part of an amap is unmapped. Consider a process with a map entry that maps ten pages and whose aref structure points to a ten-slot amap that is not referenced by any other map. Now suppose the process unmaps the middle eight pages of the map entry. In order to do this, UVM will break the map entry up into three separate map entries and dispose of the middle one. This process is shown in Figure 4.8. Before the unmap, the amap's reference count is one. After the unmap the amap's reference count is two. Note that pages in slots one to nine are no longer accessible after the unmap operation, and in fact they can never be referenced again. Whether the unmapped memory is freed immediately or whether is it freed when the amap's reference count drops to zero is a decision left up to the amap implementation.

The tradeoffs to this decision are as follows: if the amap module does not free partially unmapped sections of an amap until the entire amap's reference count goes to zero, then its code will be less complex since it has less to manage. Also, traditional processes on a Unix-like operating system seldom if ever perform a partial unmap of memory. However, newer memory allocators often make use of the `mmap` and `munmap` system calls to allocate and free memory, and these allocations take place from amaps. Also newer VM-related IPC systems can also take advantage of the amap layer to move data, thus using it in ways that traditional processes do not. If partial amap memory unmaps become frequent and the amap layer does not immediately free resources — as in the BSD VM system — it is possible for a "swap memory leak" situation to develop.

UVM's reference amap implementation handles partial unmaps without causing swap memory leaks. This requires a few extra internal functions in the reference amap module, and an `am_ppref` array in the amap structure. The `am_ppref` is a "per-page" array of reference counters[1]. This array is only allocated and active when a reference to an anonymous map has been split the way it was in Figure 4.8. When the array is first created, each page gets a reference count equal to the reference count of the amap itself. From that point onward, as parts of the amap are referenced and freed the per-page reference counters are adjusted to keep track of each page's current number of references. If a page's per-page reference count drops to zero, then any allocated memory or swap is immediately freed.

Keeping a reference counter for each page in a large amap would be expensive. For example, if the reference to an N-page region of an amap is changed due to a mapping operation, then all N per-page reference counters will have to be updated. Consider the example shown in Figure 4.8. If the amap starts off with a reference count of one, then the per-page reference counters would have to be adjusted for the unmap as shown in Figure 4.9. Each reference counter in the unmapped range will have to be changed from one to zero.

We can reduce the overhead of per-page reference counters by using one reference counter for each contiguous block of memory. UVM's reference amap implementation attempts to reduce this overhead of per-page reference counters. In the example shown in Figure 4.9, there is a large contiguous block of eight pages whose reference count is zero. Rather than having individual counters for this region that each need updating in the event of a reference count change, it would be more efficient to have one counter for the entire

[1] In this case "page" means anonymous memory page, not physical memory page.

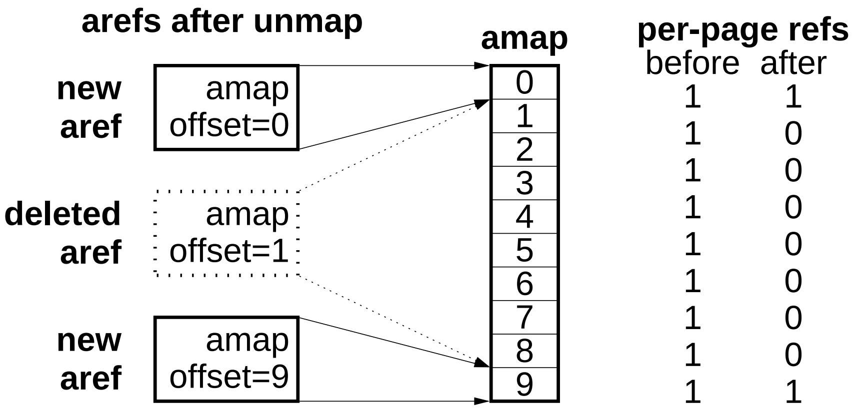


Figure 4.9: Per-page reference counters before and after an unmapping. Before the unmap, the amap's overall reference count is one, and after the unmap its overall reference count is two (one for each aref).

contiguous chunk, and then have another array to indicate the length of the chunk. An example of this encoding is shown in Figure 4.10.

Adding a second array to keep track of the length of a mapping would allow us to change the reference count on a block of pages quickly. However, it also doubles the memory needed by the per-page reference count system. UVM takes advantage of the large number of "don't care" values in the array to compact the reference and length arrays into one single array. This is done by dividing the blocks of references into two groups: those of one page in length and those of more than one page in length. Single page blocks can be distinguished from multi-page blocks by the sign bit on their reference count. If the value is positive, then it is a single page block. If the value is negative, then there is a multi-page block, and the length of the block is the following value. There is one more problem: how to handle blocks whose reference count is zero. Since zero is neither negative or positive it is impossible to tell the length of the block. This issue is addressed by offsetting all reference counts by one. Thus, the reference count and length of a block can be found with the code shown in Figure 4.11. An example of this encoding is shown in Figure 4.12.

An attractive feature of this original scheme is that its processing cost starts off small and only rises as the amap is broken into smaller chunks. Processes that do not do any partial unmaps will not have any per-page reference count arrays active. Processes that only do a few partial unmaps will have only a few chunks in their per-page reference count array.

per-page refs		per-page refs	
before	after	before (ref/len)	after (ref/len)
1	1	1/10	1/1
1	0	x/x	0/8
1	0	x/x	x/x
1	0	x/x	x/x
1	0	x/x	x/x
1	0	x/x	x/x
1	0	x/x	x/x
1	0	x/x	x/x
1	0	x/x	x/x
1	1	x/x	1/1

Figure 4.10: Per-page reference counters with length. The symbol “x” means that this value does not matter.

```
if (am_ppref[x] > 0) {
    reference_count = am_ppref[x] - 1;
    length = 1;
} else {
    reference_count = -am_ppref[x] - 1;
    length = am_ppref[x + 1];
}
```

Figure 4.11: Code to find the per-page reference count and length

4.4 Accessing Anonymous Backing Store

In the BSD VM system a `vm_pager` data structure is allocated and assigned to every `vm_object` that accesses backing store. Anonymous memory in the BSD VM system lives in `vm_object` structures whose pager is associated with the system swap area. Often these `vm_object` structures are shadow or copy objects. The location of backing store to which anonymous memory can be paged out is accessed through the object’s `vm_pager` structure. The `vm_pager` structure points off to another structure for the swap pager that translates object offsets into a location on swap. Because each page in the VM system can only belong to one `vm_object` it is easy to access backing store for that page by just following the `vm_object`’s pager pointer.

With UVM’s amap-based anonymous memory things change. An anonymous page of memory can belong either to an aobj `uvm_object` or to an anon. Accessing backing

per-page refs		encoded per-page refs	
before (ref/len)	after (ref/len)	before	after
1/10	1/1	-2	2
x/x	0/8	10	-1
x/x	x/x	x	8
x/x	x/x	x	x
x/x	x/x	x	x
x/x	x/x	x	x
x/x	x/x	x	x
x/x	x/x	x	x
x/x	x/x	x	x
x/x	1/1	x	2

Figure 4.12: Encoded per-page reference count array

store for an aobj-based anonymous page of memory is done through the `uvm_object`'s pager operations pointer. This will not work for an anon because it is not a `uvm_object` and thus does not have a pager operations pointer. Since each anon is backed by the swap area, there is no need for such a pointer. However, there still needs to be a way to map a paged-out anon to its data on backing store. In the SunOS4 VM, this is done by statically assigning a page-sized block of swap to each anon. There are several problems with this scheme. First, it forces the system's swap size to be greater than or equal to the the size of physical memory so that each anon allocated at boot-time has a location to which it can be swapped out (this problem was later addressed in Solaris [13]). Second, the static allocation of anon to swap area makes it hard to cluster anonymous memory pageouts and thus anonymous memory pageout takes longer.

Rather than statically assigning a swap block to each anon, in UVM the assignment of a swap block to an anon is done by the pagedaemon at pageout time. The swapping layer of the VM system presents swap to the rest of the kernel as a big "file." This file, known as the "drum," can be accessed by the root user via the device file `/dev/drum`. Internal to the swap layer of the VM, the drum might be interleaved across several different devices or network files. To most of UVM the drum is simply a large array of page-sized chunks of disk. Each page-sized chunk of swap is assigned a "slot" number. When an anon has backing store assigned to it, it saves the slot number in its `an_swslot` field. Swap slot number zero is reserved: it means that a backing store has not been assigned to an anon.

The data in an anon can be in one of three possible states:

resident with no backing store slot assigned: In this case the anon's an_page pointer points to the page in which the anon's data resides. The anon's an_swslot field is zero indicating that no swap slot has yet been assigned to this anon. All anons start in this state.

non-resident: An anon can be in this state if a swap slot was allocated for it, its page was paged out to its area of the drum, and then its page was freed. In this state the page pointer will be null and the swap slot number will be non-zero. An anon can enter this state from either of the other two states.

resident with backing store slot assigned: An anon enters this state if it was previously non-resident and its data was paged in from backing store. In this case it has both a page pointer and a swap slot assigned. As long as the data in the resident page remains unmodified the data on backing store is valid.

Note that in UVM it is not possible to have an anon that has neither a page nor a swap slot assigned to it.

4.5 The Effects of Amaps on Pages and Objects

Converting BSD to use amap-based anonymous memory is a major change that effects a number of other parts of the VM system. In this section we describe the effect of amaps on pages and objects. In the following section we describe the effect of amaps on forking and copy-on-write.

In the BSD VM system, each allocated page of memory can belong to a single vm_object structure at a specified offset. If the kernel wants to perform an operation on the object in which a page belongs then it uses the object pointer in the page structure to lock the object, perform the operation, and then unlock the object when done.

In UVM things are not that simple. In addition to belonging to a uvm_object, pages can also belong to an anon structure. The anon structure is likely to be referenced by one or more amaps. This means that if the kernel wants to perform an operation on the memory object that a page belongs to it must first decide whether that memory object is a uvm_object or an anon. Making this decision is quite easy to do: there is a flag bit in the page structure called PG_ANON that is true if the page is part of an anon structure and false otherwise. The issue of determining the owner of a page is of importance to the page daemon since it starts at the page queues, finds a vm_page structure that it wants to work on and then tries to lock the data structure that that page belongs to. The daemon needs to know if it is locking an anon or a uvm_object.

In UVM, it would be possible to eliminate the distinction between pages belonging to uvm_objects and pages belonging to anons by simply converting each anon into a single page uvm_object structure. However, this has a major drawback: the uvm_object structure is much larger than the anon structure and has fields in it that are not needed by anon pages. Since the system typically has a much larger number of anon's allocated than uvm_objects, it would be a great waste of kernel memory to convert all anons to uvm_objects.

4.6 The Effect of Amaps on Forking and Copy-on-write

This section describes the effects of amaps on BSD's forking and copy-on-write process. First, a description of copy-on-write is presented. Then the effect of Mach-style memory inheritance on amaps is discussed.

4.6.1 Copy-on-write

As previously described in Chapter 3, a process' virtual address space is described by a map structure. Each mapped region in the address space is described by map entry structures that are kept in a sorted linked list off the map. Each map entry maps two levels: a top-level anonymous amap layer, and a bottom-level backing object layer. Either or both of these layers can be null. For copy-on-write mappings, all copy-on-write anonymous data is stored in the amap layer. Each map entry describes the attributes of the mapping it maps. With respect to copy-on-write, there are two important boolean attributes in the map entry:

copy-on-write: If copy-on-write is true, then it indicates that the mapped region is a copy-on-write mapping and that all changes made to data in the mapping should be done in anonymous memory. If copy-on-write is false, then that indicates that the mapping is a shared mapping and that all changes should be made directly to the object that is mapped.

needs-copy: If needs-copy is true, then it indicates that this region of memory needs its own private amap, but it has not been allocated yet. The region either has no amap at all, or it has a reference to an amap that it needs to copy on the first write to the region. The needs-copy flag allows UVM to defer the creation of an amap until the first memory write occurs. The needs-copy flag is used for copy-on-write operations.

As an example, consider the data area of a process' address space. As explained earlier in Section 2.2, the data area of a process is mapped read-write copy-on-write. When this mapping is established both its copy-on-write and needs-copy flags are set to true. The mapping's amap reference (aref) is set to null to indicate that no amap has been created yet. The copy-on-write flag will remain true for the lifetime of the mapping (it is not possible to change the copy-on-write status of an existing mapping). The needs-copy flag will remain true until the first write to the mapped area occurs. At that point the `amap_copy` function will be called to copy the amap. The `amap_copy` function allocates a new amap structure, attaches it to the map entry, and then clears the needs-copy flag. Note that zero-fill mappings are handled in a similar way: they are established with both copy-on-write and needs-copy true. The difference between a zero-fill mapping and the copy-on-write mapping of a program's data area is that the zero-fill area has a null backing object, where as the data area has a file as a backing object.

When data is written to a copy-on-write area, an anon with a reference count of one is allocated in the amap to contain the data. The details of how anons are inserted into amaps during a copy-on-write operation are presented in Chapter 5.

The reference count of an anon is important in determining its copy-on-write status. If the reference count is one, then it means that only one amap is using the anon and that it is safe to write directly to the anon's data. However, if the reference count is greater than one then on a write a copy-on-write operation must be performed. This is done as follows:

1. A new anon is allocated, the new anon's reference count is one.

2. A new page is allocated and the data is copied from the old anon's page to the new anon's page

3. The old anon's reference counter is dropped by one and the new anon is installed in the current amap.

4. The copy-on-write operation is complete and the write can proceed as normal.

Thus, data in an anon can only be changed if its reference count is one. Note that the procedure described above is incomplete because it does not take into account the possibility that the system is out of memory. If the system is out of virtual memory, then the allocation of the new anon will fail. This can happen if all of the system's swap space is allocated.

To recover from this, the process can either wait in hopes that some other process will free some memory, or it can be killed. Currently in UVM the process is killed, however, further work to find a better solution might be useful. On the other hand, if the system is out of physical memory then the allocation of the new page will fail. In this case, the process must drop all its locks and wake up the pagedaemon. The pagedaemon will (hopefully) free some pages by doing some pageouts and then wake the process back up.

This is the basics of how copy-on-write works in UVM. Note that amaps combined with the anon's reference counter perform the same role in copy-on-write that object chaining does in BSD VM. However, UVM's amap-based scheme is simpler than BSD VM's object chaining scheme because it requires only a two-level lookup rather than a traversal of an object chain of arbitrary length, and it does not require complex object collapse schemes or leak swap memory. In the next section the details of copy-on-write with respect to forking are described.

4.6.2 Forking

When a process forks a child process it must create a new address space for the child process based on the parent process' address space. In traditional Unix, regions mapped copy-on-write are copy-on-write copied to the child process while regions mapped shared are shared with the child process. In the Mach operating system the forking operation is more complex because the parent process has more control over the access given to the child process. Both the BSD VM and UVM include this feature from Mach. Each map entry in a map has an inheritance value. The inheritance value is one of the following[2]:

none: An inheritance code of "none" indicates that the child process should not get access to the area of memory mapped by the map entry. The fork routine processes this by skipping over this map entry and moving on to the next one in the list.

share: An inheritance code of "share" indicates that the parent and child processes should share the area of memory mapped by the map entry. This means that both the references to any amap and backing object mapped should be shared between the two processes. Note that the copy-on-write flag of a mapping in the child process is always the same as in the parent (the child can't suddenly get direct access to a backing memory object).

[2]There is also a "donate copy" inheritance value that does not appear in Mach but is defined (but not implemented) in BSD VM.

copy: An inheritance code of "copy" indicates that the child process should get a copy-on-write copy of the parent's map entry. First, in order to achieve proper copy-on-write semantics, all resident pages in the parent's mapping are write protected. This ensures that a page fault will occur the next time the parent attempts to write to one of the pages. As part of the fault recovery procedure, the page fault handler will do the copy-on-write on the faulting page. Second, the child gets a reference to the parent's backing object, if any. Third, the child adds a reference to the parent's amap (if any) and sets the copy-on-write flag. Fourth, the needs-copy flag is set in both the parent and child. In some cases (described later) the needs-copy flag must be cleared immediately.

The default inheritance value for a region mapped copy-on-write is copy, while the default inheritance value for a region mapped shared is share. Thus, the default behavior emulates traditional Unix. However, it is possible for a process to change the inheritance attribute of a region of memory by using the `minherit` system call.

Note that in share inheritance the amap structure can be shared between mappings. When this happens a special "shared" flag is set in the amap structure to indicate that the amap is being shared. When the shared flag is set, the amap module is careful to propagate changes in the amap to all processes that are sharing the mapping. For example, when an anon is removed from an amap any resident page associated with that anon must be unmapped from the process using the amap. If the amap is known not to be shared, then simply removing the mapping for that page in the current process' pmap is sufficient. However, if the amap is being shared (and thus the shared flag is set) then not only must the mapping be removed from the current process' pmap, but it must also be removed from any other process that is sharing the mapping. Removing a mapping from the current process' pmap can be accomplished by using the `pmap_remove` function, while removing all mappings of a page can be accomplished with the `pmap_page_protect` function.

In UVM, the function that copies the address space of a process during a fork operation is `uvmspace_fork`. The C pseudo-code for this function is shown in in Figure 4.13. Note that each map entry in the parent's address space is copied to the child process based on its inheritance value.

It is often easier to understand the relationship between amaps and forking, faulting, and inheritance changes by looking at a memory diagram that shows the state of the data structure through these critical events in a process' lifetime. Table 4.2 defines the symbols used in a memory diagram. Figure 4.14 shows the arrangement of the data structure in a memory diagram. Note that in a map entry the upper left corner contains the mapping type

```
uvmspace_fork() {

    allocate a new vmspace structure for the child;
    for (each map entry in the parent's map) {

        switch(entry->inherit) {
            case VM_INHERIT_NONE:
                /* do nothing */
                break;
            case VM_INHERIT_SHARE:
                /* add shared mapping to child */
                break;
            case VM_INHERIT_COPY:
                /* add copy-on-write mapping to child */
                break;
            default: panic("uvmspace_fork");
        }
    }
    return(new vmspace);
}
```

Figure 4.13: Pseudo code for fork

Table 4.2: The symbols used in a memory diagram

Symbol	**Meaning**
A_r	an anon structure with reference count r
C	copy
F_n	process n forks
IH_n	process n changes an inheritance value
M_r	an amap structure with a reference count r
NC	needs-copy flag is true
RF_n	process n triggers a read-fault
S	share
$SHARE$	"shared" amap flag is true
WF_n	process n triggers a write-fault
x	"don't care"

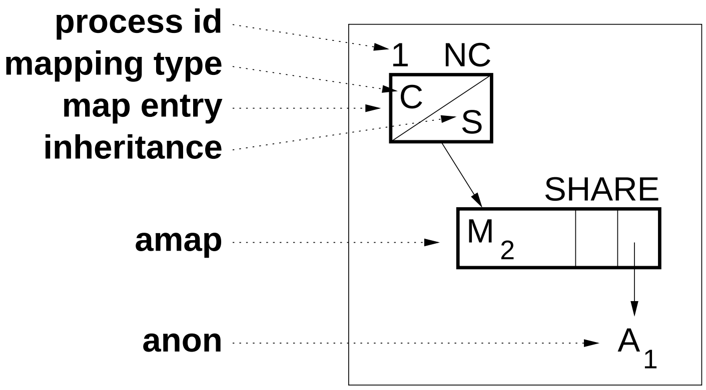


Figure 4.14: A sample memory diagram

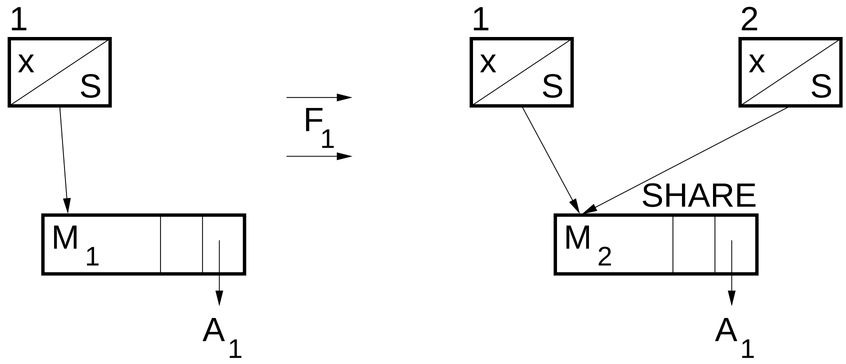


Figure 4.15: Share inheritance

and the lower right corner contains the inheritance value. Figure 4.15 shows what happens to a map entry during a fork when its inheritance is set to share. Note that the amap's share flag becomes true after the fork and the amap's reference count goes from one to two while the anon (that is shared) remains unchanged.

Figure 4.16 shows what happens to a map entry during a fork when its inheritance is set to copy. Note that in this case the amap gets an additional reference, but it is not a shared one. Both process one and process two have their needs-copy flag set. Also, note that the mappings in process one have been write protected to ensure that the copy-on-write operation will happen. The needs-copy flag will remain set on the map entries until the first write fault either in process one or process two. If process one had a write fault on anon "A" then the result would be as shown in Figure 4.17. When that write fault occurs, the needs-copy flag will first need to be cleared with `amap_copy` before the fault can be processed. This causes process one to get a new amap and both anons "A" and "B" get a reference

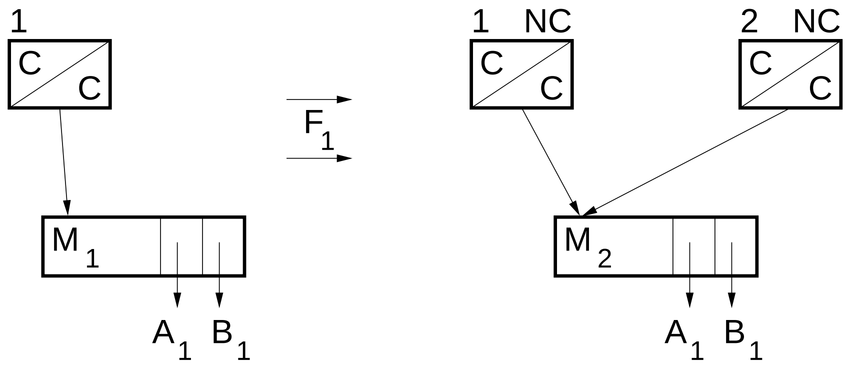


Figure 4.16: Copy inheritance

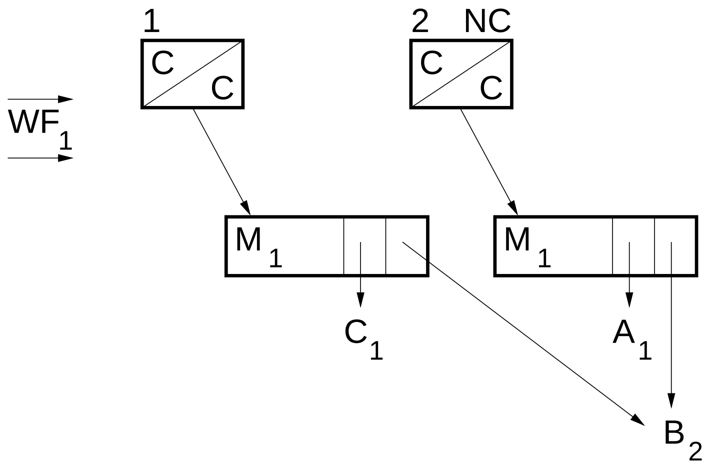


Figure 4.17: Clearing needs-copy after a copy-inheritance fork. Needs-copy was cleared because process one took a write fault on anon "A" causing a new anon ("C") to be allocated.

count of two. Once the new amap has been created, then a new anon ("C") for process one can be allocated and inserted into it. After the copy-on-write fault anon "B" is still shared between the two processes. Note that process two still has its needs-copy flag set: this is harmless. If process two has a write fault on the mapped area, `amap_copy` will be called on the mapping to clear needs-copy. The `amap_copy` function will notice that the amap pointed to by process two has a reference count of one. This indicates that process two is the only process referencing the amap, and thus the amap does not need to be copied. In this case `amap_copy` can clear the needs-copy flag without actually copying the amap.

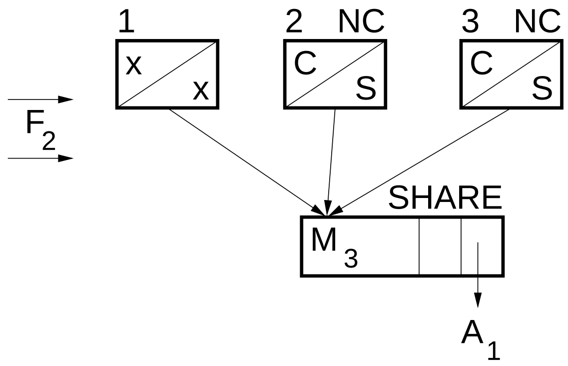


Figure 4.18: An erroneous fork. Process two is the parent process that has needs-copy set and shared inheritance. Process three is the child process. Process one is also using the amap.

4.7 Inheritance and Forking

The description in the previous section of the effect of shared and copy inheritance during a fork covers the common cases. However, there are a few special cases that must be correctly handled by the fork code in order to properly preserve the semantics defined by the memory mapping API. These special cases are described in this section.

4.7.1 Share Inheritance when Needs-copy is True

Consider a process that has a map entry in its map that has its inheritance value set to share and has an amap attached to it. When this process forks, the data in the amap should be shared between the parent and the child process, as shown earlier in Figure 4.15

Now consider what would happen if the parent process' needs-copy flag was true. The needs-copy flag of a map entry is true only if the entry is pointing to an amap that the entry needs a copy of, but has not yet made since a write fault has not occurred. This amap is typically also referenced by another process. If the pattern shown in Figure 4.15 is followed, the result is as shown in Figure 4.18. This configuration is wrong.

The problem with the configuration shown in Figure 4.18 is that both processes two and three are supposed to be sharing an amap, but they both have needs-copy set. Consider what would happen if process three had a write fault on the mapped area. The VM system would detect process three's needs-copy flag and clear it by creating a new amap for it. But by doing this process two and process three are no longer sharing the same amap and the semantics required by the shared inheritance are broken. The unfortunate result is shown

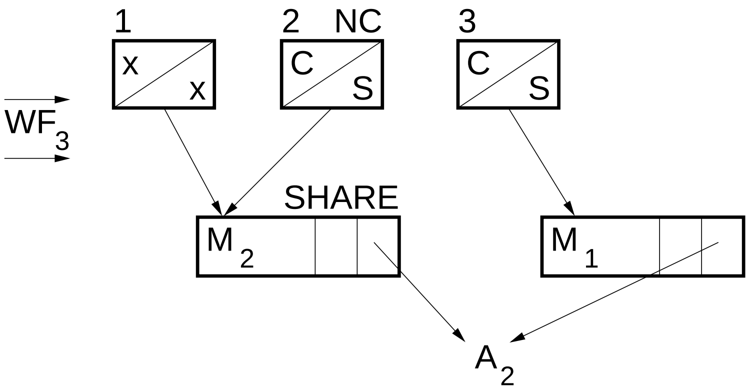


Figure 4.19: Erroneous fork after process three write-faults. Processes two and three should be sharing the same amap, but they are not.

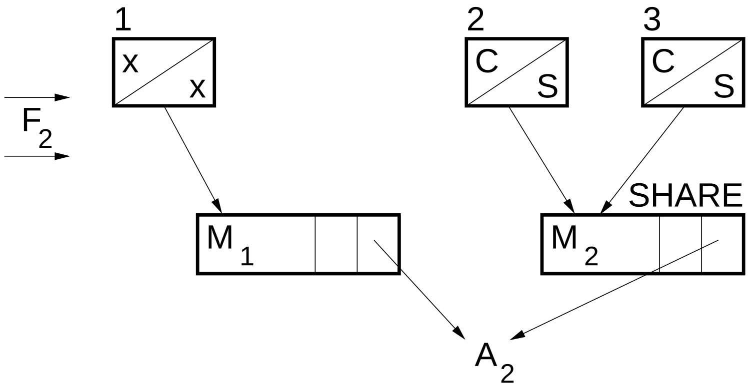


Figure 4.20: The correct handling of the erroneous fork

in Figure 4.19. Given the data structures involved there is no way for process three to know that it needs to update process two's amap pointer to point at the new amap it just created.

UVM avoids this problem by checking the needs-copy flag on the parent process during a shared-inheritance fork. If the flag is set, then UVM goes on and calls `amap_copy` immediately so that the amap to be shared is created before a problem like the one shown in Figure 4.19 can occur. The correct handling of the erroneous fork situation is shown in Figure 4.20.

4.7.2 Copy Inheritance when the Amap is Shared

When forking a map region with copy inheritance, the fork routine must also be careful with the handling of the needs-copy flag. Consider an amap that is being shared by two processes. The amap will have its share flag set and a reference count of two. Suppose

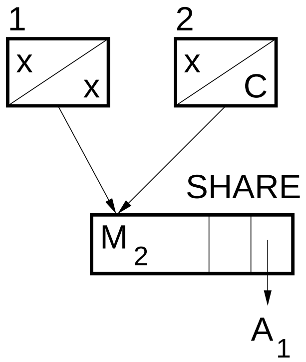


Figure 4.21: Two processes sharing an amap

that one of the processes mapping the map has its inheritance value set to copy. This configuration is shown in Figure 4.21. Note that processes one and two should be sharing the amap.

Now suppose process two forks. If the scheme shown in Figure 4.16 were followed the result would be as shown in Figure 4.22. That configuration is wrong. A write fault by any of the three processes will cause problems:

- If process one writes to a page in the amap process two will see the change because it is sharing the same amap with process one. However, process three will also see the change and it should not.
- If process two writes to a page in the amap the needs-copy flag on its mapping would cause a new amap to be allocated to store the changed page. This is wrong because process one and two are supposed to be sharing the same amap.
- If process three writes to the mapping then it will get a new amap as expected, and the copy-on-write will proceed normally. However process two will still have its needs-copy flag set after process three's write is finished, and that will lead to the problem described above.

The correct way to handle a copy-inherit fork when the amap is being shared is to use `amap_copy` during the fork to immediately clear needs-copy in the child process. The correct result of the fork is shown in Figure 4.23. The share flag has been added to the amap structure in UVM to detect this situation.

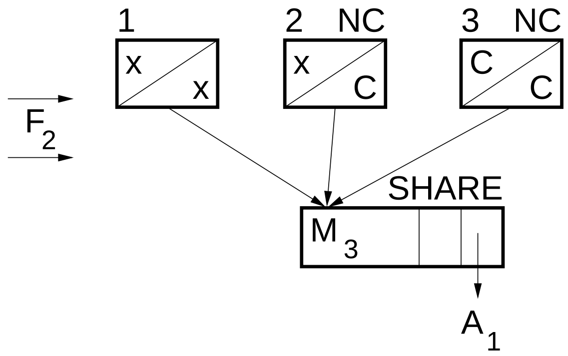


Figure 4.22: Erroneous copy-inherit fork

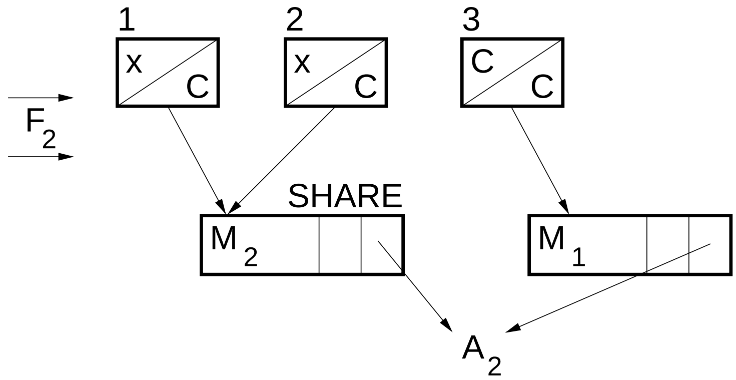


Figure 4.23: Erroneous copy-inherit fork

4.7.3 Copy Inheritance of a Shared Mapping

By default a shared mapping's inheritance is share. However, a shared mapping's inheritance can be changed to copy by the `minherit` system call. Shared mappings normally do not have amaps associated with them (because there is no need to promote their data to an amap for copy-on-write). However, with UVM certain operations can cause amaps to be allocated for shared regions. In the unusual case of a shared mapping having an amap and copy inheritance, the mapping is handled as follows by fork:

- The data in the amap layer immediately becomes copy-on-write in both the parent and child process.
- The data in the object layer remains shared in the parent process, but becomes copy-on-write in the child process.

4.7.4 Copy Inheritance when the Map Entry is Wired

Memory is said to be wired if it is always resident and thus never has to be fetched from backing store during a page fault. Wired memory is used to improve the performance of time-critical processes by reducing page fault overhead. If a process with a wired map entry whose inheritance value is copy forks, then the child process will want a copy-on-write copy of the region. However, in order to do that the parent's mapping would have to be protected. In order to keep fault time down this is to be avoided.

So, in the case of wired memory, rather than following the normal copy-on-write pattern, the fork routine calls the `amap_cow_now` function. This function ensures that the child process has an amap and that all copy-on-write faults have been resolved. By doing this, the parent pays the up-front cost of resolving all copy-on-write faults in one shot and avoids having its mappings protected for deferred copy-on-write faults.

4.8 Summary

In this chapter we have reviewed the function of anonymous memory in UVM. We introduced the amap interface and described UVM's reference amap implementation. We explained how UVM handles partial unmaps of regions of anonymous memory efficiently. We then explained the role of amaps in copy-on-write and how Mach-style memory inheritance interacts with the amap system.

Chapter 5

Handling Page Faults

This chapter describes the operation of `uvm_fault`, the UVM page fault handler. Page faults occur when a process or the kernel attempts to read or write virtual memory that is not mapped to allow such an access. This chapter includes an overview of UVM's page fault handling, a discussion of error recovery in the fault handler, and a comparison between the BSD VM's fault routine and UVM's fault routine. Note that the effect of page loanout on the fault routine is described in Chapter 7.

5.1 Page Fault Overview

The page fault handler is called by low-level machine-dependent code when the MMU detects a memory access that causes a page fault. The fault handler must use the information stored in the machine-independent layer of the virtual memory system to resolve the fault so that the faulting process can resume execution. If the page fault handler is unable to resolve the fault, an error signal is sent to the faulting process. There are two kinds of page faults: read faults and write faults. A read fault occurs when a process or the kernel attempts to read from unmapped virtual memory. A write fault occurs when a process or the kernel attempts to write to unmapped virtual memory or write to a read-only mapped page of virtual memory.

In the BSD VM system, the page fault routine handles page faults by traversing shadow and copy object chains, as described earlier in Section 2.4.5. Because UVM has a new pager interface and no longer has shadow and copy object chains, the BSD VM page fault routine cannot be used in UVM — it is no longer applicable.

UVM includes a brand-new page fault routine that is simpler, more efficient, and easier to understand than the BSD VM's fault routine. At a high-level, UVM's page fault

```
uvm_fault() {
  look up faulting address' map entry;
  if (data is in amap layer) { /* case 1 */
    handle any copy-on-write processing;
    map page and return success;
  } else if (data is in uvm_object layer) { /* case 2 */
    handle any copy-on-write processing;
    map page and return success;
  } else {  /* missing data */
    return error;
  }
}
```

Figure 5.1: High-level pseudo code for the fault routine

routine operates by first looking up the faulting address in the faulting process' VM map. Once the appropriate map entry is found the fault routine must check for the faulting page in each of UVM's two-levels of mapping. The fault routine first checks the amap level to see if the faulting page is part of anonymous memory. If not, then the fault routine requests the faulting page from the the `uvm_object` associated with the faulting map entry. In either case, if a copy-on-write fault is detected a new anon is allocated and inserted into the amap at the appropriate position. Finally, the fault routine establishes a mapping in the faulting process' pmap and returns successfully. High-level pseudo code for the fault routine is shown in Figure 5.1. The remainder of Section 5.1 describes the step-by-step operation of UVM's page fault routine.

5.1.1 Calling `uvm_fault`

UVM's fault routine is called from the machine-dependent code with the following information:

faulting virtual address: The machine-dependent code obtains from the MMU the address that was accessed in virtual memory to trigger the page fault.

faulting `vm_map`: Based on the faulting virtual address, hardware information, and the pointer to the currently running process the machine-dependent code must determine if the fault occurred in memory managed by the current process' `vm_map` or by the kernel's `vm_map`. The faulting map is passed to the fault routine.

fault type: Based on information from the hardware, the machine-dependent code must determine the type of page fault that occurred. The fault type is "invalid" if the virtual address being accessed was not mapped. The fault type is "protect" if the virtual address being access was mapped, but the access violated the protections on the mapping (e.g. writing to a read-only mapping). Finally, there is an internally used fault type, "wire," that is used when wiring pages.

access type: Based on information from the hardware, the machine-dependent code must determine if the faulting process was reading or writing memory at the time of the fault. This is the fault access type.

The fault routine will use these four pieces of information to process the fault. Once the fault routine has resolved the fault it returns a status value to the machine-dependent code that called it. The fault routine will either return "success" indicating that the fault was resolved and that the faulting process can resume running, or it will return an error code that indicates that faulting process should be sent an error signal. If there is an error and the faulting process is the kernel the system will usually crash ("panic").

5.1.2 Fault Address Lookup

The first step in handling a page fault is to determine what data is mapped in at the faulting address. This information is contained in the faulting `vm_map` structure's sorted linked list of map entries. Looking up the faulting address in the map is simply a matter of searching the linked list for the map entry structure that maps the faulting address[1]. However, there are two issues that complicate the process: map locking and map entries that point to share maps or submaps.

Every map structure has its own lock. Map are read locked while they are being read and write locked while they are being modified. A read-locked map and its corresponding map entry structures cannot be modified. Multiple processes can have a map read-locked at the same time. If a process or the kernel wants to change the mappings in a map, it must be write-locked. A map can be write-locked only if it is not already read-locked and only one process can write-lock a map at a time. When the fault routine is called, the faulting map is unlocked. The fault routine read-locks the map in order to lookup the faulting address. As long as the fault routine holds the read-lock on the map then the address lookup is valid. If

[1]Since a linear search of a linked list can get expensive if the list is long, the lookup code caches the result of the last lookup in a map as a "hint" as to where to start the next lookup. This reduces the overhead of the search.

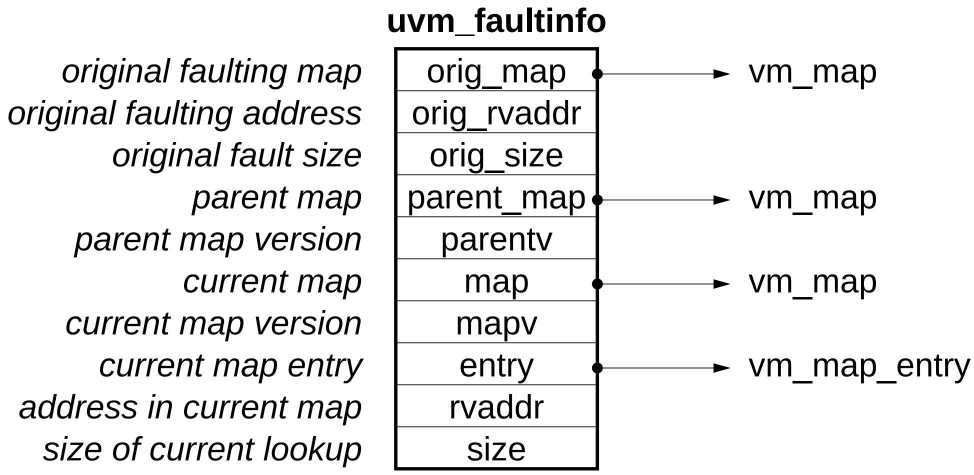


Figure 5.2: The faultinfo data structure. The first three fields are filled out at the start of the fault routine. The remaining fields are filled up by the `uvmfault_lookup` function when it does the map lookup. All virtual addresses are truncated to a page boundary.

the fault routine has to do a lengthy I/O operation to get the needed data, then it unlocks the faulting map so that other processes or threads may access the map while the I/O is in progress. Once the I/O operation is complete, the fault routine must relock the map to check that its mapping is still valid.

If a map lookup results in a map entry that points to a share map or submap, then the lookup must be repeated in that map. Since map entries in share maps and submaps are not allowed to point to share maps or submaps themselves, only two levels of maps can be encountered: the faulting map (called the "parent map") and the share or submap. Note that share maps and submaps have their own locks that must be held during lookups.

While the fault routine must handle map locking, share maps, and submaps properly, the details can be isolated in a data structure and a few helper functions so that the bulk of the fault routine code doesn't have to worry about how many maps it is dealing with. The data structure that contains information on the current mapping of the faulting address is called the `uvm_faultinfo` data structure, shown in Figure 5.2. The faultinfo data structure is often called "`ufi`" for short.

The first thing the fault routine does is to store the faulting map, address, and size in its faultinfo data structure. For page faults, the fault size is always the page size (the fault size field is for use by other functions such as page loanout).

The fault routine then calls the `uvmfault_lookup` function to look up the map entry that maps the faulting address. The fault lookup function takes a faultinfo structure with its first three fields filled in as input. It then locks the faulting map, and does a lookup on the faulting address. If the faulting address is not mapped, the lookup routine returns an error code. If the faulting address is in a region mapped by a share or submap, then the lookup routine makes the original map the "parent" map, locks the new map, and then repeats the lookup in the new map. If the lookup is successful, then the fault lookup function will fill in the remaining seven fields of the faultinfo structure. For faulting addresses that map to a share map or submap, the `parent_map` field will point to the original map, and the `map` field will point to the share or submap. If there is no share or submap, then `parent_map` will be null and `map` will point to the original map. Thus, the lowest-level map always ends up in the `map` field of the faultinfo structure. This map is sometimes referred to as the "current" map. The `entry` field is always a map entry in this map. If successful, the lookup function returns with the maps locked.

Note that the faultinfo structure contains version numbers for both the parent map and the current map. If the fault routine unlocks a map for I/O, these version numbers can be used to detect if the map was changed while unlocked. If the map was not changed, then the fault routine knows that the lookup information cached in its faultinfo structure is still valid, and thus a fresh lookup is not needed. However, if the version numbers are changed, then the fault routine must repeat the lookup in case the mappings have changed while the map was unlocked.

The UVM fault routine has four helper functions relating to map lookups. They are summarized in Table 5.1.

5.1.3 Post-lookup Checks

If the fault routine's call to the fault lookup routine fails, then an invalid address error is returned to the caller. If the lookup was successful, then the current map can be obtained through the faultinfo data structure. The fault routine is now holding read-locks on the maps in the faultinfo data structure. At this point the fault routine can perform a few quick checks before it starts looking at the mapped memory objects.

- The address' current protection is checked with the type of access that caused the fault. If the access is not allowed, then the fault routine unlocks the maps and returns a protection failure error. For example, if a process has a write fault to an area of memory mapped read-only, then the fault routine would return an error.

- The needs-copy flag of the map entry is checked. If the fault is a write fault or if the mapping is a zero-fill mapping, then the needs-copy flag will need to be cleared.

Table 5.1: Fault lookup functions

Function	Description
uvmfault_lookup	Looks up a virtual address in a map, handling share and submaps. The virtual address and map are passed to this function in a faultinfo structure. If successful, the lookup function returns with the maps locked.
uvmfault_unlockmaps	Unlocks the map pointed to by a faultinfo structure.
uvmfault_relock	Attempts to relock the maps pointed to by a faultinfo structure without doing a new lookup. This will succeed only if the maps' version numbers have not changed since they were locked.
uvmfault_unlockall	Unlocks an anon, amap, uvm_object, and a faultinfo structure.

This is done by calling the uvmfault_amapcopy function. This function drops the read-lock on the maps, obtains write-locks on the maps, and clears needs-copy by calling amap_copy. Then the fault routine restarts the fault processing.

- Once needs-copy has been cleared, the fault routine checks the current map entry to ensure that it has something mapped in at one of the two-levels. If the map entry has neither an amap nor a uvm_object attached to it, then there are no memory objects associated with it. In this case the fault routine unlocks the maps and returns an invalid address error.

5.1.4 Establishing Fault Range

After performing the post-lookup check, the UVM page fault routine establishes a range of pages to examine during the fault. This range could be just the faulting page (called a "narrow" fault), or it could include several neighboring pages. The size of the range is determined by the size of virtual memory mapped by the map entry and the map entry's memory usage hint (the "advice" field in the map entry). For sequential memory usage, pages behind the faulting page are flushed from memory. For random memory access, the range is just the faulting page. Normal memory access has a range of a few pages in either direction.

5.1.5 Mapping Neighboring Resident Pages

After establishing the range of fault, the fault routine then looks for neighboring resident pages in the mapping's amap. If a neighboring page is found to be currently unmapped, but is resident, then the fault routine establishes a mapping for it. This is done in hopes of reducing future faults by anticipating the need for a resident page and mapping it in before the fault occurs.

UVM currently ignores neighboring non-resident anon with data paged out to the swap area. A possible future enhancement to UVM would be to modify it to start a page-in operation on these pages in hopes of having them resident by the time the program faults them.

If the fault routine detected that a `uvm_object`'s page is needed to resolve the fault then it looks for resident pages in that object as well[2]. The fault routine gets these pages by calling the object's "pager get" function. Neighboring object pages are mapped into the process based on the memory usage hint. Because all data structures are locked when the get function is called, this is called a "locked get" operation. If the faulted data is not resident, the fault routine will later unlock the fault data structure and call the get function again to do the I/O to backing store. This is called an "unlocked get."

After the neighboring resident pages are mapped, the preliminary work of the fault routine is done. The rest of the fault routine focuses on getting the page that was faulted on mapped in to resolve the fault and allow the faulting process to resume execution. In UVM, page faults are divided into two major cases based on UVM's two-layer mapping scheme. Faults on data in the upper layer are "case 1" or "anon" faults. Faults on data in the lower layer are "case 2" or "object" faults.

5.1.6 Anon Faults

In an anon fault, the data being faulted resides in an anon structure. Thus, the content of the mapping's object layer is irrelevant. There are three steps involved in processing an anon fault: ensuring that the anon's data is resident, handling the two sub-cases of anon faults, and mapping in the page.

[2]Certain objects need to have complete control over the handling of their faults. In this case, the fault routine passes control to an object's fault function to let it resolve the fault (see Section 8.4).

Ensuring the Faulting Data is Resident

The first thing that the fault routine must do in an anon fault is ensure that the data in the faulting anon is resident in an available page of physical memory. This is done with the "anonget" function (called `uvmfault_anonget`). The anonget function is called by the fault routine with faultinfo, amap, and anon as arguments. These structures must be locked before the function is called. The anonget function must be prepared to handle the following three cases:

- **non-resident:** The data has been paged out to the swap area.
- **unavailable resident page:** The data is resident, but the page it resides in is currently busy.
- **available resident page:** The data is resident and its page is available for use.

Each of these cases is described below.

If the data is non-resident, then the anonget function makes it resident. This is done by allocating a new page, unlocking the fault data structures, and issuing a request to the swap I/O layer to read the data from swap into the new page. While the read is in progress the process will sleep, thus releasing the CPU in order to allow other processes to run. When the I/O is finished, the fault routine will resume execution.

If the data is resident in an unavailable page, then the anonget routine must mark the resident page "wanted," unlock the fault data structures, and then sleep until the page is available. There are two types of unavailable pages: "busy" pages and "released" pages. A page is marked busy if its data is in flux and should not be accessed. For example, pages whose data is currently being transfered between physical memory and backing store are marked busy. A process that sets the busy flag of a page is said to be its "owner." Released pages are pages that are in the process of being freed. When clearing a page's busy or released flags, it is the responsibility of the page's owner to check to see if the page is wanted. If the page is wanted, then the owner must wake any process waiting for access to the page.

If the data is resident in an available page, then the anonget routine can successfully return. The pseudo code for `uvmfault_anonget` is shown in Figure 5.3.

Note that the anonget function must be prepared to handle several error conditions, including running out of physical memory, I/O errors while reading from the swap area, and problems re-locking the fault data structures after sleeping. A discussion of error recovery in the fault routine will be presented later, in Section 5.2.

```
uvmfault_anonget() {

  while (1) {
    if (page resident) {
      if (page is available)
        return(success);    /* done! */
      mark page wanted;
      unlock data structures;
      sleep;
    } else {
      allocate new page;
      unlock data structures and read page from swap;
    }
    relock data structures;
  }  /* end of while loop */

}
```

Figure 5.3: The `uvmfault_anonget` function

Anon Fault Sub-cases

Once the anon's data is resident, the fault routine divides anon faults into two sub-cases, as shown in Figure 5.4.

In case 1A, the page in the anon is mapped directly into the faulting process' address space. Case 1A is triggered by either a read-fault on an anon's page, or by a write-fault on an anon whose reference count is equal to one. Case 1B is triggered by a write-fault on an anon whose reference count is greater than one. Case 1B is required for proper copy-on-write operation. An anon's reference count can only be greater than one if two or more

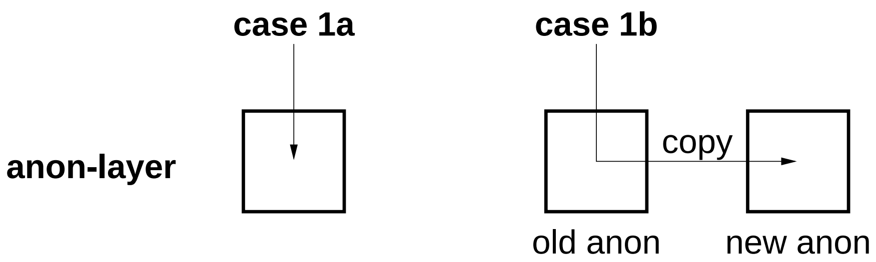


Figure 5.4: Anon fault sub-cases. The difference between case 1a and case 1b fault is that in case 1b the data must go through the copy-on-write process in order to resolve the fault.

processes are sharing its page as part of a copy-on-write operation. Thus, when writing to an anon in this state, a copy of the data must be made first to preserve the copy-on-write semantics. Likewise, for case 1A, when faulting on anons whose reference count is greater than one, the faulting page must be mapped in read-only to prevent it from being changed while the reference count is greater than one.

To handle case 1B, the fault routine must allocate a new anon containing a new page. The data from the old anon must be copied into the new anon, and then the new anon must be installed in place of the old one in the amap. The new anon will have a reference count of one, and the old anon will lose one reference. The fault routine must be prepared to handle the sort of out-of-memory errors described in Section 5.2 while handling case 1B.

Mapping the Page in

Once the copy-on-write process of case 1B has been addressed, the new page can be entered into the faulting process' pmap, thus establishing a low-level mapping for the page being faulted. This is done with a call to the `pmap_enter` function. Once the low-level mapping has been established the fault routine can return successfully. The faulting process will resume execution with the memory reference it made to the page that was just faulted on.

5.1.7 Object Faults

The second class of faults that the fault routine handles is the object fault. In this case, the data being faulted resides in a `uvm_object`. There are three steps involved in processing an object fault: fetching the data if it is not resident, handling the two sub-cases of object faults, and mapping in the page.

Fetching Non-resident Data

Earlier in the fault process, the fault routine did a locked get to find any resident pages in the object that were close to the faulting page. If the faulting page was resident and not busy, then the locked get returned the faulting page as well. The locked get sets the "busy" flag on the pages it returns. The fault routine does not have to worry about getting a page that is busy by another process because the locked get will not return such a page.

If the page is not resident, then the fault routine must issue an unlocked get request to the pager get function. To do this, the fault routine unlocks all its data structures (with the exception of the object — the pager get function will unlock that before starting I/O) and calls the pager get function. After starting I/O, the process will sleep in the pager get

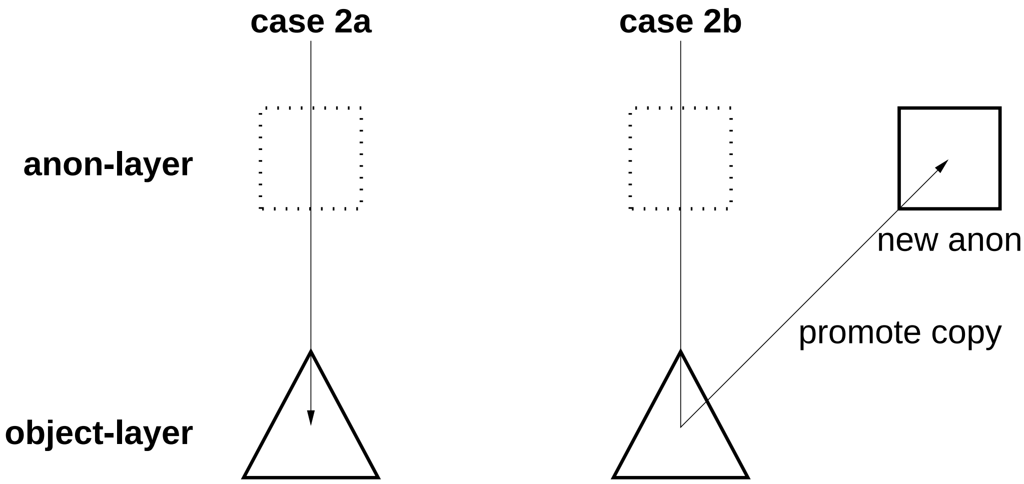


Figure 5.5: Object fault sub-cases. In case 2a the faulting process has direct access to the object's memory, while in case 2b the faulting process has copy-on-write access to the object.

function until the data is available. When the data is available, the pager get function will return with the faulting page resident and busy. The fault routine must check for errors and relock the data structures.

Object Fault Sub-cases

Once the object's data is resident, the fault routine divides object faults into two sub-cases, as shown in Figure 5.5.

In case 2A the object's page is mapped directly into the faulting process' address space. Case 2A is triggered by either a read-fault on an object's page, or by a write-fault on an object that is mapped shared (i.e. not copy-on-write). If a read-fault occurs on a copy-on-write region, then the page will be mapped in read-only to ensure that the faulting process cannot attempt to modify it without generating a page fault. Note that if the object is a memory mapped file, then changes made to its pages will be reflected back to the file and seen by other processes.

Case 2B is triggered by a write-fault on an object that is mapped copy-on-write, or by a fault on a copy-on-write region that has no object associated with it. The latter case happens with regions of virtual memory that are mapped "zero-fill." When data is copied from the lower-level object layer to the upper-level anon layer the data is said to be "promoted." To promote data, the fault routine must allocate a new anon with a new page and copy the data from the old page to the new page. The fault routine must be prepared to handle the sort of out of memory errors described in Section 5.2 while handling case 2B.

Mapping the Page in

Once the object page is resident and case 2B is taken care of, then the new page can be entered into the the faulting process' pmap with `pmap_enter`. After mapping the page, the fault routine must clear the faulting page's busy flag, and check the faulting page's wanted flag. If the page is wanted, then the fault routine wakes up any process that may be sleeping waiting for it. Once this is done, the fault routine can return successfully. The faulting process will resume execution with the memory reference it made to the page that was just faulted on.

5.1.8 Summary

In summary, a page fault happens as follows: First the faultinfo structure is filled in for the lookup operation. After the lookup operation is performed the protection is checked and needs-copy is cleared if necessary. Next, available resident neighboring pages are then mapped in. Finally, the fault routine resolves the fault by determining which case the fault is, getting the needed data, and mapping it in. Pseudo code showing the overall structure of UVM's fault routine is shown in Figure 5.6.

5.2 Error Recovery

The fault routine resolves most page faults without encountering any errors. However, errors can occur and the fault routine must be prepared to recover from them gracefully. This section describes the possible errors the fault routine may get and how it addresses them.

5.2.1 Invalid Virtual Addresses

Invalid virtual addresses are virtual addresses that are not mapped by any map entry in the faulting map. An invalid virtual address is detected by the failure of the fault lookup function. These errors are easy to handle because they are detected early on in the fault process. Since no resources have been allocated or locked, the fault routine can just return an invalid

```
uvm_fault() {
  fill in faultinfo structure;
ReFault:
  call uvmfault_lookup to look up mapping;
  check protection;
  clear needs-copy if necessary;
  check for either an amap or an object;
  establish range of mapping (for neighboring pages);
  map in resident anon pages;
  map in resident object pages if necessary;
  if (case 1) {
    ensure data is resident;
    if (case 1B) { do copy-on-write; }
    map in the correct page;
    return(success);
  } else {
    ensure data is resident;
    if (case 2B) { do copy-on-write promotion; }
    map in the correct page;
    return(success);
  }
  done!
}
```

Figure 5.6: Fault pseudo code

address error code to the caller. This error most often occurs when buggy programs attempt to reference data through an invalid pointer.

5.2.2 Protection Violations

Each map entry in a map has a protection code associated with it. Once the lookup function has found the proper map entry in the faulting map, the protection code is compared to the access type of the fault. If they are incompatible (e.g. a write to a read-only area of memory), then the fault routine handles the error by unlocking the maps and returning a protection failure error code. This error most often occurs when buggy programs attempt to write to their read-only text area.

5.2.3 Null Mappings

If a map entry has neither an amap nor an object and the mapping is not copy-on-write, then the map entry is just reserving space in the map. It is not actually mapping anything into the space. Memory access to this sort of region is an error condition. The fault routine handles null mappings by unlocking the maps and returning an invalid virtual address error. This error most often occurs when buggy programs attempt to reference data through an invalid pointer.

5.2.4 Out of Physical Memory

The fault routine allocates physical memory in two cases. First, if the faulting memory reference is on a previously unaccessed zero-fill area of memory, then the fault routine must allocate and zero a page to map into that area. Second, if the faulting memory reference is a write to a copy-on-write area of memory, then the fault routine must allocate a page of physical memory to copy the data to. This can happen either at the anon or object layer (fault cases 1B and 2B).

If the system is out of physical memory, then the fault routine's allocation request will fail. The fault routine handles this condition by unlocking all its data structures and waking the pagedaemon. It then waits for the pagedaemon to free up some physical memory by doing pageouts. When the pagedaemon has done this, it will send a wakeup to the faulting process. The faulting process then restarts the fault.

5.2.5 Out of Anonymous Virtual Memory

The fault routine can run out of anonymous virtual memory if the entire swap area and most of RAM is allocated. This case is detected by the failure of an allocation of a new anon. UVM's fault routine handles this condition by unlocking the data structures and causing the fault to fail.

There is room for improvement in UVM's handling of this condition. The BSD VM system "handles" this condition by ignoring it. If you run a BSD VM system out of virtual memory the system will deadlock. In UVM, the process that runs the system out of virtual memory will get an error signal at the time of the fault. In the future, it would be worthwhile to investigate ways to report the shortage of virtual memory at memory allocation time (when the allocating process has a chance to do something about it) rather than at fault time (when the allocating process will be unlikely to be able to handle it).

5.2.6 Failure To Relock Data Structures

If the fault routine has to wait for an I/O operation or wait for memory to become free it always unlocks all its data structures before going to sleep. Once the I/O operation is complete or the memory being waited on becomes available, the faulting process resumes execution. In this situation, the first thing the fault routine must do is re-establish locks on all the data structures associated with the fault. Since these data structures were unlocked for a period of time, it is possible that other processes could have changed or even freed them while the faulting process was waiting.

UVM uses the `uvmfault_relock` function to relock the map data structures. As described earlier, the relock function uses the map's version numbers to determine if the old lookup is still valid. If the lookup is not valid, then the fault routine recovers from this by un-busying any busy pages that it is holding and restarting the fault from the top (called a "refault").

The BSD VM system does not make use of the map's version numbers — it always repeats the lookup. Using the map's version numbers to avoid the second lookup appears to be a feature of Mach VM that was removed from BSD. The `uvmfault_relock` function restores this feature to UVM.

5.2.7 I/O Errors

If the fault routine has requested that an I/O operation be performed to fetch a faulting page from backing store it is possible that the I/O operation will fail. For example, if a disk starts to malfunction or a device driver cannot get access to resources it needs to perform the I/O, then the page-in will fail with an error.

If the page-in operation fails while attempting to page in anonymous memory, then the fault routine must restore the faulting anon back to the state it was in before the fault. That means that the page that has been associated with the anon must be removed from it and freed. If the page-in operation fails while attempting to page-in object pages the object's pager is responsible for freeing any allocated pages. The pager can return two types of error codes: one to indicate that the fault routine should re-try the fault and one to indicate that the page-in failed and that the fault routine should give up.

5.2.8 Released Pages

When the fault routine or pagedaemon is having I/O performed on a page the page's busy flag is set to indicate to other processes that this page is currently in flux and should not be accessed. While the I/O is in progress it is possible for some other process to want to free the busy page. This could be due to the page being unmapped or due to a pager flush operation. Because the page is currently in the middle of an I/O operation (and thus is marked busy) it cannot be freed. Rather than freeing it, the page's released flag is set. This tells the process that set the busy flag that the page needs to be freed as soon as the I/O is complete.

The fault routine must check a page's released flag after relocking the faulting data structures to ensure that the page is still valid. If the page is marked released, then the fault routine must dispose of the page and restart the fault. Anon pages can be marked released only if they are part of an anon that is no longer part of an amap (and thus the anon's reference count is zero). In this case, the fault routine must free both the anon's page and the anon itself[3]. Released object pages are handled by the object's page release function (described in detail in Chapter 6). Note that one possible result of calling an object's page release function is that the object itself is terminated. In this case, the fault routine must be careful to drop all references to the terminated object.

5.3 BSD VM Fault vs. UVM Fault

The BSD VM fault routine and the UVM fault routine differ in a number of ways. This section highlights the major differences.

5.3.1 Map Lookup

In the BSD VM system, map lookups are handled by the `vm_map_lookup` function. This function takes eight parameters. Three of the parameters are inputs to the function: the faulting map, the faulting address, and the fault type. The function performs the lookup, handling the presence of share and submaps. In addition, it also checks protection, creates shadow objects (if needs-copy is set), and creates zero-fill objects. When the faulting process is done with the lookup, it calls a "map lookup done" function to unlock the map.

In UVM, map lookups are handled by `uvmfault_lookup`. This function takes a faultinfo data structure and a locking flag and performs the lookup. It handles share and

[3]The `uvm_anfree` function performs both of these actions.

submaps. Issues such as protection checking and clearing needs-copy are left up to the calling process. Unlike BSD VM, UVM saves the map's versions in the faultinfo data structure and provides a relock function. This avoids the extra overhead of having to traverse the list of map entry structures when relocking the maps after an I/O.

While both BSD and UVM's lookup function ease the handling of share and submaps, the BSD's lookup function is overloaded to performs tasks not directly related to the lookup such as protection checking and object chain creation. This overloading increases the complexity of the BSD lookup function.

5.3.2 Object Chaining vs. Two-Level Mappings

Another source of complexity in the BSD VM fault routine are object chains. For each page fault, the BSD VM system must traverse down the shadow chain to find the faulting data. It then must push the changed data down the copy object chain. As pages are being written the BSD VM has to check its object chains to see if they can be collapsed to prevent them from becoming too long. There is no limit to the number of objects that the BSD VM must examine to resolve a fault! In contrast, UVM has a simple two-level mapping scheme that does not require any extra collapse processing. The faulted data can be in only one of two places: the amap layer or the object layer.

In order to properly handle a fault, the BSD VM system must do the following if it is going to unlock the maps to perform I/O:

- A busy page must be placed in the top-level shadow object of the faulting map. This prevents a race condition between the faulting process and other processes that share the same address space. Without the busy page, two processes could be trying to install the same page in the top-level object at the same time.

- Any object that the fault routine puts a busy page in must have its "paging in progress" counter incremented. In the BSD VM, any object that has a non-zero paging in process counter is not allowed to be collapsed. This prevents objects from being yanked out from under the fault routine while it is waiting for I/O to complete.

Because of UVM's two-level mapping scheme, there is no need for UVM's fault routine to worry about races down an object chain or objects being collapsed out from under it. This simplifies UVM's fault handling.

5.3.3 Mapping Neighboring Resident Pages

As explained earlier, the UVM fault routine maps in neighboring resident pages while processing a fault. This usually reduces the total number of page faults processes have to take. In contrast, the BSD VM fault routine ignores neighboring pages. If they are needed they will be faulted in later. Thus, UVM reduces process' fault overhead. For example, the number of pages faults for the command "`ls /`" was reduced from 59 to 33 by UVM (see Section 10.5.3).

It should be noted that UVM's two-level mapping scheme makes mapping in neighboring pages relatively easy. The neighboring pages are either in the amap or the object layer. In the BSD VM system to search for neighboring pages, the fault routine would have the additional overhead of walking the object chains to find neighboring pages.

5.4 Summary

In this chapter we have reviewed the function of the UVM fault routine. We explained the steps the fault routine goes through to resolve a fault. The possible errors that the fault routine may encounter and how it recovers from the were described. Finally, some of the differences between the UVM fault routine and the BSD VM fault routine were presented.

Chapter 6

Pager Interface

This chapter describes UVM's pager interface. The pager is responsible for performing operations on uvm_object structures. Such operations include changing the reference count of a uvm_object and transferring data between backing store and physical memory. This chapter contains a description of the role of the pager, explanations of the pager functions, and a comparison of UVM's pager interface with the BSD VM pager interface.

6.1 The Role of the Pager

In UVM, a pager is a data structure that contains pointers to a set of functions that perform operations on a uvm_object. Each uvm_object structure on the system has a pointer to its set of pager functions. UVM currently has three pagers:

aobj: The aobj pager is used by uvm_object structures that contain anonymous memory.

device: The device pager is used by uvm_object structures that contain device memory.

vnode: The vnode pager is used by uvm_object structures that contain normal file memory.

UVM performs all pager-related operations on a uvm_object through its pager functions. This allows UVM to treat uvm_object structures that represent different types of memory alike.

Note that there are two types of memory objects in UVM: uvm_object structures and anon structures. Pagers only perform operations on uvm_object memory objects. For the remainder of this chapter the term "object" will be used as shorthand for "uvm_object structure."

Function	Description
pgo_init	init a pager's private data structures – called when the system is booted
attach	gain a reference to kernel data structure that can be memory mapped by an object
pgo_reference	add a reference to an object
pgo_detach	drop a reference to an object
pgo_get	get pages from backing store
pgo_asyncget	start an asynchronous I/O operation to get pages from backing store
pgo_fault	object fault handler function
pgo_put	write pages to backing store
pgo_flush	flush an object's pages from memory
pgo_cluster	determine how many pages can be get or put at the same time
pgo_mk_pcluster	make a cluster for a pageout operation
pgo_shareprot	change protection of an object that is mapped by a share map
pgo_aiodone	handle the completion of an asynchronous I/O operation
pgo_releasepg	free a released page

Table 6.1: The Pager Operations. Note that the attach function is not part of the uvm_pagerops structure.

6.2 Pager Operations

A pager is composed of a set of functions that perform operations on an object. These functions are called pager operations. Pointers to all of these functions except one (the "attach" function) are defined by a uvm_pagerops structure. This section contains a description of each of the pager operations. The pager operations are summarized in Table 6.1.

6.2.1 The Init Operation

UVM keeps a global list of all pagers in the VM system. When the system is booting, UVM's startup routine will call each pager's pgo_init function. This allows each pager to set up any global data structures it needs before the pager is first used. For example, the device pager keeps a linked list of all allocated device-backed objects. The device pager's pgo_init function initializes this list to the empty state.

Note that the pager init function is not needed by all pagers. Pagers that do not need to perform any actions at system startup time can set the pgo_init pointer to NULL.

6.2.2 The Attach Operation

A pager's attach function is used to gain an initial reference to an object. Because this function must be called before the VM system has a pointer to the object and its corresponding `uvm_pagerops` structure, the attach function is not included as part of the `uvm_pagerops` structure. The attach function returns a pointer to the `uvm_object` structure.

Each of UVM's three pagers has its own attach function. These functions are:

`uao_create`: the aobj pager attach function. The `uao_create` functions allocates a new aobj-object of the specified size and returns a pointer to its `uvm_object` structure. This type of object is used to map System V shared memory.

`udv_attach`: the device pager attach function. This function takes a `dev_t` device identifier and returns a pointer to its `uvm_object`. If the device does not currently have a `uvm_object` structure allocated for it then the attach function will allocate one for the device.

`uvn_attach`: the vnode pager attach function. This function takes a pointer to a vnode structure and returns a pointer to the vnode's `uvm_object` structure. If the `uvm_object` structure is not being used, then `uvn_attach` will initialize it. The `uvn_attach` function also takes an access level as an argument. The access level is used to determine if the vnode is being mapped writable or not.

All of the attach functions return `NULL` if the attach operation fails.

6.2.3 The Reference Operation

The reference function takes a pointer to an unlocked object and adds a reference to it. The reference function is called when performing mapping operations. For example, when a process is forking a child process, the child gains a reference to any `uvm_object` structures that it inherits from the parent process.

6.2.4 The Detach Operation

The detach function takes a pointer to an unlocked object and drops a reference from it. The detach function is called during an unmap operation to dispose of any references to `uvm_object` structures the area being unmapped may have.

6.2.5 The Get Operation

The get function is used to obtain pages from an object. It is most frequently called by the page fault routine to get an object's pages to help resolve a page fault. The get routine returns one or more contiguous pages in an object.

When requesting a block of pages, the fault routine has one page that it is especially interested in — the one the page fault occurred in. That page is called the "center" page. While the get must always fetch the center page if possible, fetching the requested neighboring pages is left up to the get function. This behavior can be modified with the "allpages" flag. If set, this instructs the get function that all pages are of equal importance and that they should all be fetched.

As explained earlier in Chapter 5, the get operation has two modes of operation: a locked get and an unlocked get. In a locked get the calling function is assumed to have critical data structures (including the object) locked and does not wish to unlock them. Because returning non-resident pages requires an I/O operation and I/O operations require the calling function to sleep, which in turn requires data structures to be unlocked, only resident pages can be returned by a locked get. In an unlocked get, the pager is allowed to unlock the object and perform I/O.

The get function takes eight arguments:

uobj: A pointer to the locked object whose data is being requested. This object may be unlocked for I/O only in an unlocked get operation.

offset: The location in the object of the first page being requested.

pps: An array of pointers to `vm_page` structures. There is one pointer per page in the requested range. The calling function initializes the pointers to null if the get routine is to fetch the page. Any other value (including the "don't care" value) will cause the get function to skip over that page.

npagesp: A pointer to an integer that contains the number of page pointers in the "pps" array. For locked get operations, the number of pages found is returned in this integer.

centeridx: The index of the "center" page in the pps array.

access_type: Indicates how the caller wants to access the page. The access type will either be read or write.

advice: A hint to the pager as to the calling process' access pattern on the object's memory (for example, normal, random, or sequential). This value usually comes from the map entry data structure's advice field.

flags: Flags for the get operation. The get function has two flags: "locked" and "allpages."

The get function returns an integer code value that indicates the results of the get operation. Possible return values are:

`VM_PAGER_OK`: The get operation was successful.

`VM_PAGER_BAD`: The requested pages are not part of the object.

`VM_PAGER_ERROR`: An I/O error occurred while fetching the data. This is usually the result of a hardware problem.

`VM_PAGER_AGAIN`: A temporary resource shortage occurred.

`VM_PAGER_UNLOCK`: The requested pages cannot be retrieved without unlocking the object for I/O. This value can only be returned by a locked get operation.

The get operation sets the busy flag on any page it returns on behalf of the caller. When the caller is finished with the pages it must clear the busy flag. Note that there is also a `VM_PAGER_PEND` return value that is used for asynchronous I/O. Since the get operation is always synchronous it cannot return this value.

The get operation can be called by the fault routine or the loanout routine (described in Chapter 7). Once the calling routine looks up the faulting or loaning address in the map, it then determines the range of interest. The range includes the page currently being faulted or loaned and any neighboring pages of interest. The calling routine then checks the amap layer for the page. If the page is not found in the amap layer and there is an object in the object layer, then the calling routine performs a locked get operation using the object's get routine. The calling routine passes an array of page pointers into the get routine with the faulting or loaning page as the center page. The pointers for neighboring pages that are already in the amap layer are set to "don't care." The rest of the pointers are set to null. If the locked get finds the center page resident and available it returns `VM_PAGER_OK`, otherwise it returns `VM_PAGER_UNLOCK`. If the pager returns "ok" then the calling routine has all the pages it needs. It need only clear the busy flag on the pages it gets from the get operation before returning. Note that a get operation can still return neighboring resident pages even if it returns `VM_PAGER_UNLOCK`.

If the locked get returns VM_PAGER_UNLOCK, then the calling routine will eventually have to unlock its data structures and perform an unlocked get. Note that an unlocked get operation requires a locked object. The get routine will unlock the object before starting I/O. When I/O is complete, the get routine will return with the object unlocked. The calling routine must then relock all its data structures and reverify the map lookup before continuing to process.

6.2.6 The Asyncget Operation

The asyncget function causes the pager to start paging in an object's data from backing store. However, unlike the normal get operation, the asyncget function returns after starting I/O — it does not wait for it to complete. This allows the VM system to preload data in an object in hopes of having the data resident by the time it is needed.

UVM currently does not use the asyncget operation. It is provided for possible future use to improve the performance of the system.

6.2.7 The Fault Operation

The fault function is an operation that allows the pager more control over how its pages are faulted in. Pagers either have a "get" operation or a "fault" operation. The pager get operation takes an object and returns pointers to vm_page structures. For pagers such as the device pager this is not sufficient. The device pager manages device memory and that memory does not have or need any vm_page structures associated with it. Thus, the device pager has nothing it can return to the fault routine through the get interface.

The pager fault operation allows the pager to take over and resolve a page fault on its memory. It takes the following eight arguments:

ufi: A uvm_faultinfo data structure that contains the current state of the fault.

vaddr: The virtual address of the first page in the requested region in the current vm_map's address space.

pps: An array of vm_page pointers. The fault routine ignores any page with a non-null pointer.

npages: The number of pages in the pps array.

centeridx: The index in pps of the page being faulted on.

fault_type: The fault type.

access_type: The access type.

flags: The flags (only “allpages” is allowed).

The fault operation has the same set of return values as the get operation. In order for the fault operation to successfully resolve a fault, it must call `pmap_enter` to establish a mapping for the page being faulted.

6.2.8 The Put Operation

The put function takes modified (“dirty”) pages in an object and flushes the changes to backing store. It is called by the pagedaemon and the pager flush operation to clean dirty pages. The put routine takes the following arguments:

uobj: The locked object to which the pages belong. The put function will unlock the object before starting I/O and return with the object unlocked.

pps: An array of pointers to the pages being flushed. The pages must have been marked busy by the caller.

npages: The number of pages in the pps array.

flags: The flags. Currently there is only one flag: `PGO_SYNCIO`. If this flag is set then synchronous I/O rather than asynchronous I/O is used.

The put operation returns either one of the return values used by the get operation or — when an asynchronous I/O operation has been successfully started — `VM_PAGER_PEND`. In cases when the put operation returns `VM_PAGER_PEND` the pager will unbusy the pages and set their clean flag rather than the calling function.

If the put routine returns anything other than `VM_PAGER_PEND` then the calling function must clear the pages’ busy flags. If the put was successful it must also set the pages’ clean flags to indicate that the page and backing store are now in sync.

6.2.9 The Flush Operation

The flush function performs a number of flushing operations on pages in an object. These operations include:

- marking pages in an object inactive.
- removing pages from an object.
- writing dirty pages in an object to backing store (either synchronously or asynchronously).

The flush operation is used by the `msync` system call to write memory to backing store and invalidate pages. The vnode pager uses its flush operation to clean out all pages in a vnode's `uvm_object` before freeing it.

The flush routine takes the following arguments:

uobj: The locked object to which the pages belong. The flush function returns with the object locked. The calling function must not be holding the page queue lock since the flush function may have to lock it. The flush function will unlock the object if it has to perform any I/O.

start: The starting offset in the object.

end: The ending offset in the object. Pages from the ending offset onward are not flushed.

flags: The flags.

The flush operation returns "TRUE" unless there was an I/O error.

The flush operation's flags control how the object's pages are flushed. The flags are:

`PGO_CLEANIT`: Write dirty pages to backing store. If this flag is set, then the flush function will unlock the object while performing I/O. If this flag is not set, then the flush function will never unlock the object.

`PGO_SYNCIO`: Causes all page cleaning I/O operations to be synchronous. The flush routine will not return until I/O is complete. This flag has no effect unless the "cleanit" flag is specified.

`PGO_DOACTCLUST`: Causes the flush function to consider pages that are currently mapped for clustering when writing to backing store. Normally when writing a page to backing store the flush function looks for neighboring dirty inactive pages to write along with the page being flushed. Specifying this flag causes the flush function to also consider active (mapped) pages for clustering.

PGO_DEACTIVATE: Causes an object's pages to be marked inactive. This makes them more likely to be recycled by the pagedaemon if memory becomes scarce.

PGO_FREE: Causes an object's pages to be freed. Note that PGO_DEACTIVATE and PGO_FREE cannot both be specified at the same time. If both PGO_CLEANIT and PGO_FREE are specified, the pages are cleaned before they are freed.

PGO_ALLPAGES: Causes the flush operation to ignore the "start" and "end" arguments and perform the specified flushing operation on all pages in the object.

6.2.10 The Cluster Operation

The cluster function determines how large a clustered I/O operation can be. It takes an object and the offset of a page in that object. The calling function is interested in building a cluster of pages around that page. The cluster routine returns the starting and ending offsets of a cluster of pages built around the page passed in. Since the largest size I/O operation a device driver is required to perform is MAXBSIZE, a cluster can include at most MAXBSIZE bytes of data. For pagers that do not wish to cluster the cluster routine can just return a "cluster" of one page. The cluster operation is currently only used when building a cluster for a pager put operation. Future UVM enhancements may make use of this operation.

6.2.11 The Make Put Cluster Operation

The make put cluster function is an optional pager operation that builds a cluster of pages for a pageout operation. If the make put cluster function is null, then I/O will never be clustered on pageout. UVM provides uvm_mk_pcluster, a generic make put cluster function that pagers may use if a special function is not needed. In UVM currently all pagers that do clustered pageout use the generic function. The make put cluster routine is called by the pagedaemon when paging out data to backing store to see if any neighboring pages can be included in the I/O operation.

The make put cluster function is called and returns with both the page queues and the pager's object locked. It takes an uninitialized array of page pointers and a busy write-protected page. It uses the object's cluster operation to establish the range of the cluster, and then fills the array with pages that are candidate for being included in the cluster. These pages must be dirty pages that are not already busy. The make put cluster function will write protect and set the busy flag on any page it adds to the cluster. The caller is responsible for clearing this flag when it is done with the pages in the cluster.

6.2.12 The Share Protect Operation

The share protect operation is an optional pager operation that changes the protection of an object's pages in all maps that they are mapped in. The object must be locked by the caller. This function is called when removing mappings from a share map. If memory in a share map is unmapped, then it must be removed from all process' that are sharing the map. Since the VM system does not keep track of which processes are sharing a share map, it must play it safe and remove the mappings from all processes that map the memory. If a pager does not provide a share protect function, then objects that use that pager are not allowed to be mapped by share maps. UVM provides a generic share protect operation that is currently used by all pagers that can be used by share maps.

6.2.13 The Async I/O Done Operation

In a synchronous I/O request, the pager starts the I/O operation, waits for the I/O to complete, and then returns a status value. In an asynchronous I/O operation, the pager starts the I/O operation and immediately return `VM_PAGER_PEND` (I/O pending). This allows the calling process to continue to run while the I/O is in progress. When the asynchronous I/O operation completes the pager's async I/O done operation is called to finish the I/O operation.

The pager normally has the option of using asynchronous or synchronous I/O. Since the pager can always opt to use synchronous I/O, pagers are not required to support asynchronous I/O. A calling process has the option of forcing a pager to use synchronous I/O by specifying the `PGO_SYNCIO` flag to the pager I/O operations. This is useful if the process wants to ensure that the I/O is complete before continuing.

In synchronous I/O, the state of the I/O operation can be stored on the calling process' kernel stack. However, with asynchronous I/O, the calling process starts the I/O and then continues running (and using its kernel stack) while the I/O is in progress. So the kernel stack cannot be used to store I/O state for asynchronous I/O operations. Instead, the pager must store the state of the asynchronous I/O operation in its own memory area. Part of this memory area is the `uvm_aiodesc` structure. The `uvm_aiodesc` structure describes the pager-independent state information associated with an asynchronous I/O operation. This structure contains the number of pages in the I/O operation, the kernel virtual address those pages are mapped in at, a pointer to the async I/O done function, a linked list pointer, and pointer to pager-dependent data.

When a pager decides to perform an asynchronous I/O operation the following happens: The pager allocates memory to store state information, starts the I/O operation, and returns `VM_PAGER_PEND`. When the I/O operation completes an interrupt is generated. As part of the interrupt handling, the pager's `uvm_aiodesc` structure for the current I/O operation is placed on a global list of completed asynchronous I/O operations (`uvm.aio_done`) and a wakeup signal is sent to the pagedaemon. The pagedaemon is responsible for finishing the asynchronous I/O operation. It goes down the list of complete asynchronous I/O operations and calls the asynchronous I/O done function for each one. The asynchronous I/O done function must un-busy the pages involved in the I/O and wake up any processes that were waiting for the I/O operation to complete. It finally frees the asynchronous I/O state information.

The reason the asynchronous I/O done function is called from the pagedaemon rather than directly from the interrupt handler is that the VM data structure locking currently does not handle access from the bottom half of the kernel.

6.2.14 The Release Page Operation

The release page function is called when a released object page is encountered. This can happen only if the following is true: a process locks an object, sets the busy flag on one of the object's pages (e.g. for I/O), and then unlocks the object. If some other process locks the object to free a page only to discover that it is busy, then it must set the released flag. Eventually, the I/O will complete and the process that set the busy flag will relock the object and clear the busy flag. At that time it must check to see if the released flag is set. If so, then the process must call the object's release page pager operation to free the page.

The release page function takes a pointer to a locked object and a released page in that object. It disposes of the released page and then returns either true or false. If releasing the page caused the object to be terminated (e.g. because the last page of an object being removed from the system was freed) then the release page function returns false indicating that the object has been freed. The release page function returns true if the object is still alive after the release page operation.

6.3 BSD VM Pager vs. UVM Pager

In this section the role of the pager in BSD VM is compared to its role in UVM. UVM uses a different data structure layout than BSD VM. Pages are also accessed differently in UVM.

6.3.1 Data Structure Layout

There is a significant difference between the way the pager-related data structures are organized in BSD VM and UVM. First, in UVM the `uvm_object` structure is designed to be embedded within a memory mappable data structure. In BSD VM the `vm_object` data structure is allocated and managed separately from the object it memory maps. Second, in UVM the object structure has a pointer directly to the pager operations structure. In BSD VM, the object points to a `vm_pager`. That structure has a pointer to the pager operations structure and a private pointer to a pager-specific data structure. In UVM, the pager specific data is stored in the object in which the `uvm_object` is embedded.

Figure 6.1 shows the data structure configurations for both BSD VM and UVM for a memory-mapped file. In BSD VM the object, `vm_pager`, `vn_pager`, and vnode are separately allocated structures. The pagerops structure contains the pager operations for the vnode pager and is shared among all pagers associated with vnodes. In UVM, the object and vnode-specific data (the `uvm_vnode` structure that serves the same role as the `vn_pager` data structure in BSD VM) are embedded in the vnode structure. UVM has no need for a `vm_pager` structure. Instead, the object points directly to the pager operations.

Note that on the BSD VM side of Figure 6.1 there is no pointer from the `vm_pager` data structure to the `vm_object` structure. Instead, the BSD VM system maintains a hash table that hashes a `vm_pager` pointer to a `vm_object` pointer. This hash table appears to be provide additional functions in Mach VM that are not used by BSD VM, and thus it could be replaced easily by a simple pointer BSD VM.

So, in order to set up the initial mappings of a file the BSD VM system must allocate three data structures (`vm_object`, `vm_pager`, and `vn_pager`), and enter the pager in the pager hash table. On the other hand, UVM doesn't have to access a hash table or allocate any data structures. All the data structures UVM needs are embedded within the vnode structure.

In addition to having more complex data structure handling, the BSD VM system also has an extra layer of pager functions. For example, consider the "put pages" function shown in Figure 6.2. This function does nothing but call out to the pager's

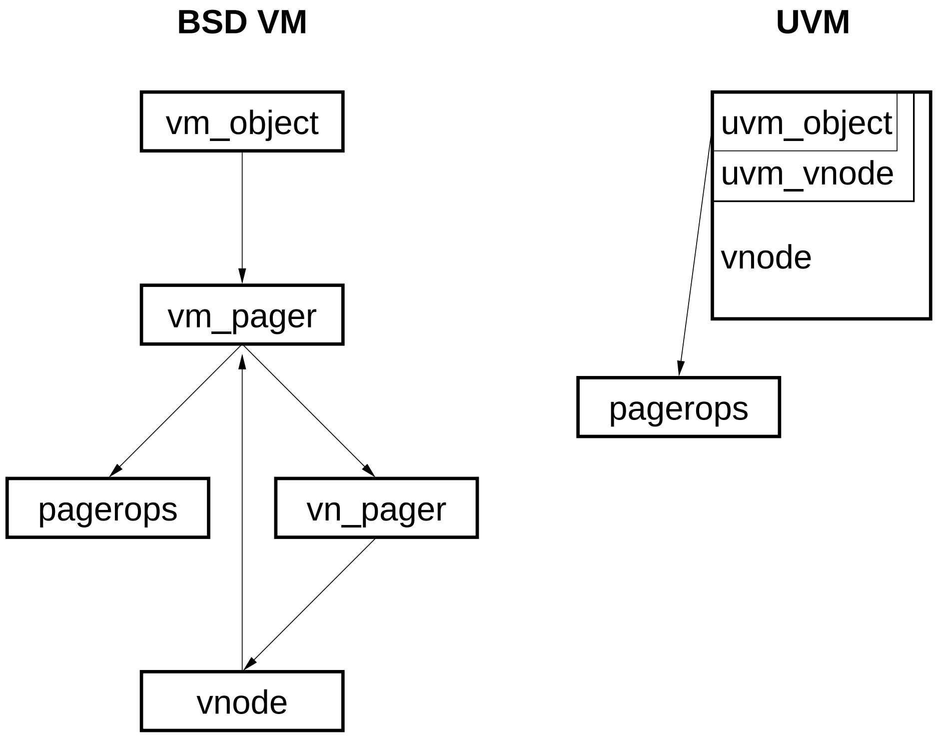


Figure 6.1: Pager data structures

`pgo_putpages` operation. The extra overhead of a function call is not needed. In UVM all pager operations are routed directly to the pager rather than through a small function like `vm_pager_put_pages`.

6.3.2 Page Access

The way pages are accessed in UVM is quite a bit different from the way they are accessed in BSD VM. In UVM all accesses to an object's pages go through the pager's get function. Resident pages are fetched with a locked get operation, and non-resident pages are fetched with an unlocked get operation. In BSD VM, the pager is only consulted if the page is non-resident. Resident pages are accessed directly by using the object-offset hash table. Since a BSD VM pager is only called when a page is non-resident, it has less information available to it on how its pages are being used. A UVM pager (and the filesystem layer underneath) can take advantage of this extra information to determine the page access pattern and make more intelligent decisions about which pages to keep resident, which pages to pre-fetch, and which pages to discard. This will be especially useful in the future when the VM and buffer caches are merged and all I/O goes through the a VM interface.

```
int vm_pager_put_pages(pager, mlist, npages, sync)

vm_pager_t pager;
vm_page_t *mlist;
int npages;
boolean_t sync;

{
  if (pager == NULL)
    panic("vm_pager_put_pages: null pager");
  return ((*pager->pg_ops->pgo_putpages)
            (pager, mlist, npages, sync));
}
```

Figure 6.2: The unnecessary `vm_pager_put_pages` function

Another difference between BSD VM and UVM is how the pager get operation gets pages. In BSD VM, the process fetching the data from backing store must allocate a free page, add it to the object, and then pass it into the pager get routine. In UVM, the process fetching the data does not allocate anything. If the pager needs a free page it allocates it itself. This allows the pager to have full control over when pages get added to the object.

A related problem with the BSD VM pager get operation is that it is required to return a `vm_page` structure. This causes problems for pagers that manage device memory because device memory does not have any `vm_page` structures associated with it. Consider what happens when a process generates a page fault on a device's page under BSD VM. The fault routine will allocate a free page and add it to the device's memory object. It will then call the pager get operation to fill the page with "data" (actually the device's memory). To service this request, the device pager must use the kernel memory allocator to allocate a fictitious `vm_page` structure that is associated with the device's memory. The device pager get operation must then remove the normal page of memory passed into it from the object, free the page, and then install the fictitious page in the object and return. This will allow the fault routine to resolve the fault. The BSD VM system must be careful that it does not allow a fictitious page to be added to the page queues or free page list as the fictitious page's memory is part of device memory.

In UVM fictitious pages are not necessary. The device pager uses the pager fault operation to map device memory into a faulting process' address space without the need for fictitious pages.

6.3.3 Other Minor Differences

There are two noteworthy minor differences between pagers in UVM and BSD VM:

- In BSD VM, the pager get operation can return `VM_PAGER_FAIL` if the offset of the requested page is valid but the page does not reside in the object. This is used when walking an object chain. If a pager returns `VM_PAGER_FAIL` then the fault routine knows to skip to the next object. Since UVM does not have any object chains, the `VM_PAGER_FAIL` error code is not used.

- UVM provides a generic structure (`uvm_aiodesc`) for maintaining the state of asynchronous I/O operations. This structure is allocated by a pager when the asynchronous I/O operation is initiated. When the device has finished all parts of an asynchronous I/O operation, the `uvm_aiodesc` structure for that operation is placed on a globally linked list of completed asynchronous I/O operations called `uvm.aio_done`. At this time the pagedaemon is woken up. As part of the state information for the asynchronous I/O operation, the `uvm_aiodesc` structure contains a pointer to an "asynchronous I/O done" pager operation. The pagedaemon will traverse the completed asynchronous I/O operations on the `uvm.aio_done` list calling each operation's asynchronous I/O done function. This function frees all resources being used for that asynchronous I/O operation, including the `uvm_aiodesc` structure.

  In contrast, BSD VM provides very little generic structure for asynchronous I/O operations. In BSD VM, the pager put operation is overloaded to be the asynchronous I/O done operation as well. If the pager put operation is called with a non-null pager, then the request is treated like a normal put operation. However, if the pager put operation is called with its pager argument set to null, then that is a signal to the pager to look for any completed asynchronous I/O operations to clean up.

  when the BSD VM pagedaemon first being running it calls each pager's put function with pager set to null — regardless of whether there are any asynchronous I/O operations waiting to be finished. This is inefficient because there often are no I/O operations waiting to be finished. BSD VM's overloading of the pager put function is confusing because it is not well documented and the completion of asynchronous I/O operations has little to do with starting a pageout. BSD VM pagers that do not support asynchronous I/O ignore put requests with null pagers.

It should be noted that the BSD I/O subsystem[1] currently requires that all pages that are in I/O be mapped in the kernel's virtual address space. For I/O operations that directly access memory (i.e. use DMA) this mapping is unnecessary because the kernel mappings are never used. In the future the I/O subsystem should be modified to take a list of pages rather than a kernel virtual address. That way only drivers that need to have their pages mapped in the kernel's address space will bother to do so and the extra overhead can be avoided for drivers that do not need kernel mappings.

6.4 Summary

In this chapter the design of UVM's pager interface was presented. The pager provides a set of functions that present a uniform interface to `uvm_object` memory objects. Functions in the pager interface include functions that adjust an object's reference count, and functions that transfer data between backing store and physical memory. The differences between the BSD VM pager interface and UVM's pager interface were also discussed. In BSD VM, each `vm_object` data structure is individually allocated and has its own private `vm_pager` data structure. In UVM, the `uvm_object` structure is designed to be embedded within a mappable kernel data structure such as a vnode. Rather than having a `uvm_pager` structure, the `uvm_object` contains a pointer to a shared set of pager operations.

[1]The operating systems that use UVM use the BSD I/O subsystem.

Chapter 7

Using UVM to Move Data

This chapter describes the design and function of UVM's page loan out, page transfer, and map entry passing interfaces. With page loanout, a process can safely "loan" read-only copies of its memory pages out to the kernel or other processes. Page transfer allows a process to receive pages from the kernel or other processes. Map entry passing allows processes to copy or move chunks of virtual address space between themselves. All three interfaces allow a process to move data using the virtual memory system and thus avoid costly data copies. These interfaces provide the basic building blocks on which advanced I/O and IPC subsystems can be built. In this chapter we explain the workings of UVM's page loanout, page transfer, and map entry passing. In Chapter 10 we will present performance data for our implementation of these functions.

Moving data with the virtual memory system is more efficient than copying data. To transfer a page of data from one page to another through copying, the kernel must first ensure that both the source and destination pages are properly mapped in the kernel and then copy, word by word, each piece of data in the page. To transfer a page of data using the virtual memory system the kernel need only ensure that the page is read-only and establish a second mapping of the page at the destination virtual address.[1]

However, moving data with the virtual memory system is more complicated than copying data. First, the kernel must use copy-on-write to prevent data corruption. Second, for pages that are going to be used by the kernel for I/O, the kernel must ensure that the data remains wired and is not paged out or flushed. Third, virtual memory based data moving

[1] On an i386 system there are 1024 words in a page, thus copying a page requires a loop of 1024 memory loads and 1024 memory stores. Transferring a page of data using the virtual memory system requires two memory loads and stores, one load-store pair to write protect the source virtual address and one pair to establish a mapping at the destination address. There is also some overhead associated with looking up the page and adjusting some counters.

only works for large chunks of data (close to page-sized). Copying small chunks of data is more efficient than using the virtual memory system to move the data. Thus, a hybrid approach is optimal.

7.1 Page Loanout

Under a traditional kernel, sending data from one process to another process or a device is a two-step procedure. First the data is copied from the process' address space to a kernel buffer. Then the kernel buffer is handed off to its destination (either a device driver or the IPC layer). The copy of the data from the process address space to the kernel buffer adds overhead to the I/O operation. It would be nice if I/O could be performed directly from the process' address space, but in a traditional kernel it is necessary to copy the data for the following reasons:

- The data may not be resident. As part of the data copying process the kernel faults in any non-resident pages in the process' address space before copying the data to the kernel buffer. This ensures that nonresident pages do not interfere with the I/O operation.

- A process may modify its data while the I/O operation is in progress. If the kernel did not make a copy of the data, then these modifications could effect the I/O operation. By making a copy of the data in a private buffer, the kernel ensures that changes the process may make to its memory after the I/O operation has started will not effect data that the process considers already sent.

- The data may be flushed or paged out by another process or the page daemon. The I/O system requires that data must remain resident while an I/O operation is in progress. Since kernel buffers are always resident and never touched by the page daemon, performing I/O on a kernel buffer is not a problem.

Page loanout is a feature of UVM that addresses these issues by allowing a process to safely loan out its pages to another process or the kernel. This allows the process to send data directly from its memory to a device or another process without having to copy the data to an intermediate kernel buffer. This reduces the overhead of I/O and allows the kernel to spend more time doing useful work and less time copying data.

At an abstract level, loaning a page of memory out is not overly complex. To loan out a page, UVM must make the page read-only and, increment the page's loan counter.

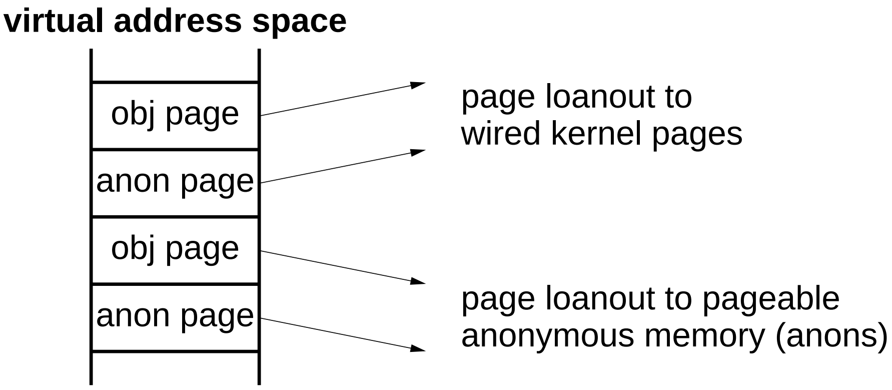


Figure 7.1: Page loanout

Then the page can safely be mapped (read-only). The complexity of page loanout arises when handling the special cases where some other process needs to make use of a page currently loaned out. In that case the loan may have to be broken.

7.1.1 Loan Types and Attributes

A page of memory loaned out by a process must come from one of UVM's two mapping layers. A process can loan its memory out to either the kernel or to another process. To loan memory to the kernel, UVM creates an array of pointers to `vm_page` structures that can be mapped into the kernel's address space. To loan memory to other processes, UVM creates an array of pointers to anon structures that can be entered into an amap that belongs to the target process. The target process can treat the loaned memory like normal anonymous memory for the most part. Page loanout is shown in Figure 7.1.

Thus, any loanout of a page of memory must fall into one of the following four loan types:

object to kernel: A page from a `uvm_object` loaned out to the kernel.

anon to kernel: A page from an anon loaned out to the kernel.

object to anon: A page from a `uvm_object` loaned out to an anon.

anon to anon: A page from an anon loaned out as an anon. Since the data is already in an anon, no special action other than incrementing the anon's reference count is required to create a loan of this type.

The procedures for performing each of these types of loans will be described in Section 7.1.5. Note that it is possible for a page from a `uvm_object` to be loaned to an anon and then later loaned from that anon to the kernel.

All types of loaned-out pages are read-only. As long as the page is loaned-out its data cannot be modified. In order to modify data in a loaned out page, the kernel must terminate the loan. This is called "breaking" the loan. Breaking a loan involves allocating a new page off the free list, copying the data from the loaned page to the new page, and then using the new page.

There are several events that can cause UVM to break a loan. These events include:

- The owner of a loaned page wants to free the page.
- A write fault on a loaned page.
- The pagedaemon wants to pageout a page currently loaned out.
- An object needs to flush a loaned page.

UVM had to be modified to handle loaned pages in each of these cases. These modifications are described later in this chapter.

Memory pages loaned to the kernel have three important properties. First, pages loaned to the kernel must be resident. If the data is on backing store then it is fetched before the loanout proceeds. Second, each page loaned to the kernel is "wired" to prevent the pagedaemon from attempting to pageout a page loaned to the kernel while the kernel is still using it. This is important because if the pagedaemon was allowed to pageout the page then the next time the kernel accessed the loaned page an unresolvable page fault would be generated (and the system would crash). Third, pages loaned to the kernel are entered into the kernel's pmap with a special function that prevents page-based pmap operations from affecting the kernel mapping. This allows UVM to perform normal VM operations on the loaned-out page without having to worry about disrupting the kernel's loaned mapping and causing an unresolvable page fault.

Memory pages loaned from an object to an anon have two important properties. First, while most pages are referenced by either a `uvm_object` or an anon, these pages are referenced by both types of memory objects. However, it should be noted that the pages are considered owned by their `uvm_object` and not the anon that they are loaned to. This means that in order to lock the fields of a page loaned from a `uvm_object` to an anon, the `uvm_object` must be locked. Second, a `uvm_object`'s page can only be loaned to one anon at a time. Future loans to an anon need only gain a reference to the anon the page is already loaned to (rather than allocating a new anon to loan to).

When an object that owns a loaned page decides to free it, the page cannot be placed on the free page list because it is still being used by the process that the page is loaned to. Rather than free the page, the page becomes ownerless. If an ownerless page was originally loaned from a `uvm_object` to an anon, then the anon is free to take over the ownership of the page the next time it holds the appropriate lock. When the loan count of an ownerless page is decremented to zero, then the page can be placed on the free list.

7.1.2 Loaning and Locking

As stated in the previous section, in order to lock a page one must lock its owner. For example, if an anon contains a page that is on loan from a `uvm_object` the object must be locked to access the `vm_page` data structure's fields. There are two problem with this. The first has to do with lock ordering. If UVM is already holding a lock on the anon, it cannot just lock the `uvm_object` since that violates lock ordering. Instead, it must "try" to lock the object, and if that fails it must drop the anon lock and then try again.

The second problem is that an anon that has a page on loan from an object does not hold a reference to that object. Thus, when attempting to lock the object it is possible for the object to terminate between the time the page's object field is read and the time the lock on the object is attempted. In order to prevent this, the page queues must be locked before trying to lock the object. Locking the page queues prevents the page's loan count from changing and thus ensures that the object cannot be terminated until the page queues are unlocked.

The option of allowing an anon that has a page on loan from an object to hold a reference to that object was considered. However this proved to be impractical in the case where the pagedaemon needs to break the loan. In that case the pagedaemon would have to drop the reference to the object, possibly triggering asynchronous I/O. This would lead to deadlock because the pagedaemon would be waiting for the asynchronous I/O operation to complete, but the services of the pagedaemon would be needed to complete the I/O. Thus, the idea of allowing an anon with a loaned page to hold a reference to the object that owned the page was rejected and instead we rely on the page queue lock to prevent an object from being freed out from under an anon with a loaned page.

7.1.3 Loanout Data Structures and Functions

In order to support page loanout, two minor changes had to be made to UVM data structures. First, a new "loan count" field was added to the vm_page structure. A page is currently loaned out if its loan count is non-zero. In order to change the loan count of a page, the kernel must be holding both the page owner lock and the page queue lock. In order to read the loan count, only one of the locks need be held.

Second, the page owner field had to be changed. Before page loanout was supported a page could belong to either a uvm_object or an anon. The owner field was a union that contained a pointer to each of these structures. Now that page loanout is supported it is possible for a page that is part of a uvm_object to be loaned out to an anon. So, the union was converted to a structure so that a page can refer to both a uvm_object and an anon at the same time.

Three major functions were added to UVM to support loanout: uvm_loan, uvm_unloananon, and uvm_unloanpage. The uvm_loan function establishes a page loanout. It takes a range of virtual addresses in a vm_map and returns either an array of page pointers or anon pointers (depending on if the loan is to the kernel or to anons). The loan function can fail if part of the address range specified is unmapped or if it was unable to get some of the pages in the specified range (e.g. due to pager errors). Once a process is finished with loaned pages, it can drop the loan by using either the uvm_unloanpage or the uvm_unloananon functions. Memory loaned to anons that is currently mapped can also be freed with the standard uvm_unmap function.

7.1.4 Anonget and Loaned Pages

An anon can point to a page loaned from a uvm_object. The uvmfault_anonget function has two features that are used to handle pages loaned from objects[2]. First, in cases where the anon does not own the page that it refers to, the anonget routine locks the object that owns the page. This means that a function that uses uvmfault_anonget must unlock the object before returning. Second, when the anonget routine detects an anon that points to an ownerless page it causes the anon to take over ownership of the page. The following is pseudo code for these parts of the anonget routine:

```
/* start with a locked anon that points to a
 * loaned page ... */
restart:
```

[2] See Section 5.1.6 for a description of uvmfault_anonget.

```
  lock(page queues);
  if (anon->page->object) {
    if (!try_lock(anon->page->object)) {
      unlock(anon, page queues);
      /* allow other processes to proceed */
      lock(anon);
      goto restart;
    }
  } else if (page is ownerless) {
    page->owner = anon;
  }
  unlock(page queues);
  /* done, continue anonget */
```

7.1.5 Loanout Procedure

In this section the procedure used to establish each of the four types of page loanout is described.

Object to Kernel Loan

To loan a page from a `uvm_object` to the kernel the following steps are taken. First the object is locked. Second, the page is fetched from the object using the object's pager get function. If the page is not resident, then the object is unlocked and an I/O is initiated. Once the page is resident the object is relocked. Third, the page queues are locked. Since both the object and the page queues are locked this allows the pages loan counter to be incremented. If the loan counter was zero, then the page must be globally write-protected to trigger a copy-on-write operation in the event that some process attempts to modify the page while the loan is established. Fourth, the page is wired (i.e. removed from the pagedaemon's queues). Finally the page queues and object can be unlocked. The pseudo code for this process is:

```
lock(object);
page = get_page(object,offset); /* make resident */
lock(page queues);
if (page->loan_count == 0)
  write_protect(page);
page->loan_count++;
wire(page);
```

```
unlock(object, page queues);
return(page);
```

Anon to Kernel Loan

To loan a page from an anon to a wired kernel page the following steps are taken. First, the anon must be locked. Second, the `uvmfault_anonget` function must be called to make the anon's page resident. Note that `uvmfault_anonget` checks for anon that have pages on loan from `uvm_objects`. In that case the anonget function will ensure that both the anon and the `uvm_object` are locked. Third, the page queues are locked and the page is write protected (if needed), the loan count is incremented, and the page is wired. Finally, the anon, `uvm_object` (if there is one), and the page queues are unlocked and the page is returned. The pseudo code for this process is:

```
lock(anon);
uvmfault_anonget(anon);  /* makes resident */
/* if anon's page has an object, it is locked */
lock(page queues);
if (anon->page->loan_count == 0)
  write_protect(anon->page);
anon->page->loan_count++;
wire(anon->page);
if (anon->page->object)
  unlock(anon->page->object);
unlock(anon, page queues);
return(anon->page);
```

Object to Anon Loan

To loan a page from an object to an anon the following steps are taken. First the object is locked and the page is looked up using the pager get function. If the page is not resident, then it is fetched. Second, UVM checks to see if the page has already been loaned to an anon. If so, it locks the anon, increments the anon's reference count, unlocks the anon and object, and returns a pointer to the anon. Third, if the page has not been loaned to an anon, then a new anon is allocated. The page queues are then locked, the page is write protected if necessary, the loan count is incremented, and a bi-direction link is established between the new anon and the page. Finally, the page queues and object are unlocked and the new anon is returned. The pseudo code for this process is:

```
lock(object);
page = get_page(object,offset); /* make resident */
if (page->anon) { /* already loaned to an anon? */
  lock(anon);
  anon->reference_count++1;
  unlock(anon,object);
  return(page);
}
allocate new anon;
point new anon at page and vice-versa;
lock(page queues);
if (page->loan_count == 0)
  write_protect(page);
page->loan_count++;
unlock(page queues,object);
return(new anon);
```

Anon to Anon Loan

Loaning a page from an anon to an anon is a simple matter since the page is already in an anon. All that needs to be done is to lock the anon, increment the anon's reference count, write protect the page (if necessary), and then unlock and return the anon.

7.1.6 Dropping and Freeing Loaned-Out Pages

When a process or the kernel is finished with loaned out pages the loan should be dropped. For pages loaned to an anon this is simply a matter of locking the anon, decrementing the anon's reference count, and unlocking the anon. If the anon's reference count drops to zero then the anon can be freed with the standard `uvm_anfree` function. This allows pages loaned to anons that are mapped in other processes to be freed with the standard `uvm_unmap` function. The `uvm_anfree` function handles pages that have been loaned to an anon. If the page in an anon has a non-zero loan count, then the anon free function first attempts to lock the true owner of the page. If the page is ownerless, then the anon will take over ownership of the page. At this point, if the page is owned by a `uvm_object` then the page queues are locked while the page's loan count is dropped and the page's pointer to the anon is zeroed. Once that is done, the page is no longer loaned to an anon and the `uvm_anfree` function need not worry about it anymore.

If the page was loaned but not owned by a `uvm_object`, then it must be loaned by the anon to the kernel. In this case, the page is passed to the `uvm_pagefree` function and then the anon is freed. If a loaned page is freed, rather than adding the page to the free page list the ownership of the page is dropped and the page becomes ownerless. This allows memory objects to free their pages without having to worry about whether the page is currently loaned out or not. Thus, this handles the case where an object's pager is asked to flush out and free the object's pages.

7.1.7 The Pagedaemon and Loaned Out Pages

The pagedaemon pages out and frees pages when free physical memory is scarce. The pagedaemon must take special care in case it encounters a loaned out page. The only type of loaned out page the pagedaemon can encounter is a page that has been loaned from a `uvm_object` to an anon. All loans to the kernel are done via wired pages, so the pagedaemon would never encounter this sort of page.

The first thing the pagedaemon will do is try and lock the page's `uvm_object`. If the lock fails, then the pagedaemon will skip to the next page. If the page is dirty then the pagedaemon can proceed as normal — it will start a pageout I/O operation. The page will already be read-only because it has been loaned out. If the page is clean, then the pagedaemon can call `uvm_pagefree` like it normally would. As described in the previous section, this will cause the object to drop ownership of the page (the anon will be free to claim ownership of the page) and the loan will be broken.

The pagedaemon can also detect ownerless pages. This can happen if an object that owned a page that has been loaned to an anon has dropped ownership of that page. If this sort of page is encountered then the pagedaemon attempts to lock the page's anon. If the attempt fails, then the pagedaemon skips to the next page. If the anon is successfully locked then the pagedaemon causes the anon to take over ownership of the page. At this point the page will no longer be loaned out and can be treated like a normal anon-owned page. It should be noted that the pagedaemon should never encounter an ownerless page that is not on loan to an anon because that can only happen with pages loaned to the kernel, and those pages are always wired.

7.1.8 Faulting on a `uvm_object` with Loaned-Out Pages

The page fault routine must also be aware of loaned pages. It must ensure two things: loaned out pages are never mapped read-write and that write faults on loaned pages cause the loans to be broken. The case of a page fault on an object (a "case 2" fault, as described in Chapter 5) is handled as follows. For copy-on-write and zero-fill faults (case 2B) no special action is required since the object's pages are never written in that case. For faults directly on the object's pages (case 2A), if the fault is a read fault on a loaned page, then the fault routine must ensure that the page is always mapped read-only regardless of the current protection on the mapping. If the fault is a write fault on a loaned page then the loan must be broken.

To break the loan the following steps are taken: the object is locked and a new page is allocated. The data from the old loaned out page is copied to the new page. Then all mappings of the old page are removed thus forcing all users of the object to refresh their mapping of the page the next time it is referenced. The old page is then removed from the object and its ownership is dropped. Then the new page is installed in the object at the old page's offset. At this point the newly allocated page has replaced the old page in the object and the fault can proceed as usual. The old page is ownerless and will be freed when the processes to which it is loaned are finished with it.

7.1.9 Faulting on an Anon With Loaned-Out Pages

Page faults on anons that point to loaned pages also require special handling by the fault routine. First, as previously described, calling `uvmfault_anonget` on an anon that points to a loaned page that has no owner will cause the anon to take over ownership of that page. Calling `uvmfault_anonget` on an anon that points to a loaned page that has an owner will lock the page's owner (if it is not that anon). If the fault is a read fault, then the fault routine must ensure that the page is entered into the faulting process' pmap read-only, even if the anon's reference count is one. If the fault is a write fault, then the fault routine must break the loan. For anon's with a reference count that is greater than one, no special action is required because the normal anon copy-on-write mechanism will handle things properly. A new anon will be created and the data copied to a new page associated with it. The original anon with the loaned-out page will remain untouched (other than having its reference count decremented).

To break the loan for a write fault on an anon with a reference count of one, the fault routine must do the following. It must first allocate a new page. If no pages are available, then the fault routine wakes the pagedaemon and waits for it to make some free pages available. Once the new page is allocated, then the fault routine must copy the data from the old page to new page. Then the fault routine removes the old page from all user pmaps to cause them to refresh their mappings on their next memory reference. It then locks the page queues, decrements the loan count on the old page and zeros the page's anon pointer. If the anon owned the page (and thus it was loaned out to the kernel), then this causes the page to become ownerless. If a `uvm_object` owned the page, then this causes the page to no longer be loaned out to an anon. At this point the fault routine can unlock the page queues and the old page's `uvm_object` (if it had one). Finally, the fault routine installs the new page into the anon and the fault can be resolved as usual. In essence, what happens is that the old loaned page is disassociated from the the anon and replaced with a freshly allocated page that is not loaned out.

7.1.10 Using Page Loanout

Page loanout can be used in a number of ways. For example, to quickly transfer a large chunk of data from one process to another the data can be loaned out to anons, and those anons can then be inserted into an amap belonging to the target process. Page loanout to the kernel can be used to improve both network and device I/O. For example, when a process wants to send data out over the network the kernel would normally copy the data from the process' memory into kernel network buffers (mbufs). However, instead of copying the process' pages, the pages could be loaned out to the kernel and then associated with mbufs, thus avoiding a costly data copy. An audio device could access pages of audio data loaned from the user process' virtual address space to the audio driver directly. This would eliminate the need to copy the data to a private kernel buffer.

Page loanout can also be used to partly replace the kernel's `physio` interface that is used for "raw" I/O to devices such as a disk. The `physio` function allows a device to read and write directly from a user process' memory without having to make a data copy. Page loaning can be used to fix several problems associated with using `physio` to write data directly from a process' address space to a device. The `physio` function works by wiring a process' memory, encapsulating that memory in a kernel buffer, starting an I/O operation, and then waiting for the I/O operation to finish. Once the I/O operation completes, the `physio` function returns. There are two problems with I/O performed using this process. First, `physio` assumes that no process other than the one performing I/O has access to the process' virtual address space, and thus the process' memory cannot change while the physio operation is in progress. While this is true for single threaded systems, it is not true for multithreaded systems where a process' address space can be shared by multiple threads. On such a system, it is possible for one thread to be performing a physio on an area of virtual memory while another thread writes to or unmaps the same area of memory. Thus physio as it is currently implemented is unsuitable for use on those systems.

The second problem with processes using physio to write data to devices is that physio only supports synchronous I/O. Process performing a physio operation must wait until the operation completes before they can continue execution. This is necessary to prevent the process from disturbing the mapping of its buffer while the I/O is in progress.

Both these problems can be addressed by modifying physio to use page loanout to loan out the pages containing the data being written to the kernel. The modified version of physio would operate as follows. First, a process would perform a write operation that invoked physio. The physio function would then use uvm_loan to loan out the process' buffers to the kernel. Second, physio would use the loaned out pages to start the asynchronous I/O operation. Third, the physio operation would return, allowing the process to continue execution. Eventually, the I/O operation would complete and the kernel would drop the loan of the process' memory. While the I/O is in progress the user's buffer will be copy-on-write, thus preventing the process from writing to the pages physio is using for I/O. Furthermore, the process can safely unmap and free its pages without disrupting the I/O operation. If the loaned out pages are freed due to an unmap, they will become ownerless loaned-out pages and will eventually be freed when the kernel is finished with them (but not before then).

7.2 Page Transfer

Under a traditional BSD kernel, when the kernel wants to move some of its data from a kernel buffer to a user process' virtual address space it uses the copyout function. This function copies the data word by word to a user buffer. If the user's buffer is not resident, then a page fault is triggered and the the buffer is paged in before the copy is started.

For large chunks of data all this copying can get expensive. Page transfer provides the kernel a way to work around this data copy by allowing the kernel to inject pages of memory into a process' virtual address space. Page transfer takes either kernel pages or anons and installs them at either a virtual address specified by the user process or if no address is specified, it installs them in an area of virtual memory reserved for page transfer.

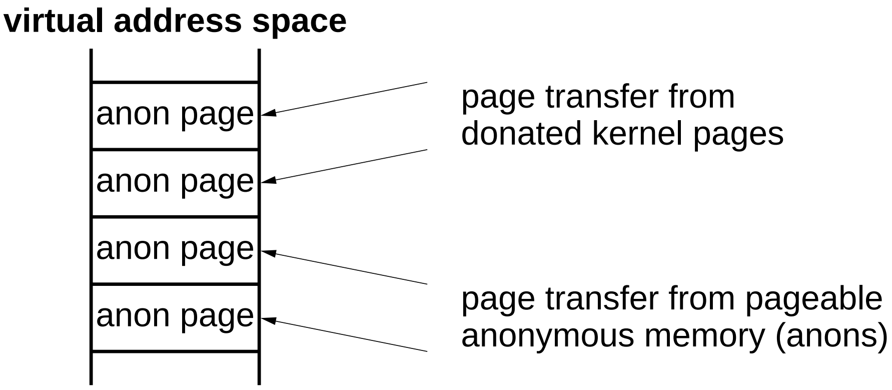


Figure 7.2: Page transfer

Once a page has been transfered into a process' address space it is treated like any other page of anonymous memory. Page transfer is shown in Figure 7.2.

There are two types of page transfer: kernel page transfer and anonymous page transfer. These two types of page transfer are described in the following sections.

7.2.1 Kernel Page Transfer

Kernel page transfer occurs when a page-sized buffer that is part of the kernel's virtual memory is transfered into a process' address space. For example, consider an audio device that is currently recording into a buffer. The audio device driver can allocate a page from UVM to act as its buffer. When the page is full the driver can make it available for page transfer. When a process issues a read on the audio device the following procedure is used. A new anon is allocated and the page with the data is attached to the new anon as its data page. Next one of two things can happen. If the user process specified a virtual address at which to put the data, then UVM ensures that that address is mapped and has an amap associated with it. If it is mapped but does not have an amap, then UVM will allocate an amap. Then the anon is installed into the amap at the appropriate address. On the other hand, if the user process did not specify a virtual address at which to place the data, then UVM chooses a free virtual address at which to place the data. The page transfer area of the virtual address space is always mapped by an amap. UVM installs the anon in this amap. At this point the data has been transfered and the user process has access to it.

The networking subsystem can also benefit from page transfer. Kernel networking buffers (mbufs) are fixed in two sizes at compile time: large and small. If the large mbufs are made page sized, then they are eligible for page transfer. The procedure is similar to what is described above, however it is more complex because mbuf pages usually start off as pages that belong to a kernel `uvm_object` allocated out of a kernel submap with kernel mappings. In this case page transfer code must remove the association between the mbuf's page and the mbuf system before the page can be transfered. This procedure is described below.

There are two possible types of large mbufs: mbufs that have normal page-sized data areas allocated out of a kernel object and mbufs that have "external" data buffers. External data buffers could be pages that are part of a device's hardware memory, or they could be pages that were loaned out from another process. Pages that are part of a device's memory must be copied because they are not managed by the VM system. Pages that were loaned out from another process can be loaned out to anons and thus transfered into the user's address space[3]. Large mbufs that have normally allocated pages can be handled by page swapping. To do this, the networking system first allocates a new page and obtains a pointer to the page that contains the data. It then removes the page that contains the data from the mbuf system, replacing it with the new page. At this point the mbuf is associated with the new page and the old page is available for page transfer.

7.2.2 Anonymous Page Transfer

Anonymous page transfer is used when the pages to be transfered into a user process' address space are already associated with anon structures. For example, this could occur when another process loans part of its memory out to anons in order for it to be used for page transfer. In this case UVM need only identify the appropriate amap structure in which to put the anons.

Anonymous page transfer can be used with page loanout (to anons) as part of an IPC mechanism. The IPC system can use these facilities to allow processes to easily exchange anon pages of data. Anonymous page transfer can also be used to improve the performance of `read` system calls on plain files if the read buffer is page aligned and page sized. Rather than copying the data from a `uvm_object`'s pages to anonymous memory, the object's pages can be loaned out to anons. Then these anons can be transfered into a process' address space. For normal files a similar effect could be achieved by using `mmap` or having the kernel establish a normal mapping underneath the `read` call. The disadvantage of doing I/O this way is that it leads to more memory mappings and thus map entry fragmentation.

[3]Thus, page loanout and page transfer can both be used at the same time.

Also, if the process writes to its buffer each mapped area will get its own amap. This requires more amap memory then anonymous page transfer and results in longer map search times. Another advantage of using page transfer is that it gives page-level granularity for mappings.

7.2.3 Disposing of Transfered Data

If a process is receiving data via page transfer, then it needs a way to release transfered pages when they are no longed needed. One way to do this is to use the standard `munmap` system call to unmap the transfered pages. The problem with this is that not only does `munmap` free the transfered pages, but it also frees the amap in which the transfered pages live. This will force the VM system to allocate a brand new amap if it wants to transfer more pages to the same area of virtual memory. To address this problem the `anflush` system call was added to UVM. This new system call removes anons from an amap without freeing the amap itself. The `anflush` system call can also be used to restore an area of anonymous memory to a zero-fill state.

7.3 Map Entry Passing

Under a traditional kernel, processes typically exchange data either using pipes or shared memory. Internal to the kernel, pipes are typically implemented as a connected pair of network sockets. When the sending process writes data on the pipe it is copied by the kernel from that process' address space into a kernel buffer (mbuf). The mbuf is placed on a queue for the receiving process. When the receiving process reads data from the pipe the data is copied from the mbuf out to the receiving process' address space. Thus, when sending data over a pipe the data is copied twice, once from the sender to a kernel buffer and once from the kernel buffer to the receiver.

In a traditional kernel shared memory between two processes can be established in three ways. First, most kernels support the System V shared memory API. This API allows a process to create a shared memory segment of specified size and protection. Once created, other processes with the appropriate permissions can attach the shared memory segment into their address space. Second, any number of processes can `mmap` a normal file `MAP_SHARED`. Changes made to the file's memory will be seen by all processes that are sharing it. Third, a process can allocate virtual memory with an inheritance code of

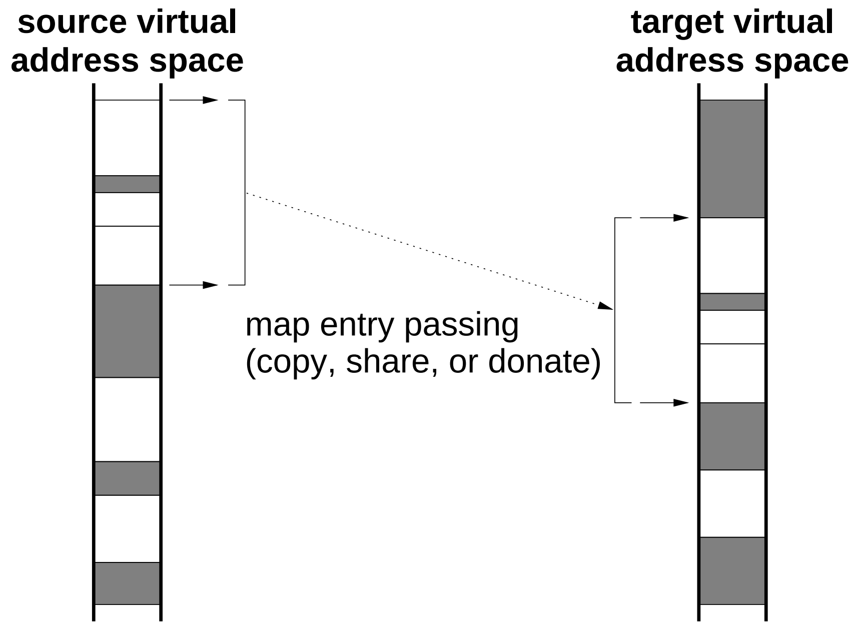


Figure 7.3: Map entry passing

"share." When that process forks a child process it will share the memory with the child process.

UVM provides a new way for processes to exchange data: map entry passing. For large data sizes, this is more efficient than pipes because data is moved without being copied. Map entry passing is also more flexible than traditional shared memory. Map entry passing does not have System V shared memory's requirement that a process pre-allocate a shared memory segment. It also does not require the services of the filesystem layer, unlike `mmap`. Additionally, map entry passing allows a range of virtual addresses to be shared. This range can include multiple memory mappings and unmapped areas of virtual memory. Map entry passing is shown in Figure 7.3.

7.3.1 Export and Import of Virtual Memory

UVM's map entry passing mechanism allows another alternative for processes to exchange data. With map entry passing a process sending data can "export" a range of its virtual address space. This allows a receiving process to "import" that memory into its address space.

To export a block of virtual memory, the sending process must create a description of the block. The `mexpimp_info` structure describes a block of virtual memory to export,

	mexpimp_info
base address	base
length	len
export type	type
export flags	flags
process ID	pid
user ID	uid
group ID	gid
mode	mode

Figure 7.4: The `mexpimp_info` structure

as shown in Figure 7.4. The base and len fields identify the block of virtual address space that is exported. Note that more than one object can be mapped in this space and it also can have unmapped gaps in it. In fact, it can even be completely unmapped. The export type can describes how the block of memory is exported. There are currently four possible values for this field (described in the next paragraph). The export flags along with the process id, user id, group id, and mode are used to control access to the exported region of memory. A process has the option of exporting memory to a specific process or establishing a file-like protection code. The export flags control which access control mechanism is used.

The four export type values are as follows:

share: causes the importing process to share the memory with the exporting process.

copy: causes the importing process to get a copy-on-write copy of the exported virtual memory.

donate: causes the memory to be removed from the exporting process' virtual address space and placed in the importing process' space. After the import the virtual space in the exporting process will be unmapped.

donate with zero: the same as "donate," except rather than leaving the exported range of virtual space unmapped after the import it is reset to be zero-fill memory.

On a successful export of memory, a "tag" structure is created for the memory. This structure is called `mexpimp_tag` and it contains a process ID and timestamp. To import virtual memory this tag must be passed in to the import system call. The tag is used to look up the exported region of memory and import it.

UVM provides two system calls for the export and import operations. Their signatures are:

```
int mexport(struct mexpimp_tag *tag, struct mexpimp_info *i)
int mimport(struct mexpimp_tag *tag, struct mexpimp_info *i)
```

Both system calls return zero on success and -1 if there is an error. For the export system call, the exporting process fills out a `mexpimp_info` structure and if successful the kernel will fill out the tag structure with the tag of the exported region of memory. For the import system call, the importing process fills out the tag structure and calls import. If the `mexpimp_info` pointer is non-null, then the info used to export that region of memory will be returned in the structure pointed to by this pointer.

7.3.2 Implementation of Map Entry Passing

Map entry passing is implemented on top of UVM's `uvm_map_extract` function. The map extraction function extracts part of the virtual memory mapped by a map and places it in another map. In addition to being used for map entry passing, the extract function is also used by the `procfs` process file system and the `ptrace` system call (used for debugging) to allow one process to access another process' virtual address space.

The `uvm_map_extract` function is called with a source map, source virtual address, length, destination map, destination address pointer, and a set of flags. The flags are:

remove: remove the mapping from the source map after it has been transfered.

zero: valid only if "remove" is set, this causes the removed area to become zero-fill after the extract operation.

contig: abort if there are unmapped gaps in the region. The contig flag is a "best effort" flag. It is possible for the extract to succeed even if there are gaps in the map in some cases.

qref: "quick" references. The qref flag is used for brief map extractions such as the ones generated by `ptrace`. With `ptrace` the following pattern is used: extract a single page from a process' map, read or write a word in that page, unmap the extracted region. Under normal VM operation this sort of operation would lead to a lot of map entry fragmentation because of the clipping used to extract the paged-sized chunks.

Quick references take advantage of the fact that the extracted mapping is going to be short lived to relax the management of references to `uvm_object` structures and amaps and avoid map entry fragmentation.

fixprot: causes the protection of the extracted area to be set to the maximum protection. This is necessary for the `ptrace` call because it needs to write break points to a process' text area. The text area is normally mapped copy-on-write with is protection set to read-only and its maximum protection set to read-write. In order for `ptrace` to be able to write to this memory, the protection must be raised from its current read-only value to its maximum value. Note that this does not effect the source map, only the destination map.

The `uvm_map_extract` operates as follows:

- A sanity check of `uvm_map_extract`'s arguments is performed. The starting address must be on a page boundary and the length must be a multiple of the page size. Also, the "remove" flag is mutually exclusive with both the "contig" and "qref" flags.
- Virtual space is reserved in the destination map for the extracted memory (done with the `uvm_map_reserve` function).
- The source map's map entry list is searched for the first map entry that contains the starting virtual address. Then a loop over the source map's virtual space is entered.
- Each source map entry in the virtual range is copied into a list of map entries that will be inserted in the destination map (if the remove flag is set, then the source map entry is removed from the source map).
- When the end of the virtual range is reached, the new map entries are inserted in the destination map.

The actual `uvm_map_extract` function is a bit more complex than this due to the need to handle quick references and data structure locking properly.

7.3.3 Usage of Map Entry Passing

Map entry passing can be used to move large chunks of data between processes without the overhead of data copying. For large data map entry passing is more efficient than pipes because the data is not copied, instead it is moved using the virtual memory system.

Map entry passing is also more flexible than memory shared with System V shared memory. With System V shared memory, the shared memory region must be allocated in advance and all shared data must be placed in it. Also, the shared memory region must be a contiguous chunk of anonymous memory. On the other hand memory shared via map entry passing can span multiple mappings, include unmapped regions, and can contain file mappings as well as anonymous memory. Any region of a process' memory can be exported on the fly, there is no need to pre-allocate a region of memory.

Map entry passing can be more efficient and flexible than memory shared with `mmap`. Regions of memory shared with `mmap` require interactions with the filesystem layer, and this can slow things down. This cost can be mitigated some by placing the memory mapped file on a RAM disk. It should be noted that using `mmap` for shared memory is still quite useful. Systems like M-NFS [43] allow memory mapped files to be shared across a network. Map entry passing cannot be used to do this (since it is not possible to pass map entries between machines on a network).

UVM's map entry passing also allows a process to grant another unrelated process access to only part of a file. In traditional Unix, if a file descriptor is passed to another process using file descriptor passing, then the receiving process gets access to the entire file at what ever permission the file descriptor allows. The only way to pass partial access of a file to another process is to open the file, `mmap` in the part of the file to be accessed, close the file, and then fork a child process that only has access to the part of the file that was memory mapped.

Using UVM's map entry passing, any process can memory map a file and then offer to send that mapping to other processes. To do this, the server process need only map in the part of the file that it wants to share and then export that mapping to the process that is to receive partial file access. The receiving process can then import the mapping of that part of the file. The receiving process will only have access to part of the file that is mapped by the exported pages. Parts of the file outside of that range will be inaccessible and will remain so as there is no way for a process to change the size of a file's mapping. This could be useful in allowing processes access to only a page-sized section of a database, for example.

Note that with some more kernel structure, it would be possible to transfer virtual address space between processes using map entry passing without the need for tags. To do this, the `uvm_map_extract` function could be modified to return a list of map entries that can be queued for a target process. Then when the target process was ready it could have this list of map entries inserted in its map.

7.4 Summary

In this chapter UVM's new data movement features were presented. These features extend the services offered by the VM well beyond what is supported in BSD VM. They can be used to build advanced I/O and IPC subsystems such as the ones presented in Section 3.4.2. Page loanout provides a way for a process to loan its memory out to other processes or the kernel. Page transfer provides a way for a process to import memory from other processes or the kernel into its address space. Both page loanout and page transfer provide page-level granularity and can be used alone or together. Map entry passing allows chunks of virtual address space to be transfered between processes (or the kernel) by using UVM's map extraction function.

Chapter 8

Secondary Design Elements

This chapter describes a number of the secondary design elements of UVM. These elements are small parts of the UVM design that differ from BSD VM and contribute to the overall function of UVM. Topics covered in this chapter include amap chunking, clustered anonymous pageout, the design of UVM's three pagers, memory mapping issues, page flags, VM startup, and pmap issues.

8.1 Amap Chunking

As explained in Chapter 2, every process running on the system has a stack area. In the BSD kernel, the size of the stack area is determined by a process' resource limits. These limits are are read and set with the `getrlimit` and `setrlimit` system calls. Resources controlled by limits include CPU time, file size, coredump size, and stack size. Each limit has two values: the current "soft" limit and the "hard" limit. A process can change the value of the soft limit to any value that does not exceed the hard limit. For the stack, typical values for the soft and hard limits are two and thirty-two megabytes, respectively. The BSD kernel reserves space in each process' map for the largest possible stack (i.e. for the hard limit). This requires two map entries. One to map the current stack zero-fill read-write, and one to map the space from the end of the current stack to the hard limit zero-fill no-access.

For a typical stack each process starts with a two megabyte read-write region for the current stack and a thirty megabyte region that is not accessible. Note that for most normal processes only a small part of the current stack gets used. Under the BSD VM system, this results in a single anonymous two megabyte `vm_object` being allocated to back the current stack. While this object is large, it is not a problem for BSD VM because the size of a `vm_object` is constant no matter how large it is. On an i386, for example, a vm_object is always 76 bytes. However, if left unchecked, this is a problem for UVM because the amount of kernel memory allocated for an amap grows with the size of the amap. This is due to the amap containing three arrays whose length is determined by the number of pages the amap maps. On a system with a 4K page size a two megabyte amap consumes a little over six thousand bytes of kernel memory[1]. Allocations of that size can quickly fill up the kernel's virtual memory space reserved for its memory allocator.

To address this problem, UVM was modified to break up very large amaps into smaller amaps when possible. This is called amap "chunking." Currently UVM uses a chunk size of 64KB. Allocating an amap that maps 64KB requires about two hundred bytes of kernel memory. The amap_copy function takes a boolean parameter that indicates if chunking is allowed or not.

As an example of the benefits of amap chunking, consider a newly created process that has not accessed its stack yet. Its stack region will be mapped by two zero-fill map entries both with needs-copy set. The first map entry reserves thirty megabytes of virtual memory in case the stack's resource limits are changed. The second map entry maps the current two-megabyte stack. When the process first accesses its stack, a page fault will be generated. Without amap chunking, the uvm_fault routine will look up the second map entry and call amap_copy to clear needs-copy. This will cause a two megabyte amap to be allocated for the second map entry and consume over six thousand bytes of kernel memory. But with amap chunking enabled, uvm_fault calls amap_copy with chunking allowed. As a result, amap_copy breaks the second map entry up into two chunks. The first chunk will map 1984KB of zero-fill stack memory with needs-copy set. The second chunk will map the faulting address and be attached to a freshly allocated 64KB amap. By chunking the amap allocation, UVM typically saves a little less than 6000 bytes of allocated kernel memory per process. UVM still uses more memory than BSD VM for this procedure because of its use of amaps. However, amaps provide per-page granularity that is useful for page loanout and page transfer and help eliminate the object collapse and swap memory leak problems found in BSD VM.

8.2 Clustered Anonymous Memory Pageout

When BSD is running low on free memory, the pagedaemon process is invoked. The pagedaemon's job is to scan the page queues in search of resident pages of memory that

[1]A two megabyte region contains 512 pages. Assuming a 32 bit machine, the size of each entry in an array is four bytes. Thus three 512 entry arrays would take 6144 bytes.

have not been used recently. When such a page is found the pagedaemon attempts to pageout the page to backing store. After a page is successfully paged out it can be reused as its data is safely non-resident in backing store. Pages that contain file data are paged out to their backing file, while anonymous memory pages are paged out to the system's swap area.

There are two types of pages that the pagedaemon can encounter: clean pages and dirty pages. Clean pages are pages that have never been modified since being loaded from backing store. As the data in a clean page is identical to the data currently on backing store, to pageout a clean page all the pagedaemon has to do is free it. On the other hand, dirty pages are pages that have been modified since they were loaded from backing store. To pageout a dirty page, the pagedaemon must first clean the page by invoking an I/O operation to write the page's data to backing store. Then the pagedaemon can free the page.

When writing data out to backing store, it is more efficient for the kernel to transfer multiple contiguous pages at the same time because it involves less overhead (e.g. I/O setup and interrupts). Paging out multiple contiguous pages at the same time is called clustered pageout. For example consider a system with a 4096 byte page size (4096 is 0x1000 in hexadecimal). Assume that there is a file that has three dirty pages at byte offsets 0x4000, 0x5000, and 0x6000. Suppose memory is scarce and the pagedaemon is running. Without clustered pageout, each of these pages will be individually paged out in their own I/O transaction. However, if clustered pageout is available then when the pagedaemon encounters the page at offset 0x5000 it will not immediately page it out. Instead it will search for dirty pages preceding and following the page at offset 0x5000. This will cause it to find the pages at 0x4000 and 0x6000. The pagedaemon will then cluster these two pages with the original page at 0x5000 to form a three page cluster. Thus, all three pages will be cleaned in a single I/O transaction. Note that if the offsets of the dirty pages were changed to 0x3000, 0x5000, and 0x7000 then the pagedaemon would not be able to cluster the pages together because they are no longer contiguous.

In additional to file memory, the pagedaemon can also cluster anonymous memory for pageout to the swap area. In the BSD VM system, all pageable anonymous memory must be part of a `vm_object` whose pager is the "swap pager." The swap pager operates by taking the object and dividing it into fixed-sized swap blocks. Each swap block is one or more pages in length (larger objects have larger swap blocks). The first time a page in a swap block is paged out to swap the swap block is assigned a contiguous location in the swap area. This assignment is static, once an anonymous page has been assigned a swap location on backing store it is always paged out to that location.

One unique property of anonymous memory is that it is completely under the control of the VM system and it has no permanent home on backing store. UVM takes advantage of this property to more aggressively cluster anonymous memory than is possible with the scheme used by BSD VM. The key to this aggressive clustering is that UVM's pagedaemon can reassign an anonymous page's pageout location on backing store. This allows UVM's pagedaemon to collect enough dirty anonymous pages to form a large cluster for pageout. Each page's location on swap is assigned (or reassigned) so that the cluster occupies a contiguous chunk of swap and can be paged out in a single large I/O operation. So, if UVM's pagedaemon detects dirty pages at offsets 0x3000, 0x5000, and 0x7000 in an anonymous object it can still group these pages in a single cluster, while the BSD VM would end up performing three separate I/O operations to pageout the same pages.

The result of this is that UVM can recover from page shortages quicker and more efficiently than BSD VM.

8.3 The Anonymous Memory Object Pager

Anonymous memory can be found at either level of UVM's two-level memory mapping scheme. At the amap layer, anonymous memory is associated with anon structures, as explained in Chapter 4. At the object layer UVM supports a `uvm_object` that is backed by anonymous memory. These sorts of objects are managed by the "aobj" pager and thus are called aobj-objects. In this section the design of the aobj pager is briefly presented.

8.3.1 The `uvm_aobj` Structure

The state information for an aobj-object is stored in the `uvm_aobj` structure. The main role of the `uvm_aobj` structure is to keep track of the size of the object and where its paged out data resides on backing store. This structure contains the `uvm_object`, structure, the size of the object (in pages), some flags, an array or hash table that maps the offset in the object to its location on swap, and pointers to maintain a linked list of all active aobj-objects on the system.

When an aobj-object is created, the aobj pager examines the size of the object. If the object contains a relatively small number of pages, then the pager allocates an array that has an entry for each page in the object. Each entry contains the location on swap where the page's non-resident data resides. Pages that have never been paged out have their entry set to zero (an invalid swap location). For large objects, such an array could consume a lot of kernel virtual memory, so if the object is larger than a certain number of pages the aobj pager allocates a hash table to map page number to swap location. This allows small objects to receive the benefit of fast array-based lookups while larger objects using a hash table do not put a large burden on the kernel memory allocator.

8.3.2 The Aobj Pager Functions

The aobj pager's pager functions of interest are the attach and I/O functions. The aobj pager's attach function is `uao_create`. The prototype for this function is:

```
struct uvm_object *uao_create(vm_size_t size, int flags)
```

This function creates a new aobj-based `uvm_object` of the specified size. The "flags" argument is used during system startup to create a special kernel object (see Section 8.8), otherwise it is zero. Once created, an aobj-object will remain allocated until its reference count drops to zero. Once that happens the aobj-object and all the resources it is holding will be released.

Due to UVM's aggressive clustering of anonymous memory for pageout all writes to the swap area are handled specially by the pagedaemon. As a result of this, the aobj pager's "put" function never gets called (and thus can be set to null). The aobj's pager "get" function is a simple front end that calls out to functions that handle I/O to the swap area. These functions are used to access both amap and aobj based anonymous memory.

8.4 The Device Pager

The device pager allows device memory to be mapped by user processes. The most common devices that allow their memory to be mapped are graphical framebuffers. The device pager's attach function takes a device specifier and returns a pointer to the `uvm_object` for that device. Internally, the device pager maintains a linked list of all active device-objects on the system. When the attach function is called, the list is first searched to see if the device already has a `uvm_object`. If so, that object's reference counter is incremented and a pointer to that object is returned. If not, then a new `uvm_object` is allocated, initialized, and returned.

The device pager is unique in that it performs no I/O. Rather than having a pager "get" function, the device pager has a pager "fault" function. When a process faults on device memory, the uvm_fault routine transfers control of the handling of the page fault to the device pager's fault routine. The fault routine consults the device driver's d_mmap function to determine the correct device page to map in and then it calls pmap_enter to enter the mapping and resolve the fault. The device pager also has a null "put" function.

Note that this is different from the BSD VM device pager. The BSD VM device pager is stuck with the BSD VM pager interface that requires it to provide a pager "get" function that returns a pointer to vm_page structure. Since device memory does not have vm_page structure, the BSD VM device pager must allocate a "fictitious" vm_page structure and return it. The rest of the BSD VM system must be careful to avoid placing a fictitious page in the general page pool (e.g. by paging it out) because its memory is device memory and cannot be used as general purpose memory. UVM's pager fault operation eliminates the need for messy fictitious pages.

8.5 The Vnode Pager

The vnode pager manages uvm_object structures associated with files. The vnode pager is heavily used by the BSD kernel as all programs are memory mapped through it. The design of UVM's vnode pager is presented in this section.

8.5.1 BSD VM Vnode Management

UVM and BSD VM handle the VM related vnode data structures quite differently. This is due to changes to the object cache and the embedding of the uvm_object. Under BSD VM, to memory map a file the following steps are taken by the vm_mmap function:

- The vm_pager_allocate function is called with the vnode as an argument. This function calls the BSD VM vnode pager's allocate function.

- The BSD VM vnode pager's allocate function checks the vnode to see if there is already a vm_pager structure associated with that vnode.

  – If there is already a pager associated with the vnode, then the allocate function lookups up the vm_object associated with that pager, gains a reference to it, and returns a pointer to the pager.

  – If there is no pager associated with the vnode, then the allocate function mallocs a new vm_pager, vn_pager, and vm_object structure for the vnode and ties them all together. The allocate function then gains a reference to the backing vnode using the VREF macro and returns a pointer to the freshly allocated pager.

- The pager is then used by vm_mmap to lookup the vnode's vm_object (again).
  - For shared mappings, the vm_mmap function calls the vm_allocate_with_pager function to enter the object in the map structure.
  - For copy-on-write mappings, a rather complex set of function calls are performed to enter the object in the map structure with the correct object chain configuration for copy-on-write[2].

It should be noted that in BSD VM the vnode's vm_object maintains an active reference to the vnode. The reference is gained with VREF when the vm_object is allocated and it is dropped with vrele when the object is freed. The vm_object has its own reference count that is used to count how many VM data structures are using it.

The BSD VM system also has a data structure called the "object cache." The purpose of the object cache is to allow vm_object structures that are not currently being used to remain active for a time after their last reference is dropped. Objects in the object cache are said to be "persisting." This is useful for vm_object structures that map files that are repeatedly used for short periods of time. For example, it is useful for the /bin/ls object to remain active even when the program is not being run because it is used frequently. Each object on the system has a flag that says whether it is allowed to persist or not. When a vm_object structure's reference count drops to zero, if its persist flag is true it is added to the object cache, otherwise it is freed.

The number of objects allowed in BSD VM's object cache is statically limited to 100 objects (vm_cache_max). If this limit is reached, an older object from the cache may be freed to make room for a new one. If a process memory maps an object that is currently in the cache then it is removed from the cache when the vm_object's reference count is incremented to one.

The main problem with the object cache is that persisting unreferenced vnode objects that are in the object cache are still holding a reference to their backing vnode. BSD's vnode layer has a similar caching mechanism to the object cache, and the fact that vnodes

[2]Describing the details of this code is beyond the scope of this section — see vm_mmap.c for details.

persisting in the VM system have an active reference prevents the vnodes from being recycled off of the vnode cache. The net result is that in BSD VM there are two layers of persistence caching code and the VM object cache interferes with the proper operation of the vnode cache. Thus, useful persisting virtual memory information may be thrown away prematurely, and the selection of a vnode to recycle may be less than optimal because vnodes with virtual memory data in the object cache cannot be recycled even though they are no longer being used.

8.5.2 UVM Vnode Management

UVM replaces BSD VM's vnode management structure with a new one created for it. In BSD VM the object cache is part of the main VM code. In UVM the object cache has been eliminated. The persistence of `uvm_object` data structures is handled by their pagers. The aobj and device pagers have no need for persisting objects.

UVM's vnode pager supports persisting objects using the existing vnode cache to control persistence. This eliminates one layer of object caching and prevents the situation of having two caching layers at odds with each other. In UVM all VM related data structures are embedded within the vnode structure. The vnode has a flag that indicates whether the VM data structures are currently in use. The vnode layer clears this flag when a vnode is first allocated. When a file is memory mapped, UVM's `uvm_mmap` function performs the following actions:

- The vnode's attach function is called with the vnode as argument. The attach function returns a pointer to the `uvm_object` associated with that vnode.
  - If the `uvm_object` is not currently in use, then the attach function will get the size of the vnode, initialize the `uvm_object` structure, gain a reference to the vnode with `VREF`, and then return a pointer to the `uvm_object` structure.
  - If the `uvm_object` is already active then the attach function will check the object's reference count. If it is zero then the object is currently persisting. In that case the reference count is incremented to one and the attach function gains a reference to the backing vnode with `VREF`. If the object is not persisting, then the only action required is to increment the reference count. A pointer to the `uvm_object` structure is returned.
- The `uvm_map` function is called to map that object in with the appropriate attributes.

When the final reference to a vnode `uvm_object` is dropped the reference to the underlying vnode is dropped (with `vrele`). This causes the vnode to enter the vnode cache. If the vnode's flag is set to allow the `uvm_object` to persist, then the persisting flag is set and processing is complete. If the object is not allowed to persist, then all of its pages are freed and it is marked as being invalid.

In order for UVM's scheme to work, UVM must be notified when a vnode is being removed from the vnode cache for recycling. The vnode pager provides a function `uvm_vnp_terminate` for this. When the vnode layer of the operating system wants to recycle an unreferenced vnode off the vnode cache it calls `uvm_vnp_terminate` on the vnode. This function checks to see if the vnode's `uvm_object` is persisting. If so, it releases all the `uvm_object`'s resources and marks it invalid. The `uvm_vnp_terminate` function is called from the `vclean` function in `vfs_subr.c`.

8.5.3 Vnode Helper Functions

The vnode pager has several helper function that are called by other parts of the kernel to perform actions on a vnode's VM data. These functions are the setsize function, the umount function, the uncache function, the terminate function and the sync function. These functions are named differently depending on which VM system is being used. For example, the setsize function is called `vnode_pager_setsize` in BSD VM, and it is called `uvm_vnp_setsize` in UVM. The role of these helper functions is described in this section.

When the I/O system changes the size of a file the **setsize** function is called so that the VM system can update its copy of the size.

The **umount** function is called when a filesystem is being unmounted. Before a filesystem can successfully be unmounted all references to the files on the filesystem must be removed. As part of this process, the BSD kernel recycles all vnodes associated with the filesystem. In the BSD VM system this presents a problem because if an unreferenced vnode has an unreferenced `vm_object` persisting in the object cache the vnode recycle operation will fail. The umount function addresses this problem. When unmounting, before recycling the filesystem's vnodes the BSD kernel calls the umount helper function. The umount function goes down the list of vnodes associated with the filesystem and ensures that any `vm_object` associated with the vnode cannot persist in the object cache. The umount helper function is not needed in UVM because there is no object cache. When the unmount operation recycles the vnodes in the vnode cache, the `uvm_vnp_terminate` function will be called to clean out any persisting VM data from the vnode.

The **uncache** function is called when the BSD kernel wants to ensure that a vnode's VM data will not persist when the last reference to it is dropped. This function simply clears the object's "can persist" flag. This function is called when ever a file is written, renamed, or removed to ensure that the VM and file data are in sync when the file is closed[3].

The UVM-specific **terminate** function is used by the vnode cache code to clean out a vnode's VM data when the vnode is being recycled. The BSD VM system does not have a terminate helper function.

The **sync** helper function was added to BSD VM to allow modified data in vnode objects to be periodically flushed out to backing store. This function is called as part of the sync system call. Before the sync helper function was added to NetBSD, modified VM data was only flushed to backing store when the object containing it was freed or the `msync` system call was used. Thus, changes made to a file via a writable memory mapping by a program that kept the file open for a long period of time were apt to be lost if the system crashed before the program closed and unmapped the file. The sync function addresses this problem. It works by building a list of all active vnode objects and flushing all the pages in them to backing store. UVM's vnode pager also includes the sync function, but it improves the design. UVM's vnode pager's attach function is used to gain access to a vnode's `uvm_object`. Its prototype is:

```
struct uvm_object *uvn_attach(struct vnode *vn,
                              vm_prot_t accessprot)
```

The second argument of `uvn_attach` is the protection access level that the caller desires. For example, if a file is to be memory mapped read-only or copy-on-write, then the access protection would be "read" to indicate that the attacher will not modify the object's pages. On the other hand, if the attacher is going to modify the object, then it specifies an access protection of "write." The vnode pager uses this information to maintain a list of vnode objects that are currently writable. Then when the vnode sync helper function is called the vnode pager only considers vnode objects on the writable list. Since most vnode objects are mapped read-only, this reduces the overhead of the sync operation because rather than traversing every page of every active vnode object to see if it needs to be cleaned only the writable vnodes need to be checked.

[3]This function is necessary in both BSD VM and UVM because the VM cache and buffer cache have not yet been merged in NetBSD.

Table 8.1: The eight BSD VM memory mapping functions. The vm_map_find, vm_allocate, vm_allocate_with_pager, and vm_mmap functions map memory at either a specified virtual address or they use vm_map_findspace to find an available virtual address for the mapping.

Function	Usage
vm_map_lookup_entry	looks for the map entry that maps the specified address; if the address is not found, then return the map entry that maps the area preceding the specified address
vm_map_entry_link	links a new map entry into a map
vm_map_findspace	finds unallocated space in a map
vm_map_insert	inserts a mapping of an object in a map at the specified address
vm_map_find	inserts a mapping of an object in a map
vm_allocate	allocates zero fill in a map
vm_allocate_with_pager	maps a pager's object into a map; the new mapping is shared (rather than copy-on-write)
vm_mmap	maps a file or device into a map; the vm_mmap function handles the special object chain adjustments required to map an object in copy-on-write

8.6 Memory Mapping Functions

In the BSD kernel memory there are four system calls that map memory into a process' address space:

exec: maps a newly executed program's text, data, and bss areas into the process' address space.

mmap: maps a file or anonymous memory into a process' address space.

obreak: grows a process' heap by mapping zero-fill memory at the end of the heap.

shmat: maps a System V shared memory segment into a process' address space.

The BSD VM system provides eight functions that are used to establish mappings for these four system calls. These eight functions are shown in Table 8.1. The call path for these functions is complex, as shown in Figure 8.1.

On the other hand, UVM provides five functions for establishing memory mappings. These functions are shown in Table 8.2. Unlike BSD VM, UVM has one main mapping

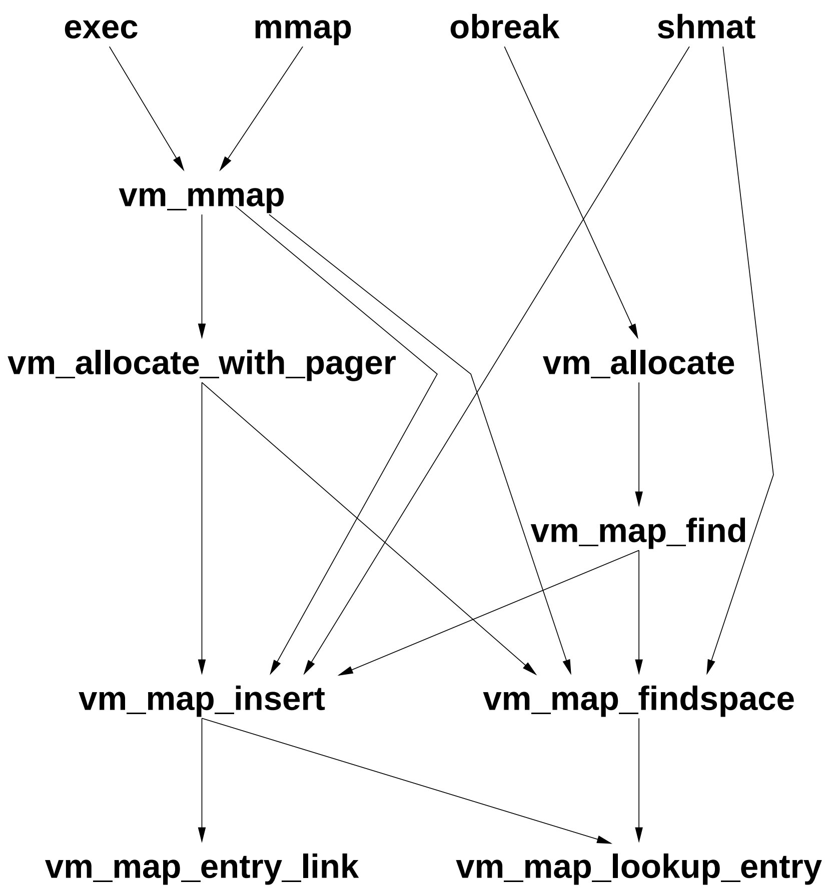


Figure 8.1: The BSD VM call paths for the four mapping system calls

function, `uvm_map`. All mapping operations use this function to establish their mappings. This makes UVM's code cleaner and easier to understand than the BSD VM code. UVM's mapping call path is shown in Figure 8.2.

The prototype for the `uvm_map` function is:

```
int uvm_map(vm_map_t map, vm_offset_t *startp,
        vm_size_t size, struct uvm_object *uobj,
        vm_offset_t uoffset, uvm_flag_t flags)
```

A call to `uvm_map` maps `size` bytes of object `uobj` into map `map`. The first page of the mapping maps to offset `uoffset` in the object being mapped. The `startp` argument contains a hint for the address where the mapping should be established. Once the mapping is established, the actual address used for the mapping is returned in `startp`. The flags, described below, are used to control the attributes of the mapping. The `uvm_map` function

Table 8.2: The five UVM memory mapping functions

Function	Usage
`uvm_map_lookup_entry`	looks for the map entry that maps the specified address; if the address is not found, returns the map entry that maps the area preceding the specified address
`uvm_map_entry_link`	links a new map entry into a map
`uvm_map_findspace`	finds space in a map and inserts a mapping of the specified object
`uvm_map`	establishes a mapping
`uvm_mmap`	attaches to a file's or device's `uvm_object` and maps it in

returns `KERN_SUCCESS` if the mapping was successfully established, otherwise it returns an error code.

The `uvm_map` function's `flags` are a bitwise encoding of the desired attributes of the mapping. This encoding is usually created with a call to the `UVM_MAPFLAG` macro. This macro takes the following arguments:

protection: the initial protection of the mapping. This protection must not exceed the maximum protection.

maximum protection: the maximum allowed protection of the mapping.

inheritance: the initial inheritance value for the mapping.

advice: the initial usage advice for the mapping.

mapping flags: further mapping flags. These flags are:

- **fixed:** the mapping must be established at the address specified in `startp`.
- **overlay:** allocate an amap for the mapping now.
- **nomerge:** do not try and merge this mapping in with its neighbors.
- **copyonw:** set the copy-on-write flag for this mapping.
- **amappad:** add some padding to the amap so that it can be grown without reallocating its memory (valid only if "overlay" is true).

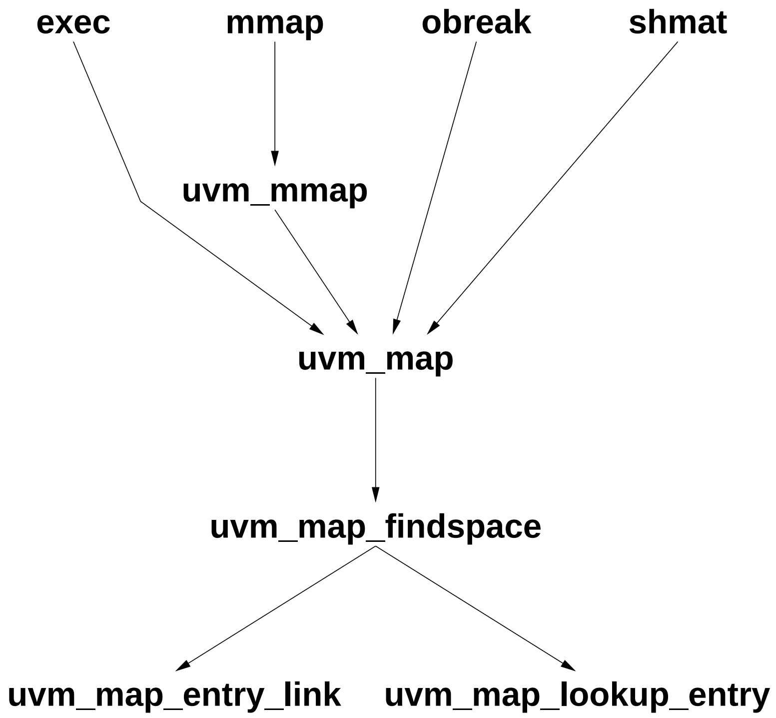


Figure 8.2: UVM's mapping call path

trylock: try and lock the map. If the map is already locked, return an error code rather than sleep waiting for the map to be unlocked.

Note that under UVM the `uvm_map` function allows all the mapping attributes to be specified at map time. This cannot be done under BSD VM. BSD VM's mapping functions always establish a mapping with a default protection and inheritance. For example, the default protection of a mapping under BSD VM is read-write. Thus, if the kernel wants to establish a read-only mapping of a file using the BSD VM system it must first use the mapping functions to establish a read-write mapping. It then must use the `vm_map_protect` function to write-protect the mapping. This is both wasteful and dangerous. It is wasteful because the mapping function locks the map, establishes the mapping with the default protection, and then unlocks the map. Then the `vm_map_protect` function must relock the map, repeat the map lookup, change the protection, and then unlock the map. Thus, under BSD VM the map must be locked and unlocked twice, and there is an extra lookup and protection change. This is also dangerous because when establishing a memory mapping there is a brief window of time between establishing the mapping and write protecting the mapping where the map is unlocked and the object is mapped read-write. On a multithreaded system this could allow a malicious process to race the kernel and successfully modify a read-only file before the kernel has a chance to write-protect the mapping. UVM's `uvm_map` function avoids both these problems by allowing the attributes of the mapping to be specified at the time the mapping is created.

If a caller to `uvm_map` does not wish to map an object in at the object layer of the mapping, it can call `uvm_map` with `uobj` set to null. Thus, a zero-fill mapping can be created by setting `uobj` to null and setting the "copyonw" flag.

Note that both BSD VM and UVM support the BSD "pmap prefer" interface for systems with virtually addressed caches (VACs) or MMUs that do not allow mappings in certain virtual address ranges (e.g. Sun sparc systems). This interface is used by the mapping functions to push a mapping address forward so that it does not create VAC cache aliases or place a mapping at an invalid address.

8.7 Unmapping Memory

In the BSD VM system, memory is unmapped with the `vm_map_delete` function. This function operates by locking the map, searching for map entries to remove, and then unlocking the map. When a map entry that needs to be removed is found, the address range that the map entry maps is purged from the pmap and the reference to the mapped object is dropped. While this works correctly, it is not optimal for the following reason. Dropping the reference to an object usually means decrementing a reference counter. This usually has a low overhead, however, if the reference counter drops to zero, then the pager may wish to free the mapped object. If the mapped object contains dirty pages, then they will need to be flushed before that object can be freed. This means that an I/O operation will have to be initiated. When the I/O operation is started the map that the object was mapped in is still locked by the unmap function. Neither the kernel nor any other process can access the map while the unmap function has it locked. Hopefully the pager will perform an asynchronous I/O operation so that the unmap operation can complete without having to wait for the I/O to complete. However, since pagers are not required to provide asynchronous I/O this may not be case.

In UVM this issue has been addressed by breaking an unmap operation into two parts. The `uvm_unmap` function first locks the map and calls `uvm_unmap_remove` to clean out the map's pmap and remove the specified map entries. Note that `uvm_unmap_remove` does not drop references to any mapped objects. Instead it returns the list of map entries removed from the map. At this point `uvm_unmap` unlocks the map, allowing the kernel and other processes access to it. Then `uvm_unmap` calls the uvm_unmap_detach function. This function goes down the list of map entries and drops the reference to any amaps or uvm_object structures that are pointed to. It then frees the map entry. Once uvm_unmap_detach returns the unmap operation is complete. Unmap operations under UVM do not leave the map locked during pager I/O.

8.8 Kernel Memory Management

The kernel's virtual address space contains the kernel's text, data, bss, and mappings. The kernel's memory is mapped by the kernel_map map. While the kernel can use uvm_map to map files and devices into its address space, the kernel also needs to be able to map private memory into its address space to support the kernel memory allocator (malloc) and other kernel subsystems. In both BSD VM and UVM there are a special set of functions used to establish and remove private kernel mappings. These functions are called kernel memory management functions. UVM uses the same API as BSD VM for the kernel memory management functions, although the functions have been renamed to be consistent with UVM coding style. In this section these functions and their usage by the kernel is described.

8.8.1 Kernel Memory Management Functions

UVM provides the following functions for the management of private kernel memory:

uvm_km_alloc: The alloc function allocates wired memory in the kernel's virtual address space. The allocated memory is uninitialized. If there is not enough physical memory available to satisfy the allocation, then the allocating process will wake the pagedaemon and sleep until memory is available. If there is not enough virtual space in the map this function will fail. BSD VM does not have a function that fills this role (but see uvm_km_zalloc below).

uvm_km_zalloc: This function operates in the same way as uvm_km_alloc except the allocated memory is zeroed by the memory allocator. This function is known as kmem_alloc in BSD VM.

uvm_km_valloc: The valloc function allocates zero-fill virtual memory in the kernel address space. No physical pages are allocated for the memory. As the memory is accessed the fault routine will map in zeroed pages to resolve the faults. If there is no free virtual space in the map this function will fail. This function is known as `kmem_alloc_pageable` in BSD VM.

`uvm_km_valloc_wait`: This function operates in the same way as `uvm_km_valloc` except that if the map is out of virtual space then the caller will sleep until space is available. This function is known as `kmem_alloc_wait` in BSD VM.

`uvm_km_kmemalloc`: This function is the low-level memory allocator for the kernel malloc. It allocates wired memory in the specified map. If there is not enough physical memory available to satisfy the allocation, then the allocating process will wake the pagedaemon and sleep until memory is available unless a “cannot wait” flag is set. If there is not enough virtual space for the allocation then this function will fail. This function is known as `kmem_malloc` in BSD VM.

`uvm_km_free`: This function is a simple front end for the standard unmap function. It is known as `kmem_free` in BSD VM.

`uvm_km_free_wakeup`: This function is the same as `uvm_km_free` except that after freeing the memory it wakes up any processes that are waiting for space in the map. It is known as `kmem_free_wakeup` in BSD VM.

8.8.2 Kernel Memory Objects

Most private kernel memory resides in large objects called kernel memory objects. The main kernel memory object is `kernel_object`. This object spans the entire virtual address space of the kernel. Thus, each virtual address in the kernel’s address space has a corresponding location in the kernel object. For example, on the i386, the kernel address space currently starts at 0xf0000000 and ends at 0xffffffff. The kernel virtual address 0xf0000000 is assigned to offset 0 in the kernel object. Likewise, kernel virtual address 0xf0004000 is assigned to offset 0x4000.

Although the kernel object is quite large, only a small part of that space is in use at any one time. The unused parts of the kernel object consumes neither virtual or physical memory. The only time that the kernel object comes into play is when it is mapped. For example, in UVM, the `uvm_km_alloc` function operates by finding a free virtual address and mapping the corresponding part of the kernel object into that virtual address.

There are two main advantages of having large kernel objects. The first is that all the kernel’s private pages are collected into a few objects. This can ease debugging kernel memory problems by reducing the number of data structures. The second advantage of large kernel objects is that it reduces map entry fragmentation by allowing map entries to be merged together. For example, if the kernel maps virtual address 0xf0008000 in and then later maps in 0xf0009000 (the following page assuming a 4096 byte page size), these two one-page mappings of the kernel object can be merged into a single two-page mapping of the kernel object at address 0xf0008000. If each mapping were in different objects this would not be possible because a single map entry cannot point to more than one object. Reducing map entry fragmentation reduces kernel memory use and lowers map search time.

As stated earlier, most of the kernel memory allocations are done through the `kernel_object` structure. However, some of the kernel's submaps have their own private kernel memory objects. For example, the `kmem_map` is used to allocate memory for the kernel memory allocator. It has its own private kernel object `kmem_object` that contains its memory. This is necessary because the `kmem_map` needs to be accessible during an interrupt and the more general `kernel_map` is not properly protected for access at interrupt time.

8.8.3 Comparison of BSD VM and UVM Kernel Memory Management

While both BSD VM and UVM use the same basic API for the allocation of private kernel memory, there are a number of implementation differences between the two systems that makes UVM's kernel memory management more efficient than BSD VM's.

Once difference between UVM and BSD VM is the way the kernel map is accessed to establish a private kernel memory mapping of a kernel object. The unusual part of such a mapping is that the offset within the kernel object depends on the virtual address chosen for the mapping and that address is not known by the function that wishes to establish the mapping. In BSD VM, one would normally use a function such as `vm_allocate` that locks the map, establishes the mapping, and then unlocks the map and returns. However, since the offset in the kernel object is not known in advance, this is not possible. Instead, the kernel memory allocation functions must lock the map, use a two-step process to establish the mapping, and then unlock the map. The first step is to find space in the kernel's map for the mapping. At this point the virtual address to be used for the mapping is known and the offset in the kernel object can be computed. The second step is to insert a mapping from the virtual address allocated in the first step to the kernel object at the correct offset. Both steps require address lookups in the target map. The second lookup is redundant because it is recomputing information already computed in the first lookup. Fortunately, the VM system caches the result of the last lookup on the map so the second lookup will not be as expensive.

In UVM such mappings are established with a single call to `uvm_map`. The calling function specifies which kernel map and which kernel object to use for the mapping. Rather than specifying an offset in the object, the calling function uses the special value `UVM_UNKNOWN_OFFSET`. By specifying an unknown offset, the calling function tells the `uvm_map` function that it needs to compute the true offset of the mapping once the virtual address has been determined. This is done by subtracting the starting virtual address of kernel space from the newly allocated virtual address. Thus, only a single mapping call and a single map lookup are needed to establish the mapping under UVM. If the function calling `uvm_map` also needs to use the offset, it can simply compute it by subtracting off the kernel's starting virtual address from the address returned by the `uvm_map` function.

UVM and BSD VM allocate pageable kernel virtual memory differently as well. Pageable kernel virtual memory is virtual memory that is mapped zero-fill but no physical memory is assigned to it until the first page fault on it. BSD VM allocates such memory by mapping in a null object copy-on-write. This causes the page fault routine to allocate a new anonymous `vm_object` to contain the memory for this area when the first page fault occurs. While this simplifies the memory mapping procedure (since there are no object offsets to worry about) it also nullifies the benefits of having a single large kernel object and can lead to map entry fragmentation and the allocation of multiple smaller anonymous kernel objects. UVM, on the other hand, uses the usual "unknown offset" mechanism to establish pageable kernel mappings, thus allowing it to use a kernel object for these mappings.

Another difference between UVM and BSD VM in kernel memory mapping is that BSD VM uses the `vm_page_zero_fill` function to zero any wired memory that it allocates. There are two problems this. First, the page zero function is a very simple function that calls out to the `pmap_zero_page` function:

```
boolean_t vm_page_zero_fill(struct vm_page *m)
{
  m->flags &= ~PG_CLEAN;
  pmap_zero_page(VM_PAGE_TO_PHYS(m));
  return(TRUE);
}
```

Since this function does not actually do anything other than clear the clean bit and call a pmap function it is unnecessary. The second problem with using the zero page function is that it assumes that the page being zeroed is not currently mapped in the kernel and will not be mapped in the kernel any time in the future. Thus, it operates by temporarily mapping the page in, zeroing it, and then unmapping the page. The resulting sequence of operations under BSD VM are allocate a page, temporarily map the page into the kernel's address space, zero the page, unmap the page (flushing the TLB and cache as necessary), and finally map the page into the kernel at its final address. Since all private kernel memory is mapped in the kernel's address space the extra overhead of establishing and removing the the temporary mapping of the page is complete unnecessary. In UVM wired kernel pages are mapping in and zeroed at their target address, thus avoiding this extra layer of mapping.

One other difference between UVM and BSD VM's kernel memory mapping is in the operation of the low-level kernel memory allocator (`uvm_km_kmemalloc` and `kmem_malloc`). In BSD VM, this function operates as follows. First BSD VM's two step mapping procedure is used to determine what virtual address to map the data in at. Then BSD VM executes two loops over that area. In the first loop, pages of memory are allocated in the specified kernel object. In the second loop the pages in the object are looked up by offset and entered in the kernel's pmap. This is wasteful for two reasons. First, there are two loops over the same address space. Second, in the first loop all the pages are allocated. Rather than saving the pointers to these pages for the second loop, the BSD VM function looks up the pages in the kernel object. This lookup is a more expensive hash table lookup. To contrast, UVM establishes the mapping with a single `uvm_map` call and then enters a loop over the allocated virtual address space that allocates a page and then maps it in. No hash table lookups are required.

8.9 Wired Memory

Wired memory is resident memory that is not allowed to be paged out to backing store. Memory is wired in BSD in the following situations:

- A user process can wire and unwire parts of its memory using the `mlock` and `munlock` system calls. This allows a process that performs time-critical operations to avoid the extra overhead of an unexpected page fault by ensuring that all possible faults in the wired area of memory are resolved in advance.

- The memory used to store the kernel's code segments (text, data, and bss) and critical data structures is wired. This is done to prevent the kernel from taking an unexpected page fault in a critical section of code. Also, certain parts of the kernel such as the memory that contains page fault handlers cannot not be paged out since this code is needed to handle future page faults.

- Each process on the system has a `proc` and a `user` structure that store the process' state. The `user` structure contains the process' kernel stack, signal information, and process control block. These three items are used only when a process is runnable. When a process is created, its `user` structure is wired in pageable kernel memory. If memory becomes scarce, a process can be "swapped out." To swap out a process the kernel unwires the process' user area and sets a flag in the process' `proc` structure that indicates that the process is swapped out. When a swapped out process needs to be run, it must first be "swapped in." To swap in a process, the kernel re-wires the process' user area and restores the process' swap flag in its `proc` structure.

- The `sysctl` system call is used by a process to query the kernel about some aspect of the operating system. The kernel responds to the `sysctl` system call by copying out the requested information into a buffer provided by the process in the process' address space. The buffer provided by the process may or may not be resident. If it is not resident, then when the kernel copies out the data a page fault may be generated. While the fault is being serviced the kernel's data may become dated. In order to prevent this and ensure that the calling process receives the freshest data possible, the kernel wires the process' buffer before copying the data out, and then unwires it once the copy is complete.

- The kernel `physio` function is used by the I/O system when a process reads or writes to a "raw" device. Such I/O operations always move data between the user's memory and the device (bypassing any resident data the kernel may have cached). In order to perform a `physio` operation, the kernel must ensure that the process' provides resident memory for the I/O operation and that that memory will not be modified or paged out during the I/O operation. To ensure this, the kernel wires the user's memory before starting the I/O operation. After the I/O operation is complete, the kernel unwires the memory.

There are two kernel interfaces for wiring memory: `vslock`/`vsunlock` and `vm_map_pageable` (`uvm_map_pageable` in UVM). The `vm_map_pageable` function takes a map, a starting virtual address, an ending virtual address, and a boolean value that indicates whether the specified memory should be wired or not. In BSD VM, the `vslock` and `vsunlock` functions are implemented as calls to `vm_map_pageable`. The `vslock` interface is only used by `sysctl` and `physio`, everything else uses `vm_map_pageable`.

The wiring of memory is counted in two places. First, each `vm_page` structure has a wire count. If this count is greater than zero, then the page is removed from the page queues, thus preventing the pagedaemon from paging out the page. Second, each map entry in a map has a wire count that indicates how many times the memory mapped by that entry has been wired. Like other map entry attributes such as protection and inheritance, if the wiring of only a part of memory mapped by a map entry is changed then a new map entry must be allocated and inserted in the map for that area.

When an area of memory is wired, the wire count of the map entry that is mapping the area is incremented. If the count was zero, then each page in the mapped area is made resident (using the fault routine), removed from the page queues, and the page's wire count is incremented. When an area of memory is unwired, the map entry's wire count is decremented. If the count reaches zero, then the wire count on each page on the mapping is decremented. If the page's wire count is decremented to zero, then the page is put on the page queues so that the pagedaemon will consider it for pageout if memory becomes scare.

8.9.1 Wiring and Map Entry Fragmentation

The problem with wiring memory is that it causes map entry fragmentation. The more map entries a map has the more kernel memory it takes and the longer it takes to search for a mapping. Such fragmentation occurs in a number of situations. For example, the kernel's map gets fragmented due to process' `user` structure being wired and unwired as part of swapout.

User process maps also get fragmented due to `sysctl` calls (and to a lesser extent `physio` I/O operations and `mlock` calls). While many programs do not call `sysctl` directly, the BSD C library makes use of `sysctl` to obtain the current configuration of the operating system.

8.9.2 Wiring Under UVM

There are two main ways to avoid map the entry fragmentation caused by the wiring and unwiring of memory. One way to avoid such fragmentation is to attempt to coalesce adjoining map entries together when ever a map or its attributes are modified. This would reduce map entry fragmentation caused by calls such as `sysctl` that wire memory, perform an operation, and then unwire memory. The other way to reduce map entry fragmentation is to avoid it in the first place. UVM attempts to avoid map entry fragmentation due to wiring whenever possible. The advantage of avoiding map entry fragmentation over coalescing map entries is that it is more efficient and less complex. It is more efficient because attempts to coalesce the map each time the map is touched are not required. It is less complex because the code needed to manage the coalescing of the map entries is not needed.

The main reason a map entry needs to be fragmented when part of the memory it maps is wired is to keep track of the wiring. When the area is unwired and the map entry's wire count reaches zero this information allows the page's state to be restored. In essence, the map is being used to store state information. In this case the state information is that "this part of memory is wired." If this state information is stored somewhere other than the map entry then the map entry's wire count does not need to be changed. UVM uses this approach to reduce map entry fragmentation in several cases:

private kernel memory: Private kernel memory is always wired and only appears within the kernel map. BSD VM stores the wired state of wired kernel memory in both the kernel map and the `vm_page` structures that are mapped by the kernel. In UVM the wired state is only stored in the `vm_page` structures, reducing kernel map entry fragmentation.

pageable kernel memory: Pageable kernel memory is used for processes' `user` structure. BSD VM stores the wired state of a processes `user` structure in both a kernel map entry and in the process' `proc` structures flag. UVM takes advantage of this. Since the wired state is stored in the `proc` structure, there is no need to store a redundant copy of the state information in the kernel map. In UVM, instead of fragmenting the kernel map the map's wire counters are left alone. When a `user` structure is wired or unwired both the flag in the `proc` structure and the wire counters in the `vm_page` structures that map the `user` structure are adjusted appropriately. This also reduces kernel map entry fragmentation.

sysctl and physio: In both of these functions, memory is wired, an operation is performed, memory is unwired, and the function returns. BSD VM stores the wired state of processes performing these functions in both the process' map, and on the process' kernel stack. In UVM the wired state of memory is only stored on the process' kernel stack. This reduces user map entry fragmentation.

It should be noted that in UVM, unlike in BSD VM, the vslock and vsunlock interface used by sysctl and physio wire memory using the vm_page structure's wire counter and do not change map entries.

In UVM, the only time a map entry's wire counter is used is for the mlock and munlock system calls. All other places that use wired memory store the wired state information elsewhere to avoid map entry fragmentation. As a result maps under UVM have fewer entries than under BSD VM.

8.10 Minor Mapping Issues

In this section two minor mapping related issues are discussed briefly.

8.10.1 Buffer Map

In BSD VM, the buffer map is used to allocate physical memory for the BSD buffer system at boot time. The way BSD VM allocates this memory is quite wasteful. A buffer is a fixed size chunk of virtual memory that is partly populated with physical memory. When booting the BSD kernel allocates a number of buffers and partially populates some of them with physical memory. Once this memory is allocated, it is no longer managed by the VM system (the buffer system uses an older interface inherited from earlier versions of BSD).

To allocate buffer memory, the BSD VM system allocates a submap of the kernel map at boot time (buffer_map). The virtual address space managed by this submap is used for the buffers. The entire virtual address space of the submap is allocated as one large chunk of zero-fill memory. To populate the buffer system with memory, BSD VM calls vm_map_pageable on parts of the zero-fill area to fault and wire in physical memory. Once this is done, the buffer map is never used again – from then on the memory is managed via the old interface.

The problem with this is that the wiring of the memory with vm_map_pageable leads to map entry fragmentation. For example, consider a system with a buffer size of 64KB and a small four buffer map. Assume each buffer should get populated with 16KB of physical memory at boot time. BSD VM would first allocate a 256KB submap for the buffer map. This space would be divided into four 64KB chunks. Initially, the entire 256KB is mapped zero-fill with a single map entry. Then BSD VM will wire down 16KB in each 64KB buffer. The results will be a submap with eight map entries, two for each buffer. Once these map entry structures are allocated they are neither freed nor used again.

In UVM there is no longer a buffer map. Instead, a chunk of the kernel map is allocated for the buffer system. Instead of using `uvm_map_pageable` to allocate the memory, UVM allocates the pages directly and maps them into the buffer area. This prevents UVM from wasting map entries on the buffer area.

8.10.2 Obsolete `MAP_FILE` Interface

The `mmap` system call supports two types of mappings: shared (`MAP_SHARED`) and copy-on-write (`MAP_PRIVATE`). These two types of mappings are standard [30]. Unfortunately, 4.4BSD also supports another mapping flag: `MAP_FILE`. The `MAP_FILE` constant is defined to be zero and the type of mapping it produces depends on what type of object is being mapped. If the object is a plain file, then the mapping is copy-on-write. However, if the object is a device, then the mapping is shared. Since `MAP_FILE` is non-standard and dependent on the type of file being mapped its use has been depreciated in UVM (and support for it may eventually be removed).

8.11 Page Flags

Each page of physical memory that is managed by the virtual memory system has a corresponding `vm_page` structure that contains information about the allocation of that page. As part of that information, each page has a set of boolean flags. In the transition from BSD VM to UVM these flags were rearranged. Initially, in BSD VM the page flags were stored in two separate integer bitfields. One bitfield was locked by the object lock and the other bitfield was locked by the page queues. At some point in the evolution of BSD VM these two bitfields were combined into a single integer that had its bits defined as C pre-processor symbols. This was unfortunate because it messed up the locking of the bits (some bits in the integer are locked by the object and others locked by the page queues). This is not currently a problem for BSD VM because fine-grain locking is not used. However, if BSD is to support fine-grain multiprocessor locking, then the page flag locking would become a problem.

In UVM the page's flags have been separated into two integers: `flags` and `pqflags`. The `flags` field is locked by the object that owns the page and the `pqflags` field is locked by the page queues. Also, in UVM some of the flags have been removed. The valid `flags` bits for UVM are:

PG_BUSY: The data in the page is currently in the process of being transfered between memory and backing store. All processes that wish to access the page must wait until the page becomes "unbusy" before accessing the page (see `PG_WANTED`).

PG_CLEAN: The page has not been modified since it was loaded from backing store. Pages that are not clean are said to be "dirty."

PG_CLEANCHK: The clean check flag is used as a hint by the clustering code to indicate if a mapped page has been checked to see if it is clean or not. If the hint is wrong it may prevent clustering but it will not cause any problems. The clean check flag is an optimization borrowed by UVM from FreeBSD.

PG_FAKE: The fake flag indicates that a page has been allocated to an object but has not yet been filled with valid data. UVM currently does not use this flag for anything other than to assist with debugging.

PG_RELEASED: The released flag indicates that a page that is currently busy should be freed when it becomes unbusy. It is the responsibility of the process that set the busy flag to check the released flag when unbusying the page.

PG_TABLED: The tabled flag indicates that the page is currently part of the object/offset hash table. When a page that is tabled is freed, it is removed from the hash table before being added to the free list.

PG_WANTED: The wanted flag is set on a busy page to indicate that a process is waiting for the page to become unbusy so that it can access it. When the process that set the busy flag on the page clears it, it must check the wanted flag. If the wanted flag is set, then it must issue a wakeup call to wakeup the process waiting for the page.

In addition to these flags there are five additional page flags that appear in BSD VM that should no longer be used: `PG_FICTITIOUS`, `PG_FAULTING`, `PG_DIRTY`, `PG_FILLED`, and `PG_LAUNDRY`. The elimination of the `PG_FICTITIOUS` flag for device pages UVM was described earlier in Section 8.4. The `PG_FAULTING` flag is used by BSD VM for debugging the swap pager and is not applicable to UVM. BSD VM uses the dirty and filled flags for debugging by the I/O system. This usage should be discontinued because the I/O system does not obtain the proper locks before accessing the flags and thus could corrupt the flags if fine-grain locking is in use. The laundry flag is used by BSD VM to indicate a dirty inactive page[4]. This flag is not needed by UVM.

Note that functions that set and clear the busy bit of a page need to check the wanted and released bits only if the object that owns the page was unlocked while the page was busy. If the object was not unlocked while the page was busy, then no other process could step in and set the wanted or released bits because they are protected by the object's lock.

The valid `pqflags` are:

`PQ_ACTIVE`: The page is on the active page queue.

`PQ_ANON`: The page is part of an anon.

`PQ_AOBJ`: The page is part of a aobj `uvm_object`.

`PQ_FREE`: The page is on the free page list. If `PQ_FREE` is set, then all other page queue flags must be clear.

`PQ_INACTIVE`: The page is on the inactive page queue.

`PQ_SWAPBACKED`: The page is backed by the swap area, and thus must be either `PQ_ANON` or `PQ_AOBJ` (and in fact, this flag is defined as the logical-or of those two flags).

8.12 VM System Startup

In this section the initialization of the VM system at system startup time is described. The kernel is loaded into memory by a low level bootstrap program. Once loaded, it is executed. When the kernel first starts the VM system has not been initialized and is not available for usage. The kernel must initialize the kernel environment enough to call the VM startup function (`vm_mem_init` in BSD VM, `uvm_init` in UVM).

8.12.1 Before VM Startup: Physical Memory Configuration

Before the VM system is started, the kernel needs to determine several pieces of information. First, it must determine how the physical memory is configured. Older systems have

[4] This flag is not well documented — it may have other uses

contiguous physical memory. Newer systems have non-contiguous physical memory. On these new systems the blocks of physical memory often correspond to memory module sized chunks.

The original version of the BSD VM system only handled systems with contiguous physical memory. If a system such as a Sparc1 was to run BSD, then it had to emulate a system with physical contiguous memory in the machine-dependent layer of the code (and in fact the sparc 4.4BSD port does this). Later versions of BSD directly support non-contiguous memory through the `MACHINE_NONCONTIG` interface. Unfortunately, this interface is different from the default interface and the BSD VM code has two different code paths for physical memory management. One of the code paths is selected at compile time (using a C pre-processor define). Both interfaces are briefly described below.

Contiguous Memory

If the `MACHINE_NONCONTIG` symbol is not defined for the C pre-processor then the old contiguous physical memory interface is selected. To use this interface, a process must set up four global variables before calling `vm_mem_init`. The variables are `avail_start`, `avail_end`, `virtual_avail`, and `virtual_end`. These variables define the start and end of available physical memory, and the start and end of available kernel virtual memory (respectively).

Non-contiguous Memory

If the `MACHINE_NONCONTIG` symbol is defined in the C pre-processor then the non-contiguous physical memory interface is selected. The non-contig interface is defined by several functions:

`pmap_virtual_space`: returns values that indicate the range of kernel virtual space that is available for use by the VM system.

`pmap_free_pages`: returns the number of pages that are currently free.

`pmap_next_page`: returns the physical address of the next free page of available memory.

`pmap_page_index`: given a physical address, return the index of the page in the array of `vm_page` structures.

`pmap_steal_memory`: allocates memory. This allocator can be used only before the VM system is started.

8.12.2 UVM's Physical Memory Configuration Interface

The two interfaces supported by BSD VM are quite different and require a number of C `ifdef` statements in the BSD VM source code. In order to address this problem UVM provides a new physical memory configuration interface that supports all physical memory configurations with a single unified interface. UVM's interface is called `MACHINE_NEW_NONCONTIG`, or MNN for short. MNN supports both contiguous and non-contiguous physical memory, and its support of contiguous memory has no extra overhead over the old interface.

In MNN, physical memory is managed using a statically-allocated array of memory segment descriptors. The maximum number of these segments is defined when the kernel is compiled. For systems with contiguous memory the number of entries in this array is simply one. For non-contiguous systems the number of entries is hardware dependent. Each entry in the array identifies the start and end of the region of physical memory it describes and contains pointers to an array of `vm_page` structures for that memory. The machine-dependent pmap module is also allowed to place some machine-dependent information in this array.

One common operation performed by the VM system is looking up the `vm_page` structure for a physical address. To do this under MNN, the array of physical memory segments must be searched. MNN allows the search to be optimized based on the likely physical memory configuration of the system. For systems with contiguous physical memory no search is necessary: the desired page is either in the first entry of the array or it is not a valid managed page. Thus, for contiguous systems no search code is compiled into the kernel. For non-contiguous systems MNN provides several search algorithms to choose from:

random: The array of physical memory segments is stored in random order and searched linearly for the requested page.

bsearch: The array of physical memory segments is sorted by physical address and a binary search is used to find the correct entry.

bigfirst: The array of physical memory segments is sorted by the size of memory segment and searched linearly. This is useful for systems like the i386 that have one small chunk of physical memory and one large chunk.

MNN encapsulates these options into a single function with the following prototype:

```
int vm_physseg_find(vm_offset_t phys_addr, int *offset);
```

This function returns the index of the entry in the array of physical memory segments for the specified physical address (or it returns -1 if the physical address is not valid). The "offset" is the offset in the segment for the given physical address.

When a system using MNN is booting, before calling the VM startup function, it must first load a description of physical memory into the array of physical memory segments. This is done with the `uvm_page_physload` function. This function takes the starting and ending physical addresses of a segment of physical memory and inserts the segment into the array, maintaining the requested sorting order. This single function replaces the two older interfaces supported by BSD VM.

8.12.3 UVM Startup Procedure

Once physical memory has been loaded into the VM system using the MNN interface then the `uvm_init` function can be called to bring up the rest of the VM system. This function is called as part of the kernel's `main` function. UVM is brought up in eight steps. First, UVM's global data structures are initialized. Second, the page sub-system is brought up. This includes initializing the page queues, allocating `vm_page` structures for available physical memory and placing them on the free list. Third, the kernel's private pool of statically allocated map entries is set up. Fourth, the kernel's virtual memory data structures, including the kernel's map and submaps, and the kernel object are allocated and initialized. Fifth, the machine-dependent `pmap_init` function is called to allow the pmap to do any last minute setup before the rest of the VM system is brought up. Sixth, the kernel memory allocator (`malloc`) is initialized. Seventh, the pagers are initialized. Finally, an initial pool of free anons is allocated. At this point the VM system is operational.

8.13 New Pmap Interface

UVM supports both the BSD VM pmap API and a new modified pmap API. The new pmap API, called `PMAP_NEW`, is required if page loanout to the kernel is to be supported (otherwise such loanouts will always fail). The pmap API was changed in five areas for `PMAP_NEW`. These changes are described below.

8.13.1 Page Functions

The pmap API has several functions that operate on all mappings of a managed page. (A managed page is one that has a `vm_page` structure and is used by the VM system.) An example of such a function is `pmap_page_protect`, a function that sets the protection on all mappings of a page. This function is used to write protect all active mappings of a page. Other such functions include functions that query and clear the referenced and modified bits of a page. In order to support such functions, the pmap module must keep track of all pmap structures that reference a page. Typically this is done with a list of mappings for each page. Each managed page has its own list of active mappings.

In the current pmap API, functions that perform operations on all mappings of a page take the physical address of the page as one of their arguments. The physical address of a page is determined through the `vm_page` structure's `phys_addr` field. Thus, to write protect all mappings of a page, one would use:

```
pmap_page_protect(page->phys_addr, VM_PROT_READ);
```

In order to process this, the pmap module must look up the physical address in the table of managed pages to determine what `vm_page` the physical page belongs to so the list of mappings for that page can be traversed. If the page is not a managed page, then there is no such list and no action is performed.

The current procedure is wasteful because the function calling the pmap page protection function knows the `vm_page` structure for the page, but does not pass this information to the pmap module. This forces the pmap module to do a page lookup to determine the identity of the physical address' page. In the new pmap interface, these functions take a pointer to a `vm_page` structure rather than a physical address so that this extra lookup operation can be avoided.

8.13.2 Kernel Pmap Enter Functions

Pmap enter functions establish a low-level mapping for a given page. The FreeBSD VM system's pmap API provides several special kernel pmap enter functions that establish a kernel mapping for a page faster than the normal `pmap_enter` function. The speedup is achieved by not entering the page's mapping on the list of mappings for that page. This allows pages that will only be mapped by the kernel (e.g. for malloc) to be entered in the kernel pmap quickly. UVM's new pmap interface supports the FreeBSD pmap API. New functions include pmap_kenter_pa, pmap_kenter_pgs, and pmap_kremove. These functions establish a kernel mapping for a physical address, establish a kernel mapping for an array of page structures, or remove kernel mappings.

While FreeBSD's motivation for these functions was to speed kernel pmap access, UVM also uses the functions to support page loanout to the kernel. When a page that is mapped by a user process is loaned out to the kernel, it is mapped with pmap_kenter_pa. Since the new mapping is not entered on the list of mappings for the page, it will not be effected by pmap_page_protect. The benefit of this is that it provides a way for the VM system to remove all mappings of a page except for the ones the kernel is using for page loanout. This allows the pagedaemon to page out a page loaned to the kernel by removing all its user-level mappings with pmap_page_protect without disturbing the kernel's mappings. Then the pagedaemon, following the rules presented in Chapter 7, causes ownership of the page to be dropped, thus allowing the page to remain allocated until the kernel is done with the loan. This effect is impossible to achieve with the old pmap interface.

8.13.3 Other Changes

The other three pmap API changes are minor. They are:

- The FreeBSD pmap_growkernel function was added to the pmap API. If enabled, this pmap function is called when UVM allocates a chunk of kernel virtual memory for the first time. This allows a pmap module to allocate page table pages for the memory before it is used.

- The "size" argument to the pmap_create function has been removed. This argument is a holdover from Mach VM and is not used in either BSD VM or UVM.

- The pmap_clear_reference and pmap_clear_modify functions were modified to return a boolean value. They return false if the specified bit is already clear. Otherwise, they clear the bit and return true. This allows the kernel to test and clear the bit at the same time.

8.14 New I386 Pmap Module

UVM includes a new pmap module for the i386 processor. The pmap module has been rewritten to address problems with the old pmap, support UVM's new pmap API, and incorporate improvements from other operating systems such as FreeBSD. The new pmap uses the `PMAP_NEW` interface.

The i386 pmap module maintains two main types of data structures: pmap structures and pvlist structures. The pmap structure contains the hardware-specific state information for a single virtual address space. Each process has its own pmap. The pvlist structure is a per-page structure that contains a linked list of a page's current mappings. An element on this list is called a pventry. Each pventry contains a pmap pointer, a virtual address, and a pointer to the next pventry in the pvlist. The pvlist is used to change the protection of all mappings of a page and to keep track of whether the page is referenced or modified.

The three most common pmap operations are establishing a mapping, removing a mapping, and changing the mapping of a specific page in all pmaps that reference it. The pmap module performs these operations as follows:

- When `pmap_enter` is called to establish a mapping of a page in a pmap, it first ensures that there is a valid page table for the requested virtual address. If there is currently no page table, a free page is allocated to contain the new page table. Since the size of a page table on the i386 is equal to the size of a page, a page that contains a page table is called a "page table page." Once a page table is allocated, the pmap module allocates a pventry structure to record the mapping on the page's pvlist. Finally, an entry for the mapping is entered in the page table.

- When `pmap_remove` is called to remove a mapping of a page the procedure is reversed. The page's entry is removed from the page table, and the pventry structure is removed from the pvlist. If the final entry was removed from the page table, then the page table page is freed. Additionally, when removing addresses from a pmap, the i386's hardware cache of current page table entries must be flushed. This is called a translation lookaside buffer (TLB) flush.

- When `pmap_page_protect` is called to change the protection of a page in all pmaps that refer to it, the page's pvlist is traversed. For each pventry on the pvlist the mapping is changed. If the new protection is `VM_PROT_READ` then the page is write-protected. If the new protection is `VM_PROT_NONE`, then the page is removed from all pmaps referencing it (in this case the pvlist must be zeroed).

8.14.1 UVM Pmap Features From Other Operating Systems

UVM's new i386 pmap includes several new features inspired by other operating systems. These features are:

- Making use of the single page TLB flush instruction. When removing page mappings, the only way the original i386 processor could flush the TLB was to flush all entries out. Not only does this flush the TLB entry of the mapping being removed, but it also flushes all the other valid entries as well, thus causing them to be reloaded. This is inefficient. Starting with the i486 processor, Intel added a new instruction that flushes a single entry out of the TLB. The FreeBSD pmap makes use of this instruction to avoid the unnecessary flushing of valid TLB entries. UVM's new pmap includes code based on FreeBSD's code to use this new instruction.

- Intel also added a new flag to the Pentium for a page table entry. The new flag, `PG_G`, indicates that if this mapping is cached in the TLB it should only be flushed by the single page flush instruction. The instruction to flush the entire TLB will not effect mappings with `PG_G` set. The benefit of this is that all the kernel's memory can be mapped with `PG_G`. When a context switch occurs, the entire TLB is flushed, but the kernel's TLB entries will persist because `PG_G` is set on them. UVM's code to manage the `PG_G` bit was based on FreeBSD.

- Under BSD VM, a user process' page tables are placed in the top part of the user's address space and managed as anonymous memory. This is a problem for two reasons. First, when the final entry of a page table is removed the page table is not freed. Instead, the page table persists until the process exits. Second, page tables must be prefaulted before the kernel attempts to access their PTEs. In the common case of a page fault, an extra call to the fault routine can fault in the page table pages, however, there are other paths such as `mlock` that bypass this pre-fault code. If the pre-fault code is bypassed and the desired page table is not present then the BSD kernel crashes with a "`ptdi`" panic. This problem is solved in FreeBSD by having the pmap module directly manage the page table pages. UVM uses a similar approach.

- The BSD VM pmap does not have any locking on its data structures. The Mach VM pmap contains a scheme for data structure locking. Each pvlist has a lock and each pmap has a lock. In UVM a locking scheme based on the Mach VM scheme has been implemented. The locking scheme had to be adapted for the i386 processor.

8.14.2 UVM-Specific Pmap Features

UVM's i386 pmap has two new features that are UVM specific. First, the new pmap has been carefully designed to manage pmap resources without deadlocking the system. The locking scheme of the machine-independent layer of both BSD VM and UVM expect that pmap operations will never block. Although this is expected in BSD VM, the old i386 pmap does not deliver on this promise. For example, the `pmap_enter` function allocates a pventry structure when establishing a mapping. The BSD VM system uses the `kmem_alloc` function to allocate a pool of pventry structures. But the `kmem_alloc` locks the kernel map during allocation and sleeps if not enough memory is available. If the function calling `pmap_enter` calls when there is a shortage of pventry structures and it already has the kernel map locked, or if there is not enough memory to allocate new pventry structures, the system will deadlock. This usually does not occur because the old pmap module keeps a pool of allocated pventry structures available for usage. However, the behavior of the old i386 pmap module is clearly wrong.

UVM's `pmap_enter` uses several different strategies to avoid deadlock when allocating pventry structures. First, like BSD VM, it maintains a pool of free pventry structures and attempts to keep the pool reasonable full. However, if the pool runs out, UVM uses a second strategy. UVM has a special allocation function used to allocate more memory for pventrys. The function attempts to lock the kernel map. But rather than sleeping if the kernel map is already locked, the allocation function fails instead. If this function fails, then UVM falls back to a final strategy. The final strategy is to lock a pvlist structure and attempt to "steal" a pventry structure from it. The pventry structures can be stolen because the information that they store can be reconstructed based on the information in the `vm_map` structure. In the highly unlikely event that this final strategy fails UVM will panic. UVM also allocates page table pages using a similar strategy. If there are no pages on the free list, the pmap module will attempt to "steal" a page table page from another pmap. To do this, it must first remove all mappings from the page table. By allocating pventry structures and page table pages this way, the UVM version of `pmap_enter` (and related functions) never block machine-independent code (unlike BSD VM).

The second UVM-specific feature is the dynamic handling of TLB flushes. The i386's TLB can be flushed in one of two ways. Either all the entries can be flushed, or a single entry can be flushed. If a large number of pages are being unmapped, it is more efficient to flush the entire TLB than to flush each page one at a time. However, if only a few entries are going to be flushed, then it is more efficient to flush the pages one at a time. UVM's new pmap includes code that dynamically chooses between these two strategies depending on how many pages are going to be removed. Note that this decision is based on the number of pages removed, not the size of area being removed. Large sparsely populated regions of VM may still benefit from single-page TLB flushing, and UVM allows for this.

8.15 Summary

In this chapter a number of secondary design elements of UVM were presented. These elements include:

- the chunking of amap allocation to reduce the amount of kernel memory allocated for a sparsely populated amap.
- the aggressive pageout clustering of anonymous memory that allows UVM to quickly recover from a memory shortage.
- the design of UVM's three pagers: the aobj pager, the device pager, and the vnode pager.
- the `uvm_map` memory mapping function that provides a single uniform interface to memory mapping under UVM.
- the unmapping function that separates unmapping memory and dropping references to memory objects.
- the management of kernel memory maps and objects that reduces the number of VM calls needed to establish kernel memory mappings.
- changes in the management of wired memory that prevents unnecessary map entry fragmentation and thus reduces map lookup time.
- the redesign of the page flags to accommodate fine grain locking and to eliminate unneeded flags.
- `MACHINE_NEW_NONCONTIG`, the new unified scheme used to manage a computer's physical memory.
- the new pmap interface that supports features incorporated from FreeBSD and also kernel page loanout.

- the new i386 pmap module that manages pmaps more efficiently and does not block machine-independent code.

While the design elements are secondary, they cumulate to produce a beneficial effect on UVM's design and performance, and thus play an important role within UVM.

Chapter 9

Implementation Methods

This chapter describes the methods used to implement and debug UVM. First, the strategy behind the UVM implementation is presented. Then the tools used to assist with debugging are presented. These tools include the UVM history mechanism and functions that can be called from the kernel debugger. Finally, two user-level applications that display the status of UVM are presented.

9.1 UVM Implementation Strategy

Implementing a software system as large as UVM is a major task, especially when almost all of the work is being done by a single person. The choice of implementation strategy for such a project is critical if one hopes to bring the project to a successful conclusion. A poor implementation strategy could lead to one of two possible conclusions. First, the project could become so complex that it cannot be completed. Second, the project could be completed in such a way that it is so difficult to use and install that no one bothers to consider using it. Either one of these alternatives is unacceptable for UVM. We want UVM to be completed, make an impact, and be *used* by people.

Three implementation strategies for UVM were considered. The first strategy was to re-design and re-implement the whole VM system from scratch. This strategy was rejected because it was too much work for too little gain. There were certain aspects of the BSD VM system that worked fine and did not need to be modified or reinvented. For example, BSD VM's machine-independent/machine-dependent layering is perfectly adequate for UVM. In fact, making machine-dependent design changes is highly unattractive because of the number of platforms BSD runs on (NetBSD currently has eighteen different pmap modules). The second implementation strategy considered was to start with the source tree, completely remove the BSD VM source code from it, and write UVM to replace it (using the same API). The problem with this strategy is that it is very difficult to debug because most of UVM has to be written before it can be debugged. Also, all debugging would have to be done at a very low level, without the assistance of the kernel debugger, since the system would not be able to fully boot without a working VM system. The third UVM implementation strategy, described in greater detail below, involves implementing the UVM system in parallel with the existing VM system.

UVM was implemented in four main phases. In the first phase we studied the current BSD VM. In the second phase we implemented the core of UVM's mapping and fault mechanism within BSD VM. In the third phase we put all user process' memory under the control of UVM, keeping the kernel under the control of BSD VM. In the fourth and final phase we put the kernel under UVM and removed BSD VM from the system. Throughout the implementation process, the UVM source tree was managed in such a way that UVM was easy to use and install. The rest of this section discusses these phases in more detail.

9.1.1 Phase 1: Study of BSD VM

Before even attempting to write UVM code, it was important to understand how BSD VM functions and the set of features it provides. This was no easy task, as the BSD VM system is not well documented and at the time the UVM project was started the only real documentation on the BSD VM system were some papers put out by the Mach project. So most of the study of BSD VM was based on information collected from the BSD VM source code. The result of phase one of UVM was a set of notes that described the detailed operation of BSD VM and the desired features and structure of UVM. Much of what we learned in this phase is documented in Chapter 2.

9.1.2 Phase 2: UVM Under BSD VM

The second phase of the UVM implementation involved implementing UVM's core memory mapping and fault handling routines. This was done by taking advantage of one of the attractive features of BSD VM. In BSD VM the machine-independent and machine-dependent code are cleanly separated. Thus, the machine-dependent pmap layer can be used to establish and remove low-level memory mappings without involving the upper layer of the BSD VM. This allowed UVM's core memory mapping and fault handling routines to be developed in a live kernel running the BSD VM system.

To achieve this, several minor changes were made to the BSD kernel. First, we modified the kernel so that after the system was booted and the BSD VM system was brought up, the UVM testing subsystem would allocate a number of pages from the BSD VM system to be used for testing. Second, another pointer to a `vm_map` structure was added to the `vmspace` structure for UVM. When a process is created this map pointer is set to null. Third, the BSD VM fault routine was modified to detect faults on memory managed by the UVM testing subsystem and delegate them to the new `uvm_fault` routine. Fourth, the kernel was modified to free any UVM related resources a process is holding when the process exits. Finally, a UVM testing system call was added to the kernel to allow normal user-level processes to manipulate UVM.

In order to test UVM using this setup, a process performs the following steps. First the process uses the UVM test system call to establish a block of virtual memory that will be managed by UVM. The system call uses BSD VM to allocate a chunk of the specified size out of its map. A null `vm_object` is mapped there. Second, a new `vm_map` structure is allocated and initialized to manage this newly allocated area of virtual memory. The new map is installed in the process `vmspace` structure's second `vm_map` pointer.

At this point, the testing process can use the UVM test system call to establish shared and copy-on-write mappings to a file, or establish a zero-fill mapping. The process can then perform reads and writes on the UVM block of virtual memory. The reads and writes can trigger page faults. The BSD VM fault routine detects the UVM faults and delegates the handling of them to the UVM fault routine.

As a result of the changes implemented in Phase 2, a user-level process on a system running the BSD VM system can easily test a number of UVM features such as mapping, shared memory, zero-fill memory, and copy-on-write. As copy-on-write is handled with amaps and anons, the test program also tests these subsystems. If an error occurs, either the process will get an error signal, or the kernel will crash. Since the kernel is running under the fully operational BSD VM system, the normal kernel debugger (`ddb`) can be used to debug and improve UVM.

9.1.3 Phase 3: Users Under UVM

At the completion of the second phase of the UVM implementation strategy the memory mapping, anonymous memory, and fault code were in reasonable shape. In the third phase all user processes' virtual memory was placed under the control of UVM. The kernel continued to operate under BSD VM, thus allowing UVM to be debugged with the normal kernel debugger.

To place all user processes' virtual memory under the control of UVM, the standard user-level VM system calls were redirected from the BSD VM system to UVM. Also, the number of pages allocated by UVM from BSD VM at system startup time was increased. When a testing kernel boots, the first user-level process is `/sbin/init`. The kernel uses the code already tested in phase two to memory map and fault init's text, data, and bss into its address space.

In phase three, the main part of UVM that was tested was the code that handles the duplication of a process' address space during a `fork` operation. Indeed, it took many attempts before init was successfully able to fork off a `/bin/sh` process.

9.1.4 Phase 4: Kernel Under UVM

The final phase involved switching the management of kernel memory to UVM. The first step in this process was to address the differences between machines with contiguous and non-contiguous physical memory. As part of this effort, `MACHINE_NEW_NONCONTIG` (MNN) was developed (see Section 8.12.2). Once MNN was created, the kernel had a single unified interface for managing physical memory. The second step in this phase was to write the code that managed the kernel's memory (see Section 8.8 on the `uvm_km` functions). Once this step was completed, a BSD kernel could run multiuser under UVM with only two features missing: the ability to configure and pageout to swap, and System V shared memory. These final two features were added to UVM in step three with the assistance of Matthew Green and Chuck Silvers — two NetBSD contributors.

9.1.5 Source Tree Management

One of our goals for UVM was to get the system integrated into one or more of the freely available BSD systems. This put two constraints on the development of UVM. First, UVM had to keep up with current developments in the BSD systems. UVM was implemented within the NetBSD kernel. NetBSD was selected because we were already familiar with it and it was the only BSD operating system than ran on the sparc hardware that UVM was being developed on at the time. The NetBSD kernel is a moving target — it is constantly evolving. Fortunately, the NetBSD source tree is managed with the Concurrent Versions System (CVS). CVS allows multiple developers to use the same source tree at the same time. CVS allowed UVM to be developed in parallel with the NetBSD kernel. Periodically, changes made to the master NetBSD source tree were merged into the UVM tree using CVS.

The second constraint placed on UVM development was that UVM presence in the source tree could not disrupt the BSD VM source code. The transition from BSD VM to UVM was made as easy as possible for both users and developers. To achieve this, UVM was written in such a way to allow the selection of VM system (BSD VM or UVM) to be made when compiling the kernel. Both BSD VM and UVM source live within the same source tree. This allows a user to easily switch between VM systems without having to have more than one source tree. Also, MNN was ported to BSD VM. This allowed kernel developers to move their ports to UVM with a two step process. First, MNN support is added to the machine-dependent part of the port. Second, UVM support is added to the port. Having MNN run under BSD VM allows kernel developers to test MNN machine-dependent code separately from UVM.

9.2 UVMHIST: UVM Function Call History

One important tool used in the implementation and debugging of UVM is the UVMHIST function call history subsystem. UVMHIST provides a way for UVM to produce a trace of requests being made to the VM system and the corresponding functions called to service these requests. Such a trace can be produced without disrupting the normal operation of the operating system. This allows a kernel developer to easily trace the path taken through the UVM source code and determine if either an incorrect or inefficient path was taken.

UVMHIST is implemented as a circular array of log records. Each record contains the following information:

- The name of the function that generated the log message.
- The call number of the function. The call number indicates which invocation of the function generated the log. Call numbers start at zero and are incremented by one each time the function is called. The call number is useful for separating events and detecting recursion.
- A timestamp collected from the system's clock.
- A pointer to a string containing a `printf` format.

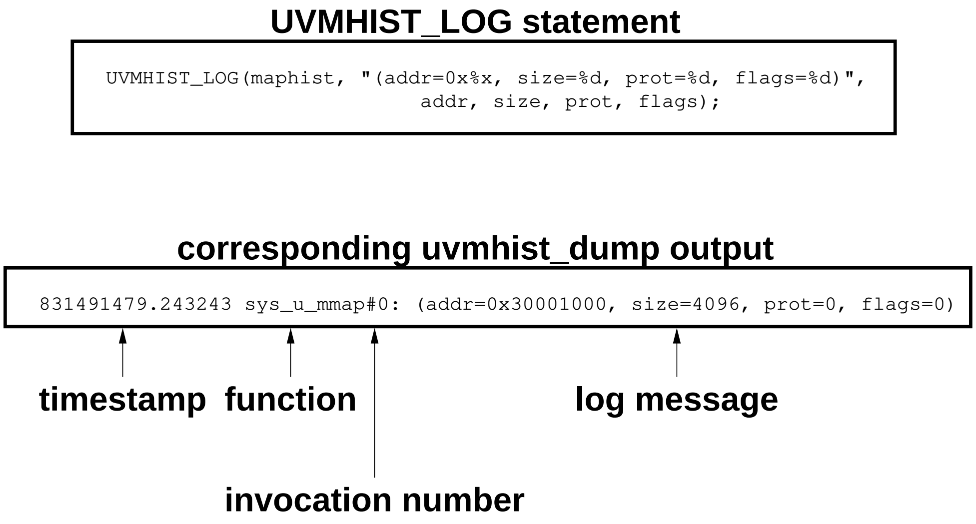


Figure 9.1: UVMHIST usage

- Four integers that contain the data for the `printf` format.

UVM supports multiple UVMHIST arrays.

Log entries are entered into the UVMHIST log using the `UVMHIST_LOG` macro. Calls to the `UVMHIST_LOG` are distributed throughout the UVM source code. Note that on systems with UVMHIST disabled the `UVMHIST_LOG` macro is null. This allows the `UVMHIST_LOG` calls to remain in production code with no overhead.

UVMHIST includes a function called `uvmhist_dump` that prints the contents of an array of history entries. An example of a `UVMHIST_LOG` call and the corresponding output from `uvmhist_dump` is shown in Figure 9.1.

UVMHIST has had a beneficial effect on the UVM project in several ways. First, it is an effective educational tool for learning about the VM system. Second, data from UVMHIST traces has found several redundant calls to the VM system in the BSD source code. Eliminating these calls reduced VM system overhead. Third, data from UVMHIST traces has also been used to modify UVM's design to reduce the overhead of certain common operations. The remainder of this section provides details on the benefits of UVMHIST in these three areas.

9.2.1 UVMHIST As an Educational Tool

The primary way to learn about the operation of most kernel subsystems is to either read the source code or try and find a paper or book that describes the subsystem's operation.

While such information resources do explain the data structures and functions, this is only part of the story. Source code and papers only present a static view of a software system. It is difficult to fully understand and appreciate a kernel subsystem without being familiar with the dynamic aspects of the system. In other words, understanding what functions are commonly called and the order in which they are called in response to a request is an important part of understanding a system. This is one of the reasons that understanding the detailed function of BSD VM is difficult. It has many functions and no roadmap to explain how all the functions interrelate during normal VM system operation.

UVMHIST allows kernel developers to more easily educate themselves on the operation of UVM. A developer can run a program that makes a request to the VM system. Once the request is complete, the developer can dump out the UVMHIST and easily understand the path taken through UVM to satisfy that request. An especially useful exercise is to enable UVMHIST and boot a system single user and review the UVM calls necessary to to start `/sbin/init`.

9.2.2 UVMHIST and Redundant VM Calls

The traces produced by UVMHIST during single user operation were used to eliminate several redundant calls to the VM system in the BSD kernel. Redundant calls are harmless to the operation of the kernel, but they add needless extra VM system overhead to the life time of a process. The following redundant calls were found in the BSD kernel using UVMHIST:

- When forking a process, the `uvm_fork` function uses `uvm_map_inherit` to set the inheritance value of the user's page tables to "none" before copying the address space. This call is necessary to prevent the child process from interfering with the parent's processes page tables. However, after copying the address space, the `uvm_fork` performs two redundant VM operations on the newly created child process' map. First, it unmaps the area of the child's address space used for page table pages. Second, after re-allocating that space for the child's page tables it calls `uvm_map_inherit` to set the inheritance value of the page table space to "none."

  The unmap call is redundant because the `uvm_fork` function set the inheritance of the page table space to "none" before copying the address space, thus it is not possible for any page table mappings to be active after the address space is copied.

  The second inheritance change is there to prevent a child of the child's process from inheriting the child's page tables. This call is unnecessary because if the child forks the uvm_fork function will reset the inheritance of this region of memory before copying the child's address space.

- In the exec code, the function vmcmd_map_pagedvn is used to memory map the text or data portion of a vnode into a process' address space. Initially, it used the uvm_mmap function with MAP_FIXED to establish this mapping. At the time this mapping is established the process' address space is being filled for the first time with the new program, so there should never be a previously established mapping that would block the mapping being created. However, due to the requirements of the mmap system call, if the MAP_FIXED flag is set, the uvm_mmap function always calls the unmap function to clear out any previously established mapping. This results in the UVMHIST trace containing the following VM operations at exec time:

  1. allocate a new vmspace (the new space has no active mappings)
  2. unmap text area [redundant]
  3. map text
  4. unmap data area [redundant]
  5. map data

  Clearly both unmap operations are redundant. To fix this problem, the exec code was modified to call uvm_map directly, since the special handling of uvm_mmap is not needed in UVM.

- UVMHIST traces also detected a redundant call to uvm_map_protect in the exec code path. The function vmcmd_map_zero establishes zero-fill mappings. It is used to memory map the bss and stack segments of a process. As previously described, in BSD VM all mappings are established with the default protection (read-write). Early versions of UVM's memory mapping function also always used the default protection. It turn out that in the BSD code the vmcmd_map_zero function always used uvm_map_protect to set the protection after the mapping was established — even if the desired protection was the default protection. This results in the following trace during an exec:

  1. zero-map the bss segment (the protection is set to the default, read-write).
  2. set the protection of the bss segment to read-write [redundant]
  3. zero-map the reserved stack area

4. set the protection of the reserved stack area to "none"
5. zero-map the active stack area
6. set the protection of the active stack area to read-write [redundant]

The two redundant `uvm_map_protect` calls set the protection to the value it is already set to. This trace was one of the motivating factors for allowing protection to be specified in the flags of the `uvm_map` function. Thus, in UVM not only are the redundant `uvm_map_protect` calls eliminated, but the call to set the protection of the reserved stack area to "none" is also eliminated since that protection can now be set while memory mapping the area.

- When a process exits, it must unmap its memory to free it. The UVMHIST traces show that the BSD kernel was unmapping an exiting process' address space in three places on i386 systems: once in the main exit function, once in the machine-dependent `cpu_exit` function, and once in the `uvmspace_free` function. Based on this information the unmap in `cpu_exit` was removed and the unmap operation in `uvmspace_free` was modified to only occur if the map being freed still has entries.

9.2.3 UVMHIST and UVM Design Adjustments

In addition to being used to detect redundant VM calls, UVMHIST traces were also used to make two adjustments to UVM's amap code. One change motivated by the UVMHIST logs was the amap chunking code previously described in Section 8.1. Using the UVMHIST trace produced over part of a system's bootup, it was determined that before amap chunking was added to the system 11186 pages of virtual memory were covered by the amap system. After adding amap chunking, UVMHIST shows that in the same bootup operation amaps covered only 107 pages of virtual memory.

Another area where the UVMHIST traces contributed to a UVM design change was in the handling of the `sys_obreak` system call. This system call is used to expand a process' heap area for `malloc`. Expanding the heap area involves extending the size of the amap that allocates that area. This can be somewhat expensive since the data in the amap may have to be copied in order to increase its memory allocation. When booting single user, the UVMHIST trace showed that the `/sbin/init` process calls `sys_obreak` eleven times before it gets to the point where it can prompt for a shell on the system's console. Each time the break grew, the corresponding amap had to be extended. To reduce the overhead of this, the "amappad" flag was added to the `uvm_map` function. This flag causes the mapping function to allocate an amap for the mapped region that is larger than the region itself. This allows the amap to be extended in-place without the need for allocating a new amap and copying the data from the old amap to the new amap. After adding this flag, the UVMHIST trace shows that the number of amap extends that cause an amap reallocation was reduced from eleven to one.

9.3 UVM and the Kernel Debugger

Another useful tool used in debugging the implementation of UVM is DDB, the kernel debugger built into BSD kernel. If DDB is compiled into the kernel, the kernel can be stopped at almost any point and the VM data structure can be examined. Because DDB is much easier to enable and use than source-level kernel debuggers such as KGDB [63] that require special serial line setup, it is extremely useful for kernel developers debugging VM problems reported over the Internet. In these cases the developer often does not have access to the hardware that is experiencing the problem but they can guide the person reporting the problem through the process of using DDB to diagnose the problem.

Kernels with DDB enabled experience little if any overhead due to DDB's presence as DDB is usual not active. DDB can be invoked in one of three cases: the system crashes, the kernel calls the `Debugger` function to invoke DDB, or the user hits a special escape sequence to break the kernel into DDB[1]. Once running, DDB prompts for input with the "`db>`" prompt. At that point the system operator can examine most UVM data structures. Useful UVM related DDB commands are shown in Table 9.1. In addition to these commands, the standard commands used to set and clear breakpoints and to single-step through kernel functions are also useful. Kernel text addresses displayed by DDB can be translated to their line number in the source code by using GDB on the object file that contains the function in question.

9.4 UVM User-Level Applications

In addition to kernel-level tools such as UVMHIST and DDB, UVM includes two user-level applications that display UVM status information. In traditional systems, such applications require special privileges to run because they need read access to kernel virtual memory (`/dev/kmem`) and the file containing the currently running kernel. However, in UVM this

[1]On an i386 the DDB escape sequence is CTL-ALT-ESC. On a sparc it is L1-A or BREAK.

Table 9.1: Useful UVM DDB Commands

Command	**Usage**
`ps`	Lists all active processes on the system. If the "a" option is specified ("`ps/a`"), then the kernel address of each processes' `vm_map` structure is printed.
`show map`	Shows the contents of the specified `vm_map` structure. If the "f" option is specified, then the list of map entries attached to the map is also printed. For user processes, use the "`ps/a`" command to get map addresses. For the kernel map, use the "`show map *kernel_map`" command.
`show object`	Shows the contents of the specified `uvm_object` structure. If the "f" option is specified, then a list of all pages in the object is also printed.
`show page`	Shows the contents of the specified `vm_page` structure. If the "f" option is specified, the extra sanity checks on the page are performed.
`call uvm_dump`	Shows the current values of the counters used by UVM and the kernel addresses of kernel objects.

is no longer the case. UVM supports a `sysctl` system call that any process can use to get UVM status information without the need for special privileges. This gives normal users the flexibility to write their own status gathering programs without the need to bother their system administrator.

The first UVM status program is called `uvmexp`. It is a simple text-based program that gets the status of UVM, prints it, and exits. This is much like the "`call uvm_dump`" command in DDB. The second UVM status program is an X11-based program called `xuvmstat`. This program creates a window and graphically displays various system counts (updating the display once a second). A screen dump of the `xuvmstat` program is shown in Figure 9.2.

9.5 Summary

In this chapter we have reviewed the methods used to implement and debug UVM. UVM was implemented in four phases. First the design of the BSD VM was studied. Second, UVM's core memory mapping and page fault functions were implemented and tested under a system running BSD VM. Third, virtual memory management of all user-processes was switched from BSD VM to UVM. Finally, the kernel's memory management was switched over to UVM. Throughout the implementation process, changes made to the BSD source tree were merged into the UVM kernel source tree using CVS. This eased of integration of UVM into the main source tree. Additionally, UVM was designed to co-exist in the source tree with BSD VM. This allows for a smooth transition for BSD users from BSD VM to UVM.

The UVMHIST tracing facility, UVM DDB commands, and UVM status programs were also presented in this chapter. UVMHIST allows kernel developers to produce dynamic code traces through the VM system. This helps developers better understand the operation of UVM and also was used to detect redundant calls in the VM system. The UVM DDB commands allow a user to easily examine the state of UVM on a running system. The UVM status programs are built on top of a non-privileged system call that allows users to query the status of UVM.

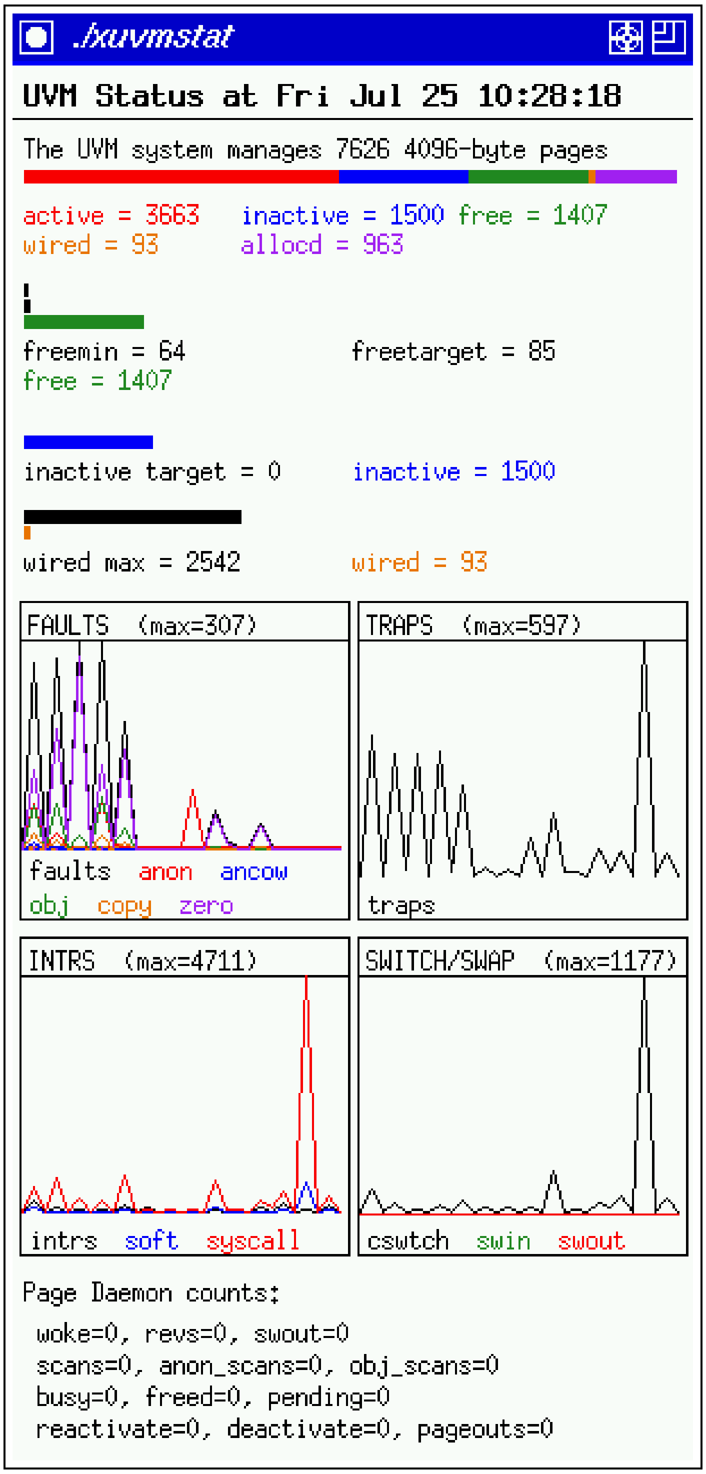


Figure 9.2: A screen dump of the `xuvmstat` program

Chapter 10

Results

In previous chapters we described the design and implementation of new VM features first introduced to BSD by UVM. The design of features such as page loanout, page transfer, and map entry passing provide the I/O and IPC subsystems with multiple data movement options. But such gains in flexibility are useful only if they are more efficient than traditional data copying. In this chapter we describe five sets of tests designed to measure the performance of UVM's new features. The results of these measurements show that page loanout, page transfer, and map entry passing can significantly improve I/O and IPC performance over traditional systems. We also measure the effects of UVM's secondary design elements. Our results show that due to our secondary design improvements UVM outperforms BSD VM in several critical areas, even when our new data movement mechanisms are not used.

All our tests were performed on 200 MHz Pentium Pro processors running NetBSD 1.3A with either UVM or BSD VM. Our Pentium Pros have a 16K level-one cache, a 256K level-two cache, and 32MB of physical memory. The main memory bandwidth of our Pentium Pros is 225 MB/sec for reads, 82 MB/sec for writes, and 50 MB/sec for copies, as reported by the `lmbench` memory benchmarks[1] [40].

10.1 Page Loanout Performance

Page loanout allows a process to loan a read-only copy of its pages of memory out to the kernel or another process, thus avoiding costly data copies. The loaned out pages are

[1] We used version 1.2 of `lmbench`'s `bw_mem` programs to make these measurements.

marked copy-on-write in the process so that changes made by the user to loaned out pages do not interfere with the loan.

A convenient place to measure the effect of page loanout on I/O is in the transfer of data from a user process to a device. In a traditional system, this would involve copying the data from the user's address space to a kernel buffer, and then performing I/O on that buffer. With page loanout, such an I/O operation could be done by loaning the pages from the user's address space directly to the device, thus avoiding the data copy.

Networking devices are often chosen for such measurements, so we decided to measure the effect of page loanout on the BSD networking subsystem. Our goal is to find out what effect page loanout has on the amount of data we can pump through the networking subsystem.

The BSD networking subsystem consists of three layers: the socket layer, the protocol layer, and the network interface layer. The socket layer is responsible for queuing and transferring data between user memory and kernel network buffers (mbufs). In a traditional kernel, data is transfered between the user and kernel through data copies. The protocol layer is responsible for taking mbufs from the socket layer, encapsulating them in a packet, and passing the packet to the networking interface layer for transmission. The protocol layer also receives inbound packet in mbufs from the network interface layer. These inbound packets are checked and either discarded, passed up to the socket layer, or sent to the network interface layer for transmission on a different network interface (e.g. if the system is acting as a router). The network interface layer controls the networking hardware and transmits packets it receives from the protocol layer and stores received packets in mbufs that it passes up to the protocol layer.

One problem with measuring the performance of page loanout with the networking subsystem is that current processors can easily generate data faster than current generation network interfaces can transmit it[2]. For example, the Pentium Pro processor on our test machine can easily swamp our 100Mbps fast Ethernet and 155Mbps ATM cards. This results in either the transmitting process being stalled until the network interface layer can catch up, or in data being discarded at the network interface layer due to a full network queue. If one of these network interface cards was used in measuring page loanout, any positive results achieved by page loanout would be seen as a reduction in processor load rather than increased bandwidth.

[2] Next-generation network interfaces such as Gigabit Ethernet and higher-speed ATM interfaces can transmit data at higher speeds.

10.1.1 Socket Layer Modifications

Measuring bandwidth is both easier and more accepted than measuring processor load. Since the network interface layer introduces the same delays irrespective of the use of page loanout, these delays can be safely factored out of performance measurements. To achieve this, a new "null" protocol layer was introduced into the BSD kernel. When the null protocol layer receives data from the socket layer for transmissions it discards it rather than passing it on to a network interface. This allows the data transfer overhead of copying verses page loanout to be measured with bandwidth rather than processor load.

The socket layer is of primary interest in the context of UVM because it handles the transfer of data between user memory and mbufs. We had to modify the socket layer in order to test the page loanout function of UVM.

An mbuf is a fixed sized structure that can contain either a small amount of data or a pointer to another buffer. Mbufs that contain data are called small mbufs. Small mbufs usually hold one hundred eight bytes of data. Mbufs that contain a pointer to another buffer are called large mbufs (because the buffer is usually larger than the size of data a small mbuf can hold). A large mbuf's buffer cannot cross a page boundary, and thus is limited in size to one page. Mbufs can be linked together into an mbuf chain. An mbuf chain typically stores one packet. Chains of mbufs can be linked together to form a queue of packets.

There are two types of large mbufs: cluster mbufs and external mbufs. A cluster large mbuf points to a memory buffer that is managed by the kernel mbuf subsystem. An external mbuf points to an area of memory that is not managed by the mbuf system. External mbufs also contain a pointer to a "free" function that is called to free the external buffer. External mbufs allow network hardware devices that have their own private memory to "loan" it to the mbuf system for short periods of time. We also used external mbufs to help integrate UVM page loanout in the socket layer.

Before describing the modifications done to the socket layer to add page loanout, it is important to understand the socket layer data transmission code path. A process can request that data be transfered over a socket by passing the socket file descriptor, a pointer to the data buffer, and the buffer's length to the `write` system call. This system call causes the `sys_write` kernel function to be called. This function does two things. First, it wraps the specified buffer in a `uio` structure[3]. Second, it looks up the file operations structure (`fileops`) associated with the file descriptor passed to `write` and calls the appropriate "write" file operation for that file descriptor. There are currently two types of

[3]The `uio` structure is used within the kernel to describe one or more areas in an address space.

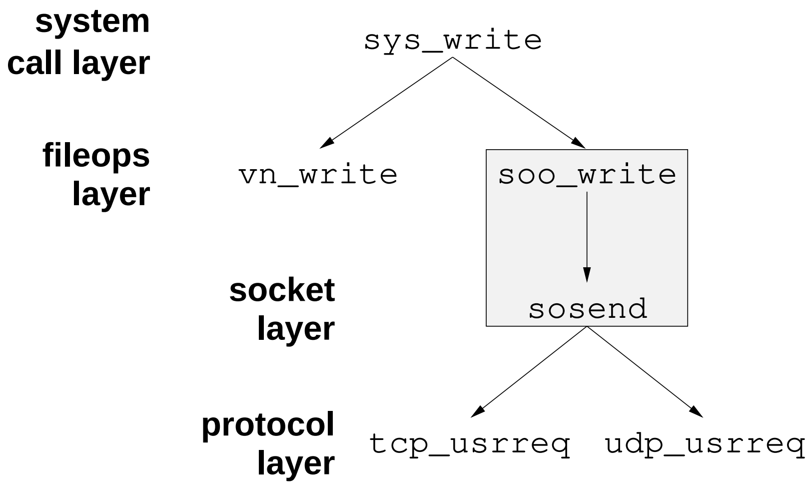


Figure 10.1: Socket `write` code path. The code in the box has been duplicated and modified to use page loanout for our measurements. The modified code is accessed by an alternate write system call.

file descriptors: sockets and vnodes. The write file operation for sockets is the `soo_write` function. The write operation for vnodes is `vn_write`.

The `soo_write` function is a one line function that translates the write call into a `sosend` call. The `sosend` function does the lion's share of the work of the socket layer including handling all the numerous options supported by the socket layer. If the `sosend` function determines that it ok to send data on the socket it will allocate an mbuf chain for the data, copy the data from the user buffer pointed to by the `uio` structure into the chain, and then pass the newly created mbuf chain to the protocol layer's user request function for transmission[4]. Once the socket layer has passed the data to the protocol layer, its role in the transmit process is complete. The code path for a `write` is shown in Figure 10.1.

To enable the socket layer to use page loanout rather than data copying for data transmission a second `write` system call was added to the kernel. Having two write system calls in the kernel allows us to easily switch between data copying and page loanout when taking measurements. The new write system call calls a modified version of `soo_write` when a write to a socket is detected. The alternate `soo_write` function calls `xsosend`. The `xsosend` function is a modified version of `sosend` that looks for large user buffers.

When `xsosend` detects a large user buffer, it uses UVM's page loanout to loan the user's pages out to kernel pages. It then maps those pages into the kernel address space,

[4]This is the `PRU_SEND` request.

and allocates an external mbuf for each one. Each external mbuf points to a special free function that will drop the loan once the pages have been transmitted.

10.1.2 Page Loanout Measurements

To measure the effect of page loanout we wrote a test program that uses the null protocol to transmit large chunks of data through the socket layer. The actions performed by the test program are similar to programs such as ftp, web, and video servers. Such programs operate by opening data files and transmit the file's content over the network.

The test program transmits data in one of two ways. It can use the normal `write` system call, in which case the data will be copied from user space into kernel buffers. Or it can use the modified `write` system, in which case the `xsosend` function will use UVM's page loanout to move the data.

The test program operates as follows:

1. The parameters are parsed.
2. A null socket is created for program output.
3. A buffer is allocated.
4. The entire buffer is written to the null socket in fixed sized chunks (the "write size") a specified number of times.

The test program takes as parameters the size of the buffer, the number of times the buffer is transmitted, and the amount of data to pass to the `write` system call (the write size).

The test program was run using a two megabyte buffer that was transmitted 1024 times. Thus, two gigabyte of data was transmitted for each run of the test program. Each run of the program was timed so that the bandwidth of the null socket could be determined. The write size was varied from 1K to 2048K. The results of the test using copying and page loanout are shown in Figure 10.2.

For write sizes that are less than the hardware page size (4K), copying the data produces a higher bandwidth than using page loanout. This is due to the page loanout mechanism being used to loan out pages that are only partially full of valid data. For example, when the write size is 1K for each page loaned out there is 3K of data in the page that is not being used. Once the write size reaches the page size, page loanout's bandwidth overtakes data copying. As the write size is increased to allow multiple pages to be transmitted in a single `write` call, the data copying bandwidth increases until the write

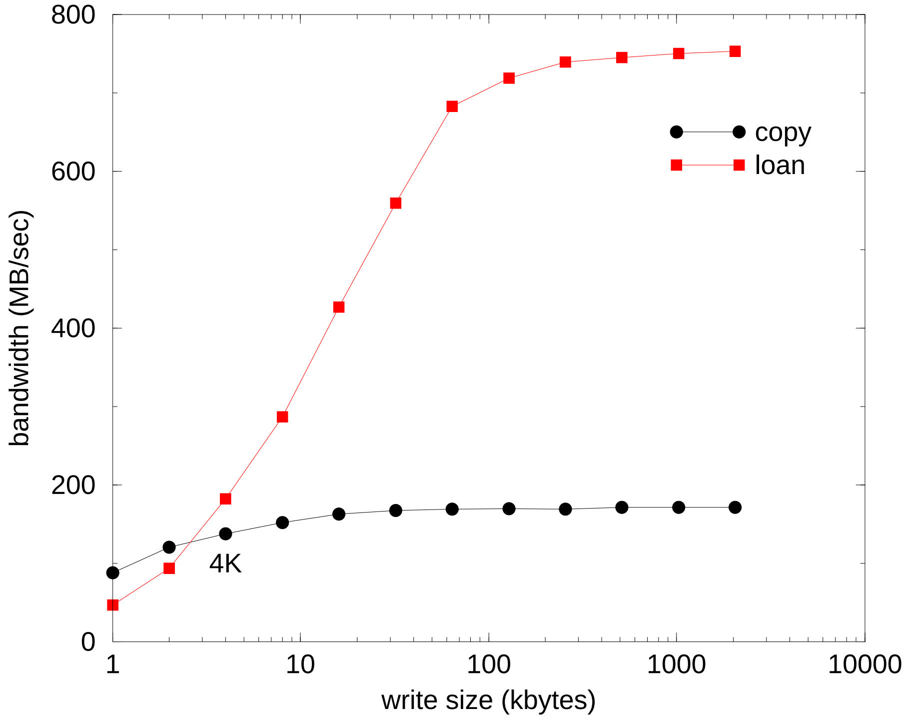


Figure 10.2: Comparison of copy and loan procedures for writing to a null socket

size reaches 32K. The data copy bandwidth never exceeds 172 MB/sec. Meanwhile, the loanout bandwidth rises sharply as the write size is increased. When the write size is 32K the loanout bandwidth is 560 MB/sec. The loanout bandwidth levels off at 750 MB/sec when the write size is 512K. Clearly, page loanout improved the I/O performance of the test program by reducing data copy overhead.

Note that the high bandwidth achieved by page loanout is through virtual memory based operations rather than through data copying. The results of the `lmbench` memory bandwidth benchmarks show that such high bandwidths cannot be achieved if each byte of the data is read or copied. Not all applications require the data to be accessed. For example, the multimedia video server described in [12] transmits data read from disk without reading or touching it and thus could benefit from page loanout.

10.2 Page Transfer Performance

Page transfer allows a process to receive pages of data from the kernel or another process. Pages received with page transfer are mapped into the receiving process' page transfer area of virtual memory. Once a page is transfered into a process' address space it becomes part of that process' anonymous memory and requires no further special treatment.

Page transfer moves data in the opposite direction that page loanout does. In page loanout, a process loans its data out to some other process or the kernel. In page transfer, a process receives pages from other processes or the kernel. The pages received may either be loaned from another process or their ownership may be donated to the receiving process. The benefit of using page transfer is that the transferred data is moved into the receiving process' virtual address space without the overhead of a data copy.

The effect of page transfer on the overhead of I/O operations can be measured using the kernel's networking subsystem in a way that is similar to the procedure used to measure the effect of page loanout. In a traditional system, when data is read from a socket it is copied from a kernel mbuf into the receiving process' virtual address space. With page transfer, the network read operation on larger chunks of data can be done by using page transfer to transfer a large mbuf's data area from the kernel directly into the receiving process' virtual address pace, bypassing the copy. For this to be feasible, the kernel should be configured so that the buffer size of a cluster mbuf is equal to the system page size[5].

Unfortunately, the measurement of the performance of page transfer on network I/O operations is effected by a similar problem to the one that that effected the page loanout measurements, namely that current processors can pass received data to user processes faster than the current generation of network interfaces can receive it. Thus any positive effects produced by page transfer will result in a reduction of processor load rather than an increased in-bound data bandwidth.

Fortunately, this problem can be worked around. Since the kernel's protocol and network interface layers have the same overhead regardless of whether or not page transfer is used, their overhead can be safely factored out of the page transfer performance bandwidth. This was achieved by introducing a new "null" socket read into the BSD kernel. When the socket layer detects a null socket read, it allocates an mbuf chain of the requested size (containing random data) and uses it to satisfy the socket read request. This allows us to directly measure the effect of page transfer on I/O overhead by observing the bandwidth of null socket reads.

[5]The current default cluster mbuf size on the i386 is 2048 bytes, or half a page. This value must be doubled in order to be able to use page transfer on cluster mbufs.

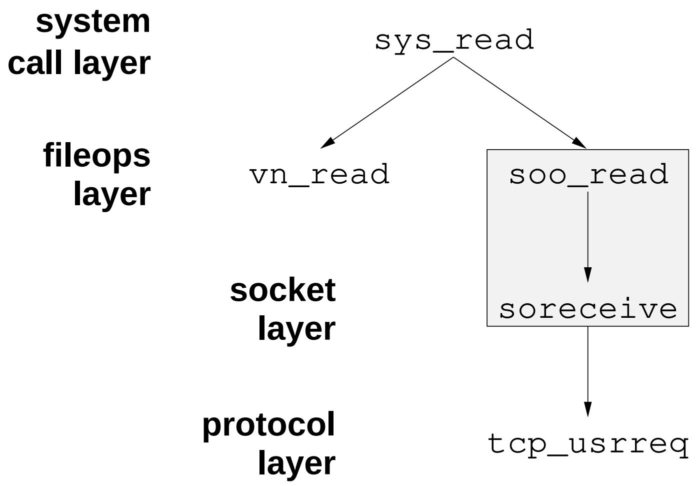


Figure 10.3: Socket `read` code path. The code in the box has been duplicated and modified to use page transfer for our measurements. The modified code is accessed by an alternate read system call.

10.2.1 Socket Layer Modifications

The socket layer of the BSD kernel had to be modified to support the new null socket read system call. The null socket read system call moves data either by copying it or by using page transfer. The code path for a socket read is shown in Figure 10.3. When the protocol layer receives data for a socket from the network interface layer it enqueues the data on the socket's receive socket buffer mbuf queue. This data waits in the socket buffer queue until either a process issues a read for the data or the socket is closed (in which case the data is discarded).

When a process issues a read on a socket, the `sys_read` kernel function is called. This function does two things. First, it wraps the pointer to the user's buffer up in a `uio` structure. Second, it looks up the file operations structure associated with the file descriptor passed to `read` and calls the appropriate "read" file operation for that file descriptor. The two read file operations functions are `soo_read` and `vn_read` (for sockets and vnodes, respectively).

The `soo_read` function is a one line function that translates the write call into a `soreceive` call. The `soreceive` function's job is to move data from a socket's receive buffer queue to the address space of the user who issued the receive request. If not enough data is available, the `soreceive` function may suspend the requesting process until more data is available. Once the `soreceive` function has moved the data, it may issue a user request to the protocol layer of the socket to inform the protocol that the user read some data. Sliding window protocols such as TCP require this callback so that they can update their window sizes as data is read out of the socket buffer. Other protocols such as UDP do not require this callback.

To implement the null socket read operation a second read system call ("`xread`") was added to the BSD kernel. This system call detects null read operations by checking for a special invalid file descriptor. If detected, the `xread` function calls a special version of `soreceive` to handle the null read request. Otherwise, `xread` uses its normal code path to handle the read request. The `xread` system call returns both the number of bytes read and a pointer to the virtual address where the pages were transfered (if page transfer was used).

The special null-read version of `soreceive` allocates an mbuf chain (with random data) large enough to satisfy the read request, and then moves the data to the user using the technique requested by the user (either data copying or page transfer). If page transfer is specified, a new function called `so_mcl_steal` is called to remove the page from a cluster mbuf. It performs the following actions:

1. Unused portions of the cluster mbuf's data area are zeroed to ensure data security.
2. A new anon and new page are allocated.
3. The new page is mapped into the cluster mbuf's data area and the old page is attached to the new anon.
4. The new anon is returned (it is ready to be transfered into the receiving processes' address space).

Once the data pages have been removed from the cluster mbufs and placed into anons, the anons can be transferred into the user's address space using the normal page transfer functions.

10.2.2 Page Transfer Measurements

To measure the effect of page transfer we wrote a test program that uses null socket reads to receive large chunks of data through the socket layer. Thus, the test program is similar to programs that receive large chunks of data from the network or from other processes.

The test program can request that the kernel use either data copying or page transfer to move the data. The test program operates as follows:

1. The parameters are parsed.

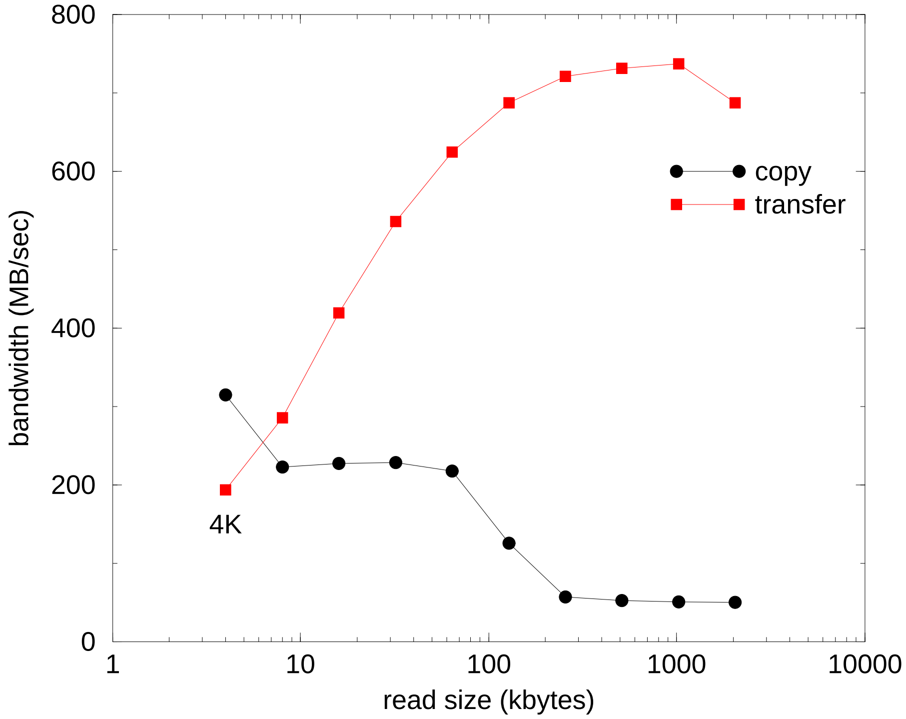


Figure 10.4: Comparison of copy and transfer procedures for reading from a null socket

2. A buffer is allocated if data copying is being used. This buffer will be used with the `read` system call (its size is called the "read size"). If page transfer is being used, then the operating system will provide the buffer.

3. The data is read using null socket reads in fixed sized chunks.

4. If page transfer is used, the transfered chunk of memory is freed using the `anflush` system call after the read operation.

The test program takes as input parameters the amount of data to read, and the number of bytes the program should read in one system call (the read size).

The test program was used to transfer 1GB of data first using data copying and then using page transfer. The read sized was varied from 4K (one page) to 2048K, and the bandwidth was measured. The results of the test are shown in Figure 10.4.

When copying data, smaller reads produce a greater bandwidth. As the read size increases, the null socket read bandwidth decreases from 315 MB/sec to 50 MB/sec. This decrease is due to data caching on the Pentium Pro. Memory accesses to the cache are significantly faster than memory accesses to main memory. If an application's buffer entirely fits within a cache then memory references to it will be faster than the bandwidth of main memory. For example, when the test program uses 4K buffers both the source and destination buffers of the data copy fit within the level-one cache producing a bandwidth of 315 MB/sec. When the size of the test program's buffers are increased to 8K they no longer fit within the level-one cache, but they do fit within the level-two cache. This results in a bandwidth drop to 225 MB/sec. The bandwidth remains nearly constant for read sizes from 8K to 64K. However, between 128K and 256K the bandwidth drops to 50 MB/sec as the size of the data buffers becomes larger than the level-two cache. Finally, the bandwidth eventually levels out at 50 MB/sec (the same value we got from `lmbench`'s memory copy benchmark). The 4K read size gets the full benefit of the high bandwidth of the Pentium Pro's level-one cache since it was the only process running on the system when the measurements were taken (so its data was always fresh in the cache).

On the other hand, the cache does not play as great a role in page transfer. The bandwidth starts around 190 MB/sec for page sized transfers, and then increases up to 730 MB/sec for a 1MB page transfer. The final dip in the page transfer curve is due to the cache. As the transfer size gets larger, the page transfer code touches more kernel data structures, and at some point this causes touched data structures to exceed the size of the level one cache.

Note that the high bandwidth achieved by page transfer is through virtual memory based operations rather than through data copying. Such high bandwidths cannot be achieved if each byte of the data is read or copied. Not all applications require the data to be accessed. For example, the multimedia video recorder used in our video server project receives data from an ATM network and immediately writes it to disk without reading or touching the data [12].

10.3 Using UVM to Transfer Data Between Processes

In the previous two sections we discussed the effect of page loanout and page transfer on the movement of data between a process and a network interface. In this section, we examine using UVM to move data between two processes using a pipe.

In a traditional system, a pipe is implemented as a pair of connected sockets. Thus, when data is written to a pipe the kernel uses the `sosend` function to transmit the data. The `sosend` function copies the data being written into an mbuf chain and places that chain on the receive socket buffer of the socket associated with the other end of the pipe.

When the data is read from the pipe with the `read` system call the `soreceive` function removes the mbuf chain from the receive socket buffer, copies the data from the chain to the receiver's memory, and then frees the mbuf chain. Thus, each byte of data written to the pipe is copied twice. UVM's page loanout and page transfer mechanisms can be used to reduce kernel overhead by eliminating both these copies.

10.3.1 Socket Layer Modifications

The kernel's socket layer must be further modified in order to use UVM to eliminate data copies when using pipes. While the page loanout modifications described in Section 10.1.1 are sufficient for use with a pipe, the page transfer modifications described in Section 10.2.1 for the null socket read operation are not.

In order to support page transfer for pipes, the `xread` system call was further modified to support a new read operation function that was added to the `fileops` structure. When page transfer is used, the virtual address to which the data pages have been transfered must be returned to the process performing the `xread` system call. The interface of the traditional read operation function has no way to return this address, but the new read operation does. We decided that for testing adding a second read operation to the `fileops` structure would be less disruptive to the system than modifying the original read operation.

The `xread` system call uses the new read operation function only if the file descriptor being read supports it and if page transfer was requested. The new read function for sockets is `soo_read2`. This function operates as follows:

1. The parameters are checked to see if they are favorable to page transfer. If they are not, then `soo_read2` uses the traditional `soo_read` function to move the data.

2. If there is no data to receive, then the `soo_read2` function causes the calling process to sleep until data arrives (unless the socket is in non-blocking mode, in which case an error is returned).

3. Once `soo_read2` has data, the mbuf chain the data resides in is checked to see if its data can be transferred using page transfer. In order to do this, the mbuf chain must consist of either cluster mbufs or of external mbufs whose data areas are pages loaned out from some other process. If the mbuf chain is not a candidate for page transfer, then `soo_read2` uses a traditional data copy to deliver the data.

4. For mbuf chains that can be used with page transfer, the `soo_read2` function calls the `so_transfer` function to transfer the data to the user process. It then returns the address to which the pages were transfered to the `xread` function.

The `so_transfer` function handles the page transfer of an mbuf chain. It operates as follows:

1. Space in the receiving process' page transfer area is reserved for the data.

2. The mbuf chain is then traversed. Each mbuf's data area is loaded into an anon. For cluster mbufs, `so_mcl_steal` is used (see Section 10.2.1). For external mbufs, the `so_mextloan_reloan` function is used (see below).

3. Finally, the anons are inserted into the receiving process' address space.

The reloan function is used for external mbufs whose data area is on loan from a process' address space due to page loanout caused by the `xwrite` system call. This function looks up the page associated with the mbuf. If the page is attached to an anon, then the anon's reference count is incremented. Otherwise, a new anon is allocated and the page is attached to it. The reloan function returns the anon.

These changes allow a process to receive data from a pipe using page transfer.

10.3.2 Process Data Transfer Measurements

To measure the effect of both page loanout and page transfer on the transfer of data between two processes over a pipe, we wrote a test program that operates as follows:

1. The parameters are parsed and a pipe is created.

2. A child process is forked off.

3. The child process writes a specified amount of data into the pipe, using page loanout if requested. Once the data is written the child process closes the pipe and exits.

4. The parent process reads data out of the pipe, using page transfer if requested. After a read operation, the data is released using the `anflush` system call. Once all data is read, the parent process exits.

The test program takes as parameters the amount of data to transfer over the pipe and the read and write sizes.

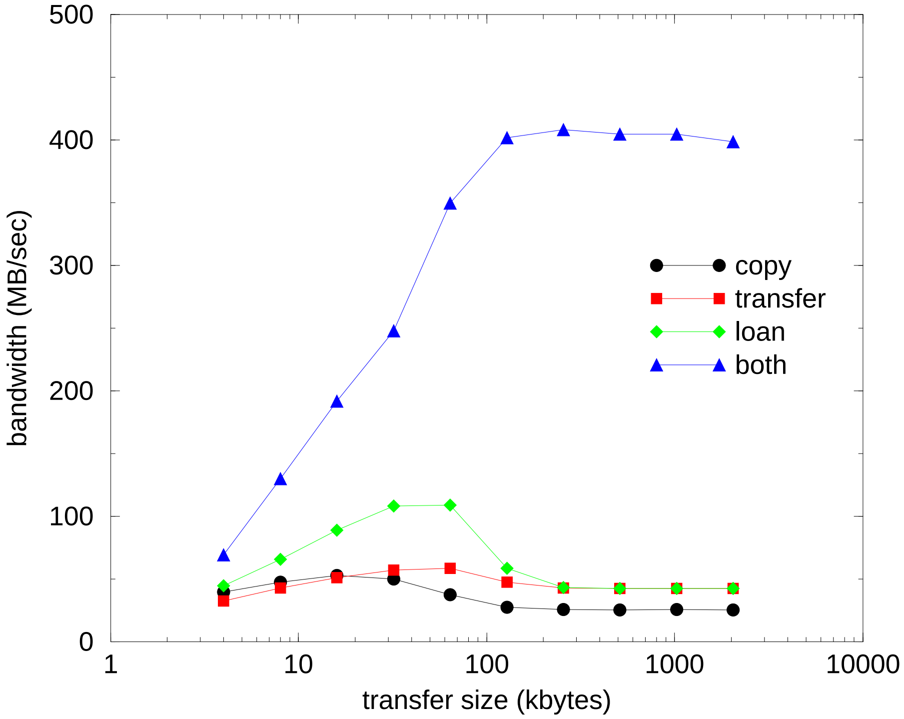


Figure 10.5: Comparison of copy, transfer, loan, and both procedures for moving data over a pipe

We used the test program to produce four sets of data. For each set of data we transfered 1GB of data varying the read and write sizes from 4K to 2048K. The four data sets are:

copy: Data was copied on both the sending and receiving ends of the pipe.

transfer: Data was copied on the sending side and moved with page transfer on the receiving side of the pipe.

loan: Data was loaned out on the sending side and copied on the receiving side of the pipe.

both: Data was loaned out on the sending side and moved with page transfer on the receiving side of the pipe.

The results of this test are shown in Figure 10.5.

For page sized data transfers (4K) copying produces a bandwidth of 40 MB/sec, while page transfer and page loanout produce bandwidths of 33 MB/sec and 45 MB/sec, respectively. Using both page transfer and page loanout at the same time with page sized transfers produces a bandwidth of 70 MB/sec.

As the transfer size is increased, the bandwidth of data copying drops and levels off at 26 MB/sec due to the effects of the cache. Meanwhile, the bandwidth of both transfer and loanout levels off at 43 MB/sec as the transfer size is increased. Since both these tests still have data copying at one end of the pipe, they are effected by caching in the same way as data copying, however their bandwidth is almost double that of copying because they are only touching the data on one end. The page loanout curve rises higher than the page transfer curve in mid-range transfer sizes because page loanout has less overhead than page transfer. Page transfer has the added complication of having to replace the data page removed from cluster mbufs with another page. The bandwidth of the test in which both loanout and transfer are used rises sharply and levels off at 400 MB/sec for large transfer sizes. In this case the data is not touched or copied at all, instead a copy-on-write mapping to the sending process' page is added to the receiving process.

10.4 Map Entry Passing Performance

Map entry passing allows a process to send a part of its virtual address space to another process. The receiving process can share the received virtual space with the parent, receive a copy-on-write copy of the virtual space from the parent, or take over full ownership of the virtual space.

Map entry passing is achieved using the export/import system call interface described in Section 7.3.1. No further kernel changes are necessary to test it. In this section we compare the performance of using map entry passing to move data to using data copying.

10.4.1 Data Exchange With Map Entry Passing Measurements

Data passing between programs is quite common on Unix-like systems. Many programs communicate through pipes or sockets. If an application passes large amounts of data is over a pipe, it might benefit from map entry passing. To measure the effect of map entry passing, we wrote two test programs that pass a fixed-sized chunk of data between two processes a specified number of times. One test program passes data between processes by sending it over a local socket. This causes the kernel to copy the data from the sending process into a mbuf chain, and then to copy it from the mbuf chain to the receiving process. The other test program passes data between processes by using map entry passing. The test program can optionally “touch” the data after it arrives to simulate data processing.

The test programs operate as follows:

1. The parameters are parsed and buffers are allocated.
2. A pair of connected sockets is created.
3. A child process is forked off.
4. For the test program that uses copying to move the data, the parent process initializes the buffer (if data "touching" is requested), and writes the buffer into the socket. The child process then reads the data from the socket. If data touching is requested, then the child process modifies the data. Then the child process writes the data back into its end of the socket. The parent process reads the buffer back. This entire process is repeated for the specified number of times.
5. For the test program that uses map entry passing to move the data, the procedure is similar to the data copying procedure except that the export and import system calls are used. The parent process initializes the data (if touching), exports the buffer, and writes the export tag to the socket. The child process reads the export tag from the socket and uses that to import the buffer from the parent process (thus removing it from the parent's address space). The child process then modifies the data buffer (if touching), exports the buffer, and writes the export tag to the socket. The parent reads the tag from the socket and imports the buffer back from the child. This entire process is repeated for the specified number of times.

Both test programs take as parameters the size of the buffer to transfer and the number of times the buffer should be transfered.

Each test program was run using transfer sizes ranging from 4K to 4096K. The number of times the data was transferred from parent to child process and back was adjusted so that the test program ran for at least thirty seconds. Each run of the program was timed so that the bandwidth could be determined.

As shown in Figure 10.6, the bandwidth curves for the test program that uses data copying via the socket pair start off near 35 MB/sec for a one page transfer size. The bandwidth decreases as the transfer size increases due to caching effects. The bandwidth of the data-copying tests level off at 12 MB/sec. On the other hand, the bandwidth curve for the map entry passing test program peaks around 96 MB/sec for a transfer size of 64K and then caching effects on kernel VM data structures cause it to fall and level off at 34 MB/sec.

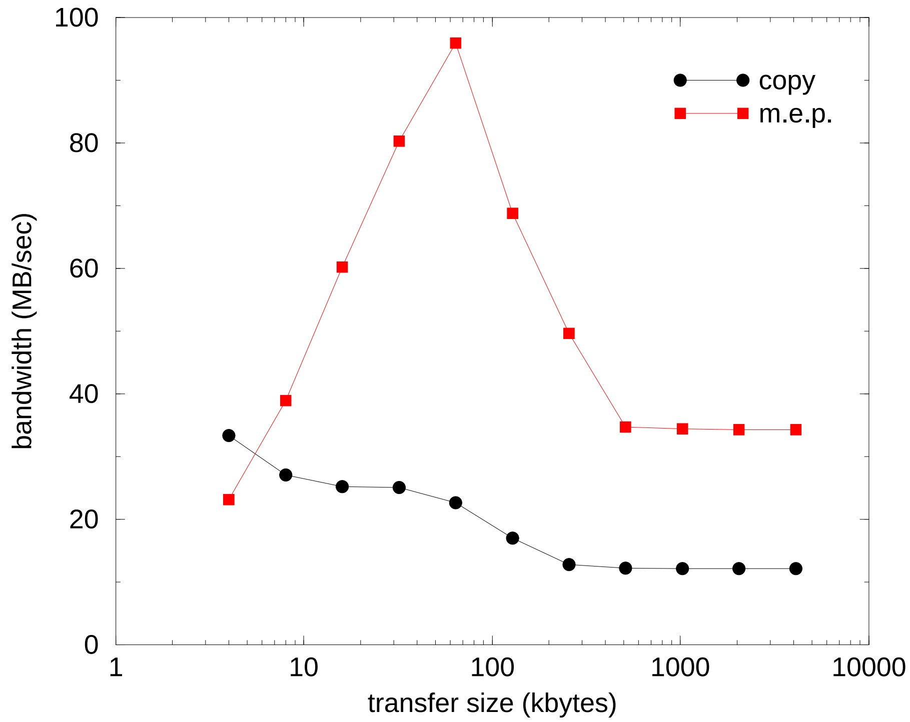


Figure 10.6: Comparison of copy and map entry passing (m.e.p.) transferring memory between two processes

If the application does not touch the data after it is transfered then the bandwidth for data copying levels off at 18 MB/sec rather than 12 MB/sec. On the other hand, if the map entry passing test program does not touch the data, then the bandwidth rises rapidly, reaching 18500 MB/sec for a transfer size of 4096K. This high bandwidth occurs because the data being transfered is accessed neither by the application program nor the kernel, so the data is never loaded into the processor from main memory.

10.4.2 Data Pipeline With Map Entry Passing and Page Loanout

Another test program we wrote combines the use of map entry passing with page loanout. The test program consists of three processes. The first process allocates and initializes a data buffer of a specified sized and then sends it to the second process. The second process modifies the data and then sends it to the third process. The third process receives the data and writes it to a null protocol socket. Once the data has been written to the null protocol socket, then the first process can generate another buffers worth of data. Such

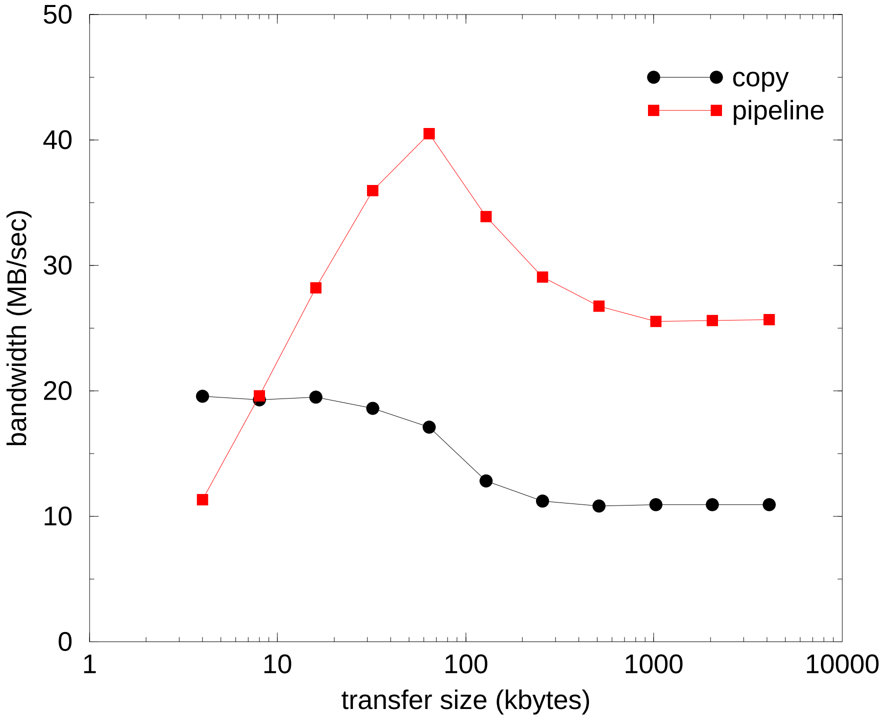


Figure 10.7: Comparison of copy and map entry passing/page loanout pipeline

a data pipeline application is similar to the multimedia "I/O-pipeline model" described in [53].

In the test program the data can be exchanged between the processes either with data copying or with map entry passing. The final process can write the data out with either data copying or page loanout. We ran the test program with transfer sizes from 4K (one page) to 4096K. The number of transfers was adjusted so that the test program ran for at least thirty seconds. The results of the the tests are shown in Figure 10.7.

When using data copying, the bandwidth started at 20 MB/sec and then dropped and leveled off at 11 MB/sec once the cache size was exceeded. When using map entry passing and page loanout, the bandwidth starts at 11 MB/sec for a page-sized transfer, and increases to a peak of 40 MB/sec for a 64K transfer size. It then levels off at 26 MB/sec as the kernel data structures exceed the size of the cache.

10.5 Secondary Design Element Performance

In this section we examine the effect of some of UVM's secondary design elements on the performance of the BSD kernel. We discuss process creation and deletion, clustered pageout, resident page faulting, and map entry fragmentation.

10.5.1 Process Creation and Deletion

When a process is created, the virtual memory system must allocate and initialize a virtual address space for the new process. A process can create a child process by using either the `fork` or `vfork` system calls. In a `fork` system call, the child process' new address space is created immediately. In a `vfork` system call, the child process borrows the parent process' address space. The parent process is suspended until the child process either creates a new address space by executing another program or exits. The `vfork` operation is more efficient for programs such as shells that fork off child processes that immediately execute another program because copy-on-write operations for the parent process' address space are bypassed. Address spaces are freed when a process exits.

A large part of the overhead associated with creating and deleting processes can be attributed to the virtual memory system. For example, new processes must have their maps initialized based on the inheritance values of the parent process' map. For regions with copy inheritance, all mappings in the parent process' address space must be write protected to ensure proper copy-on-write semantics.

To compare the overhead of BSD VM and UVM in the area of process creation and deletion we wrote a small test program. The test program operates as follows:

1. The parameters are parsed.
2. A fixed size chunk of anonymous memory is allocated and zeroed.
3. The test program then repeatedly uses `fork` or `vfork` to create a child process. The child process immediately exits.
4. The time a single fork takes is computed.

The test program takes as parameters the size of anonymous memory to allocate, whether to use `fork` or `vfork`, and whether or not to touch the anonymous memory after each `fork` operation.

The test program was run under both BSD VM and UVM for memory allocations of 0MB to 16MB. We had the test program perform 10,000 forks for each allocation size.

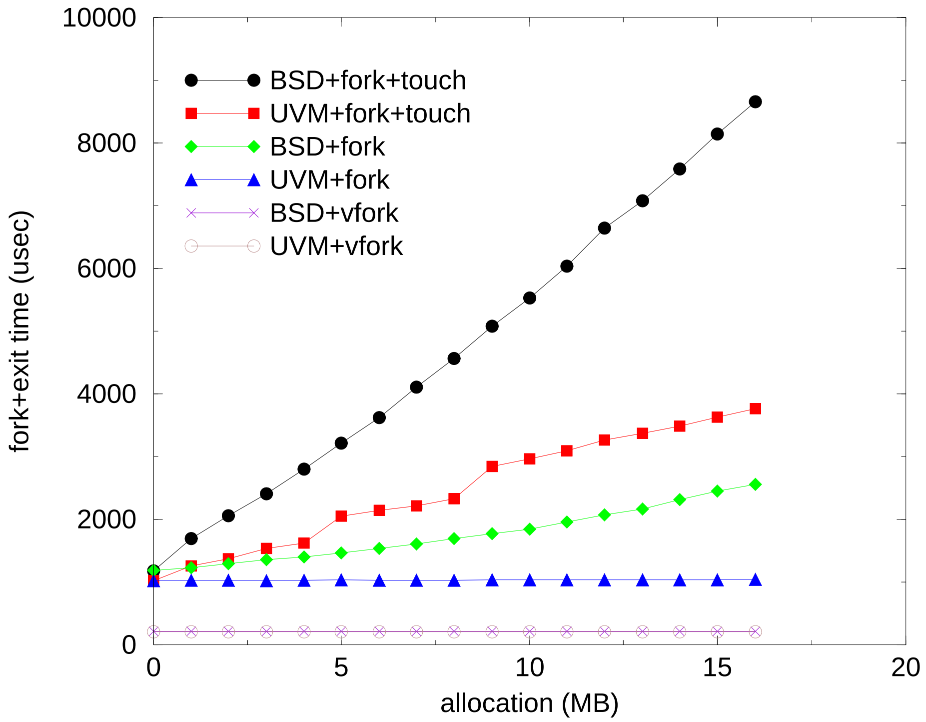


Figure 10.8: Process creation and deletion overhead under BSD VM and UVM

For the `fork` operation, we ran the test program with and without touching the allocated memory area. The results of this test are shown in Figure 10.8.

For `vfork` operations, the time it took for a process to fork and exit remained constant regardless of the memory allocation for both BSD VM (218 microseconds) and UVM (210 microseconds). This was expected because with `vfork` the child process never gets an address space of its own. Note that a normal `fork` operation takes at least five times as much time to complete as a `vfork` operation.

For `fork` operations where the allocated data is not touched for each `fork` operation there is a major difference between BSD VM and UVM. For UVM, the fork and exit time remained constant at 1035 microseconds. On the other hand, the time for BSD VM grows with the size of the memory allocation. UVM's behavior is easy to explain. At the time of the first `fork` operation, the map entry in the parent's map that maps the allocated area of memory is write protected and marked "needs-copy." Since the parent process never writes to the allocated area memory between `fork` operations, the "needs-copy" flag remains set. This indicates that the allocated area of memory is already set for copy-on-write. As explained in Section 4.6 and Section 4.7, the UVM `fork` code takes advantage of this to avoid repeating the copy-on-write setup operation, thus bypassing the overhead associated with the memory allocation. On the other hand, the BSD VM code appears to perform some shadow-object chain related operations regardless of the needs-copy state of its map entry, thus contributing to the overhead of the fork operation.

If the allocated data is touched between `fork` operations, then needs-copy gets cleared and both virtual memory systems are forced to do all copy-on-write related operations on the allocated memory each time through the loop. In this case the time for a fork and exit for both BSD VM and UVM increases with the size of the memory allocation. For an allocation of zero, the times for BSD VM and UVM are 1177 microseconds and 1025 microseconds, respectively. For an allocation of 16MB, the loop times for BSD VM and UVM rise to 8652 microseconds and 3764 microseconds, respectively. BSD VM's fork processing time increases more rapidly than UVM's due to the expense of maintaining object chains.

10.5.2 Clustered Pageout

We also measured the performance of UVM's clustered anonymous memory pageout routines (described earlier in Section 8.2). The performance of anonymous memory pageout effects how quickly an operating system responds to sudden demands for memory from applications that make heavy use of virtual memory such as Lisp programs. We wrote a test program that allocates a large fixed-sized memory buffer and then writes to each page of the buffer. We then timed how long it took for the program to completely initialize the buffer.

For our test runs, the program's buffer size was varied from one megabyte to fifty megabytes. Each test was run on a freshly booted system with thirty-two megabytes of RAM that was running only an X server with minimal applications. The results of our test are shown in Figure 10.9.

Since our test system had thirty-two megabytes of RAM, it did not start paging until the memory allocation reached twenty megabytes. As can be seen from the figure, UVM takes significantly less time to pageout memory than BSD VM. For example, it took the test program sixty-five seconds to zero a fifty megabyte buffer, where as it took UVM only eleven seconds. One reason BSD VM performs so poorly is because it does not cluster pageout, where as UVM can always cluster by dynamically relocating anonymous memory on backing store as needed to form large clusters. If BSD VM had object based

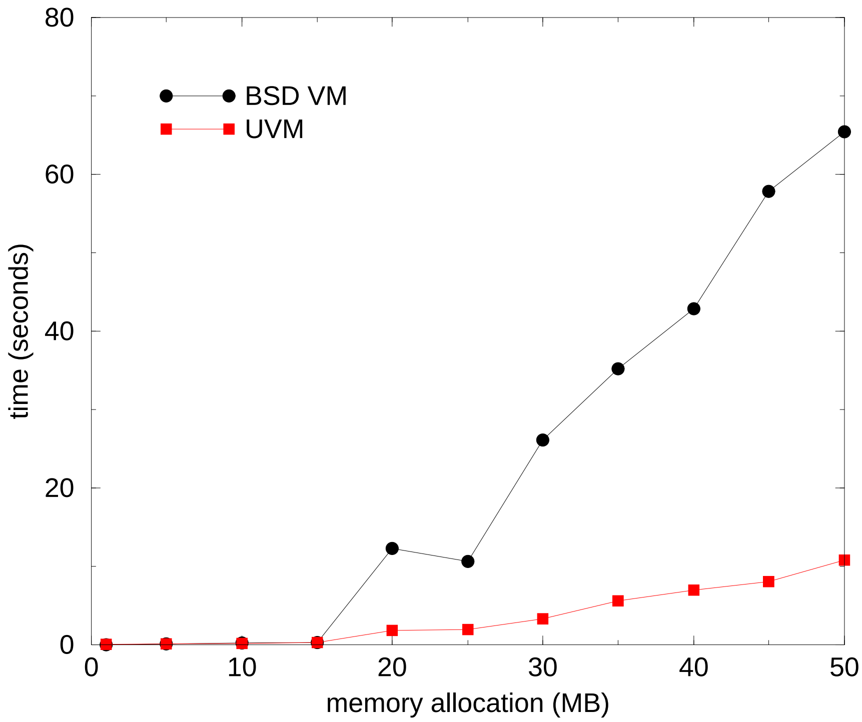


Figure 10.9: Memory allocation time under BSD VM and UVM

clustering its performance would improve, provided that the pagedaemon found contiguous dirty pages to build clusters around.

10.5.3 Resident Page Faulting

The number of times the fault routine is called under UVM is less than under BSD VM. In Section 5.1.5 we described how the UVM fault routine maps in neighboring resident pages to reduce the number of page faults that a program takes. On the other hand, the BSD VM fault routine never maps in any additional pages. Furthermore, the BSD VM fault routine is also used to fault user-level page tables into a process' address space for the old i386 pmap module. Table 10.1 shows some fault counts for a few small applications. Note that UVM currently only reduces the page fault count for pages that are already in memory (resident). Once the BSD I/O system is modified to support asynchronous pagein UVM could be easily modified to start pagein I/O operations for non-resident pages that are expected to be needed soon.

Table 10.1: Page fault counts. The `cc` command was run on a "hello world" program.

Command	BSD VM Faults	UVM Faults
`ls /`	59	33
`finger chuck`	128	74
`cc`	1086	590
`man csh`	114	64
`newaliases`	229	127

10.5.4 Map Entry Fragmentation

In Section 8.8 and Section 8.9 we discussed how UVM avoids map entry fragmentation in kernel maps and for wired memory. Map entry fragmentation is undesirable because it wastes kernel memory and increases virtual address lookup time in VM maps. To verify that UVM reduces map entry fragmentation, we modified both UVM and BSD VM to keep count of the number of map entries currently allocated. We counted both the total number of allocated map entries, and the number of statically allocated map entries that are used exclusively by the kernel.

We measured the number of allocated map entries in five cases, with a varying number of active processes. These cases are:

1. in single user mode — four processes.
2. after booting multiuser, before logging in — eighteen processes.
3. after logging in and starting X — twenty-seven processes.
4. running a single "`cat`" process from X — twenty-eight processes.
5. running ten "`cat`" processes from X — thirty-seven processes.

When measuring the number of map entries under BSD VM, we did not count the map entries allocated for the "buffer map" (see Section 8.10.1). The results of these measurements are shown in Table 10.2.

The number of statically allocated map entries under UVM remains nearly constant (seventeen or eighteen) as the number of processes on the system increases. On the other hand, the number of statically allocated map entries under BSD VM grows with the number of processes, reaching ninety-four entries when there are thirty-seven processes active on

Table 10.2: Comparison of the number of allocated map entries (the BSD VM numbers do not include 859 dynamically allocated map entries used for the buffer map)

System State	**Number of Processes**	**Number of Allocated Map Entries**			
		Static		**Total**	
		BSD	**UVM**	**BSD**	**UVM**
Single-user prompt	4	28	14	50	26
multiuser, no logins yet	18	57	17	400	242
after starting X	27	74	18	675	428
running 1 "`cat`"	28	76	17	686	433
running 10 "`cat`"	37	94	17	785	487

the system. This is due to the way BSD VM wires each process' "`user`" structure into kernel memory.

The total number of allocated map entries rises with the number of active processes for both BSD VM and UVM, with BSD VM's map entry allocation rising more rapidly than UVM's. For example, for thirty-seven processes BSD VM requires 785 map entries, while UVM requires only 487.

10.6 UVM Compared to Related Work

Comparing the performance of UVM data movement mechanisms with related work is difficult due to the wide variety of hardware platforms and different testing procedures used across projects. However, we can make some rough estimates of relative performance improvements based on our results and results published in the literature. For example, the Solaris zero-copy TCP mechanism with checksum in hardware [14] produces a factor of 1.4 improvement over data copying. UVM's page loanout facility (without the overhead of protocol processing) produces a factor of 2.6 improvement for the same transfer size. In the Container Shipping project [3] the performance of an IPC pipe using the container shipping mechanism to send, receive, and both send and receive was compared to the performance of using data copying. The reported improvement factors are 1.2, 1.6, and 8.3, respectively. A comparable experiment under UVM using page loanout, page transfer, and both mechanisms at the same time produced improvement factors of 2.1, 1.7, and 14. Other projects report results that appear to be comparable. Our rough comparisons show that UVM produces improvements on the same order of magnitude as the related work.

10.7 Summary

In this chapter we have described a set of tests that show the improved performance of UVM. We showed that page loanout and transfer tests using null sockets and protocols can achieve greater bandwidth when moving page-sized chunks of data. We showed that page loanout and page transfer can be used to improve the performance of a pipe. We showed that map entry passing can be used to quickly transfer part of a virtual address space between processes, and that map entry passing can be combined with page loanout in a data pipeline. We also showed how UVM's secondary design elements improve BSD kernel performance for forking, clustered pageout, faulting, and map entry allocation.

Applications that move large chunks of data can benefit by using one or more of UVM's three data movement mechanisms instead of data copying. The page loanout mechanism can be used without an API change if the kernel is modified to automatically use page loanout rather than a data copy for large requests. The page transfer mechanism can also be used without an API change when the data buffer is properly aligned. To take full benefit of all three mechanisms new system calls should be added to the kernel that allow a process to explicitly control its data movement.

To derive maximum benefit from UVM's new mechanisms applications programmers should note the following guidelines:

- Because applications using UVM's data movement mechanisms benefit most when the size of the data being transferred is larger than the cache, we recommend that applications that use UVM's mechanisms use large multi-page buffers. For example, consider a video server application that transmits a memory mapped video file over the network using page loanout. Rather than transmitting the file one page at a time, the application should transmit it in large multi-page chunks so that UVM will loan out multiple pages at the same time. On the other hand, applications that do not use UVM's new mechanisms (and therefore must copy data) should choose data buffer sizes so that both the source and destination buffers are small enough to fit in the cache. This allows programs to take advantage of the speed of cache memory.

- Map entry passing should be used for interprocess communication, and not for transferring memory to hardware devices. Map entry passing is preferable to page loanout and page transfer when memory is to be shared or when the region being transferred is very large (multi-megabyte) or contains unmapped areas.

- Page loanout and page transfer should be used when passing copies of data between processes or when passing copies of data between processes and devices. These mechanisms should be used for interprocess communication when the region being transferred is a medium-sized contiguous region of virtual memory.

Chapter 11

Conclusions and Future Research

In this dissertation we have presented the design and implementation of the UVM virtual memory system. UVM was designed to meet the needs of new computer applications that require data to be moved in bulk through the kernel's I/O system. Our novel approach focuses on allowing processes to pass memory to and from other processes and the kernel, and to share memory. The memory passed or shared may be a block of pages, or it may be a chunk of virtual memory space that may not be entirely mapped. This approach reduces or eliminates the need to copy data thus reducing the time spent within the kernel and freeing up cycles for application processing. Unlike the approaches that focus exclusively on the networking subsystem, our approach provides a general solution that can improve efficiency of the entire I/O subsystem. UVM also improves the kernel's overall performance in other key areas. In this chapter we describe the contributions of the UVM project and directions for future research.

11.1 Contributions

UVM provides the BSD kernel with three new mechanisms that allow processes to exchange and share paged-sized chunks of memory without data copies. These three new mechanisms are page loanout, page transfer, and map entry passing. The design of these mechanisms was based on several key principles that we believe should be applicable to other operating systems:

- First, although traditional virtual memory systems operate at a mapping-level granularity, we believe that page granular operations are also important and should be provided. This gives kernel subsystems such as the I/O and IPC system the ability to add and extract pages from a process' address space without disrupting or fragmenting the high-level address space mappings.

- Second, a virtual memory system should be designed to avoid complex memory lookup mechanisms. Simple memory lookup mechanisms are more efficient and ease implementation of VM-based data movement mechanisms. For example, UVM's two-level memory mapping scheme improves performance over BSD object chains and simplifies traditional functions such as the fault routine. It also eased the implementation of UVM's page granulary data movement mechanisms.

- Third, a virtual memory system should be designed in such a way that they allow their pages to be shared easily between processes, the virtual memory layer, and kernel subsystems. This allows data to be moved throughout the operating system without it having to be copied. While, BSD VM assumes that it always has complete control over virtual memory pages, UVM does not.

While it would be foolish to use UVM's data movement mechanisms on small data chunks, our measurements clearly show that for larger chunks of data they provide a significant reduction in kernel overhead as compared to data copying. For example:

- UVM's page loanout mechanism provides processes with the ability to avoid costly data copies by loaning a copy-on-write copy of their memory out to the kernel or other processes. Our measurements show that loaning a single 4K page's worth of data out to the kernel rather than copying it can increase bandwidth by 35%. Larger sized chunks of data provide an even greater performance improvement. For example, loaning out a 2MB buffer rather than copying it increases the bandwidth by a factor of 4.5.

- UVM's page transfer mechanism allows a process to receive anonymous pages of memory from other processes or the kernel without having to copy the data. Once a page has been received with page transfer it becomes a normal page of anonymous memory and requires no further special treatment from UVM. Our measurements show that page transfer pays when transferring two or more pages of data. A two-page page transfer increases the bandwidth over copying by 28%. Increasing the page transfer size to 1MB (256 pages) yields a bandwidth that is 14 times greater than the data copying bandwidth.

- In addition to being used separately, we have shown that page transfer and page loanout can be used together to improve the bandwidth delivered by a pipe by a factor of 1.7 for page-sized chunks of data, and by a factor of 16 for 1MB-sized chunks of data.

- UVM's map entry passing mechanism allows processes to easily copy, share, and exchange chunks of virtual memory from their address space. Measurements show that map entry passing between two processes is more efficient than data copying for two or more pages. For a two-page exchange, map entry passing provides a 44% improvement over data copying. For a 512K (128 page) data exchange, map entry passing outperforms data copying by factor of 2.8.

- We have also shown that map entry passing can be combined with page loanout to improve the performance of a data pipeline application. Our measurements show that the pipeline using map entry passing and page loanout performs better than data copying for data chunks of two or more pages. While map entry passing and page loanout yield a modest 2% improvement for data chunks of two pages, they improve performance by a factor of 2.4 for sixteen-page data chunks.

Our measurements also clearly show the effects of the kernel copying large buffers of data multiple times on the cache. These data copies effectively flush all the data out of a system's cache, thus causing it to have to be reloaded. UVM's new mechanisms eliminate this cache-flushing effect.

In addition to providing new features, UVM also improves the BSD kernel's performance in traditional areas. These areas include the following:

- Forking time has been reduced under UVM. The `vfork` system call time has been reduced by 4%. The `fork` system call time has been reduced by 13% for small processes. As the process size increases, UVM's improvement increases as well. For example, a process with a 16MB region forks 56% faster under UVM than under BSD VM.

- Pageout time has been reduced under UVM. For example, on our 32MB system the time to satisfy a memory allocation of 50MB has been reduced by 83%.

- The number of calls to the pagefault routine has been reduced under UVM. For example, when compiling a "hello world" program the page fault routine is called 46% less often under UVM.

- Map entry fragmentation has been greatly reduced under UVM. Static kernel map entries are no longer allocated each time a process is created under UVM. And UVM's improved handling of wired memory reduces the number of map entries required for traditional processes. For example, a system freshly booted multiuser under UVM uses 40% fewer map entries than it would under BSD VM.

UVM also includes numerous design improvements over BSD VM. These improvements include: the removal of object chaining and fictitious pages, the merging of the vnode and VM object cache, the removal of deadlock conditions in the pmap, the elimination of the troublesome "ptdi" panic condition, a unified mechanism to register the configuration of physical memory with the VM system, and new unprivileged system calls that allow users to obtain VM status without the need for "root" access.

In addition, UVM's well documented implementation methods show how a project of this size can be accomplished by a single person.

UVM is now part of the standard NetBSD distribution and is scheduled to replace the BSD VM system for NetBSD release 1.4. UVM already runs on almost all of NetBSD's platforms and is expected to run on every NetBSD platform soon. A port to OpenBSD is also expected.

11.2 Future Research

UVM represents a major step forward from the BSD VM system. Although much has been accomplished, there are still plenty of tasks that need attention. Future research could be done in the following areas:

- The kernel could be modified to dynamically decide if page loanout or page transfer would be more efficient than data copying based on the size of the data being transfered and the available memory resources. Page loanout and page transfer should be configured in such a way that they are options that the kernel can choose if there is an advantage to be gained.

- The performance of page loanout and page transfer on systems that have virtually addressed caches (VACs) should be evaluated. VACs cache data based on virtual address rather than physical address. The advantage of a VAC is that data can be fetched from the cache without having the MMU perform an address translation. However, a major drawback of using VACs is that ambiguous situations can develop if a single page of memory is mapped to more than one physical address. In this case, the VAC may cache the same data from the physical page of memory in more than one place.

On systems with VACs the VM system must do extra work to ensure that such ambiguous situations do not occur. Sometimes this problem can be avoided by ensuring that all of a page's mappings are aligned to a VAC boundary, but for other cases the only solution is to either uncache all mappings of a page or to kick out all the mappings except one. Since page loanout and page transfer can cause a single page to have multiple mappings a VAC could effect the performance of these mechanisms.

As we expect operating system software to make more use of multiply-mapped page as time goes on, we hope that systems with VACs become less common because of the burden they place on the VM system.

- Applications should be adapted to take advantage of new UVM features to improve their performance. Some features such as page loanout can be used without having to change the system call API. However, features such as page transfer and map entry passing require the use of new or modified system calls. We believe that such system call API changes can be hidden behind a properly designed middleware layer such as the ACE wrappers [60], although this has yet to be seen.

- UVM currently operates in a kernel that has separate fixed sized VM and buffer cache. These two caches need to be merged to provide the operating system with more flexibility. Merging the two caches will require a new I/O layer in the VM system.

- UVM should be ported to a multiprocessor system so that its fine-grain locking system can be fully tested.

The UVM system is well organized and documented, thus making it well suited for additional enhancements. Given the work we've already accomplished with UVM, taking on these tasks for future research should be an interesting, but not insurmountable challenge. Additional efforts in this area will allow applications to take full advantage of the benefits of UVM.

Appendix A

Glossary

address space: a numeric range that identifies a portion of physical or virtual memory.

aobj: a `uvm_object` that contains anonymous memory.

amap: a data structure also known as an anonymous map. An amap structure contains pointers to a set of anons that are mapped together in virtual memory.

anonymous memory: memory that is freed as soon as it is no longer referenced. This memory is referred to as anonymous because it is not associated with a file and thus does not have a file name. Anonymous memory is paged out to the "swap" area when memory is scarce.

anon: a data structure that describes a single page of anonymous memory. Information contained in an anon includes a reference counter and the current location of the data (i.e. in RAM or in backing store).

aref: a data structure that is part of a map entry structure and contains a reference and an offset to an amap structure.

BSD VM: the VM system imported into 4.4BSD from the Mach operating system.

backing store: a non-volatile area of memory, typically on disk, where data is stored when it is not resident. Examples of backing store include plain files and the system's swap area.

bss area: the area of a process' virtual address space where un-initialized data is placed. The VM system fills this area with zeroed pages as it is referenced.

busy page: a page that should not be accessed because it is being read or written.

center page: the page around which a cluster is built.

chunking, amap: the process of breaking up the allocation of large amaps into smaller allocations.

clean page: a page whose data has not been modified.

clustered I/O: I/O in which operations on smaller contiguous regions are merged into a single large I/O operation.

copy-on-write: a way of using the VM system to minimize the amount of data that must be copied. The actual copy of each page of data is deferred until it is first written in hopes that it will not be written at all. There are two forms of copy-on-write: private copy-on-write and copy copy-on-write.

copy copy-on-write: a form of copy-on-write where the copy is a complete snapshot of a memory object at the time of the copy. Changes made to the backing object after the copy are never seen.

copy object: in the BSD VM, an anonymous memory object that stores the original version of a changed page in a copied object. Copy objects are necessary to support the non-standard copy copy-on-write semantics. A list of objects connected by their copy object pointers is called a **copy object chain**.

data area: the area of a process' virtual address space where initialized data is placed. The data area comes from the file containing the program being run and is mapped copy-on-write.

dirty page: a page whose data has been modified. Dirty pages must be cleaned before they can be paged out.

faultinfo: a data structure used to store the state of a page fault.

fictitious page: a `vm_page` structure used by the BSD VM system to refer to device memory.

hardware exception: when a running program encounters a condition — which it may or may not have caused — that prevents it from continuing to run. The kernel must resolve the problem or kill the process. This is also referred to as a trap.

heap: the area of a process' virtual address space from which dynamic memory is allocated (e.g. with `malloc`). The heap is an extension of the bss area and it grows as the process allocates more memory.

kernel: the program that controls a computer's hardware and software resources. The kernel has several subsystems including networking, I/O, process scheduling, and virtual memory. In a Unix-like operating system the kernel is stored in a file such as `/vmunix`, or `/netbsd`. The kernel is part of an operating system.

Mach: an operating system developed at Carnegie-Mellon University. Mach's VM system was imported into BSD.

memory management unit (MMU): the part of a computer's hardware that translates virtual addresses to physical addresses.

memory mapped object: a memory object that has pages which are currently mapped into a user's address space. Examples of memory mapped objects include **memory mapped files** and **memory mapped devices**.

memory object: any kernel data structure that represents data that can be mapped into a virtual address space.

object collapse problem: a problem with the BSD VM system where memory remains allocated even though it can no longer be accessed. This eventually leads to the kernel running out of swap space.

operating system: the kernel and associated system programs that come with it (e.g. system utilities, windowing system, etc.).

page fault: a hardware exception triggered by the MMU when virtual memory is accessed in a way that is inconsistent with how it is mapped. Page faults may be triggered, for example, by accessing a virtual address that is not currently mapped to physical memory, or by writing to a virtual address that is mapped read-only. There are several kinds of page faults. A **read fault** occurs when an unmapped area of memory is read. A **write fault** occurs when an unmapped or write-protected area of memory is written.

panic: a system crash due to a fatal error within the kernel.

physical address space: the address space where physical memory resides. Physical memory need not be contiguous in the physical address space, although it is more efficient for the VM system if it is.

physical memory: the system's RAM or memory chips.

released page: a page that needs to be freed but cannot be freed because it is busy. Such a page is marked released and will be freed when it is no longer busy.

resident memory: memory that is in RAM rather than on backing store.

share map: a map which defines a range of shared virtual address space. Mapping changes in a share map are seen by all processes sharing the map.

shared mapping: a mapping in which changes made to memory are reflected back to the backing object. Thus these changes are shared by any process mapping the object.

shadow object: in the BSD VM, an object that stores the changed version of an object mapped copy-on-write. A list of objects connected by their shadow object pointers is called a **shadow object chain**.

stack area: the area of a process' virtual address space where the program execution stack is placed. This area is filled with zeroed pages by the VM system as it is referenced.

submap: a map used by the kernel to break up its address space into smaller chunks.

swap space: the area of backing store where anonymous memory is paged out to when physical memory is scarce.

swap memory leak: in the BSD VM, when anonymous `vm_object` structures contain memory that is not accessible by any process, those pages eventually get paged out to swap space. Swap space eventually gets filled, and the operating system can deadlock.

text area: the area of a process' virtual address space where the executable instructions of a program are placed. The text area comes from the file containing the program being run and is mapped read-only.

virtual address space: an address space in which virtual memory resides. In a Unix-like operating system each process on the system has its own private virtual address space.

virtual memory: memory that must be mapped to a page of physical memory by the VM system before it can be referenced.

`vm_object`: a data structure that describes a file, device, or zero-fill area which can be mapped into virtual address space.

vnode: a kernel data structure that describes a file.

UVM: a new virtual memory system introduced in this dissertation.

wanted page: a page that is needed but currently busy. A process can mark a busy page wanted and then wait for it to become un-busy.

wired page: a page of virtual memory that is always resident in physical memory.

zero-filled memory: memory that is allocated and filled with zeros only when it is referenced.

References

[1] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in Chorus. In *Proceedings of the Progress in Distributed Operating Systems and Distributed Systems Management Workshop*, April 1989.

[2] V. Abrossimov, M. Rozier, and M Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[3] E. Anderson. *Container Shipping: A Uniform Interface for Fast, Efficient, High-Bandwidth I/O*. PhD thesis, University of California, San Diego, 1995.

[4] J. Barrera. A fast Mach network IPC implementation. In *Proceedings of the USENIX Mach Symposium*, pages 1–11, November 1991.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[6] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-level interprocess communication for shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[7] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. Spin - an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, February 1994.

[8] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, 1996.

[9] D. Bobrow et al. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM*, 15(3), March 1972.

[10] J. Brustoloni. *Effects of Data Passing Semantics and Operating System Structure on Network I/O Performance*. PhD thesis, Carnegie Mellon University, September 1997.

[11] J. Brustoloni and P. Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *Proceedings of IEEE INFOCOM 1997*, pages 1124–1132, April 1997.

[12] M. Buddhikot, X. Chen, D. Wu, and G. Parulkar. Enhancements to 4.4 BSD UNIX for networked multimedia in project MARS. In *Proceedings of IEEE Multimedia Systems'98*, June 1998.

[13] H. Chartock and P. Snyder. Virtual swap space in SunOS. In *Proceedings of the Autumn 1991 European UNIX Users Group Conference*, September 1991.

[14] H. Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX Conference*, pages 253–264. USENIX, 1996.

[15] C. Cranor and G. Parulkar. Universal continuous media I/O: Design and implementation. Technical Report WUCS-94-34, Washington University, 1994.

[16] C. Cranor and G. Parulkar. Design of universal continuous media I/O. In *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 83–86, 1995.

[17] R. Dean and F. Armand. Data movement in kernelized systems. Technical Report CS-TR-92-51, Chorus Systèmes, 1992.

[18] P. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–89, September 1970.

[19] P. Denning. Virtual memory. *ACM Computing Surveys*, 28(1), March 1996.

[20] Z. Dittia, J. Cox, and G. Parulkar. Design of the APIC: A high performance ATM host-network interface chip. In *The Proceedings of IEEE INFOCOM 1995*, pages 179–187, 1995.

[21] Z. Dittia, J. Cox, and G. Parulkar. Design and implementation of a versatile multimedia network interface and I/O chip. In *Proceedings of the 6th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, April 1996.

[22] Z. Dittia, G. Parulkar, and J. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *The Proceedings of IEEE INFOCOM 1997*, 1997.

[23] P. Druschel. *Operating System Support for High-Speed Networking*. PhD thesis, University of Arizona, August 1994.

[24] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

[25] J. Dyson, D. Greenman, et al. The FreeBSD VM system. See `http://www.freebsd.org` for more information.

[26] K. Fall. *A Peer-to-Peer I/O System in Support of I/O Intensive Workloads*. PhD thesis, University of California, San Diego, 1994.

[27] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *Proceedings of USENIX Winter Conference*, pages 327–333. USENIX, January 1993.

[28] R. Gingell, J. Moran, and W. Shannon. Virtual memory architecture in SunOS. In *Proceedings of USENIX Summer Conference*, pages 81–94. USENIX, June 1987.

[29] R. Gopalakrishnan and G. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. In *Proceedings ACM SIGMETRICS*, May 1996.

[30] The Open Group. *The Single UNIX Specification, Version 2*. The Open Group, 1998. See `http://www.UNIX-systems.org/online.html`.

[31] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of USENIX Summer Conference*, pages 147–160. USENIX, June 1993.

[32] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2nd edition, 1996.

[33] Berkeley Software Design Inc. The BSD/OS operating system. See `http://www.bsdi.com` for more information.

[34] M. Kaashoek, D. Engler, G. Ganger, M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, October 1997.

[35] Y. Khalidi and M. Nelson. The Spring virtual memory system. Technical Report SMLI TR-93-09, Sun Microsystems Laboratories, Inc., February 1993.

[36] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD Operating System*. Addison-Wesley, 1989.

[37] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–188, 1993.

[38] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, H. Gernot, N. Islam, and T. Jaeger. Achieved IPC performance. In *Proceedings of Hot Topics in Operating Systems (HotOS) 1997*, pages 28–31, 1997.

[39] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[40] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of USENIX Conference*, pages 279–294. USENIX, 1996.

[41] F. Miller. *Input/Output System Design For Streaming*. PhD thesis, University of Maryland, 1998.

[42] F. Miller and S. Tripathi. An integrated input/output system for kernel data streaming. *Multimedia Computing and Networking*, 1998.

[43] R. Minnich. Mether-NFS: A modified NFS which supports virtual shared memory. In *Proceedings of USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, September 1993.

[44] J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P. Kougiouris, P. Madany, M. Nelson, M. Powell, and S. Radia. An overview of the Spring system. In *Proceedings of Compcon Spring 1994*, February 1994.

[45] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, and T. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of Hot Topics in Operating Systems (HotOS) 1995*, 1995.

[46] J. Moran. SunOS virtual memory implementation. In *Proceedings of the Spring 1988 European UNIX Users Group Conference*, April 1988.

[47] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *OSDI 1996*, pages 153–168, 1996.

[48] M. Nelson. Virtual memory for the Sprite operating system. Technical Report UCB/CSD 86/301, Computer Science Division, EECS Department, University of California, Berkeley, June 1986.

[49] M. Nelson and J. Ousterhout. Copy-on-write for Sprite. In *Proceedings of USENIX Summer Conference*. USENIX, June 1988.

[50] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer Conference*, pages 247–256. USENIX, 1990.

[51] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.

[52] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. Technical Report TR97-294, Department of Computer Science, Rice University, 1997.

[53] J. Pasquale, E. Anderson, and P. Muller. Container Shipping: Operating system support for I/O intensive applications. *Computer*, 27(3), March 1994.

[54] The NetBSD Project. The NetBSD operating system. See `http://www.netbsd.org` for more information.

[55] The OpenBSD Project. The OpenBSD operating system. See `http://www.openbsd.org` for more information.

[56] S. Rago. *Unix System V Network Programming*. Addison-Wesley, 1993.

[57] 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. *IEEE Transactions on Computing*, 37(8), August 1988.

[58] D. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[59] C. Schimmel. *Unix System for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley, 1994.

[60] D. Schmidt. The adaptive communications environment: Object-oriented network programming components for developing client/server applications. In *Proceedings of the 12th Sun Users Group*, June 1994. An earlier version of this paper appeared in the 11th Sun Users Group conference.

[61] M. Schroeder and M. Burrows. The performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[62] D. Solomon. *Inside Windows NT*. Microsoft Press, 2nd edition, 1998. Based on the first edition by Helen Custer.

[63] R. Stallman and Cygnus Support. *Debugging with GDB*. The Free Software Foundation, 1994.

[64] A. Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*. PhD thesis, Carnegie Mellon University, December 1987.

[65] M. Thadani and Y. Khalidi. An efficient zero-copy I/O framework for Unix. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, May 1995.

[66] L. Torvalds et al. The Linux operating system. See `http://www.linux.org` for more information.

[67] S. Tzou and D. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software - Practice and Experience*, 21(3):251–267, March 1991.

[68] USL. *Unix System V Release 4.2 STREAMS Modules and Drivers*. Prentice-Hall, 1992.

[69] U. Vahalia. *Unix Internals: the New Frontiers*. Prentice-Hall, 1996.

[70] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 2nd edition, 1996.

[71] M. Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. PhD thesis, Carnegie Mellon University, November 1989.

[72] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[73] W. Zwaenepoel and D. Johnson. The Peregrine high-performance RPC system. *Software - Practice and Experience*, 23(2):201–221, February 1993.

Vita

Charles D. Cranor

Date of Birth	June 14, 1967
Place of Birth	Camden, South Carolina
Degrees	B.S., Electrical Engineering, May 1989 M.S. Computer Science, May 1992 D.Sc. Computer Science, August 1998
Professional Societies	Association for Computing Machines Institute of Electrical and Electronics Engineers

Publications

Gigabit CORBA — High-Performance Distributed Object Computing, Gigabit Networking Workshop (GBN'96), 24 March 1996, San Francisco, in conjunction with INFOCOM '96 (with D. Schmidt and G. Parulkar).

Half-Sync/Half-Async: A Architectural Pattern for Efficient and Well-structured Concurrent I/O, in *Pattern Languages of Program Design*, (Coplien, Vlissides, and Kerth, eds.), Addison-Wesley, Reading, MA, 1996 (with D. Schmidt).

Design of Universal Continuous Media I/O, in *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (NOSSDAV '95), pp 83–86, April 1995 (with G. Parulkar).

The 3M Project: Multipoint Multimedia Applications on Multiprocessor Workstations and Servers, in *Proceedings of IEEE Workshop on High Performance Communication Systems*, Sept. 1993 (with M. Buddhikot, Z. Dittia, G. Parulkar, C. Papadopoulos).

An Implementation Model for Connection-Oriented Internet Protocols, in *Journal of Internetworking: Research and Experience*, Vol. 4., pp 133–157, Sept. 1993 (with G. Parulkar).

An Implementation Model for Connection-Oriented Internet Protocols, *Proceedings of IEEE INFOCOM '93*, Vol. 3, pp 1135–1143, April 1993 (with G. Parulkar).

An Implementation Model for Connection-Oriented Internet Protocols, M.S. thesis, Department of Computer Science, Sever Institute of Technology, Washington University, St. Louis, Missouri, May 1992.

August 1998