- Table of Contents
- Index

HP-UX 11i Internals

By Chris Cooper, Chris Moore

Publisher: Prentice Hall PTR
Pub Date: January 22, 2004
ISBN: 0-13-032861-8
Pages: 432

To maximize the performance, efficiency, and reliability of your HP-UX sysem, you need to know what's going on under the hood. *HP-UX 11i Internals* goes beyond generic UNIX internals, showing exactly how HP-UX works in PA-RISC environments.

HP experts Cooper and Moore systematically illuminate HP-UX kernel data structures and algorithms for memory management, process and thread scheduling, I/O control, files and file systems, resource management, and more. They focus on HP-UX 11i, while also offering valuable insight for those using earlier versions.

- PA-RISC architecture: register set, virtual memory, key instructions, and procedure calling conventions
- HP-UX kernel organization: hardware-dependent and independent data structures
- Process and thread management: proc tables, memory management, scheduling, and the complete process/thread lifecycle
- System-wide memory resources: allocation and mapping to physical memory
- HP-UX paging and swapping
- Files and filesystems: traditional UNIX filesystems, HFS, VFS, and dynamic buffer cache
- I/O and device management: addressing, DMA, interrupts, device files, I/O configuration, device driver assignments, and I/O request pathways
- Logical Volume Management (LVM): abstracting physical disks from the disk I/O system
- HP-UX multiprocessing: challenges, data structures, and interfaces
- Kernel communication services: semaphores, message queues, shared memory, signals, and the kernel "callout" system

- Signaling in complex threaded environments
- System initialization, from vmunix to init: running HP-UX on diverse platforms

Whether you administer HP-UX, tune it, troubleshoot it, or write kernel modules for it, you'll find HP-UX 11i Internals indispensable.

< Day Day Up >

- Table of Contents
- Index

HP-UX 11i Internals

By Chris Cooper, Chris Moore

Publisher: Prentice Hall PTR
Pub Date: January 22, 2004
ISBN: 0-13-032861-8
Pages: 432

Copyright

Library of Congress Cataloging-in-Publication Data

Cooper, Chris.
HP-UX 11i internals/Chris Cooper, Chris Moore.
p. cm. -- (Hewlett-Packard professional books)
ISBN 0-13-032861-8
1. Operating systems (Computers) 2. HP-UX. I. Moore, C.G. (Chris G.) II. Title.
III. Series.

QA76.76.)63C6644 2004
005.4'3--dc22
2003069034

Editorial/production supervision: *Nicholas Radhuber*

Cover design director: *Jerry Votta*

Cover design: *DesignSource*

Manufacturing manager: *Maura Zaldivar*

Acquisitions editor: *Jill Harry*

Editorial assistant: *Brenda Mulligan*

Marketing manager: *Dan DePasquale*

Published by Prentice Hall PTR
Pearson Education, Inc.
Upper Saddle River, New Jersey 07458

Prentice Hall offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact:

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:
International Sales
1-317-581-3793
international@pearsontechgroup.com

Printed in the United States of America

1st Printing

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd.

Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education — Japan
Pearson Education Malaysia, Pte. Ltd.

Hewlett-Packard® Professional Books

HP-UX

Cooper/Moore	HP-UX 11i Internals
Fernandez	Configuring CDE
Madell	Disk and File Management Tasks on HP-UX
Olker	Optimizing NFS Performance
Poniatowski	HP-UX 11i Virtual Partitions
Poniatowski	HP-UX 11i System Administration Handbook and Toolkit, Second Edition
Poniatowski	The HP-UX 11.x System Administration Handbook and Toolkit
Poniatowski	HP-UX 11.x System Administration "How To" Book
Poniatowski	HP-UX 10.x System Administration "How To" Book
Poniatowski	HP-UX System Administration Handbook and Toolkit
Poniatowski	Learning the HP-UX Operating System
Rehman	HP-UX CSA: Official Study Guide and Desk Reference
Sauers/Ruemmler/Weygant	HP-UX 11i Tuning and Performance
Weygant	Clusters for High Availability, Second Edition
Wong	HP-UX 11i Security

UNIX, LINUX, WINDOWS, AND MPE I/X

Mosberger/Eranian	IA-64 Linux Kernel
Poniatowski	UNIX User's Handbook, Second Edition
Stone/Symons	UNIX Fault Management

COMPUTER ARCHITECTURE

Evans/Trimper	Itanium Architecture for Programmers
Kane	PA-RISC 2.0 Architecture
Markstein	IA-64 and Elementary Functions

NETWORKING/COMMUNICATIONS

Blommers	Architecting Enterprise Solutions with UNIX Networking
Blommers	OpenView Network Node Manager
Blommers	Practical Planning for Network Growth

Brans	Mobilize Your Enterprise
Cook	Building Enterprise Information Architecture
Lucke	Designing and Implementing Computer Workgroups
Lund	Integrating UNIX and PC Network Operating Systems

SECURITY

Bruce	Security in Distributed Computing
Mao	Modern Cryptography: Theory and Practice
Pearson et al.	Trusted Computing Platforms
Pipkin	Halting the Hacker, Second Edition
Pipkin	Information Security

WEB/INTERNET CONCEPTS AND PROGRAMMING

Amor	E-business (R)evolution, Second Edition
Apte/Mehta	UDDI
Chatterjee/Webber	Developing Enterprise Web Services: An Architect's Guide
Kumar	J2EE Security for Servlets, EJBs, and Web Services
Mowbrey/Werry	Online Communities
Tapadiya	.NET Programming

OTHER PROGRAMMING

Blinn	Portable Shell Programming
Caruso	Power Programming in HP OpenView
Chaudhri	Object Databases in Practice
Chew	The Java/C++ Cross Reference Handbook
Grady	Practical Software Metrics for Project Management and Process Improvement
Grady	Software Metrics
Grady	Successful Software Process Improvement
Lewis	The Art and Science of Smalltalk
Lichtenbelt	Introduction to Volume Rendering
Mellquist	SNMP++
Mikkelsen	Practical Software Configuration Management
Norton	Thread Time
Tapadiya	COM+ Programming
Yuan	Windows 2000 GDI Programming

STORAGE

Thornburgh	Fibre Channel for Mass Storage
Thornburgh/Schoenborn	Storage Area Networks
Todman	Designing Data Warehouses

IT/IS

Missbach/Hoffman	SAP Hardware Solutions

IMAGE PROCESSING

Crane	A Simplified Approach to Image Processing
Gann	Desktop Scanners

Preface

While there are many books about the UNIX kernel in its various versions, this is the first to deal specifically with HP-UX and Hewlett-Packard's PA-RISC architecture. This book is a technical resource for anyone who has to support HP-UX, write kernel modules, or just wants to know how it all works. It is suitable as a text for a class in operating system concepts or HP-UX internals.

Chapter 1 PA-RISC 2.0 Architecture

Chapter 1 presents some of the important features of the PA-RISC architecture, still the predominant hardware platform for HP-UX. It covers the register set and how the registers are used, then explains virtual memory concepts, address translation and caching, the mechanisms provided by the architecture to support virtual memory, and some of the most common PA-RISC instructions and their use.

Chapter 2 Procedure Calling Conventions

This chapter examines the conventions used when calling from one procedure to another: how the stack is used and how stack frames are set up, how the registers are used, how arguments are passed, and how results are returned to the calling process. Stack usage for both narrow and wide modes is explained.

Chapter 3 The Kernel: Basic Organization

This chapter examines the basic organization of the kernel data structures, both hardware-dependent and hardware-independent. The discussion includes tables, the kernel as a resource manager, and kernel memory allocation. In addition, many common kernel algorithms and methodologies are presented for the reader's consideration.

Chapter 4 Programs, Processes, and Threads

The process has long been the workhorse of the UNIX environment. Modern UNIX operating systems have adopted the thread as the new schedulable entity. Chapter 4 examines both the process and the thread, and their relationship to resources and task management. One of the most important aspects of understanding the process/thread environment is in flow of control followed when a system call is requested. The transistion from user mode to system mode and the return to user mode are examined in detail in Chapter 4.

Chapter 5 Process and Thread Management from the Process's Viewpoint

This chapter explores processes and threads: how they are created, how they are terminated, and how they are scheduled to run. The internal data structures that keep track of the processes and threads are explained. We cover how memory is managed for processes and how addresses are mapped, and we look at process priorities and how they change throughout the life of the process.

Chapter 6 Managing Memory

Chapter 6 looks further into memory management by looking at systemwide memory resources. It explains how the per-process memory areas are allocated from systemwide memory and how that memory is mapped to physical memory. This chapter addresses physical and virtual memory, page regions, and address aliasing.

Chapter 7 The HP-UX Paging System

To complete our tour of the kernel's memory management system, Chapter 7 discusses the HP-UX paging system (commonly called the swapper). The following topics are included in the discussion:

- Demand paging
- Swap versus paging
- `lotsfree` and `vhand`
- Deallocation
- Pseudo-swap
- Swap chunks

Chapter 8 Files and File Systems

This chapter looks at the structures both in memory and on disk that make up the HP-UX file systems. HP-UX supports a number of file system layouts. This chapter looks in detail at the UNIX File System layout and the Virtual File System layer, which allows the kernel to work easily with the various file system types. It also explains various caching systems and how the dynamic buffer cache is used to complete disk transfers.

Although the Veritas File System (VxFS) is part of HP-UX, its internals are proprietary to Veritas and are not covered in depth. The merits of a journal file system and extent-based disk space allocation, however, are discussed in this chapter.

Chapter 9 The Process Life Cycle, Cradle to Grave

Chapter 9 explores the process's life cycle in detail: how it is created, how it is maintained, and how it is terminated. Shared objects are discussed from the process point of view and the following kernel routines and features are presented in detail:

- `fork()`
- `exec()`
- `wait()`
- `exit()`
- Copy-on-write
- Copy-on-access

Chapter 10 I/O and Device Management

Chapter 10 explores how the system performs I/O. It explains how I/O is performed at a low level in the PA-RISC architecture, including addressing, DMA, and interrupts. It then discusses how I/O works from the application's point of view, including device files and I/O configuration. We cover the steps in between with a discussion of how device drivers claim devices and how I/O requests make their way from the device files through the device drivers to the physical I/O devices. I/O is presented within the umbrella of the General I/O subsystem.

Chapter 11 The Logical Volume Manager

The logical volume manager (LVM) is a disk management subsystem. It provides a layer of abstraction between the physical disks and the disk I/O system. This chapter looks at how LVM combines physical disks into volume groups, divides those volume groups into logical disk volumes, and provides mirroring for data redundancy. The structures used within the kernel as well as the on-disk structures are discussed.

Chapter 12 Multiprocessing and HP-UX

This chapter covers the elements of the kernel that facilitate multiprocessing. We look at some of the issues of having multiple processors sharing main memory and I/O devices. Then we look in detail at the data structures and interfaces that provide for consistent access to these resources among multiple processors. This includes spinlocks and semaphores for controlling exclusive access to resources and also load-balancing between processors.

Chapter 13 Kernel Services

Chapter 13 investigates the mechanisms in the kernel for supporting communication between processes: semaphores, message queues, shared memory, and signals. The kernel structures that support these features and the kernel tunables that control them are explained. We look at the user interfaces to these facilities and examine the kernel callout system, which allows a process to request a future signal from the kernel.

Chapter 14 Signals

Signals were traditionally sent from one process to another and usually meant that a process needed to terminate. With the advent of threads, the entire signal system has become more complex. Chapter 14 discusses both traditional and new uses and methods for signals. We will discuss in turn the following HP-UX kernel supported signaling schemes; System-V, Berkeley, and POSIX.

Chapter 15 System Initialization

HP-UX runs on many different platform designs, using a variety of boot sequences. Chapter 15 explains what takes place once the specific boot criteria has been satisfied and the kernel is off and running. This sequence of events is broken down into the real main, virtual main, and I/O initialization steps.

Chapter 16 Tools Overview

Chapter 16 explains the practical use of two important tools, adb and q4.

Acknowledgments

Chris Cooper:

I am in my twentieth year with Hewlett Packard and have been privileged to work with and learn from the "best in class" engineers, solution architects, consultants, managers, and designers this industry has to offer. This work would not have been possible without their collective support and the spirit of knowledge sharing fostered by the Hewlett Packard workplace environment.

I would like to recognize and thank Stephen Ciullo, Senior Technical Consultant with HP Engineering Services, who has constantly amazed me with his depth and breadth of knowledge about all things HP-UX; and D. Paul Klein, HP Education Services consultant and HP-UX Internals instructor supreme, whom I have had the pleasure of knowing and co-teaching with in the corporate level internals training class for the past several years. I would like to offer a special thanks to my immediate manager, Deeko Patel, for her support and encouragement during this project.

Writing a book requires a major investment of time and thought, and my family has shared this experience by allowing me to invest the time to make this book happen and for that I am eternally grateful. I thank my wife, Debbi Cooper, and my daughter, Jessica Cooper, for their love and understanding.

Chris Moore:

I would like to express my thanks to the members of Hewlett Packard's Worldwide Technology Expert Center. The material presented in this book is gathered from and based on an incredible amount of information created by this very talented group of engineers. There were countless times that I referred to information written by this team or asked them for clarification on details.

I would also like to thank Glen Rosen of Actel Corporation, Ronald McCarty of Penn State University, and Stan Moravec of Hewlett Packard for their help in reviewing the contents of the book. Their input has tremendously improved the accuracy and readability of this book.

Finally, I would like to thank my wife, Carrie, and my sons, Sean and Tommy, for their never-ending patience during the writing process. There were many times when I spent more time being an author than a Dad and I appreciate their support and understanding.

List of Figures

List of Tables

PREV < Day Day Up > NEXT

List of Listings

Chapter 1. PA-RISC 2.0 Architecture

Before we dive into HP-UX itself, we should have a look at the underlying system architecture. The very nature of an operating system requires that it be aware of the architecture it's running on. The operating system's kernel insulates the user applications from the hardware, making the architecture mostly transparent to the applications. But the kernel itself has to work directly with the hardware and so has to be designed to run on a particular type of hardware.

HP-UX has been implemented on a variety of platforms, among them the Motorola 68000 family, HP's PA-RISC family, and the new Itanium family. In this book we concentrate on the PA-RISC architecture. The 68000 versions of HP-UX are long obsolete, and the Itanium versions are new enough that there are few implementations in use. The vast majority of HP-UX systems are running on PA-RISC hardware.

Why study architecture at all in a software book? The kernel has architecture-specific code for handling interruptions, dealing with I/O devices, managing multiprocessor issues, and for many other tasks. In fact, parts of the kernel are written in assembly language for performance reasons. Before we get into how the kernel works, it's important to have at least a general idea of how the hardware works.

We won't get into every detail of the architecture. That's covered thoroughly in *PA-RISC 2.0 Architecture* by Gerry Kane.[1] In this book, we cover just those implementation details that are critical to an understanding of the internals of HP-UX. In particular, we look at the general hardware building blocks, the registers that are available and how they are used, provisions for virtual memory addressing, and some of the assembly instructions that are important to the implementation of the kernel.

[1] Gerry Kane, *PA-RISC 2.0 Architecture* (Pearson Education POD, 1995).

We also look a little at the differences between the PA-RISC 2.0 and PA-RISC 1.1 architectures. PA-RISC 2.0 expanded the PA-RISC architecture to a 64-bit model and added many enhancements for improving system performance. Systems built on the PA-RISC 1.1 architecture run 32-bit kernels. A PA-RISC 2.0 system can run either a 32-bit or a 64-bit kernel. A 64-bit kernel in turn can run either 32-bit or 64-bit user applications.

RISC Architecture

PA-RISC, as its name implies, is a RISC, or Reduced Instruction Set Computer, architecture. But there is more to RISC than just a smaller instruction set. There are several characteristics that are common to all RISC architectures, including PA-RISC, that make these systems fast and easy to implement. Among these are:

- Fixed instruction size— All instructions are 32 bits wide.
- Small number of addressing modes— PA-RISC has only three methods of specifying a memory address.
- Simplified memory access— The only instructions that access memory are explicit loads and stores. Computational operations are done between registers.

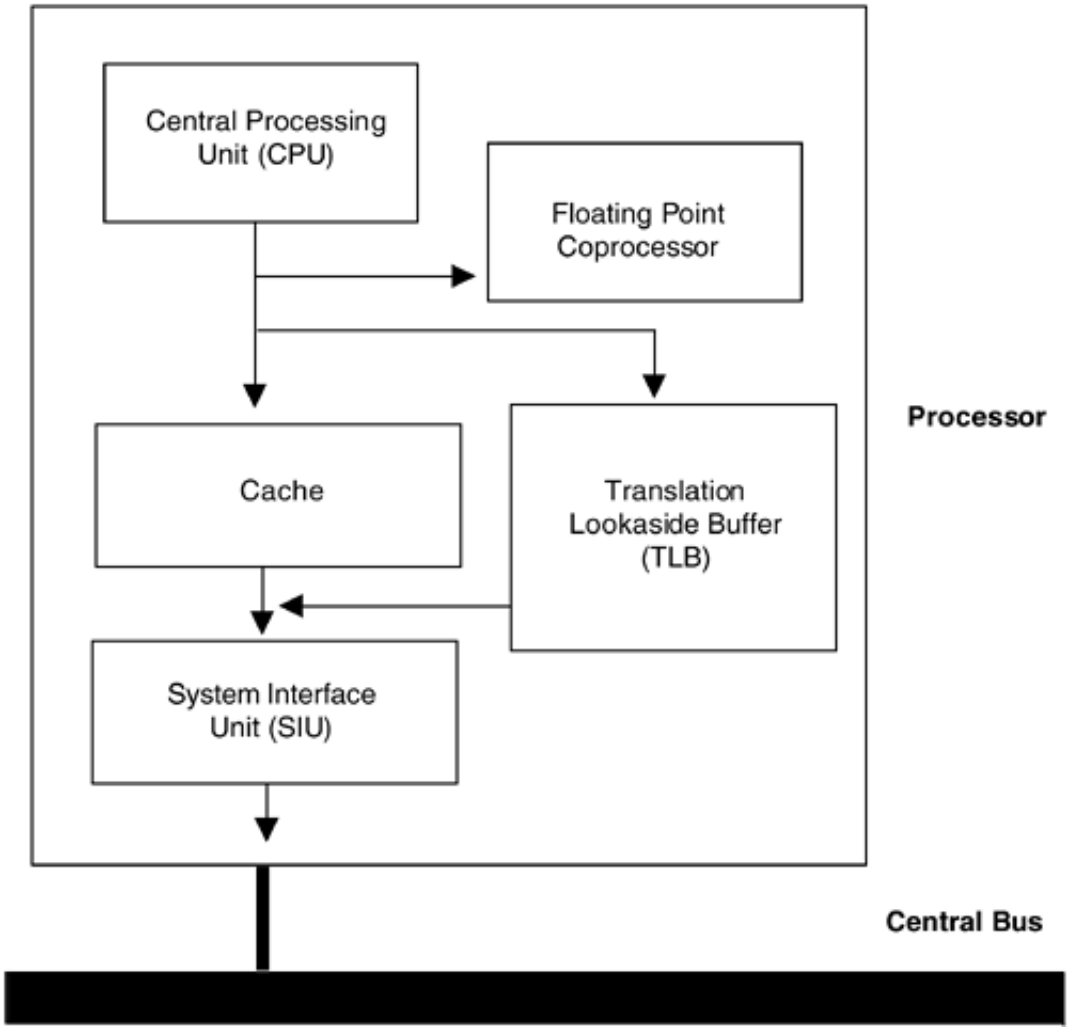
These characteristics make for a fast, simple hardware implementation. Performance gains can then be accomplished through pipelining of instructions and through optimizing of code by compilers.

The PA-RISC architecture also has many features not normally associated with RISC. The instruction set, while smaller than older CISC architectures, has many instructions designed specifically to streamline commonly executed sequences of instructions. In addition, the hardware is designed to speed up typically expensive instructions such as bit operations.

Hardware Modules

Figure 1-1 shows a typical PA-RISC processor. The architecture provides for a translation lookaside buffer (TLB), which is used to assist in translating virtual addresses to physical addresses. The inclusion of a cache is optional, but most PA-RISC processors do have at least some amount of cache. The architecture also provides for assist processors such as a floating-point processor to assist in complex operations.

Figure 1-1. Processor Block Diagram



Central Processing Unit

The CPU itself contains the system's register set, control logic, and execution logic. The register set, discussed in detail below, is where all of the computation takes place in the processor. The only instructions

that access memory are explicit load and store operations. Thus, to operate on a value, the value must be loaded, manipulated, then stored. Frequently used data is kept in registers to bypass the load and store operations. The CPU's control logic is responsible for fetching and decoding instructions. The CPU can prefetch instructions, and multiple instructions are pipelined within the CPU for parallel execution. The execution logic performs the actual operations on the data. This section of the CPU consists of the arithmetic logic unit (ALU), shift and merge units, mask registers, and a variety of other hardware for executing the instructions.

PA-RISC provides for four different privilege levels that determine what operations the CPU is permitted to perform and what data it can access. The HP-UX kernel uses only two of these privilege levels: 0 for kernel and 3 for user. The system checks access rights on a per-page basis. Each page of memory in the system has an Access Rights entry, which indicates privilege levels required to read, write, or execute that page. The system also provides for a gateway instruction, which is used for promotion from one privilege level to another. This instruction is used by the kernel when transitioning from user mode to kernel mode. This process is discussed in detail in Chapter 3.

Translation Lookaside Buffer

The TLB is used in translating virtual addresses to physical addresses. Each entry in the TLB holds a virtual page number, the corresponding physical page number, and the access rights for that particular page. The system may use separate TLBs for instructions (ITLB) and data (DTLB), or it may use a combined TLB.

PA-RISC version 2.0 includes provisions for variable-sized pages. Prior to PA-RISC 2.0 all pages were a fixed 4 KB. PA-RISC 2.0 can use pages ranging from 4 KB to 64 MB. This allows for more efficient use of the TLB when very large data structures are used. A large data structure can be placed in a single large page and will use only one TLB entry rather than requiring several.

When an instruction accesses a particular virtual page, that page is looked up in the TLB to find the corresponding physical address and the access rights. If a matching entry is not present, a *TLB miss* interruption is generated. The system must then look up the correct entry in the Page Directory (PDIR), insert the entry into the TLB, then restart the instruction. The details of how this happens are covered in Chapter 6, "Managing Memory."

Cache

PA-RISC uses a visible cache architecture. The processor has instructions to explicitly manipulate the cache, flushing or invalidating cache entries where required. Systems can have separate instruction and data caches, or they may have a single combined cache.

In order to have the required data in the cache whenever possible, the cache is capable of prefetching data and instructions. To determine which data should be fetched, the PA-RISC processor uses *branch prediction* to guess which way conditional branches will go. Compilers can code a prediction into branch instructions. For example, if a branch is used in a loop, and the loop will run for 100 iterations, then the end-of-loop branch will be taken 100 times, not taken just one time. The compiler can indicate to the processor that it is more likely that the branch is taken, and the cache prefetch circuitry can then prefetch the correct instructions. PA-RISC 2.0 also adds a *dynamic branch prediction* functionality. The processor keeps track of how many times a branch actually went the way that the compiler predicted. If the branch prediction indicator is wrong more than right, then the system starts prefetching the opposite of the way the branch prediction indicates.

In a multiprocessor system, the cache must also be aware of the memory activity of other processors. This is called *cache coherence*. The cache circuitry continually monitors the system bus for load and store instructions from other processors. In this way, it will know if the data in the cache is no longer valid due to access from a different processor.

Assist Processors

The PA-RISC architecture provides for extensibility in the form of assist processors. Two kinds of assist processors are defined: special function units (SFUs) and coprocessors. An SFU is a hardware module that has direct access to the CPU's general registers but operates outside of the CPU's execution unit. Although the architecture defines how SFUs are to be implemented, none are currently used. A coprocessor is also an assist processor, but it is less closely coupled to the CPU. It has its own set of registers, and data must be explicitly passed to and from the coprocessor for processing.

An understanding of the assist processors is not required for an understanding of the kernel, so we do not go into detail beyond this brief mention. Suffice it to say that PA-RISC processors do have a floating-point coprocessor, and if you find yourself looking at assembly code, you may occasionally see references to it.

Registers

The PA-RISC architecture provides for several sets of registers. Of particular interest to us are the general registers, space registers, and some of the control registers.

General Registers

PA-RISC has 32 general purpose registers. In PA-RISC 2.0, these are 64-bit registers; prior to PA-RISC 2.0, these registers are 32 bits wide. The architecture itself defines specific uses for only a few of these registers:

- GR0— Permanent zero. Reads from this register always return a zero. Writes to this register are discarded.
- GR1— Target of `ADDIL`. When the `ADD IMMEDIATE LONG (ADDIL)` instruction is executed, the result is always placed in GR1. This register can also be used for other purposes.
- GR2— Target of `B,L`. This register is used as the target of the `BRANCH AND LINK` instruction, which is used for procedure calls. The link, or return, address is placed in GR2. This register can also be used for other purposes.
- GR31— Target of `BLE`. This register is similar to GR2 but is used for the return address from a `BRANCH AND LINK EXTERNAL` instruction.

Beyond these requirements there is also a software convention that specifies how registers will be used by HP-UX. These conventions are slightly different for the 64-bit version of HP-UX. Figure 1-2 shows the usage of general registers for the 32-bit kernel, and Figure 1-3 shows the usage for the 64-bit kernel. Most of these requirements for register usage come into play when performing procedure calls and are covered in detail in Chapter 2, "Procedure Calling Conventions." GR27, the data pointer, points to the base of the data segment. Memory references to the data segment are made relative to GR27. These are primarily global values in the kernel. GR30 is the stack pointer.

Figure 1-2. 32-Bit General Registers

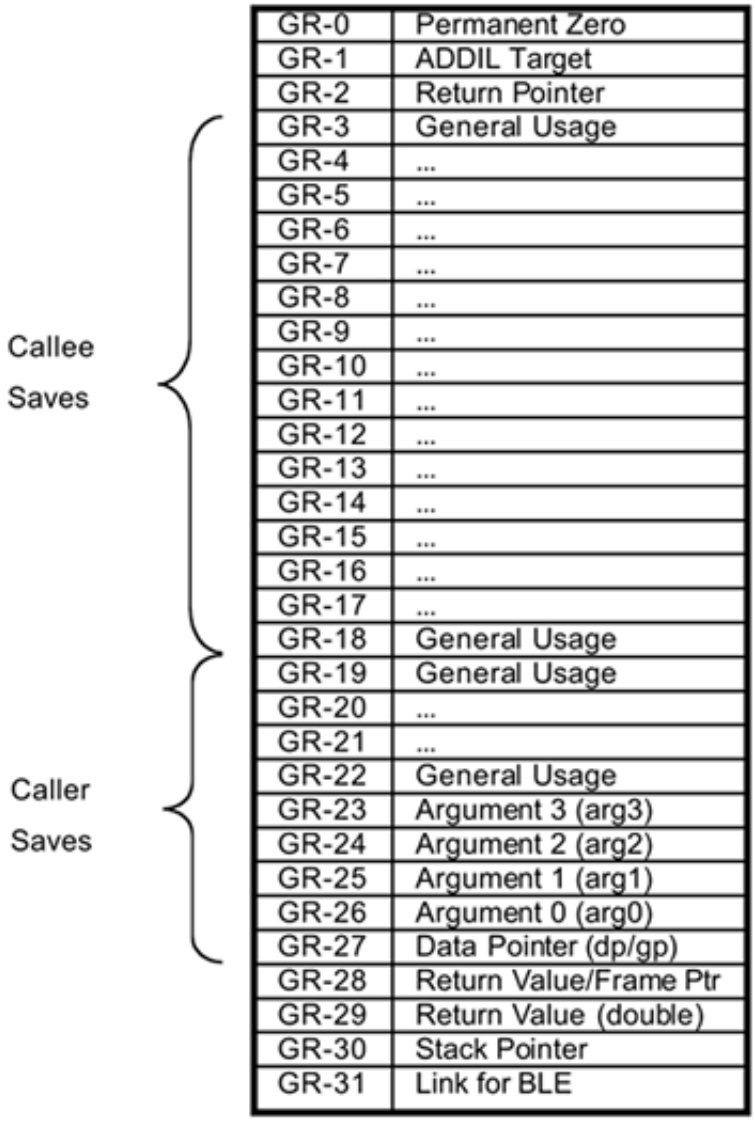
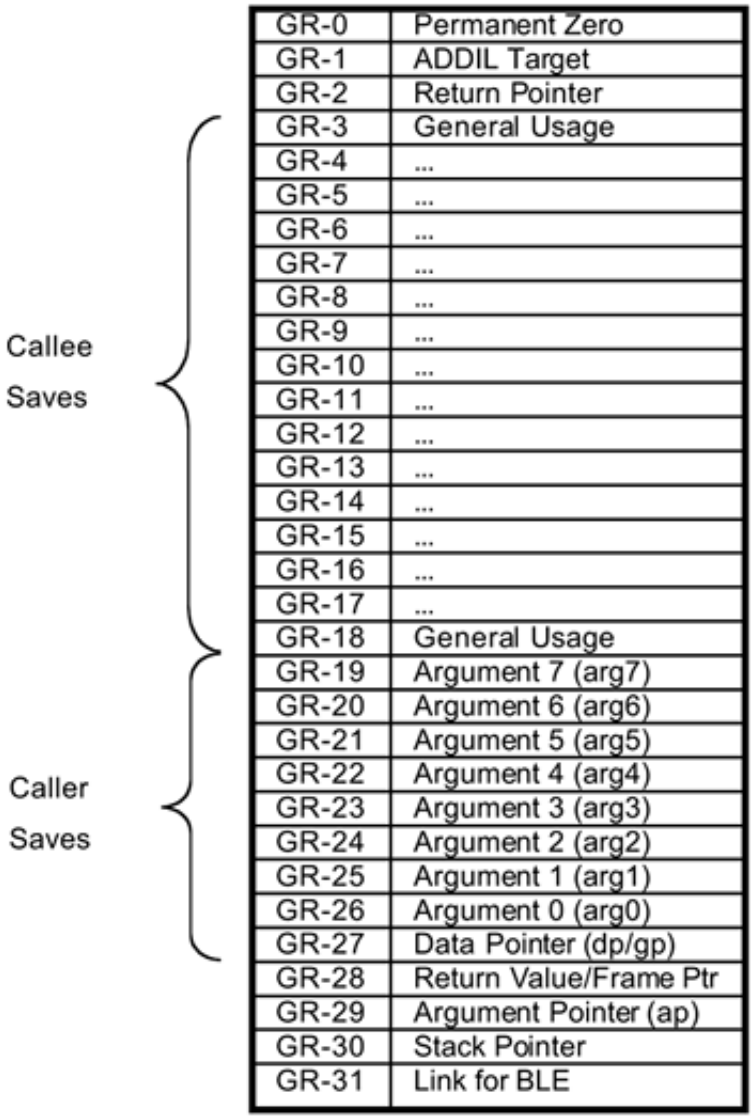


Figure 1-3. 64-Bit General Registers



Space Registers

PA-RISC provides eight space registers. As with the general registers, the architecture provides for 64-bit wide registers on PA-RISC 2.0 and 32-bit wide registers on PA-RISC 1.1. However, on all current implementations of PA-RISC 2.0, the space registers are actually only 32 bits wide. The space registers are used in combination with the general registers to construct a virtual address. The space register can be either explicitly supplied in the instruction or implied by the value in the general register. The use of the space registers is covered in detail in the section "Virtual Memory Support."

Control Registers

PA-RISC provides 32 control registers (see Figure 1-4). Again, in PA-RISC 1.1 these are 32-bit registers, and in PA-RISC 2.0 they are 64-bit registers. Most of these are used when handling interruptions and are discussed in detail in the section "Interruption Handling."

Figure 1-4. Control Registers

CR0	Recovery Counter
CR1-CR7	Reserved
CR8-CR9	Protection IDs
CR10	Coprecessor Configuration Record (CCR)
CR11	Shift Amount Register (SAR)
CR12-CR13	Protection IDs
CR14	Interrupt Vector Address (IVA)
CR15	External Interrupt Enable Mask (EIEM)
CR16	Interval Timer
CR17	Interrupt Instruction Address Space Queue (IIASQ)
CR18	Interrupt Instruction Address Offset Queue (IIAOQ)
CR19	Interruption Instruction Register (IIR)
CR20	Interruption Space Register
CR21	Interruption Offset Register
CR22	Interruption Processor Status Word
CR23	External Interrupt Request Register
CR24-CR31	Temporary Registers

Processor Status Word

One more register of interest is the Processor Status Word (PSW). This is a 64-bit register (32-bits on PA-RISC 1.1) whose bits indicate the state of the system. Some of these bits are controlled explicitly by the kernel, and others are set implicitly as the result of some action such as an interruption or taken branch. Table 1-1, from Gerry Kane's *PA-RISC 2.0 Architecture*, lists the bits in the PSW and their meanings.

Table 1-1. Processor Status Word Bits

Field	Description
W	Wide 64-bit address formation enable. When 1, full 64-bit-offset addressing is enabled. When 0, addresses are truncated to 32-bit offsets for compatibility with existing PA-RISC 1.0 and 1.1 applications.
E	Little endian memory access enable. When 0, all memory references are big endian. When 1, all memory references are little endian. Implementation of this bit is optional. If it is not implemented, all memory references are big endian and this bit is a reserved bit.
S	Secure Interval Timer. When 1, the Interval Timer is readable only by code executing at the most privileged level. When 0, the Interval Timer is readable by code executing at any privilege level.
T	Taken branch trap enable. When 1, any taken branch is terminated with a taken branch trap.
H	Higher-privilege transfer trap enable. When 1, a higher-privilege transfer trap occurs whenever the following instruction is of a higher privilege.
L	Lower-privilege transfer trap enable. When 1, a lower-privilege transfer trap occurs whenever the following instruction is of a lower privilege.

N	Nullify. The current instruction is nullified when this bit is 1. This bit is set to 1 by an instruction that nullifies the following instruction.
X	Data memory break disable. The X-bit is set to 0 after the execution of each instruction, except for the `RETURN FROM INTERRUPTION` instruction, which may set it to 1. When 1, data memory break traps are disabled. This bit allows a simple mechanism to trap on a data store and then proceed past the trapping instruction.
B	Taken branch. The B-bit is set to 1 by any taken branch instruction and set to 0 otherwise. This is used to ensure that the `BRANCH` instruction with the `,GATE` completer (the privilege-increasing instruction) cannot be used to compromise system security.
C	Code (instruction) address translation enable. When 1, instruction addresses are translated and access rights checked.
V	Divide step correction. The `DIVIDE STEP` (integer division primitive) instruction records intermediate status in this bit to provide a nonrestoring divide primitive.
M	High-priority machine check mask. When 1, high-priority machine checks (HPMCs) are masked. Normally 0, this bit is set to 1 after an HPMC and set to 0 after all other interruptions.
C/B	Carry/borrow bits. The following instructions update the PSW carry/borrow bits from the corresponding carry/borrow outputs of the 4-bit digits of the ALU: ADD* ADDI DS SHLADD* SUB SUBI The instructions marked with an asterisk set the carry/borrow bits only if the `L` (logical) completer is not specified. After an add which sets them, each bit is set to 1 if a carry occurred out of its corresponding digit and set to 0 otherwise. After a subtract which sets them, each bit is set to 0 if a borrow occurred into its corresponding digit, and set to 1 otherwise. Bits {24..31} hold the digit carries from the upper half of the ALU, and bits {48..55} hold the digit carries from the lower half.
O	Ordered references. When 1, virtual memory references to pages with the corresponding TLB O-bit 1 and all absolute memory references are ordered. When 0, memory references (except those explicitly marked as ordered or strongly ordered) may be weakly ordered. Note that references to I/O address space, references to pages with the TLB U-bit 1, semaphore instructions, and TLB purge instructions are always strongly ordered.
F	Performance monitor interrupt unmask. When 1, the performance monitor interrupt is unmasked and can cause an interruption. When 0, the interruption is held pending. Implementation of this bit is required only if the performance monitor is implemented and the performance monitor has the ability to interrupt. If it is not implemented, this bit is a reserved bit.
R	Recovery Counter enable. When 1, recovery counter traps occur if bit 0 of the recovery counter is a 1. This bit also enables decrementing of the recovery counter.
Q	Interruption state collection enable. When 1, interruption state is collected. Used in processing the interruption and returning to the interrupted code, this state is recorded in the Interruption Instruction Address Queue (IIAQ), the Interruption Instruction Register (IIR), the Interruption Space Register (ISR), and the Interruption Offset Register (IOR).
P	Protection identifier validation enable. When this bit and the C-bit are both equal to 1, instruction references check for valid protection identifiers (PIDs). When this bit and the Dbit are both equal to 1, date references check for valid PIDs. When this bit is 1, probe instructions check for valid PIDs.
D	Data address translation enable. When 1, data addresses are translated and access rights checked.
I	External interrupt, power failure interrupt, and low-priority machine check interruption unmask. When 1, these interruptions are unmasked and can cause an interruption. When 0, the interruptions are held pending.

Virtual Memory Support

In a virtual memory system, pages are mapped from one set of addresses, virtual addresses, to another set, physical addresses. Each time the processor accesses memory, it does so using a virtual address. This address then gets translated to a physical address. This mapping takes place in units of *pages*. When a program addresses a virtual address, that address is first broken up into a virtual page number and an offset into that virtual page. The virtual page is then translated into a physical page. It's possible that the virtual page might be on disk rather than in physical memory. In this case, a page fault is generated and the page must then be brought in from disk to memory. In this section we look at how addresses get translated. The case where data is resident on disk rather than in physical memory is covered in Chapter 6.

PA-RISC processors operate in one of two modes: narrow or wide. PA-RISC 1.1 processors are always narrow, and they operate on 32-bit addresses. PA-RISC 2.0 processors can be narrow or wide at any time depending on the setting of the W-bit in the PSW. Because the addresses are handled differently in these two cases, we look at both possibilities. We also look at the memory layout for both kernel mode and user mode. Recall that on a PA-RISC 1.1 system, the kernel and user applications are always narrow (32-bit). On a PA-RISC 2.0 system, the kernel can be narrow or wide (64-bit); on a wide kernel, applications can be either narrow or wide.

Virtual Memory Layout, Narrow Mode

As discussed, the entire set of virtual memory is divided into *spaces*. Every address is made up of a space and an offset into that space.

In narrow mode, we use a 32-bit offset into a space, which means that every space is 2^{32} or 4 GB in size. This 4 GB is in turn divided into four *quadrants* of 1 GB each.

Each process on the system has four quadrants of address ranges that it can use. This doesn't mean that it has 4 GB of memory set aside—just that it has that many unique addresses that it can access. However, *these four quadrants are not necessarily in the same space.* For example, a process might have quadrant 1 from space 0x123, quadrant 2 from space 0x234, quadrant 3 from space 0x345, and quadrant 4 from space 0x456. At the same time, some other process might have its quadrant 2 taken from space 0x123. To put this another way, each space is divided into four quadrants, and each quadrant belongs to a different process. The reason for this will become apparent when we talk about explicit and implicit addressing.

The quadrants are determined by simply dividing the space into four equal pieces as determined by the upper two bits of the offset. Table 1-2 shows how address ranges map to quadrants.

Table 1-2. Quadrants in 32-Bit Mode

Quadrant	Virtual Address Range
Quadrant 1	`0x00000000` to `0x3fffffff`
Quadrant 2	`0x40000000` to `0x7fffffff`
Quadrant 3	`0x80000000` to `0xbfffffff`
Quadrant 4	`0xc0000000` to `0xffffffff`

Virtual Memory Layout, Wide Mode

In wide mode, the virtual memory concepts are the same as in narrow mode, but the register sizes are different and the calculations are done differently. The PA-RISC 2.0 architecture allows for space registers of

up to 64 bits, although all current implementations us 32-bit space registers. Of these 32 bits, only the upper 22 bits are used; the lower 10 bits are always zero. General registers, which are used for the offset, are 64 bits wide. However, only the lower 42 bits are used for the offset. Like in narrow mode, the two highest bits indicate the quadrant. The result is that each quadrant is 4 TB, and each space is 16 TB. Table 1-3 shows how address ranges map to quadrants.

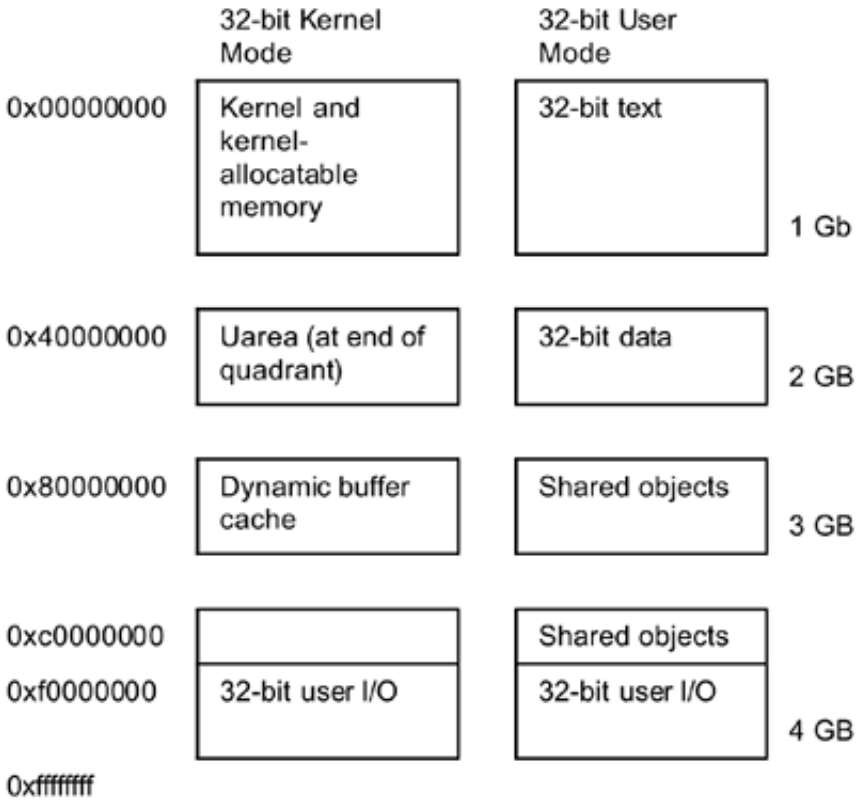
Table 1-3. Quadrants in 64-Bit Mode

Quadrant	Virtual Address Range
Quadrant 1	`0x00000000 00000000` to `0x000003ff ffffffff`
Quadrant 2	`0x40000000 00000000` to `0x400003ff ffffffff`
Quadrant 3	`0x80000000 00000000` to `0x800003ff ffffffff`
Quadrant 4	`0xc0000000 00000000` to `0xc00003ff ffffffff`

32-Bit Address Space Layout

Figure 1-5 shows the layout of memory in both 32-bit user mode and 32-bit kernel mode. Thirty-two-bit kernel mode is used only on PA-RISC 1.1 systems. On a PA-RISC 2.0 system, the kernel itself is always in 64-bit mode. Applications on a PA-RISC 2.0 system can be either 32- or 64-bit applications. On a PA-RISC 1.1 system, applications are always in 32-bit mode.

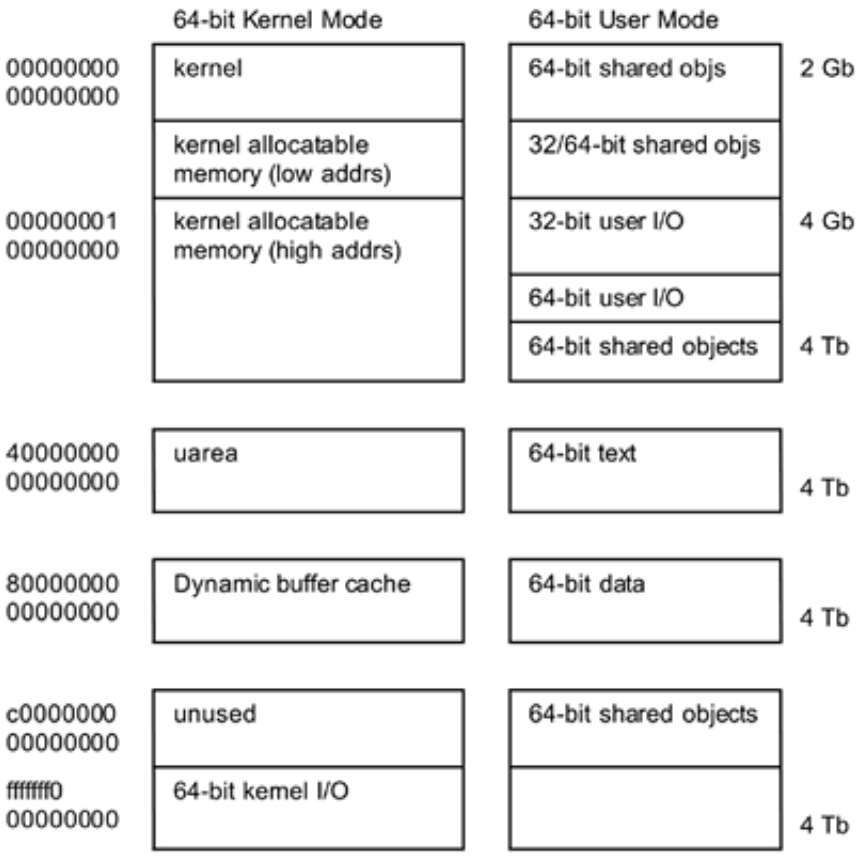
Figure 1-5. 32-Bit Address Space Layout



64-bit Address Space Layout

Figure 1-6 shows the layout of memory in 64-bit mode, both for the kernel and for 64-bit user applications. One of the requirements for the 64-bit user mode is that a 64-bit application must be able to share memory with a 32-bit application. This is called mixed-mode access. In order to accommodate mixed-mode access, an area is set aside in quadrant 1 of the 64-bit layout. This area, starting at the 2 GB offset, corresponds with the beginning of quadrant 3 in 32-bit mode (also at the 2 GB offset).

Figure 1-6. 64-Bit Address Space Layout



Explicit and Implicit Pointers, Narrow and Wide Mode

When operating in narrow mode, we have to work with a 16-bit space ID and a 32-bit offset for a total of 48 bits. For performance reasons, it is desirable to have a method of specifying an address in a single 32-bit register. For this purpose, PA-RISC uses the concept of a *short pointer*. A short pointer uses the upper two bits of the register to specify a space register and the remaining 30 bits to specify an offset. Recall also that those upper two bits indicate the quadrant in which an address resides. Table 1-4 shows the relationship between the high-order bits of an address, the space register used to address it, and the quadrant.

Table 1-4. Space Registers and Quadrants

Bits 0–1	Space Register	Quadrant
00	SR4	Quadrant 1
01	SR5	Quadrant 2
10	SR6	Quadrant 3
11	SR7	Quadrant 4

As an example, the short pointer `0x40000782` refers to offset `0x782` in the space pointed to by space

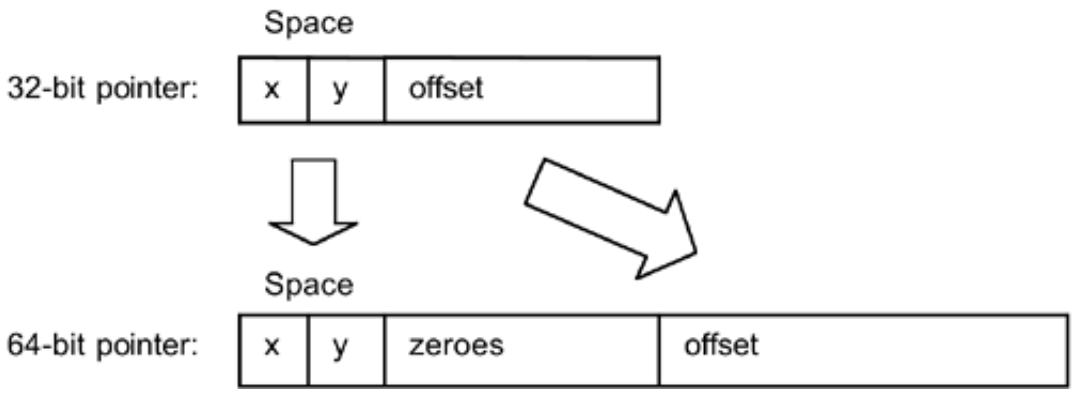
register 5. Another way of saying this is that `0x40000782` is in quadrant 2, which uses space register 5.

In wide mode, our short pointers are 64 bits long. We still use the first two bits to indicate the space register or quadrant, as described in Table 1-4. The following 20 bits in the short pointer are always zero, followed by 42 bits of offset. This gives us the address range of 4 TB in a quadrant. For example, the address `0x40000001'201a39c3` is in the space pointed to by space register 5 at offset `0x1201a39c3`.

Address Swizzling

Address swizzling is the process of converting a 32-bit address to a 64-bit address. When a 32-bit application is running on a 64-bit kernel, any addresses passed to the kernel must be swizzled to 64-bits. This process simply involves copying the upper 2 bits of the 32-bit address into the upper 2 bits of the 64-bit address, zeroing out the next 32 bits of the 64-bit address, and copying the lower 30 bits of the 32-bit address into the lower 30 bits of the 64-bit address. See Figure 1-7.

Figure 1-7. Address Swizzling



This process extends the 32-bit value to a 64-bit value while maintaining the correct quadrant. Note that this happens only for addresses, not for arithmetic values, which implies that the kernel must know that a value passed to it is meant to be an address.

Global Virtual Addresses

PA-RISC 2.0 introduced the concept of a Global Virtual Address (GVA). The GVA is a combination of the space and offset values, which results in a 64-bit unique address. In narrow mode, the space value is 32 bits long and the offset is 32 bits. We simply concatenate these two values to get a 64-bit GVA. For example, if we have an address of `0x3120.0x7ff33aeb`, the GVA is just the concatenation of the two: `0x00003120'7ff33aeb`.

In wide mode, recall that we use a 32-bit space register, but only the upper 22 bits are used; the lower 10 bits are always zeroes. The offset is a 64-bit register, but the upper 22 bits are zeroed, and only the lower 42 bits are used. This allows us to concatenate the 22 bits of space register and 42 bits of offset to give a 64-bit GVA. We simply OR the 32-bit space register value with the upper 32 bits of the offset to get the 64-bit GVA. For example, if we have an address of `0x624b400.0x3ff'ff7c34e8`, the resulting GVA is `0x0624b7ff'ff7c34e8`.

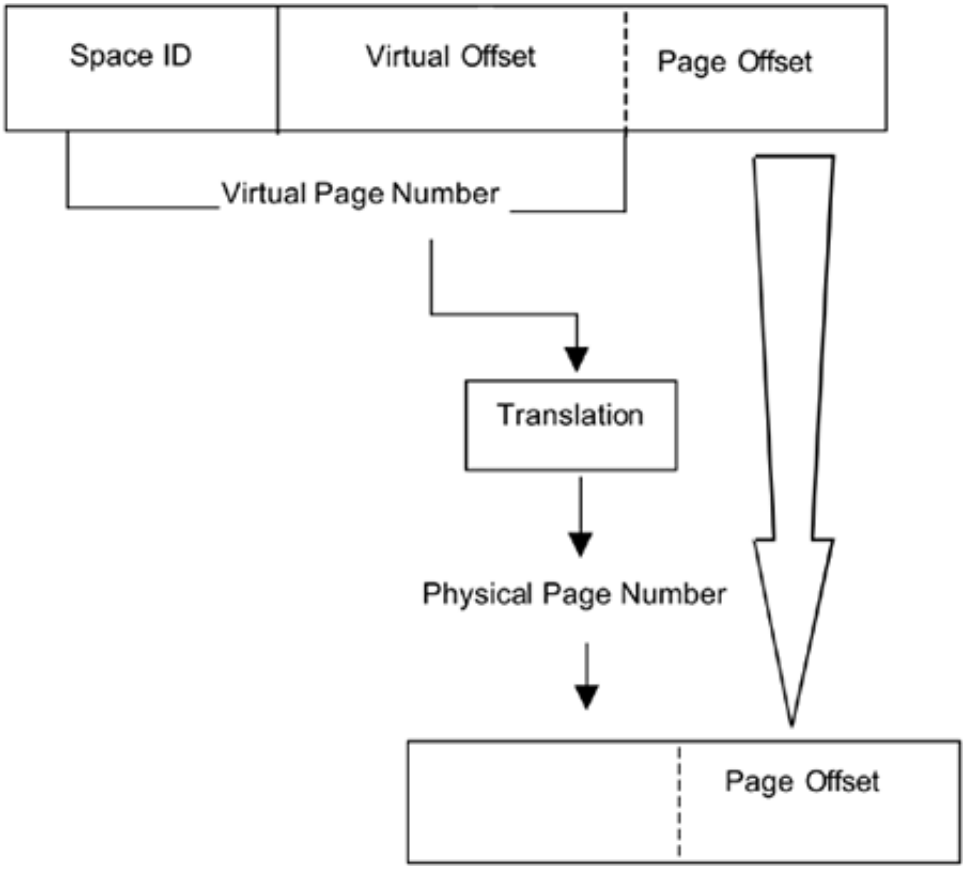
This 64-bit GVA is used as the basis for address translation. In the next section we see how the 64-bit GVA is translated to a physical address.

Address Translation

Address translation is the process of translating a virtual address to a physical address. Before we begin the translation process, the virtual address is divided into a virtual page number (VPN) and an offset into that page. The VPN is then translated into a physical page number (PPN), and the offset is used as the offset into that physical page.

In PA-RISC 1.1, all pages are a fixed 4 KB long. This means that the offset portion of a virtual address is the lower 12 bits, and the VPN is the upper 20 bits. With PA-RISC 2.0, we introduced the concept of variable-sized pages. This means that the physical pages can be from 4 KB to 64 MB. However, for the purposes of address translation, we assume a 4-KB page size. Thus, we still use the upper 20 bits as the VPN. Once the VPN is translated to a PPN, we can determine how big the actual page is and thus how many bits of the address are used as the offset into the page. Figure 1-8 illustrates this translation process.

Figure 1-8. Translation of Virtual Address to Physical Address



Address Translation Components

There are several components that are used to translate a virtual address to a physical address. The first of

these is the PDIR, a table that lists pages currently in memory. It includes the VPN and PPN of each page and flags indicating the status of the entry.

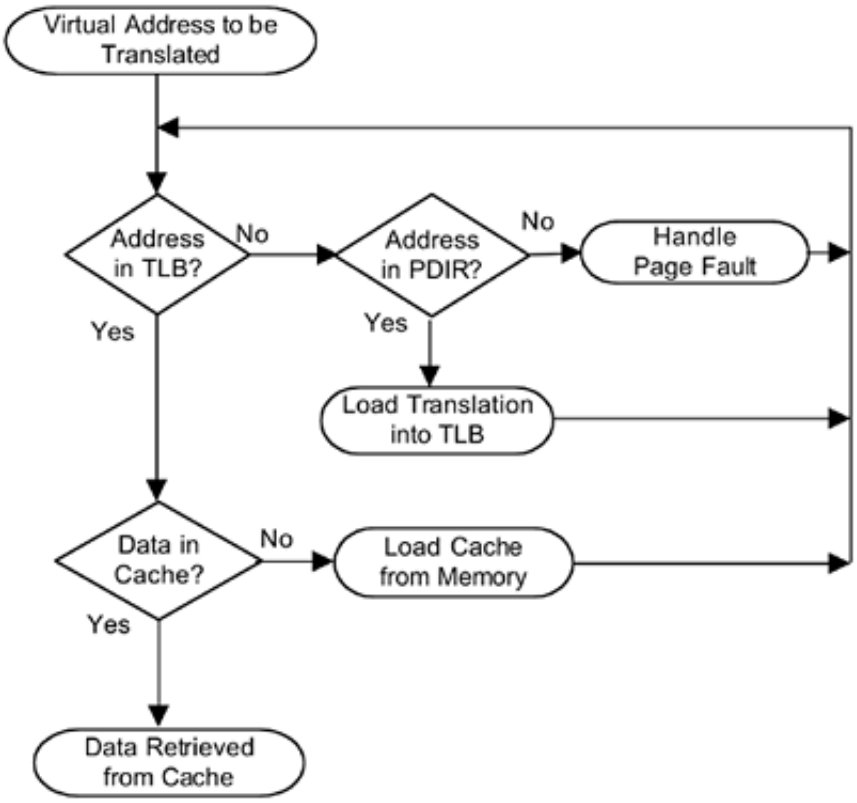
PA-RISC systems also have a TLB. This is a hardware version of the PDIR. It is not large enough to hold all of the PDIR entries, but it is much faster to reference the TLB than it is to reference the PDIR. Recently and frequently used PDIR entries are kept in the TLB for fast access. In this way, the TLB functions like a cache for PDIR entries.

The system also has one or two memory caches. Systems may have separate caches for instructions and data, or they may have a combined cache. While the cache isn't strictly used for address translation, it is part of the process of finding the required data given a virtual address.

When the system wants to access a particular virtual address, it first checks the TLB to see if the VPN is in there. If it is not, then it goes to the PDIR to see if the entry is there. If the entry is found in the PDIR, it is inserted into the TLB and the process starts over. If the entry is not found in the PDIR, a page fault is generated and the kernel must bring the page in from disk.

Once the PPN has been obtained from the TLB, the cache is checked to see if the address is in the cache. If so, then the data is loaded from the cache. If not, the cache is loaded from memory and the process is restarted. The flowchart in Figure 1-9 illustrates this process.

Figure 1-9. Address Translation through the TLB and Cache



Translation Lookaside Buffer

The TLB is a hardware unit that holds mappings from virtual pages to physical pages. Each TLB entry has the VPN and the corresponding PPN. In addition, the TLB entry is used for controlling access to pages. Each

entry has a field for access rights and an access ID. In PA-RISC 2.0, with its support for variable-sized pages, the TLB also must have a field indicating the size of the physical page. Finally, each entry has a set of flags indicating the state of the entry. The actual layout and operation of the TLB is not significant. For an understanding of the HP-UX kernel, it is sufficient to know that the TLB is there and what it does.

Access Control

The TLB is also responsible for controlling access to pages. This is handled on two different levels. The first makes use of protection IDs and the access ID field in the TLB. This determines whether a particular process or context has access to a page. The second makes use of the instruction privilege level and the access rights field in the TLB. This determines whether the CPU's current privilege level is sufficient to access the page.

Each TLB entry has an access ID field, which is 15 to 18 bits long on PA-RISC 1.1 and 15 to 31 bits long on PA-RISC 2.0. Each processor also has several protection IDs associated with it. These protection IDs are stored in control registers 8, 9, 12, and 13. Each protection ID is 32 bits long. On PA-RISC 1.1, with its 32-bit wide registers, four protection IDs are available. On PA-RISC 2.0, each register can hold two protection IDs for a total of eight. Once a match is found in the TLB for a particular VPN, the system then checks the access ID field in the TLB against each of the protection IDs. The access ID must match one of the protection IDs for the access to be allowed. The lowest bit of each protection ID is a write disable bit. If the operation is a write, then at least one of the matched protection IDs must have its low bit off, indicating the protection ID is valid for a write operation.

Each TLB entry also has an access rights field, which indicates what privilege level is required for reading, writing, and executing addresses on the page. Recall that PA-RISC provides for four privilege levels, but only two of these are used by HP-UX. Privilege level 0 is used by the kernel, and privilege level 3 is used for user processes. The current privilege level is coded into the lower 2 bits of the instruction address. Because every instruction is 32 bits long, the address of an instruction must always be a multiple of four bytes. In other words, the lower 2 bits of the instruction address are never needed for addressing an instruction. Instead, these 2 bits encode the current privilege level. The access rights field in the TLB insures that only processes with the correct privilege level can access a page. An example of this is requiring privilege level 0 in order to access kernel private data. If a user process attempts to access kernel data directly, it will fail the access rights check.

Both of the above access checks must pass in order for an access to be allowed. If either check fails, a Protection Fault as generated.

TLB Miss Handlers

Recall that the TLB is a limited resource and holds only a subset of the translations in the PDIR. If the system does not find the VPN it is looking for in the TLB, it generates a TLB miss. The information must then be retrieved from the PDIR and inserted into the TLB.

Some PARISC 1.1 systems have a hardware TLB miss handler, which can search the PDIR in memory looking for a match. The entries in the PDIR are actually stored using a hashed chain structure, discussed in detail in Chapter 6. Because of this the hardware, TLB miss handler cannot see all entries in the PDIR. In addition, the TLB miss handler doesn't do access checking, so once the new entry is inserted into the TLB, it still needs to be have its access verified.

There also is a software implementation of the TLB miss handler for systems without the hardware TLB miss handler and for cases where the hardware is unable to find the PDIR entry.

Both the hardware and software miss handlers perform the same function: check the PDIR for a translation of a VPN. If the translation is found, it is inserted into the TLB. If the translation is not found, a Page Fault is generated so that the page can be brought in from disk.

PA-RISC provides instructions for inserting entries into and deleting entries from the TLB. These instructions are used by the TLB miss handlers to manage the TLB.

Cache Organization

Once the address has been translated, the next step is to retrieve the data from that address. The cache stores data in blocks of data called *cache lines.* A cache line is the unit of data passed between main

memory and cache. This size can vary between systems, but a typical implementation uses a 32-byte cache line. Each cache line has a *cache tag* that goes along with it and describes the data. This cache tag includes the physical page that the data came from as well as flags indicating the state of the data.

The cache is accessed by a hash function using the VPN—the same hash function used by the TLB. This means that given a VPN, the system can simultaneously check the TLB for the translation and access rights and also look for the data in the cache. Once the system has the PPN from the TLB and the PPN from the cache, it compares the two to see if the cached data is really from the correct page. Because the hash function can map multiple VPNs to a single entry, it is possible that the cache data found by the hash function will belong to a different physical page.

If the PPNs from the cache and the TLB match, then the system can use the data from the cache. If the PPNs don't match, then we have a *cache miss*, and the system must retrieve the data from main memory.

PA-RISC provides instructions for flushing cache to memory. The cache data will be written to main memory if the "dirty" flag is set in the cache tag, indicating that the cache data is newer than the memory data. There is also an instruction for purging the data cache, which simply flags the tag as invalid. Note that there are no instructions for inserting data into the cache. Cache insertion is handled automatically by the CPU when main memory is accessed.

Interruptions

Normally, the system executes instructions sequentially, one after the next, unless it hits a BRANCH instruction. The BRANCH instruction is covered in Chapter 2 when we talk about procedure calling conventions. The other time the system will interrupt the normal flow of processing is in the case of an interruption. An example might be an I/O card signaling that a transfer is complete or a page fault due to a page not being present in memory.

Interruptions are categorized into four classes:

- A *fault* happens when the system cannot execute the current instruction due to a temporary problem, such as a page being unavailable. The instruction will usually be restarted after the condition is corrected.
- A *trap* happens when the current instruction causes an error, such as an arithmetic overflow or a violation of protection rights on a page. The instruction normally will not be restarted.
- An *interrupt* is generated by an entity outside of the current instruction stream, such as an I/O card requesting attention.
- A *check* occurs when the system hardware has detected a hardware malfunction such as a parity error.

While these terms are well-defined in the PA-RISC architecture, they are not strictly adhered to within the HP-UX kernel. For example, HP-UX refers to a "Protection Fault" when this is strictly a trap, not a fault. However, all four types of interruptions are handled in the same way, so it's really just a matter of terminology. Also note that the terms *interrupt* and *interruption* are sometimes confused. Strictly speaking, an event (other than a BRANCH) that causes the flow of control to change is an *interruption*. One particular type of interruption is an *interrupt*, which is a signal from an external device.

Each possible type of interruption is given a predetermined priority within the system, and the various types are organized into interruption groups. Table 1-5, from Gerry Kane's *PA-RISC 2.0 Architecture*, shows the interruption groups, interruption numbers, and the description of the type of interruption.

Table 1-5. Interrupts and Interruption Groups, in Priority Order

Group	Interruption	Description
Group 1:	1	High-priority machine check
Group 2:	2	Power failure interrupt
	3	Recovery counter trap
	4	External interrupt
	5	Low-priority machine check
	29	Performance monitor interrupt
Group 3:	6	Instruction TLB miss fault or instruction page fault
	7	Instruction memory protection trap
	8	Illegal instruction trap
	9	Break instruction trap

	10	Privileged operation trap
	11	Privileged register trap
	12	Overflow trap
	13	Conditional trap
	14	Assist exception trap
	15	Data TLB miss fault or Data page fault
	16	Nonaccess instruction TLB miss fault
	17	Nonaccess data TLB miss fault or Nonaccess data page fault
	26	Data memory access rights trap
	27	Data memory protection ID trap
	28	Unaligned data reference trap
	18	Data memory protection trap or Unaligned data reference trap
	19	Data memory break trap
	20	TLB dirty bit trap
	21	Page reference trap
	22	Assist emulation trap
Group 4:	23	Higher-privilege transfer trap
	24	Lower-privilege transfer trap
	25	Taken branch trap

Interruption Vector Table

PA-RISC uses an Interruption Vector Table (IVT) to determine how to handle each type of interruption. The address of the beginning of the IVT is store in control register CR14. Each entry in the table is 32 bytes (8 words) long. Note that these entries are not just pointers to routines—they are actual code, up to eight instructions. In many cases that code will just be a `BRANCH` to the correct handler, but this doesn't have to be the case.

Interruption Handling

The four different groups of interruptions become significant when we look at how interruptions are handled.

Group 1 consists of just the HPMC. This interruption is generated when the hardware detects an unrecoverable error. It can occur at any time asynchronously with the instruction stream and is acted on immediately.

Group 2 interruptions are checked immediately at the beginning of an instruction cycle. These interruptions are handled before the next instruction is fetched.

The system next checks the N-bit in the PSW, which indicates a nullified instruction. A nullified instruction is an instruction that should be skipped. In Chapter 2, we talk about how and why it is used. For now, it's enough to know that if the N-bit is on, the next instruction isn't even fetched—we just increment the program counter and start the next cycle.

If the N-bit is not set, the system fetches the next instruction, begins execution of the instruction, and advances the program counter. If the execution of the instruction causes a Group 3 interruption, the program counter is decremented, the PSW is reset to its previous state, and the interruption is handled. This means that when the interruption handler finishes, the instruction will be restarted. If the instruction causes a Group 4 interruption, the handler is branched to directly without backing up the program counter, so the instruction will not be restarted.

Interruption Save State

When an interruption occurs, the system must save its current state before branching to the interruption vector. Control registers 17 through 22 are used to save the interruption state. For each interruption, the system takes the following actions:

- The PSW is saved in CR22. For groups 2 and 3, this will be the PSW at the beginning of the instruction. For group 4, it will be the PSW at the end of the instruction.
- The PSW is cleared except for the W-, E-, and M-bits. This has the effect of disabling all lower priority interruptions.
- The current instruction space and offset are stored in control registers 17 and 18.
- The privilege level is set to 0.
- The instruction that was executing at the time of the interruption is stored in CR19.
- The address associated with the instruction is stored in control registers 20 (space) and 21 (offset). In this way, a faulting address is passed to the fault handlers.
- General registers 1, 8, 9, 16, 17, 24, and 25 are saved by copying them to the shadow registers.
- The system begins executing at the interruption vector address + (32 * interruption number).

Summary

We've gone through an in-depth look at the basic building blocks of the PA-RISC architecture, the way it does virtual memory addressing, and the way it handles interruptions. With these foundation concepts under our belts, we're ready to dive into how the kernel works within this architecture.

Chapter 2. Procedure Calling Conventions

In the preceding chapter, we discussed the system architecture—the design and implementation of the PA-RISC hardware. But in order to make this useful, we need to make some decisions about how we'll use the hardware. For example, we need to agree on how procedure calls are made, how arguments are passed, and how return values are returned. This is known as the runtime architecture of the system. In this chapter, we look at these conventions, and we also take a look at some of the commonly used PA-RISC instructions.

Register Usage

Recall from our discussion of the PA-RISC architecture in Chapter 1 that only four registers have specific meaning: GR0, GR1, GR2, and GR31. The runtime architecture specifies guidelines for many of the remaining registers.

Registers 3 through 18 are designated as *callee save* registers. This means that if these registers are going to be overwritten, it is the responsibility of the *callee* or the *called function* to save their contents. These registers are commonly used for storage of local variables in a function and are stored to the stack before being used. They then must be restored to their original values before returning to the calling program.

Registers 19 through 26 are *caller save* registers. If the calling function wants to preserve these values, it must save them before making the procedure call. The called function is free to modify them without saving them.

For most procedure calls, the arguments to the call are placed in the registers before the call is made. PA-RISC 1.1 specifies four registers for passing arguments. Argument 0 is stored in GR26, Argument 1 in GR25, Argument 2 in GR24, and Argument 3 in GR23. Many of the tools use alternate names for these registers to indicate their usage—arg0 instead of GR26, arg1 instead of GR25, and so on. In addition, the tools generally refer to the registers as r26, r25, and so on rather than GR26 and GR25. In PA-RISC 2.0 the number of argument registers was expanded, so Argument 4 is in register 22 (GR22), Argument 5 is in GR21, Argument 6 is in GR20, and Argument 7 is in GR19. If a procedure call uses more arguments than the number of architected registers, the remaining arguments are passed by putting them on the stack.

General register 30 is used as the stack pointer and is frequently called by the mnemonic instead of GR30. The stack grows toward higher addresses in memory, so as data are added to the stack, the value of GR30 gets larger, and as data are removed from the stack, the value of GR30 gets smaller. This sometimes gets confusing because, by convention, we draw stacks as growing *down*—larger addresses are at the bottom of the stack diagram. PA-RISC does not have explicit PUSH and POP stack operations. The stack is grown or shrunk by incrementing or decrementing GR30.

PA-RISC Instructions

The PA-RISC instruction set can be divided into seven general categories of instructions:

- Memory reference instructions— Instructions that explicitly move data to or from memory.
- Computation instructions— Instructions that perform basic arithmetic, logical operations, shifts and rotates, and bit operations.
- Branch instructions— Instructions to alter the order of instructions execution, either conditionally or unconditionally, with or without saving a link pointer to return to the current instruction stream.
- Multimedia instructions— Instructions that perform multiple parallel operations on 16-bit values packed into a single register. These instructions are designed to speed operations typically used in multimedia applications such as signal processing and compression.
- Floating-point instructions— These instructions perform floating-point operations on values stored in the system's floating-point registers
- System control instructions— These instructions perform control functions such as handling interrupts and managing the cache and TLB.
- Long immediate instructions— Instructions that allow the use of an immediate value.

You don't need to know everything about every instruction available in order to get a good grasp of the kernel, but there are a few instructions that are used commonly or that are important to understanding how procedure calls are handled. We look at the memory reference instructions, branch instructions, and long immediate instructions in more detail. Gerry Kane's *PA-RISC 2.0 Architecture* book has complete details on all the instructions.

Before we get into the details of the instructions, we should mention the format of the mnemonics. Many of the instructions have a *completer* to indicate different types of similar instructions. For example, the mnemonic for a branch instruction is simply `B`. If we want to save a return address, called a branch and link instruction, the mnemonic is `B,L`. The `L` is a completer to the branch instruction. Similarly, the store word instruction, `STW`, can become a store word and modify with a completer of `M: STW,M`.

Some tools will also use simplified mnemonics. The branch and link instruction is used to call a procedure, so it is sometimes shown in a listing as a `CALL` instruction. As another example, there is no "no operation" instruction in PA-RISC, but the assembler will use `OR 0,0,0` as a no-op, and this instruction is frequently decoded as `NOP` in assembly listings.

Memory Reference Instructions

The PA-RISC architecture uses explicit `LOAD` and `STORE` operations for all memory access. `LOAD` instructions bring data from memory to registers, and `STORE` instructions move data from registers to memory. Since PA-RISC uses memory-mapped I/O, the `LOAD` and `STORE` instructions are also used for all I/O operations. There are no explicit I/O operations in the instructions set.

`LOAD`s and `STORE`s can operate on a byte, a halfword, a word, or a doubleword. These instructions are summarized in Table 2-1.

Table 2-1. `LOAD` and `STORE` Instructions

Size of Operand	Load Instruction	Store Instruction

8 bits	`LDB` – Load byte	`STB` – Store byte
16 bits	`LDH` – Load halfword	`STH` – Store halfword
32 bits	`LDW` – Load word	`STW` – Store word
64 bits	`LDD` – Load doubleword	`STD` – Store doubleword

The most common addressing mode for memory uses a *base register* and a *displacement* to specify the target address. The displacement is added to the contents of the register to get the target address. The data at that address is then copied to the specified register (for a `LOAD`), or the contents of the register are copied to that address (for a `STORE`). An example is the instruction

```
LDW 4(r31),r27
```

This instructs the system to add 4 to the value in GR31, use the resulting sum as an address into memory, and load the value from that address into GR27. This base and displacement technique is frequently used for referencing members of a structure through a pointer—a common operation in the kernel. Consider the following code sample:

```
   struct mystruct {
     int           x;          /* 32 bit integers */
     int           y;
     int           z;

};
struct mystruct *s;

q = s->z;
```

At compile time, the compiler knows that the member `z` of the structure is 8 bytes past the beginning of the structure. To load the value of `s->z` into `q`, it first loads the current value of the pointer into a general register, for example, GR3. The load instruction might then be

```
LDW 8(r3), r4
```

The displacement portion of the instruction is the offset from the beginning of the structure, which is known at compile time. The base register holds the pointer to the beginning of the structure, which is not known until runtime.

Base Register Modification

Another variation of the `STORE` and `LOAD` instructions provides *base register modification*. That is, in addition to moving data to or from memory, the instruction also modifies the specified base register. If the displacement is positive, then the base register is modified *after* the load or store is done. This is a *post-increment* operation. If the displacement is negative, the base register is modified *before* the load or store

is done. This is a *pre-decrement* operation.

As an example, consider the following instruction:

```
STW,M r3,0x80(r30)
```

This instruction tells the system to store the value in GR3 at the address specified by GR30, then add 0x80 to GR30.

In contrast, consider the following instruction:

```
STW,M r3,-0x80(r30)
```

Because the offset is negative, the system first subtracts 0x80 from GR30, then stores the contents of GR3 at that address.

We will frequently see an `STW,M` instruction at the beginning of each called procedure. The compiler uses this instruction to allocate a stack frame—that is, to move the stack pointer forward to allow room for storage of local variables. We'll see this in more detail when we look at stack usage.

Branch Instructions

Branch instructions can be divided into two types: conditional and unconditional. As the names imply, a conditional branch may or may not be taken depending on some condition. An unconditional branch is always taken. Within these two types are several different instructions depending on the range of the branch and whether or not a return address is saved.

Unconditional Branch

Table 2-2 shows the types of unconditional branches. The external branches are used to branch to a location in a different space; the vectored branches branch to an address held in a register; and the link branches save the return address in a register.

Table 2-2. Unconditional Branches

Instruction	Definition
B	Branch
B,L	Branch and link
BLR	Branch and link register
BV	Branch vectored
BVE	Branch vectored external
BE	Branch external
BE,L	Branch external and link

When the `L` completer is used on an instruction, the instruction must specify which register is the link

register—that is, which register is to get the return address. Typically, GR2 is used for calls within the space and GR31 for calls to another space. The reason GR2 is used for the `B,L` instruction is that there is a version of `B,L` that implicitly uses GR2 as the link register. By doing this, the 5 bits in the instruction that specify the link register can be used to specify the branch target. When using GR2 as the link register, the `B,L` instruction has a range of ±8 MB from the current instruction. Register GR2 is commonly called rp, for return pointer.

The `BLR` and `BV` instructions are used for branches to target addresses that are farther than 8 MB from the current instruction but still within the same space. `BLR` computes the target address by adding an offset to any of the general registers. This address is then used relative to the current instruction. This extends the range of the branch. The `BV` instruction is not relative to the current instruction—it uses two registers, a base register and a displacement register, to compute the target address. A common use of the BV instruction is to return from a procedure call. By using GR0 (which is always zero) as a displacement and rp as the base, we branch back to the return address saved by the `B,L` instruction that made the call. This instruction is sometimes seen as `BV r0(rp)` and sometimes just as `RET`.

The "external" versions of the branch instruction are essentially the same. The key differences are that they allow the specification of both space and offset as the target address, and they save both space and offset of the return address. They also store the offset of the return address into GR31 rather than GR2.

Conditional Branch

There are four types of conditional branches in PA-RISC:

- Move and branch
- Compare and branch
- Add and branch
- Branch on bits

Each has two forms: one that compares two registers and another that compares a register and an immediate value. The immediate value is 5 bits, giving it a range of –16 to +15. The conditions for branching are specified as completers to the branch instruction. The mnemonics for these instructions are shown in Table 2-3.

Table 2-3. Conditional Branch Instructions

Instruction	Definition
`MOVB`	Move and branch
`MOVIB`	Move immediate and branch
`CMPB`	Compare and branch
`CMPIB`	Compare immediate and branch
`ADDB`	Add and branch
`ADDIB`	Add immediate and branch
`BB`	Branch on bit

The `MOVB` and `MOVIB` instructions either copy a register to another register or copy an immediate value to a register. They then branch to the target address if the specified condition is true based on the value that was moved. Table 2-4 lists the possible conditions for the `MOV` and `MOVIB` instructions.

Table 2-4. `MOVB` and `MOVIB` Instruction Conditions

Condition	Definition
	Never
=	All bits in word are 0
<	Leftmost bit in word is 1
OD	Rightmost bit in word is 1
TR	Always
<>	Some bits in word are 1
>=	Leftmost bit of word is 0
EV	Rightmost bit of word is 0

For example, the instruction `MOVB,= r4,r5,main+0x16` would move the contents of GR4 into GR5. If the value moved was zero, a branch would be taken to the address `main+0x16`; otherwise, execution would continue in sequence.

The `CMPB` and `CMPIB` instructions either compare two registers or compare an immediate value and a register. The instruction then branches to the target address if the specified condition is true; otherwise execution continues in sequence. Table 2-5 shows the possible conditions for the `CMP` and `CMPIB` instructions.

Table 2-5. Conditions for `CMP` and `CMPIB` Instructions

Condition	Definition
	Never
=	Operand 1 is equal to operand 2
<	Operand 1 is less than operand 2 (signed)
<=	Operand 1 is less than or equal to operand 2 (signed)
<<	Operand 1 is less than operand 2 (unsigned)
<<=	Operand 1 is less than or equal to operand 2 (unsigned)
SV	Operand 1 minus operand 2 overflows (signed)
OD	Operand 1 minus operand 2 is odd
TR	Always
<>	Operand 1 is not equal to operand 2
>=	Operand 1 is greater than or equal to operand 2 (signed)
>	Operand 1 is greater than operand 2 (signed)
>>=	Operand 1 is greater than or equal to operand 2 (unsigned)
>>	Operand 1 is greater than operand 2 (unsigned)
NSV	Operand 1 minus operand 2 does not overflow (signed)
EV	Operand 1 minus operand 2 is even

For example, the instruction `CMPIB,<= 7,r3,main+0x18` would compare the value in GR3 to the immediate value 7. If 7 <= GR3 (or, to put it another way, if GR3 > 7), the instruction would branch to `main+0x18`.

The `ADDB` and `ADDIB` either add two registers or add an immediate value to a register. If the result satisfies the condition, then the branch is taken. Table 2-6 lists possible conditions for the `ADDB` and `ADDIB` instructions.

Table 2-6. Conditions for `ADDB` and `ADDIB`

Condition	Definition
	Never
=	Operand 1 is equal to negative of operand 2
<	Operand 1 is less than negative of operand 2 (signed)
<=	Operand 1 is less than or equal to negative of operand 2 (signed)
NUV	Operand 1 plus operand 2 does not overflow (unsigned)
ZNV	Operand 1 plus operand 2 is zero or no overflow (unsigned)
SV	Operand 1 plus operand 2 overflows (signed)
OD	Operand 1 plus operand 2 is odd
TR	Always
<>	Operand 1 is not equal to negative of operand 2
>=	Operand 1 is greater than or equal to negative of operand 2 (signed)
>	Operand 1 is greater than negative of operand 2 (signed)
UV	Operand 1 plus operand 2 overflows (unsigned)
VNZ	Operand 1 plus operand 2 is nonzero and overflows (unsigned)
NSV	Operand 1 plus operand 2 does not overflow (signed)
EV	Operand 1 plus operand 2 is even

As an example, consider the instruction `ADDIB,OD 17,r4,main+0x18`. The value 17 is added to the contents of GR4, and the result is placed in GR4. If the result is odd, then the branch to `main+0x18` will be taken.

The branch on bit instruction examines a bit in a register and conditionally branches based on that bit. The arguments to the instruction are the register, the position of the bit, and the target address. Table 2-7 lists the possible conditions for the `BB` instruction. The position argument can be a fixed bit position, or it can be `SAR`, which refers to the Shift Amount Register (CR11).

Table 2-7. Conditions for `BB` Instruction

Condition	Definition
<	Leftmost bit in word is 1
>=	Leftmost bit in word is 0

Delayed Branching

All branching in PA-RISC uses a feature known as *delayed branching*. This means that the instruction immediately following the branch will be executed *even if the branch is taken.* This instruction following a branch is called the *delay slot*. Consider the following code:

```
LDW0x20(r30), r2

COMIB,>10,r2,main+80

LDW0x24(r30), r3
```

The first instruction loads the value from `0x20` past GR30 into GR2. Next, the compare instruction compares that value to 10 and branches to `main+80` if GR2 < 10. The next `LDW` instruction is the instruction in the delay slot. This instruction will be executed even if the branch is taken. So, if GR2 is in fact less than 10 and we branch to `main+80`, GR3 will get loaded from `0x24` past GR30 before we execute the instruction at `main+80`. Of course, the load will also get executed if we don't take the branch—we'll fall through to it as the next instructions following the branch.

This applies also to a branch and link instruction used for a procedure call. Very frequently you'll find one of the call arguments being loaded by the instruction in the delay slot. For example, consider a call to `func1(x,y)`, where x is at `0x24(r30)` and y is at `0x28(r30)`. The resulting instructions might look like this:

```
LDW0x24(r30),r26

B,L       func1,r2

LDW       0x28(r30),r25
```

The `LDW` instruction in the delay slot loads the second argument into a register before making the procedure call.

The instruction in the delay slot can be ignored or *nullified* by using the `,N` completer on a branch. The instruction in the delay slot may or may not be nullified based on whether the branch is taken and the direction of the branch. If the branch is a forward branch, then the instruction in the delay slot is nullified if the branch is taken. This is because for a forward branch, the construct is usually "if this condition is true, skip forward over the following code." The instruction in the delay slot is part of the code that is skipped if the condition is true. In the normal case where the condition is not met, we fall through to the next instruction as expected.

If the branch is backwards, the delay slot is nullified if the instruction is *not* taken. The reasoning is that a backward branch is typically at the bottom of a loop. The compiler will generally copy the first instruction in the loop into the delay slot of the test at the end. Then the branch back to the top of the loop will branch to the second instruction of the loop. By doing this, we take advantage of the speed gained by using the delay slot. If the test fails and the branch is not taken, we continue on, nullifying the instruction in the delay slot.

Immediate Instructions

An immediate instruction is one in which one of the operands is a constant value, encoded as part of the instruction. Because the PA-RISC architecture uses a fixed instruction size of 32 bits, the size of an immediate operand is limited to only 21 bits. PA-RISC has some interesting tricks for dealing with this limitation, and we look at those here.

The most common example is the add immediate left instruction (`ADDIL`). `ADDIL` takes the 21-bit immediate value, shifts it left by 11 bits, then adds it to a specified register. The result is always placed in GR1. This

gives the system a way to load the upper portion of a 32-bit value as an immediate.

In the kernel, this is commonly seen when loading a global. The address of the global is calculated by the compiler relative to the data pointer (GR27). The compiler then generates an `ADDIL` instruction to load the upper portion of the address into GR1, then an `LDW` relative to GR1 to add the lower portion of the address. For example,

```
ADDIL     -0x2000,dp

LDW       0x328(r1),rp
```

This loads the value from `dp-0x2000+0x328` or `dp-0x1cd6` into register GR2. Some of the disassembly tools recognize this combination of instructions and identify the address that is being loaded. For example,

```
ADDIL     -0x2000,dp            ; r1 = 0x891c40

LDW       0x328(r1),rp          ; rp = st_logging_enabled
```

Procedure Call Model

When a procedure call is made, the following steps must take place.

1. Save all caller save registers (registers 19 through 26). This means if the information in these registers is to be saved, the calling procedure is responsible for saving it on the stack.

2. Store arguments to be passed. The arguments that are passed to the called procedure are placed into registers and/or on the stack. The first four arguments, arg0 through arg3, are placed in registers r26 through r23. If the procedure call requires more than four arguments, the remaining arguments are placed on the stack.

3. Branch to the called procedure. A branch is taken to the called procedure using a `B,L` or `BE,L` instruction. The return pointer is placed into GR2 as a result of the branch.

4. Save the return pointer. The called procedure first saves the return pointer, GR2, onto the stack.

5. Allocate a stack frame for any calls that will be made. The called procedure next allocates space on the stack by incrementing the stack pointer. This space is used for local storage as well as for any procedure calls that it might be making.

6. Save any callee save registers that will be used. Any of the callee save registers (GR3 through GR18) that will be altered by the procedure must be saved on the stack.

Procedure Return Model

1. Extract the return pointer from the stack frame. Before returning to the calling procedure, the called procedure first retrieves the return pointer from the stack and places it in r2.
2. Restore any callee save registers. Any of the callee save registers that have been modified are restored.
3. Deallocate the stack frame. The stack space that has been allocated for this procedure must be deallocated by decrementing the stack pointer.
4. Branch to the return pointer. A BV instruction to GR2 is used to branch back to the calling procedure.
5. Restore the caller save registers. Any of the caller save registers that were saved before the procedure call are restored by the calling procedure.

Stack Usage

PA-RISC uses a stack to keep track of local data and procedure call return information for each procedure call. The PA-RISC architecture does not provide explicit stack manipulation instructions. To manage the stack, the runtime architecture specifies that general register GR30 will be used as a stack pointer. This register is commonly referred to as the sp register.

The stack in PA-RISC grows upward in memory. That is, as data is added to the stack, it goes at increasing addresses and the stack pointer increases. When data is removed from the stack, the stack pointer decreases.

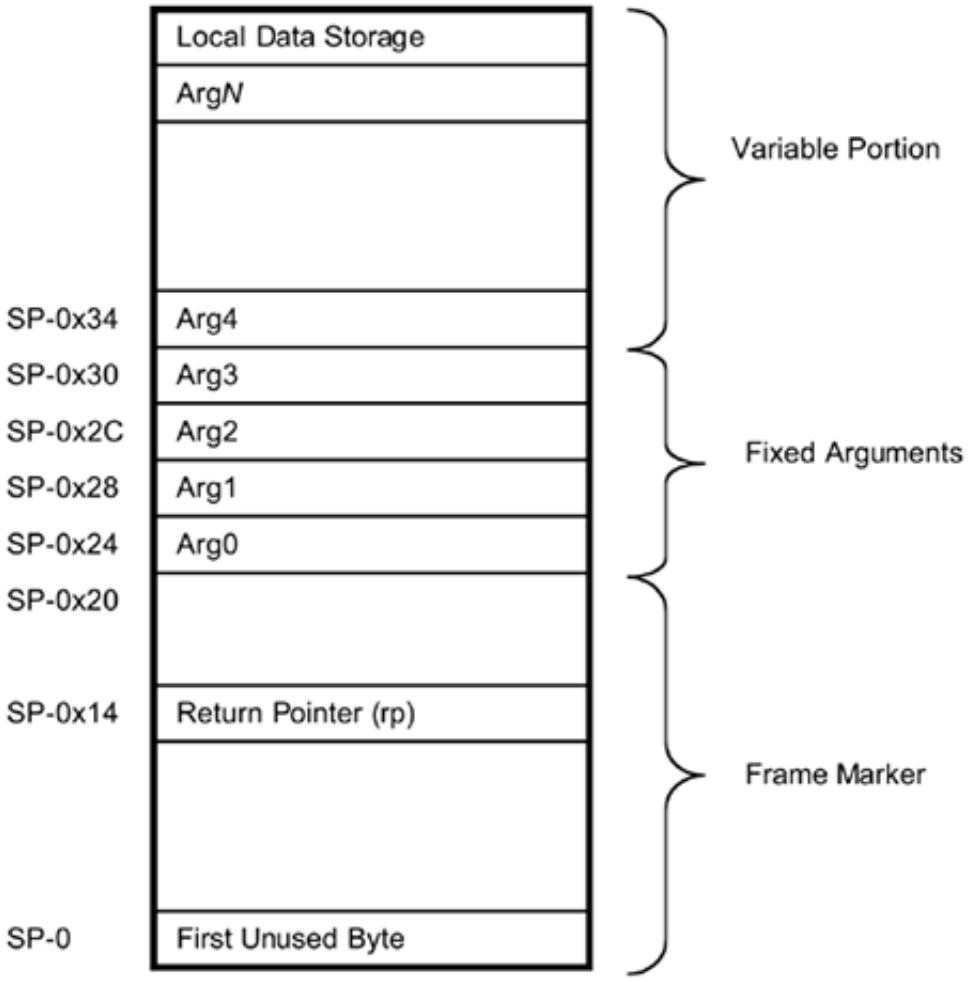
Each called procedure begins by increasing the stack pointer, thus allocating itself memory for local data storage and linkage information. This area is called a *stack frame*. The amount of space allocated for a stack frame depends on how much local data storage the procedure needs.

Stack usage differs between narrow mode and wide mode. We discuss the narrow mode stack usage first, then look at the differences when running in wide mode.

Stack Usage in Narrow Mode

Figure 2-1 shows the stack frame for a narrow-mode call. By convention, we draw the stack with lower addresses higher on the page, and the stack grows *down* to *higher* addresses. The first thing on the stack is a *frame marker*, which is always eight words long. The frame marker for the current process is located at SP-4 through SP-0x20. The space in the frame marker is reserved for a variety of uses, but the only one we need be concerned with is at SP-0x14. SP-0x14 contains the Return Pointer (RP) *of a called procedure.* In other words, when a procedure is called, it saves the return pointer at SP-0x14 in the *calling* procedure's stack frame.

Figure 2-1. Stack Usage in Narrow Mode



The area at SP-0x24 through SP-0x30 is reserved for arguments zero through three. This space is always allocated even if there are fewer than four arguments. Beyond that is a variable-sized area for storing any arguments beyond four. This area is used only if the called procedure takes more than four arguments. Finally, beyond the variable arguments area, the called procedure can allocate additional space to store callee save registers and for local data storage.

Listing 2.1 is an example of C code and the corresponding assembly showing the linkage in the procedure call.

Listing 2.1. Narrow Mode Procedure Call Example

```
#include <stdio.h>

int proc(int x);

main()
{
  int i;
```

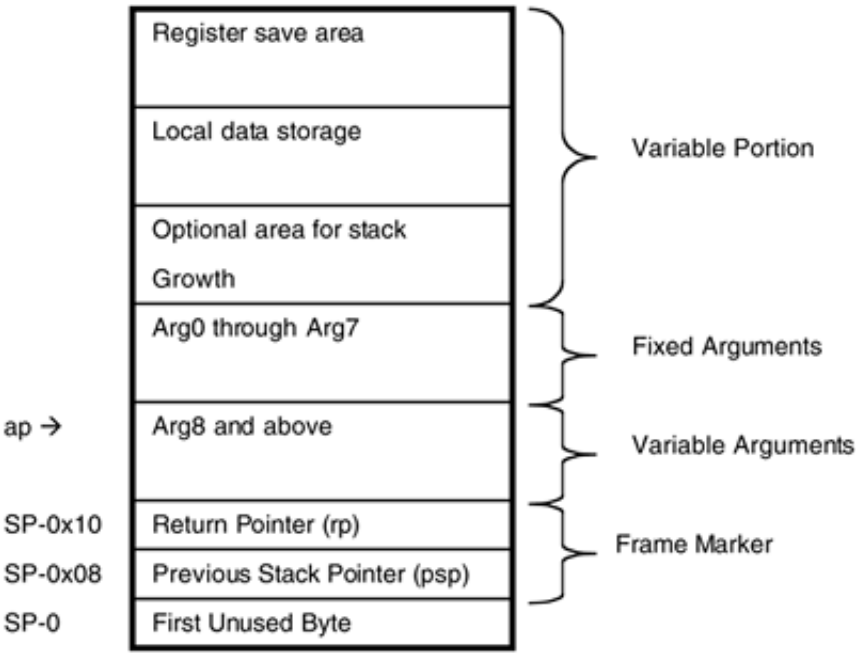
```
  int j;

  i = 10;
  j = proc(i);
  printf("j is %d\n", j);
}
int proc(int x)
{
  int r;

  r = 2*x;
  return r;
}
main starting at 0x28d8:
+0x0          NOP
+0x4          STW         rp,-0x14(sp)          ; save the return pointer
+0x8          LDO         0x40(sp),sp           ; grow the stack
+0xc          LDI         0xa,r1                ; i = 10
+0x10         STW         r1,-0x38(sp)          ; save I on the stack
+0x14         LDW         -0x38(sp),arg0        ; arg0 = i
+0x18         LDIL        0x2800,r31            ; calculate address for proc
+0x1c         BE,L        0x124(sr4,r31)        ; call proc
+0x20         COPY        r31,rp
+0x24         STW         ret0,-0x34(sp)        ; save the return value in j
+0x28         LDIL        0x2800,r31            ; load address of constant
+0x2c         LDO         0x768(r31),arg0       ; string into arg0
+0x30         LDW         -0x34(sp),arg1        ; load j into arg1
+0x34         LDIL        0x2800,r31            ; calculate address of printf
+0x38         BE,L        0xc0(sr4,r31)         ; call printf
+0x3c         COPY        r31,rp
+0x40         LDW         -0x54(sp),rp          ; get the return pointer from
+0x44         BV          (rp)                  ; our caller and branch there
+0x48         LDO         -0x40(sp),sp          ; restore the stack pointer

proc starting at 0x2924:
+0x0          NOP
+0x4          LDO         0x40(sp),sp           ; Grow stack
```

```
+0x8        STW       arg0,-0x64(sp)        ; Store arg0
+0xc        LDW       -0x64(sp),r19         ;  r19 = arg0
+0x10       SHLADD    r19,1,0,r20           ; r20 = 2 * r19
+0x14       STW       r20,-0x28(sp)         ; save r20
+0x18       LDW       -0x28(sp),ret0        ; ret0 = r20
+0x1c       COPY      ret0,ret0
+0x20       BV        (rp)                  ; Branch to rp
+0x24       LDO       -0x40(sp),sp          ; Shrink stack
```

Stack Usage in Wide Mode

Figure 2-2 shows the stack usage for a wide-mode procedure call. Recall that in wide mode all our registers are 64 bits wide. Each entry on the stack is also treated as a 64-bit doubleword.

Figure 2-2. Stack Usage in Wide Mode



The frame marker for wide mode is only two doublewords long to allow for storing two values. The return pointer is stored at SP-0x10. SP-0x08 is reserved for storing the previous stack pointer if needed.

After the frame marker, eight doublewords are allocated to store eight arguments. Additional space may be allocated for arguments if there are more than eight. Unlike in narrow mode, these are stored at *increasing* addresses, not decreasing addresses. In addition, arguments are no longer addressed relative to the stack pointer. Instead, GR29 is set to point to the address of the ninth argument (even if there are fewer than nine arguments, the pointer is set to point to the doubleword after the location reserved for the eighth

argument).

After the area for arguments are optional areas to accommodate stack growth for allocated memory, local storage, and callee save registers.

Listing 2.2 is the assembly language for the same code as above, this time compiled in wide mode. Note that gr9 is decoded as `ret1` by this tool because of its alternate use as a return value.

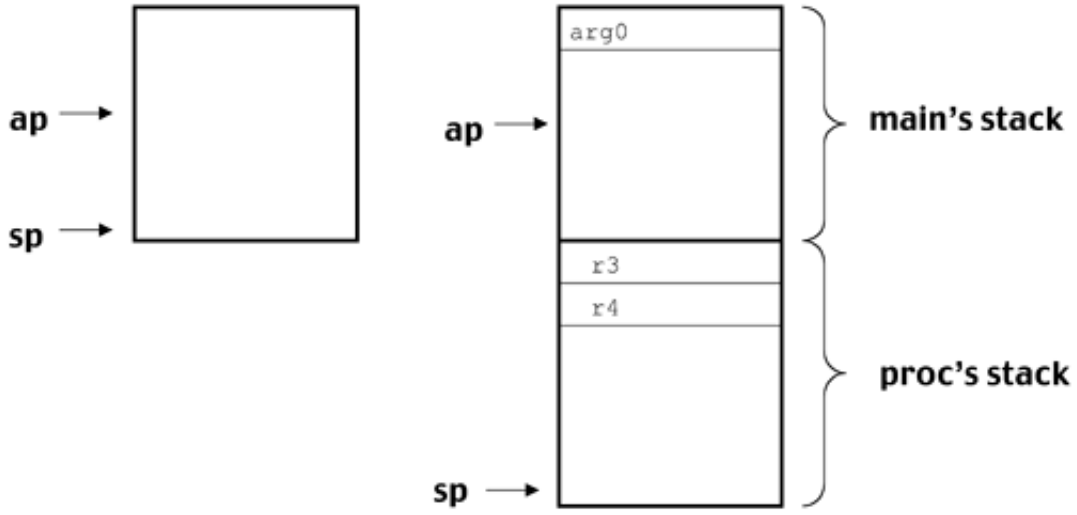
Listing 2.2. Wide Mode Procedure Call Example

```
main starting at 0x4000000000001e08:
+0x0          STD         rp,-0x10(sp)
+0x4          STD,MA      r3,0x90(sp)              ; save r3, grow stack
+0x8          STD         r4,-0x88(sp)           ; save r4
+0xc          COPY        ret1,r3
+0x10         STD         dp,-0x80(sp)
+0x14         MFIA        r1
+0x18         ADDIL       -0x800,r1
+0x1c         LDO         0x7ec(r1),r31
+0x20         LDI         0xa,r19                  ; i = 10
+0x24         STW         r19,-0x78(sp)            ; store i
+0x28         LDW         -0x78(sp),arg0           ; arg0 = i
+0x2c         BLL         0x1e88,rp                ; call proc()
+0x30         LDO         -0x30(sp),ret1           ; ap = SP-0x30

+0x34         LDD         -0x80(sp),dp
+0x38         STW         ret0,-0x74(sp)
+0x3c         ADDIL       -0x800,dp
+0x40         COPY        r1,ret0
+0x44         LDD         0x760(ret0),r19
+0x48         LDW         -0x74(sp),r20
+0x4c         COPY        r19,arg0
+0x50         COPY        r20,arg1
+0x54         BLL         0x1e78,rp
+0x58         LDO         -0x30(sp),ret1

+0x5c         LDD         -0x80(sp),dp
+0x60         LDD         -0xa0(sp),rp
+0x64         LDD         -0x88(sp),r4
+0x68         BVE         (rp)
+0x6c         LDD,MB      -0x90(sp),r3
```

```
proc starting at 0x4000000000001e88:
+0x0          NOP
+0x4          STD,MA     r3,0x50(sp)             ; save r3,
                                                 ; grow stack
+0x8          STD        r4,-0x48(sp)            ; save r4
+0xc          COPY       ret1,r3                 ; r3 = ap
+0x10         NOP
+0x14         STW        arg0,-0x3c(ret1)        ; save arg0 at
                                                 ; ap-0x3c
+0x18         LDW        -0x3c(ret1),r19         ; r19 = arg0
+0x1c         SHLADD     r19,1,0,r19             ; r19 *= 2
+0x20         STW        r19,-0x38(sp)           ; save r19
+0x24         LDW        -0x38(sp),ret0          ; ret0 = r19
+0x28         COPY       ret0,ret0
+0x2c         LDD        -0x48(sp),r4            ; restore r4
+0x30         BVE        (rp)                    ; branch to rp
+0x34         LDD,MB     -0x50(sp),r3            ; shrink stack,
                                                 ; restore r3
```

Figure 2-3 shows the stack usage for the call to `proc()`. You can see that the stack is first grown by 0x50; then GR3, GR4, and G26 (arg0) are stored on the stack. GR3 and GR4 are callee saves, so `proc()` has to store them in its own stack to be restored later. GR26 gets stored into the caller's argument area.

Figure 2-3. Stack Usage for the Call to `proc()`




```
+0x0    NOP
+0x4    STD,MA    r3,0x50(sp)        ; save r3,
                                     ; grow stack
+0x8    STD       r4,-0x48(sp)       ; save r4
+0xc    COPY      ret1,r3            ; r3 = ap
+0x10   NOP
+0x14   STW       arg0,-0x3c(ret1)   ; save arg0 at
                                     ; ap-0x3c
```

Summary

We've seen how the HP-UX kernel uses a stack architecture even though the hardware doesn't explicitly support stacks. We've also looked at how the stack is used in a procedure call—who stores what where. These conventions are used within user applications and within the kernel, guaranteeing that procedures written in one language will be able call procedures in another language.

We've also had a brief look at the PA-RISC assembly language, which will help us in understanding how the kernel works as we dig further into it.

Chapter 3. The Kernel: Basic Organization

The study of a modern operating system leads down many paths and requires that we consider a number of different challenges and their solutions. The HP-UX kernel is a multitasking, multiuser, multiprocessor, multithreaded, load-leveling, modular operating system with real-time scheduling extensions—to list just the highlights. To support such capabilities requires many levels of design abstraction, data tables, and lists as well as a host of subsystems, drivers, and dynamic modules.

In this chapter, we examine the basic organization of the kernel and its data structures, and we consider which are dependent on the underlying hardware platform (hardware dependent-layer, HDL) and which are independent (hardware-independent layer, HIL).

Before addressing HP-UX-specific topics, let's stop and think about what an operating system really is.

A Generic Overview

Approaching an operating system as a whole can be a bit overwhelming, so let's break it down a bit. Think of an operating system simply as a bootable piece of application code. True, it is a somewhat large piece of application code, and it employs many diverse and complex functionalities (even an abstracted form of self-modifying code), but in the final analysis it is still just an executable image!

A programmer must design and create data structures to store and manipulate data in a logical and efficient manner to support the operation and design goals of the program. An operating system designer faces the same challenge, only in spades. Understanding and identifying the resulting data structures and their interaction is a major focus of this book.

The UNIX operating system design is very simple.

This statement usually draws an assortment of looks or comments; indeed, some may question its basic premise or perhaps the qualifications of the one who spoke it! The essence that the statement is meant to draw out is that in the design of a UNIX operating system there are a few key elements (see Figure 3-1).

- First, and most important, a UNIX operating system is responsible for the scheduling and management of individual threads of execution. Later, we discuss processes and threads, but for now let's just consider a thread of execution as an operational piece of code, something that is accomplishing "work" on behalf of a "user." Who gets to run, when, and how long are the issues under the control of the operating system.

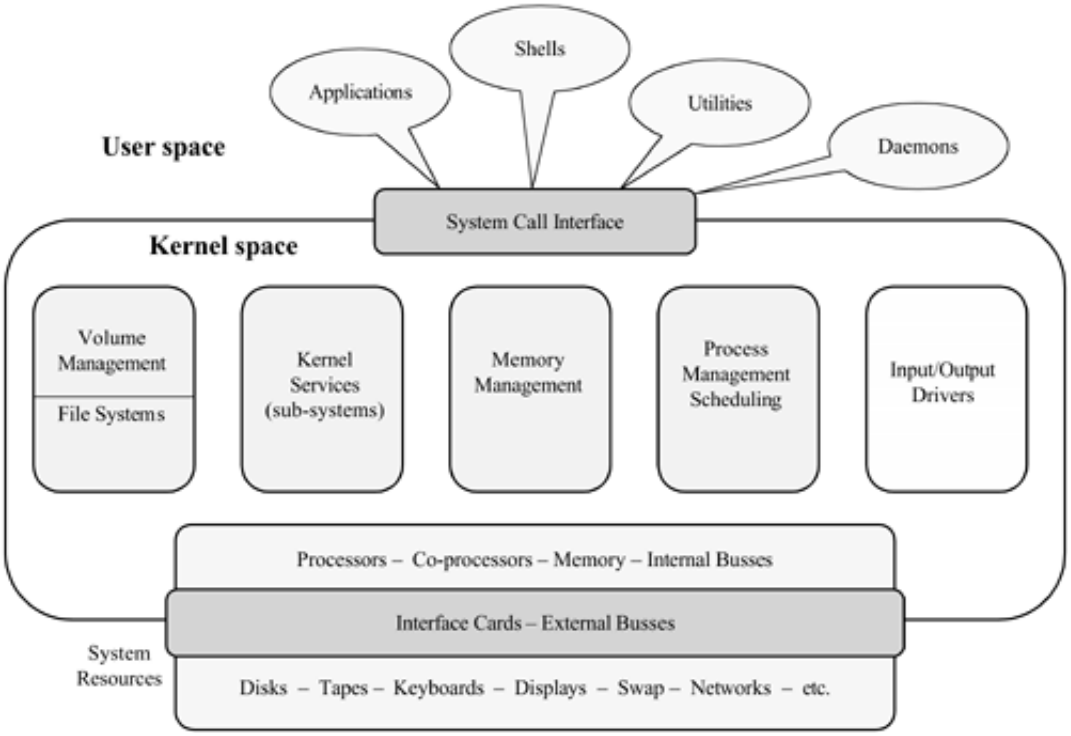
- UNIX is the ultimate control freak—it is prosecutor, judge, and jury for all code that attempts to circumvent its authority. The kernel manages access to all system resources. The only way executing code may make use of a system resource is to request access through well-defined programmatic kernel hooks (called *system calls*).

- All I/O is file I/O. That is to say that all devices are defined by "special" device file handles and treated as if they are simple files.

- All forms of synchronous and asynchronous interruptions are handled by kernel routines. Simply put, if it is unexpected, then let the kernel figure it out.

- System resources should be available to all requestors in a balanced manner when possible.

Figure 3-1. The Big Picture



These are the prime directives of a UNIX kernel's statement of design. While this may seem an oversimplification, it is our hope that by setting these elements as our focal points and by relating other topics and features back to them, we will stay on track.

The UNIX kernel is very modular in its design and evolution. The design was influenced in large part by the C programming language—itself a very modular language—in which the bulk of UNIX kernel code is written. A lot of the personality of the original "kernel hackers" (a term used with great respect) is still present in the heart and soul of today's UNIX incarnations.

When studying a UNIX operating system, always be vigilant for similarities between different sections of the kernel code; that is, look for variations on a theme. It will help your comprehension to compare and contrast the various programmatic tools and tricks of the trade as they come into focus during subsequent chapters.

All I/O Is File I/O

Continuing with the theme of simplification, let's think about the fundamentals of process I/O and the UNIX kernel. The UNIX kernel is charged with policing all forms of I/O as part of its resource management duties. To simplify this task, UNIX has reduced all types of I/O to the level of file I/O. By representing all external devices as files, the system needs only one type of access-control mechanism. This is the infamous "rwx rwx rwx," user group and other (or UGO) security model presented in all introductory-level UNIX training courses and books. UNIX basically follows the KISS (keep it simple, stupid) principle of design.

Abstraction: A Fundamental of Kernel Design

How is an external tape drive or the key on a keyboard reduced to a file path? Through the use of abstraction, a filename (known as a *device file* or *special file*) references a driver in the kernel and passes operational parameters to it (more on this in Chapter 10, "I/O and Device Management." For now, suffice it to say that things are often not what they appear to be at first glance. The kernel contains many layers of abstraction and indirection—smoke and mirrors, my friend, smoke and mirrors. Our challenge is to blow away the smoke and study the reflections in the mirrors.

Is It Real or Is It Virtual?

A major portion of the UNIX operating system is devoted to the management of and translation between "real" and "virtual" addressing modes. From the viewpoint of a process, all possible memory locations fall within a logical address range: 32-bit applications are called *narrow*, and 64-bit applications are called *wide*. The kernel also comes in both narrow and wide versions, usually dictated by the width of the processor architecture on which it is running.

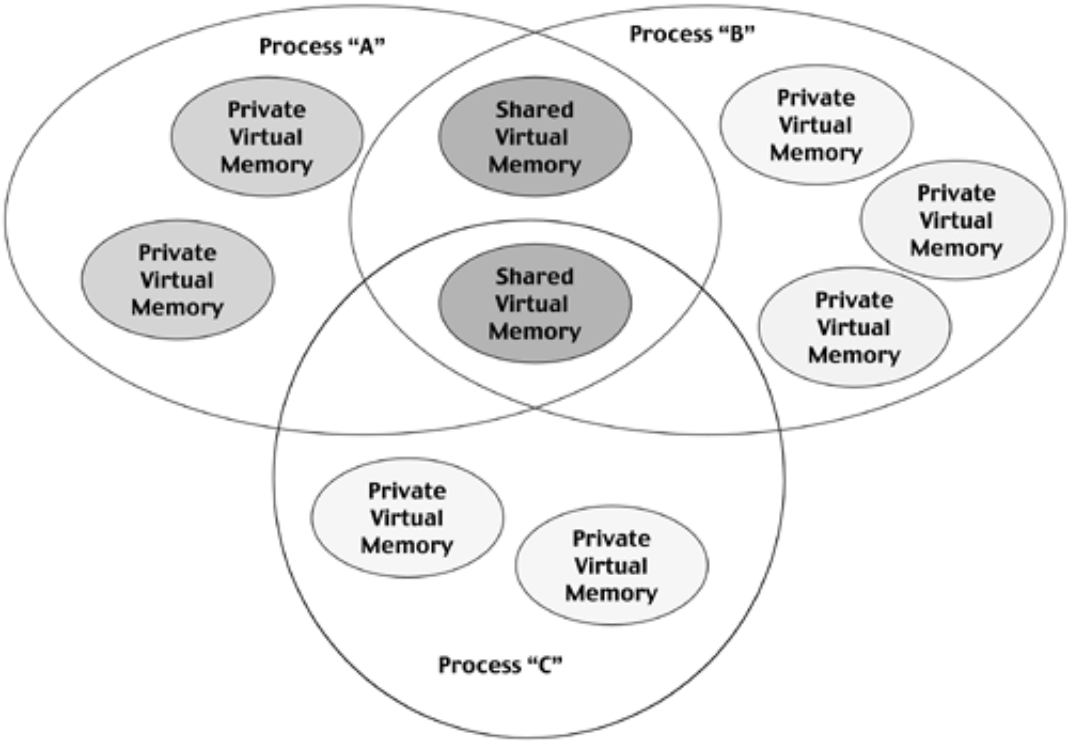
When a program's source code is compiled, references to individual execution modules, library routines, and data elements are stored as symbolic names. This type of code module is called an *object module*. A symbol table is required to define the name and attributes of each item in the table.

When all the related modules of a program have been collected, a linking-loader is used to create the final executable image. The *loader* orders all the individual items within the executable image, and the *linker* replaces all the symbolic references with the logical addresses of the objects that have been loaded. The resulting "image" is fixed within the process's logical address range. As the kernel and all of its processes must share the available physical, or "real," memory, some type of translation must be performed between the process's logical address and the system's physical address. To facilitate this abstraction, an address translation scheme is used.

Most UNIX operating systems employ a concept known as *virtual addressing*. In a virtual memory system, the kernel maintains and manages an address space that is many times larger than the physical memory size addressable by the hardware. This address space exists only as an organizational definition and requires constant translation to true physical addresses during program execution. A virtual memory system requires hardware support as well as implementation in kernel code.

The major advantage of a virtual memory system implementation is that it allows many processes to coexist within the virtual address space (VAS). Each process is allowed its own logical view. Some regions of the virtual space are kept private for a single process; others may be maintained by the kernel as shared regions (see Figure 3-2). This is the basis for shared code and data objects, and is the focus of an entire chapter later in this book (Chapter 10, "Memory Management").

Figure 3-2. Virtual Memory Objects, Private and Shared


Process "A"
Private Virtual Memory
Private Virtual Memory
Shared Virtual Memory
Process "B"
Private Virtual Memory
Private Virtual Memory
Private Virtual Memory
Shared Virtual Memory
Private Virtual Memory
Private Virtual Memory
Private Virtual Memory
Process "C"

Abstraction Layers

We have used the term *abstraction* and should spend a moment talking about what it means with regard to kernel design. When we consider the task of resource management, we must determine the level of control we wish to implement in our management scheme. One of the tricks of the trade involves grouping individual components of a system resource into uniform-sized blocks, chunks, extents, pages, and so on. The availability of the resource is then tracked at the level of these granular units, thus reducing the complexity of kernel data structures.

A classic case is that of memory management. A computer's physical RAM consists of circuits representing single bits of data storage; these are combined into sets of eight and referred to as bytes. An operating system combines bytes into words (for HP-UX, a word is 32 bits, or 4bytes; this is true for both narrow and wide kernels). The word is still a very small amount of storage space, and if the kernel needed to manage each word (by manage, we mean keep track of which words are currently being used, which are free, and who is using what), the amount of memory needed to build such structures could easily require as much space as or more than the memory being managed!

To reduce this management overhead, we make the managed unit size larger than a word. In UNIX, this is accomplished by combining sequential physical words of memory into units called *page frames* (on HP-UX the page frame is 4096 bytes, or 1024 words). Now the task of keeping track of what is free and what is in use becomes much simpler. This is a very basic layer of abstraction; the kernel manages page frames, which you and I know are actually blocks of words made up of bytes that are 8-bits each.

A modern UNIX kernel may use multiple layers of abstraction. Let's continue with our discussion of fundamental memory management. UNIX kernels often employ a scheme whereby a process that requires a number of page frames to hold its code is assigned an appropriately sized *region* in a virtual page-frame map. Virtual page frames are mapped to specific physical page frames by means of processor hardware and supporting kernel tables (discussed later in this book). This additional layer of abstraction greatly simplifies issues such as allowing two or more processes to share the same view of executable code, shared libraries, shared memory, and other process-level shared objects.

There are structures in the kernel to keep track of each element at each abstraction layer. Entities at higher layers simply point to the tracking structures at the lower layers. Lower level resource attributes are inherited by the upper abstraction layers.

Care must be taken in deciding the size of each management unit—too large and you may waste a limited resource; too small and the overhead of the tracking structures may be excessive. The kernel designer constantly walks a fine line between convenience and efficiency. As you study resource management, note the granularity of control the kernel has over its charges.

Some Generic Kernel Techniques

The discussion of operating system internals presents many challenges and opportunities to an author. Our approach is to discuss each area of the kernel, consider the challenges faced by kernel designers, and then explore the path taken toward the final solution implemented in the HP-UX code.

Before we talk about HP-UX specifics, let's discuss some generic challenges faced by kernel designers. As with any programming assignment, there are frequently many different ways to approach and solve a problem. Sometimes the decision is based on the programmer's past experience, and sometimes it is dictated by the specific requirements of each kernel design feature. As an operating system matures, these individual point solutions are often modified or "tweaked" in order to tune a kernel's operating parameters and bring them into alignment with performance objectives or system benchmark metrics. The HP-UX operating system is the product of a continuous improvement process that has enabled the refinement of core features and introduced many enhancements and services over the years.

Kernel Data Structures

Programmers often use algorithms or borrow coding techniques from a "bag-of-tricks" that belongs to the software development community at large. This common knowledge base contains many elements of special interest to those who craft operating system code. Let's explore some of the challenges that kernel programmers face and try to develop a basic understanding of a few of the common programmatic solutions they employ.

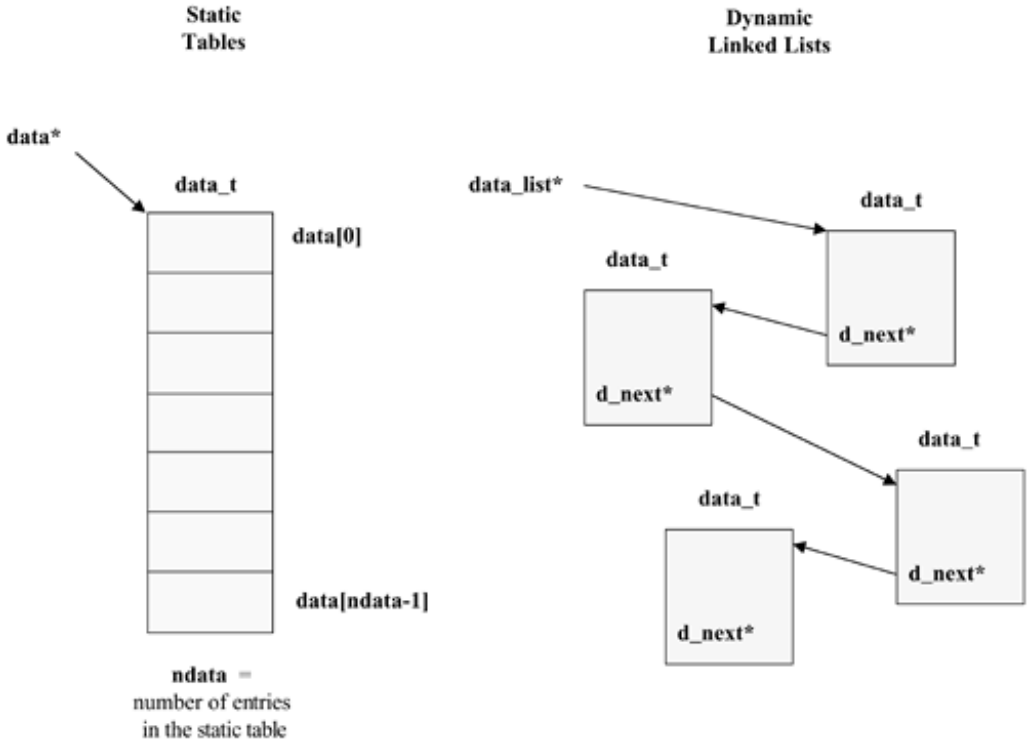
Static Lists (Static Tables)

The kernel often needs to maintain an extensive list of parameters related to some data structure or managed resource. The simplest way to store this type of data is to create an ordered, static list of the attributes of each member of the list.

Each data structure is defined according to the individual pieces of data assigned to each element. Once each parameter is typed, the size (in bytes) of the structure is known. These structures are then stored in contiguous kernel space (as an array of structures) and may be easily indexed for fast access.

As a general observation, and by no means a hard and fast rule, the naming convention of these lists may resemble the following pattern (see Figure 3-3). If the name of the data structure for a single member of the list is defined as `data_t`, then the kernel pointer to the start of the list would be `data*`. The list could also be referenced by an array named `data[x]`, and the number of elements would be stored in `ndata`. Many examples in the kernel follow this convention, but by no means all of them.

Figure 3-3. Tables and Lists



Pros

The space needed for a static list must be allocated during system initialization and is often controlled by a *kernel-tunable parameter*, which is set prior to the building of the kernel image. The first entry in a static data table has the index value of 0, which facilitates easy calculation of the starting address of each element within the table (assuming a fixed size for each member element).

> **Example**
>
> Assume that each data structure contains exactly 64 bytes of data and that the beginning of the static list is defined by the kernel symbol `mylist`. If you wanted to access the data for the list member with an index number of 14, you could simply take the address stored in the kernel pointer `mylist*` and add 14 x 64 to it to arrive at the byte address corresponding to the beginning of the 15th element in the list (don't forget that the list index starts with 0). If the structure is defined as an array, you could simplify the access by referencing `mylist[14]` in your code.

Cons

The main drawback to this approach is that the kernel must provide enough list elements for all potential scenarios that it may encounter. Many system administrators are considered godlike, but very few are truly clairvoyant! In the case of a static list, the only way for it to grow is for the kernel to be rebuilt and the system rebooted.

Another consideration is that the resource being managed must have an index number associated with it wherever it needs to be referenced within the kernel. While this may seem simple at first, think about the scenarios of initial assignment, index number reuse, resource sharing and locking, and so on.

Summary

While this type of structure is historically one of the most common, its lack of dynamic sizing and requirement to plan for the worst case has put it on the hit list for many kernel improvement projects.

Dynamic Linked Lists (Dynamic Tables)

The individual elements of a list must be maintained in a manner that allows the kernel to monitor and manage them. Unlike the elements in a static list, all the elements of a dynamic list are not neatly grouped together in a contiguous memory space. Their individual locations and relative order are not known or predictable to the kernel (as the name "dynamic" indicates).

It is a relatively simple task to add elements to a list as they are requested (providing the kernel has an efficient kernel memory-management algorithm, which is discussed later). Once a data structure has been allocated, it must be linked with other structures of the same list. Linkage methods vary in complexity and convenience.

Once a structure has been allocated and the correct data stored, the challenge is in accessing the data in a timely manner. A simple index will not suffice due to the noncontiguous nature of the individual list elements. The choice is to "walk" the list by following forward pointers inserted into each list element as a means of building a continuous path through the list or to implement some other type of index data structure or hash function. While a hash greatly reduces the access/search time, it is a calculated overhead and must be used each time an item in the list is needed.

An additional challenge comes when it is time for the kernel to clean up a structure that is no longer needed. If the individual elements of the list have been simply linked by a single forward-linkage pointer, then the task of removing a single element from the list can be time consuming. The list element, which points to the element to be removed, must be identified in order to repair the break in the chain that the removal will cause. These requirements lead to the development of bidirectional linkage schemes, which allow for quicker deletion but require additional overhead during setup and maintenance.

Pros

The main attraction to the dynamic list is that the resources consumed by the list are only allocated as they are needed. If the need arises for additional list elements, they are simply allocated on the fly, and a kernel rebuild and reboot are not needed. In addition, when a list element is no longer needed, its space may be returned to the kernel pool of available memory. This could reduce the overall size of the kernel, which may positively affect performance on a system with tight memory size constraints.

Cons

Nothing is free when it comes to programming code! The convenience of dynamic lists comes with several associated costs. The kernel must have an efficient way to allocate and reclaim memory resources of varying sizes (different list structures have different element size requirements).

The challenge of how to link individual list elements together increases the complexity and size of each data structure in the list (more choices to be evaluated by the kernel designer!). The dynamic list creates additional challenges in the realm of searching algorithms and indexing.

Summary

The current movement is clearly toward a totally dynamic kernel, which necessitates incorporation of an ever-increasing number and variety of dynamic lists. The challenge for the modern kernel designer is to help perfect the use and maintenance of dynamic lists. There is ample opportunity here to think outside the box and create unique solutions to the indexing and linkage challenges.

Resource Allocation

An early challenge for a kernel designer is to track the usage of a system resource. The resource may be memory, disk space, or available kernel data structures themselves. Any item that may be used and reused throughout the operational life cycle of the system must to be tracked by the kernel.

Bit Allocation Maps

A bitmap is perhaps one of the simplest means of keeping track of resource usage. In a bitmap, each bit represents a fixed unit of the managed resource, and the state of the bit tracks its current availability.

A resource must be quantified as a fixed unit size, and the logic of the bitmap must be defined (does a 0 bit indicate an available resource or a used resource?). Once these ground rules have been determined, the map may be populated and maintained.

Example

In practices a resource bit map requires relatively low maintenance overhead. The actual size of the map will vary according to the number of resource units being mapped. As the unit size of the resource increases, the map becomes proportionally smaller, and vice versa. The size of the map comes into play when it is being searched: the larger the map, the longer the search may take. Let's assume that we have reserved a contiguous 32-KB block of kernel memory and we want to store data there in 32-byte structures. It becomes a fairly simple issue to allocate a 1024-bit bitmap structure (128 bytes) to track our resource's utilization. When you need to find an available storage location, you perform a sequential search of the bitmap until you find an available bit, set the bit to indicate that the space is now used, and translate its relative position to indicate the available 32-byte area in the memory block.

Pros

The relative simplicity of the of the bitmap approach makes it an attractive first-pass solution in many instances. A small map may be used to track a relatively large resource. Most processors feature assembly language–level bit-test, bit-set, and bit-clear functions that facilitate the manipulation of bitmaps.

Cons

As the size of the bitmap increases, the time spent locating an available resource also increases. If there is a need for sequential units from the mapped space, the allocation algorithms become much more complex. A resource map is a programmatic agreement and is not a resource lock by any means. A renegade section of kernel code, which ignores the bitmapping protocol, could easily compromise the integrity of the bitmap and the resource it manages.

Summary

If a system resource is of a static size and always utilized as a fixed-sized unit, then a bitmap may prove to be the most cost-effective management method.

Resource Maps

Another type of fixed resource mapping involves the utilization of a structure known as a *resource map* (see Figure 3-4). The following is a generic explanation of the approach as there are many differing applications of this technique. In the case of a resource map, you have a resource of a fixed size against which individual allocations of varying sizes need to be made.

Example

For our example, let's consider a simple message board. The message board has 20 available lines for message display; each line has room for 20 characters. The total resource has room for 400 characters, but individual messages must be displayed on sequential lines. Consider posting the two following messages:

House broken

Beagle puppy

Free to good home

12-year-old boy

Seeks lawns to mow

If the lines of each message were not assigned sequential space on the message

board, you could end up with the following mess!

Beagle puppy

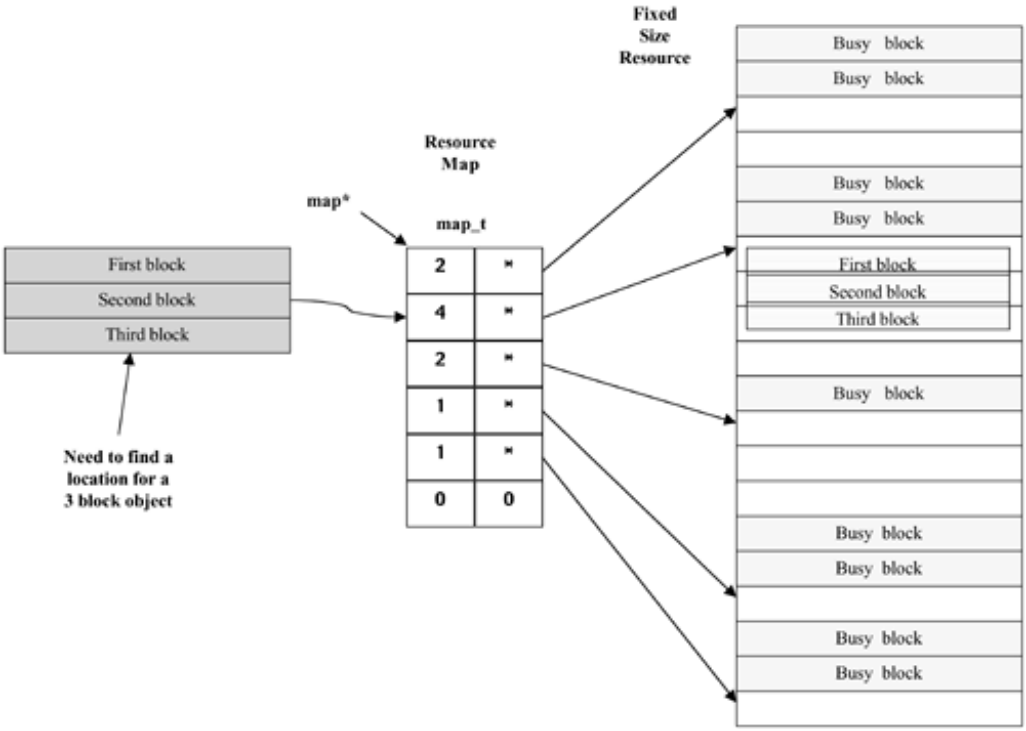
Seeks lawns to mow

House broken

12-year-old boy

Free to good home

Figure 3-4. Resource Maps



To avoid such a situation, a resource map could be employed to allocate sequential lines. Each entry in the resource map would point to the next block of available line(s) on the board.

> If the message board were blank, then there would be only one entry in our resource map pointing to the first line and stating that 20 lines were sequentially available. To list the first message, we would allocate the first three lines from the board, adjust our resource map entry to point to the fourth line, and adjust the count to 17. To add the second message to the board, we would allocate two more lines and adjust the first entry in the map to point to the sixth line, with the count adjusted to 15.

In effect a resource map points to the unused "holes" in the resource. The size of the resource block tracked by each map entry varies according to usage patterns.

Pros

A resource map requires relatively few actual entries to manage a large number of resources. If the allocation block size varies and needs to be contiguously assigned, then this may be your best bet.

Cons

Map entries are constantly being inserted and deleted from the maps. This requires constant shifting of the individual map entries (the saving grace here is that there are relatively few entries). Another concern is the size of the resource map itself: if you run out of entries, then freed resources may not be accounted for and in effect will be lost (a type of memory leak) to system usage until a reboot.

Summary

Resource maps have long been utilized by System V Interprocess communication kernel services, and if care is taken in their sizing, they are very efficient.

Searching Lists and Arrays

Where there are arrays and lists of data structures, there is always a need to search for specific elements. In many cases, one data structure may have a simple pointer to related structures, but there are times when all you know is an attribute or attributes of a desired structure and not an actual address.

Hashtables: An Alternative to Linear Searches

Searching a long list item by item (often called a sequential or linear search) can be very time consuming. To reduce the latency of such searches, *hash* lists are created. Hashes are a type of indexing and may be used with either static arrays or linked lists to speed up the location of a specific element in the list.

To use a hash, a known attribute of the item being searched for is used to calculate an offset into the hashtable (hash arrays are frequently sized to a power of two to assist in the calculation and truncation of the hashed value to match the array size). The hashtable will contain a pointer to a member of the list that matches the hashed attribute. This entry may or may not be the actual entry you are looking for. If it is the item you seek, then your search is over; if not, then there may be a forward *hash-chain* pointer to another member of the list that shares the same hash attribute (if one exists). In this manner, you will have to follow the hash-chain pointers until you find the correct entry. While you will still have to perform a search of one or more linked items, the length of your search will be abbreviated.

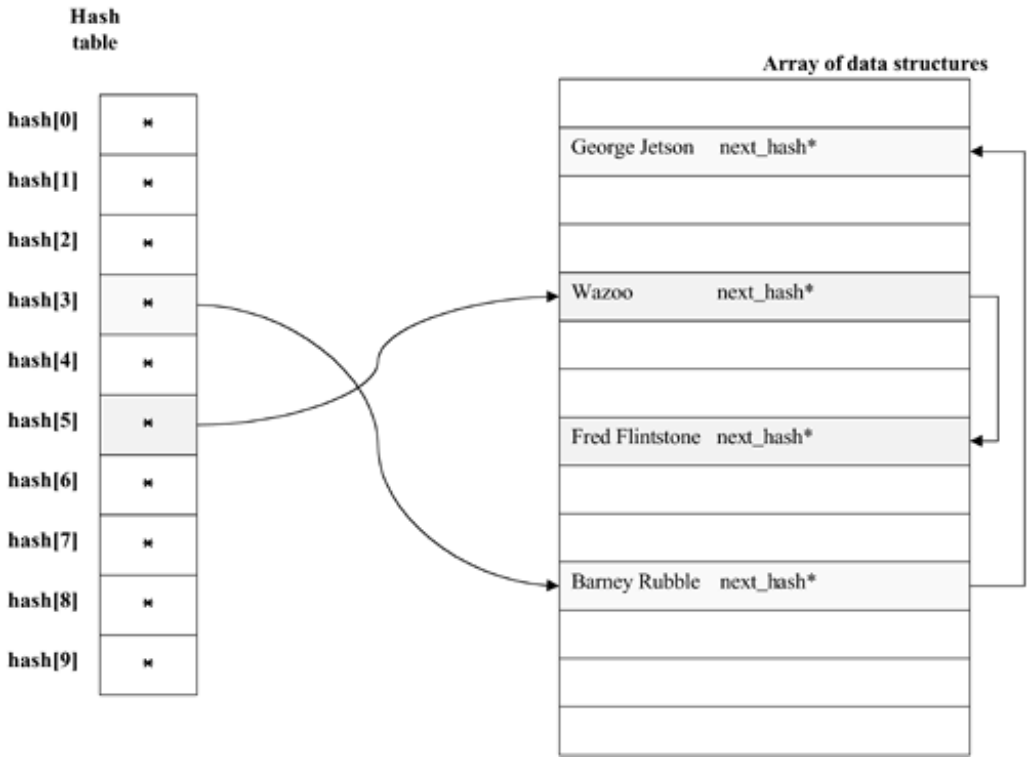
The efficiency depends on how evenly distributed the attribute used for the hash algorithm is within the members of the list. Another key factor is the overall size of the hashtable.

Example

Suppose you have a list of your friends' names and phone numbers. As you add names and numbers to the list, they are simply placed in an available storage slot and not stored in any particular order. Traditionally, you might sort the list alphabetically each time an entry is made, but this would require "reordering the deck." Consider instead using a hashtable.

As each entry is made, the number of letters in each name is counted; if there are more than nine, then only the last digit of the count is kept. A separate *hashtable* array with 10 entries is also maintained with a pointer to a name in the list with that *hash count*. As each name is added to the list, it is linked to all the other list members with the same hash count. See Figure 3-5.

Figure 3-5. Hashtables and Chains



If your list had 100 names in it and your were looking for *Fred Flintstone*, the system would first total the character count (*Fred* has 4 and *Flintstone* has 10, a total of 15 counting the space between the names), which would send us to `hash[5]`. This entry would point to a name whose hash count is 5; if this were not the *Fred Flintstone* entry, you would simply follow the embedded hash-chain pointer until you found Fred's entry (or reached the end of the list and failed the search).

If there were 100 entries in the table and 10 entries in the hashtable, using a standard distribution, then each hash chain would have 10 entries. On average, you would have to follow an individual chain for half of its length to get the data you wanted. That would be a five-linkage pointer search in our example. If we had to perform a sequential search on the unordered data, the average search length would have been 50 elements! Even considering the time required to perform the hash calculation, this could result in considerable savings.

While this example is greatly simplified, it does demonstrate the basics of hash-headers and hash chains to speed up the location of the "right" data structure in a randomly allocated data table.

Pros

Hashing algorithms offer a versatile indexing method that is not tied to basics such as numerical sequence or alphabetic order. Relevant attributes are used to calculate an offset into the hash-chain header array. The header points to a linked list of all items sharing the same hash attribute, thus reducing the overall search time required to locate a specific item in the list.

Cons

The specific attributes used for the hash may be somewhat abstract in concept and must be carefully considered to assure that they are not artificially influenced and do not result in uneven distributions to the individual chains. If the size of the resource pool being hashed grows, the length of the individual chains may become excessively long and the payback may be diminished.

While the basic concept of hashing is very simple, each implementation is based on specific attributes, some numeric, some character-based, and so on. This requires the programmer to carefully study the data sets and identify which attribute to use as a key for the hash. Frequently, the most obvious one may not be the most efficient one.

Summary

Hashing is here to stay (at least for the foreseeable future). Make your peace with the concept, as you will see various implementations throughout all areas of kernel code.

Binary Searches

When it comes to searching a fixed list for a value, there are many approaches. The brute-force method is to simply start with the first element and proceed in a linear manner through the entire list. In theory, if there were 1024 entries in the list, the average search time would be 512 tests (sometimes the item you are looking for would be at the front of the list and sometimes toward the end, so the average would be 1024/2 or 512).

Another method involves the use of a *binary search algorithm*. The decision branch employed by the algorithm is based on a binary-conditional test: the item being tested is either too high or too low. In order for a binary search to work, the data in the list must be ordered in an increasing or decreasing order. If the data is randomly distributed, another type of search algorithm should be used.

Consider a 1024-element list. We would start the search by testing the element in the middle of the list (element 512). Depending on the outcome of this test, we would then check either element 256 or 768. We would keep halving the remaining list index offset until we found the desired element.

Pros

Following this method, the worst-case search length for our theoretical 1024-element list would be 10! Compare this to 1024 for the brute-force linear search method.

Cons

While the reduction in the number of individual comparisons is impressive, the basic list elements must be ordered. The impact of this on list maintenance (adding items to or removing them from the list) should not be underestimated. An unordered list may be easily managed through the use of a simple *free-list* pointer and an embedded linkage pointer between all the unused elements of a list. If the list is ordered, then many of its members may need to be moved each time an item is added or removed from the list.

Summary

We have considered only a very basic form of binary search. Kernels employ many variations on this theme, each tuned to match the needs of a specific structure.

Partitioned Tables

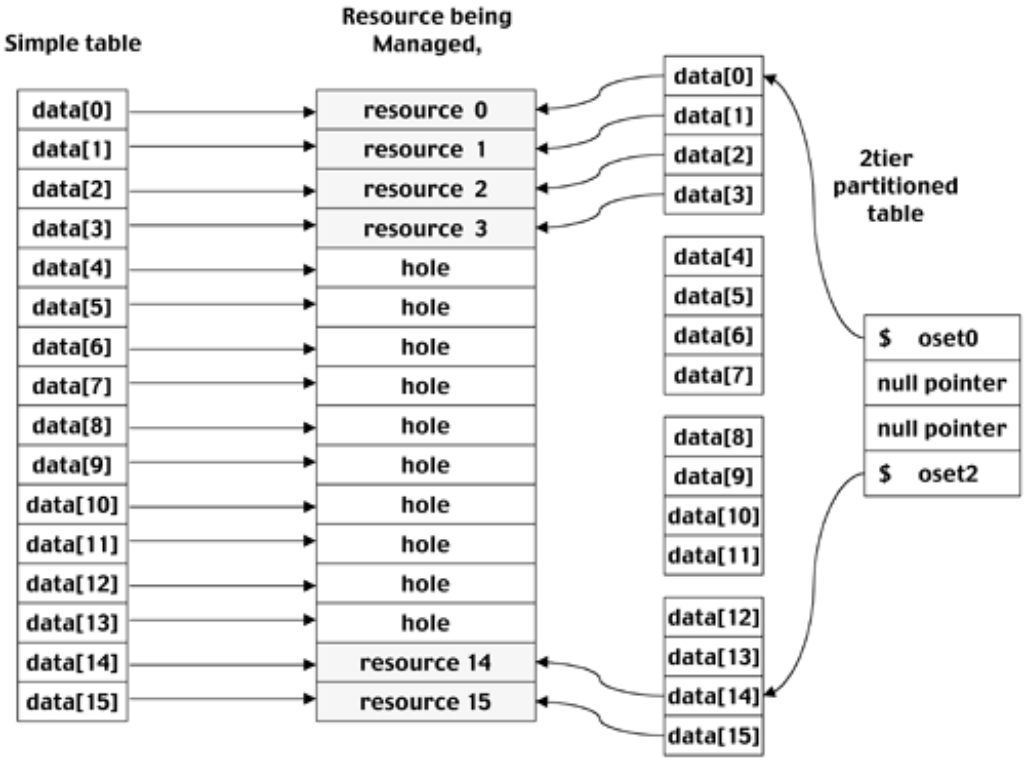
Modern architectures present kernel designers with many challenges; one is the mapping of resources (both contiguous and noncontiguous). Consider the task of tracking the page-frames of physical memory on a system. If physical memory is contiguous, then a simple usage map may be created, one entry per pageframe, the page number would be the same as the index into the array.

On modern cell-oriented systems, there may be multiple memory controllers on separate busses. Often, the hardware design dictates that each bus be assigned a portion of the memory address space. This type of address allocation may result in "holes" in the physical memory map. The use of *partitioned tables* offers a method to efficiently map around these holes.

Example

Consider the greatly simplified example in Figure 3-6. In order to manage a resource of 16, items we could use a simple 16-element array (as shown on the left side of the figure). In this example, there is a hole in the resource allotment; physically, the 5th through 14th elements are missing. If we use the simple array method, 16 elements will still be needed in the array if we want to preserve the relationship between the array index and the corresponding address of the resource.

Figure 3-6. Partitioned Tables



By switching to a two-tier partitioned table, we can map the resources on both sides of the hole and reduce the amount of table space needed. The first tier is a simple array of four elements, each either a valid pointer to a block of data structures or a null pointer signifying that the associated block of resources does not exist.

In addition to the pointer, an offset value is stored in each element. This is used in the case where the hole extends partially into a block's range (as does the last pointer in our example). The offset allows us to skip to the element containing the first valid data structure.

Let's compare the effort required to locate a data structure. If you needed the information related to the 15th resource and were using the simple array approach, you would only have to index into the 15th element of the array (`data[14]`).

If the partitioned approach were being used, you would first divide the element address by the size of the second-tier structures. For our example that would be 14/4, which would yield 3 with a remainder of 2. You would then index into the first-tier array to the fourth element (index = 3), follow the pointer found there, and use the remainder to offset into the partitioned table to the third element (index = 2).

In our simplified example, the single array approach required room for 16 data structures even though there were only six resources being mapped. The partitioned approach required room for only eight data structures (in two partitioned tables of four elements each) plus the very simple four-element first-tier structure.

At first glance, it may not seem that the payback is worth the extra effort of navigating two tables, but this is a very simple example. As we mentioned earlier, the approach is used to manage tables large enough to map all the physical page-frames of a modern enterprise server! There can be millions of potential pages needing their own data structures (and large holes may exist). We will see partition tables in use when we discuss memory management.

Pros

The value of partitioned tables is in the reduction of kernel memory usage. The less memory used by the kernel,

the more available for user programs!

Cons

The method actually has very few drawbacks; the referencing of a map element is now a two-step process. The map element number must be divided by the number of elements in each partitioned table structure (second-tier structure) to yield an index into the first-tier structure. The remainder from this operation is the offset into the second-tier structure for the desired element. In practice, the partitioned tables are often sized to a power of two, which reduces the calculation of the offsets to simple bit-shifting operations.

Summary

Partitioned tables are dictated by the architecture and are a necessary tool in the kernel designer's bag of tricks.

The B-Tree Algorithm

The *b-tree* is a somewhat advanced binary search mechanism that involves making a series of index structures arranged in a type of relational tree. Each structure is known as `bnode`; the number of `bnode`s depends on the number of elements being managed. The first `bnode` is pointed to by a `broot` structure, which defines the width and depth of the overall tree.

One of the interesting and powerful aspects of the b-tree is that it may be expanded on the fly to accommodate a change in the number of items being managed. B-trees may be used to map a small number of items or hundreds of thousands by simply adjusting the depth of the structure.

The simple `bnode` consists of an array of key-value pairs. The key data must be ordered in an ascending or descending manner. To locate a needed value, a linear search is performed on the keys. This may seem to be an old-fashioned approach, but let's consider what happens as the data set grows.

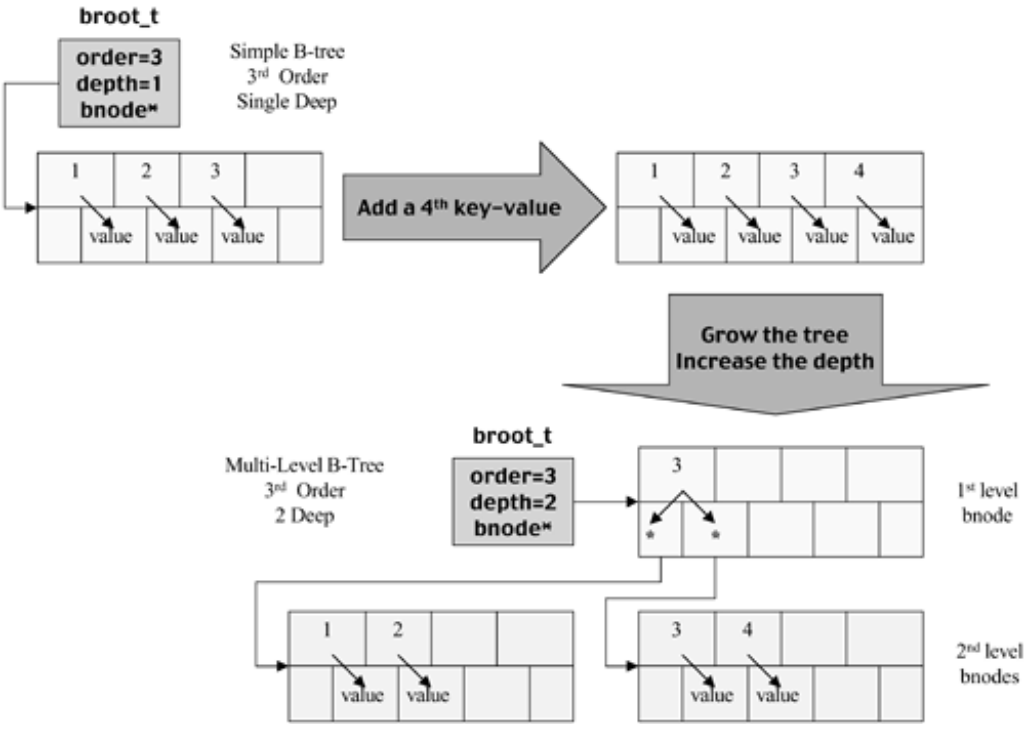
The first issue is the size of the `bnode`. A b-tree is said to be of a particular order. The order is the number of keys in the array structure (minus 1—we will explain this as we discuss a simple example). If we have a third-order b-tree, then at most we would have three keys to check for each search. Of course, we could only reference three values with this simple structure!

In order to grow the scope of our b-tree's usefulness, we have to grow the depth of the tree. Once a b-tree expands beyond its order, additional `bnode`s are created and the depth of the tree is increased.

Only `bnode`s at the lowest level of the tree contain key-value data. The `bnode`s at all other levels contain key-pointer data. This means that in order to find a specific value, we must conduct a search of a `bnode` at each level of the tree. Each search, on average, requires half as many compare operations as there are keys in the `bnode` (the order). This means that the average search length is defined as (order/2) x depth. Optimization is achieved by adjusting both the order and the depth of the b-tree.

> **Example: Growing the b-tree**
>
> From Figure 3-7, consider a very simple example of a third-order b-tree. The original `bnode` has keys: 1, 2, 3. Everything fits in a single `bnode`, so the depth is simply 1.

Figure 3-7. B-Trees



When we add a fourth key, 4, to the tree, it fills the `bnode` to capacity and causes the growth of the tree. If the number of keys in a `bnode` exceeds the order, then it is time to split the node and grow the tree.

To grow this simple tree, we create two new `bnode` structures and move half of the existing key–value pairs to each. Notice that the data is packed into the first two entries of each of the new `bnode`s, which allows for room to grow.

Next, we must adjust the depth value in the `broot` data structure that defines the tree. The original `bnode` must also be reconfigured. First, it is cleared, and a single key is created.

Let's take a closer look at the `bnode` structure. Notice that there are actually more value slots than there are key slots. This allows for two pointers to be created in relation to each key entry. The pointer down and to the left is used if you are seeking a key of a lower value. The pointer down and to the right is used if you are looking for one that is greater than or equal to the key.

Let's try locating a value given a key = 2:

> Follow the `broot` pointer to the first-level `bnode`.

Search for a key = 2. Because there is no perfect match, we look for a key that is > 2 and follow the pointer down and to the left of that key to the next `bnode`.

Search for a key = 2. A match here will yield the appropriate value. We know that this is a value and not another pointer, since the `broot` structure told us we had a depth of two!

Note that the key values do not have to be sequential and may be sparse as long as they are ordered. Searches on the lowest level of the tree return either a value or a "no match" message.

The search for a key always attempts to find a perfect match yielding either a value or a pointer to the next lower level. If no match is found and we are not on the lowest level, the following logic is used. If your search key lays between two adjacent key entries, the pointer to the next level lies below and between them. A search key less than the first key entry uses the first pointer in the `bnode`. If your search key is larger than the last valid key in the `bnode`, then the last valid pointer is used.

Pros

B-trees may be grown dynamically, while their table maintenance is modest. This makes them ideal for managing kernel structures that vary in size. Key values may be packed or sparse and added to the table at any time, providing for a flexible scope.

Cons

Another benefit is that given a sufficiently sized order, a b-tree may grow to manage a large number of items while maintaining a fairly consistent search time. Consider a 15th-order b-tree: the first depth would map 16 items, the second depth would map 256 items, and the third depth would yield a scope of 4096 while only requiring the search of three *bnode*s! This type of exponential growth makes it very popular for management of small to large resources.

The b-tree, binary-tree, and balanced-tree belong to a family of related search structures. While the b-tree has a modest maintenance overhead due to its simple top-down pointer logic, its growth algorithm may result in sparsely populated `bnode`s. This increases the number of nodes required to map a given number of data values. As with most approaches, we trade table size for maintenance cost.

Another issue is that while the b-tree may grow its depth to keep up with demand, it may not change its order (without basically cutting it down to the ground and rebuilding it from scratch). This means that designers need to pay attention to its potential usage when sizing the order in their implementations.

Summary

The b-tree requires a bit of study prior to its implementation but offers an effective method for the mapping of ordered dynamic lists ranging in size from slight to huge. We will see a practical application of the b-tree when we examine kernel management of virtual memory region structures.

Sparse Tables

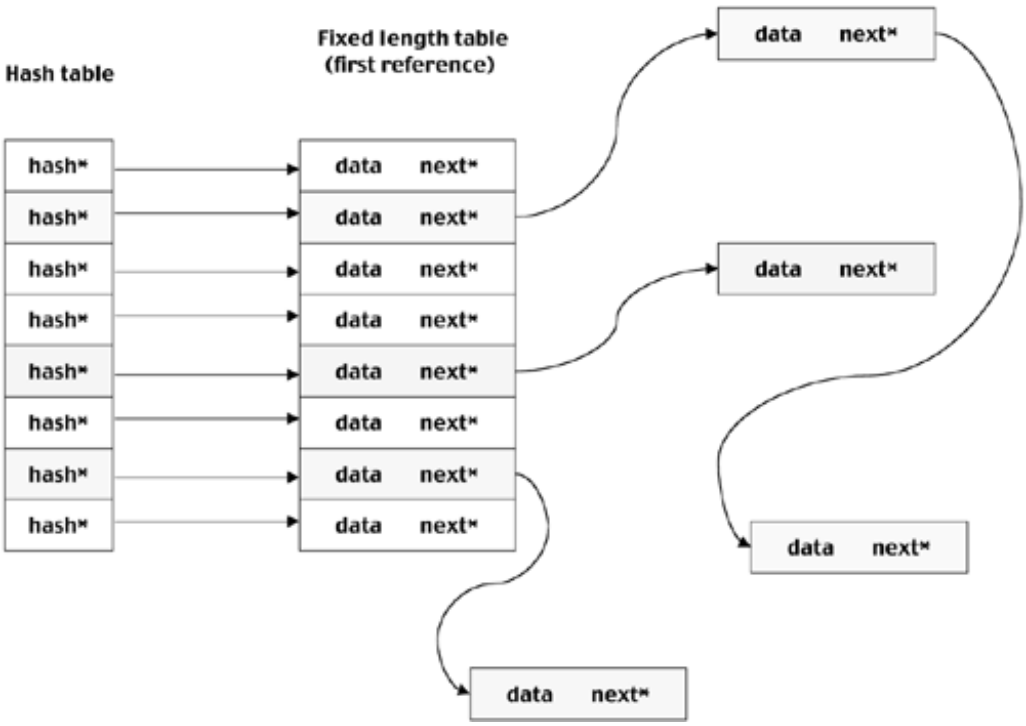
We discussed the use of a hash to speed access to members of a static, unordered array. What would happen if the hash size were aggressively increased (even to the size of the array it referenced or larger)? At first you might think this would be a great solution: simply create a hashing algorithm with enough scope, and lookups become a single-step process. The problem is that the kernel data array and its corresponding hash could become prohibitively large in order to guarantee a unique reference.

A compromise is to size the data structure large enough to hold your worst-case element count and hope that the hashing algorithm is fully distributive in its nature. In an ideal situation, no two active elements would share the same hash.

In the less-than-ideal real world, there will be cases where two data elements do share a common hash. We may solve the problem by dynamically allocating an additional data structure outside the fixed array and creating a forward hash-chain link to it.

Usually, this step is not necessary if the hash formula is sufficiently distributive. In practice, a forward pointer may only be needed in a very small percentage of the cases (less than 1% or 2%). In the very rare case where a third element must share the same hash, an additional structure would be chained to the second, one and so on (reference Figure 3-8).

Figure 3-8. Sparse Tables



Pros

Sparse lists greatly reduce the average search time to locate items in unordered tables or lists.

Cons

Sparse lists require the kernel to manage the available sparse data-element locations as yet another kernel resource. As there is a possibility that the data element first pointed to may not be the actual one you are searching for, the target structure must contain enough data to validate that it is or is not the one you want. If it isn't, a routine needs to be developed to "walk the chain."

Summary

Spare lists work best when there is some known attribute(s) of the desired data set that may be used to generate a sufficiently large and distributive hash value. The odds of needing to create a forward chain pointer decrease greatly as the scope of the hash increases. We will see an example of this approach in the HP-UX kernel's virtual-to-physical page-frame mapping. In actual use, it is a one-in-a-million chance to find a hash-chain with more than three linked elements!

The Skip List

In the world of search algorithms, the skip list is a new kid on the block. Its use was first outlined in the 1990s in a paper prepared for the Communications of the Association for Computing Machinery (CACM) by William Pugh of the University of Maryland. For additional information, visit *ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.ps.Z*.

The algorithm may be employed to reduce search times for dynamic linked lists. The individual list elements must be assigned to the list according to some ordered attribute. This approach works well for linked lists with only a dozen or so members and equally as well for lists of several hundred members.

At first glance, skip lists appear to be simply a series of forward- and reverse-linkage pointers. Upon closer

examination, we see that some point directly to a neighbor, while others skip several in-between structures. The surprising part is that the number of elements skipped is the result of random assignment of the pointer structures to each list member as it is linked into the list.

List maintenance is fairly simple. To add a new member element, we simply skip through the list until we find its neighboring members. We then simply link the new structure between them. The removal of a member follows the same logic in reverse.

When a new member is added, we must decide at which levels it will be linked. The implementation used in the HP-UX **`pregion`** lists uses a skip pointer array with four elements. All members have a first-level forward pointer. The odds are one in four that it will have a second-level pointer, one in four of these will have a third-level pointer, and one in four of these will have a fourth-level pointer. As elements may be added or removed from the list at any time, the actual distribution of the pointers takes on a pseudorandom nature.

To facilitate the method, a symbolic first element is created, which always contains a forward pointer at each of the skip levels. It also stores a pointer to the highest active level for the list. See Figure 3-9.

Example

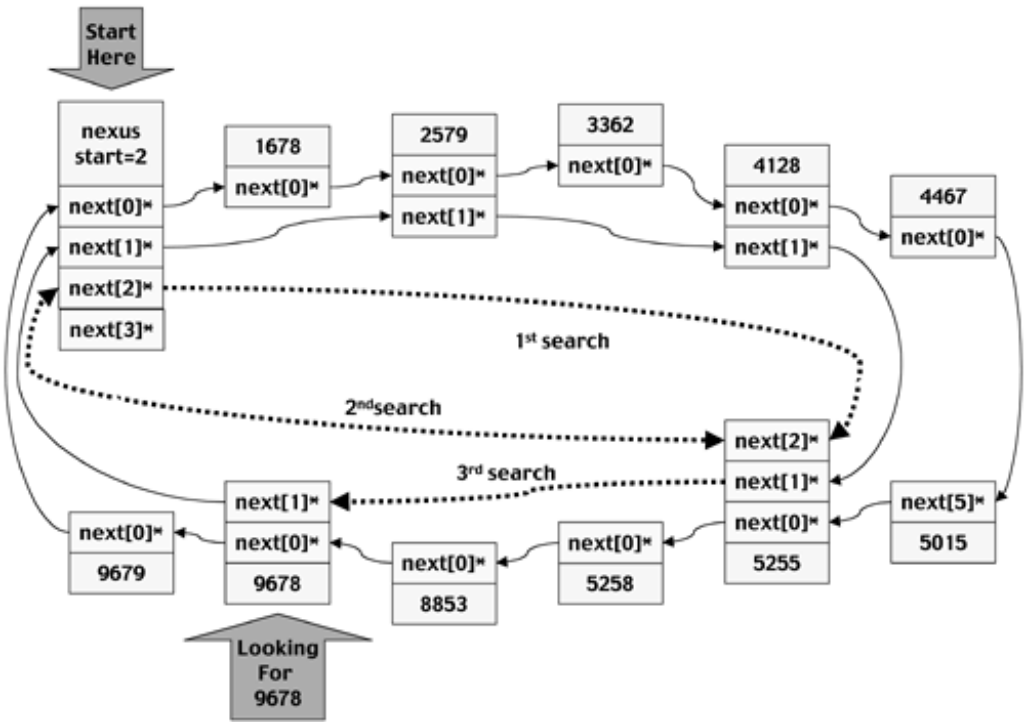
From Figure 3-9, let's assume that we need to locate the structure with the attribute 9678. In the list nexus structure, we see that the highest level active pointer is at `next[2]`, so we follow it. This structure has an attribute value of 5255, so we need to continue our search at this level.

We arrive back at the starting point structure, so we backtrack to the 5255 structure, drop down a level to `next[1]`, and continue.

We now arrive at the structure with the 9678 attribute—it's a match! Our search is over.

In the example, it took only three searches. A simple binary search would have taken four searches.

Figure 3-9. Skip List



Pros

The skip list offers an interesting approach for searching that often results in a reduction of search times when compared to a simple binary method. Adding and removing members to and from the list is reasonably quick.

Cons

It requires the creation of a multielement array for the forward linkages. The random nature of the pointer assignment does not take into account the relative size or frequency of use of the various list elements. A frequently referenced structure might be inefficiently mapped by the luck-of-the-draw (in our example we beat the binary method, but other members of our list would not: try searching for the 5015 structure).

Summary

Despite the random nature of this beast, the overall effect may be a net-sum gain if the ratio between the number of items and the number of levels is carefully tuned.

Operations Arrays

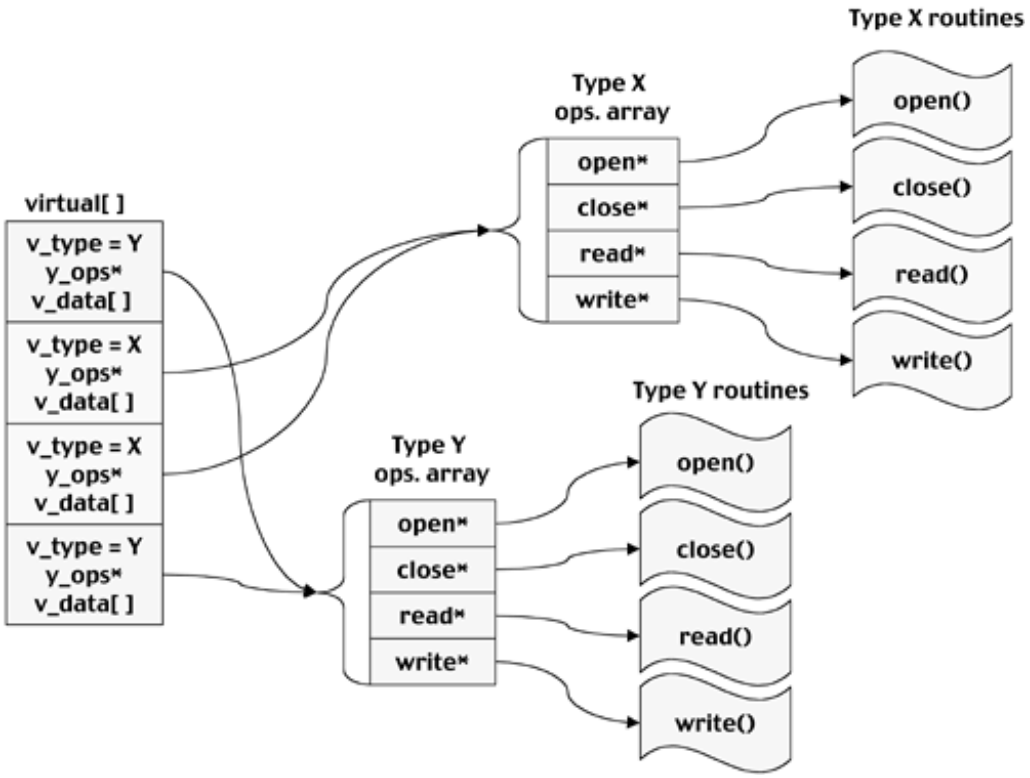
Modern kernels are often required to adapt to a variety of different subsystems that may provide competing or alternate approaches to the same management task. A case in point is that of a kernel that needs to support multiple file system implementations at the same time.

To accomplish this goal, specific file systems may be represented in the kernel by a virtual object. Virtual representation masks all references to the actual object. This is all well and good, but what if kernel routines needed to interact with the real object required code and supporting data dependent upon type-specific attributes? An operations array, or vectored jump table, may be of service here.

Example

Consider Figure 3-10. Here we see a simple kernel table with four elements, each representing a member of a virtual list. Each list member has its actual `v_type` registered, a type-specific `v_data[]` array, and a pointer to a `v_ops[]` operational array.

Figure 3-10. Operations Arrays: A Vectored Jump Table



For this model to work, the number of entries in the operational array and the functions they point to must be matched for each type the kernel is expected to handle. In our example, there are four operational target functions: `open()`, `close()`, `read()`, and `write()`. Currently, our system has only two variations labeled type: `X` and `Y`.

When a routine is called through a vectored jump referenced through `v_ops[x]`, it is passed the address of the virtual objects `v_data[]` array. This allows the final type-specific function to work with a data set type that it understands.

The end result is that all other kernel objects need only to request a call to `v_ops[0]` to instigate an `open()` of the virtual object without concern or knowledge of whether it is of type `X` or `Y`. The operations array will handle the redirection of the call. In practice, we will see many examples of this type of structure in the kernel.

Pros

The cost of redirecting a procedure call through a vector jump table is very low and for the most part transparent to all that use it.

Cons

In debugging, this is yet one more level of indirection to contend with.

Summary

The vectored jump table, or operational array, provides a very effective abstraction layer between type-specific, kernel-resident functions, and other kernel subsystems.

The HP-UX Kernel Overview

Now that we have spent some time considering a generic UNIX kernel, the tools of the trade, and some of the challenges faced by the kernel designers, let's turn our attention to the specifics of the HP-UX kernel.

The current release of the Hewlett-Packard HP-UX Operating System is HP-UX 11.i (the actual revision number is 11.11). We concentrate on the current release, but as many production systems are still running HP-UX 10.20 and HP-UX 11.0, where appropriate we try to cover material relevant to these releases as well.

The HP-UX kernel is a collection of subsystems, drivers, kernel data structures, and services that has been developed and modified for the past 20 years. This legacy has yielded the kernel we present in this book. Over the years, virtually no part of the kernel has gone undisturbed: the engineers and programmers at HP have shown an unwavering commitment to the continuous process-improvement cycle that defines the HP-UX kernel. The authors of this book tip our collective hat to their continuing efforts and vision.

In its current incarnation HP-UX runs primarily on systems built on the Hewlett-Packard Precision Architecture processor family. This was not always the case. Early versions ran on workstations designed on the Motorola 68xxx family of processors. As in the past when HP-UX was ported to the HP-PA RISC chip set, today we are on the threshold of another port of this operating system to an emerging new platform: the Intel IA-64 processor family. In this book, we concentrate on the HP PA-RISC implementation.

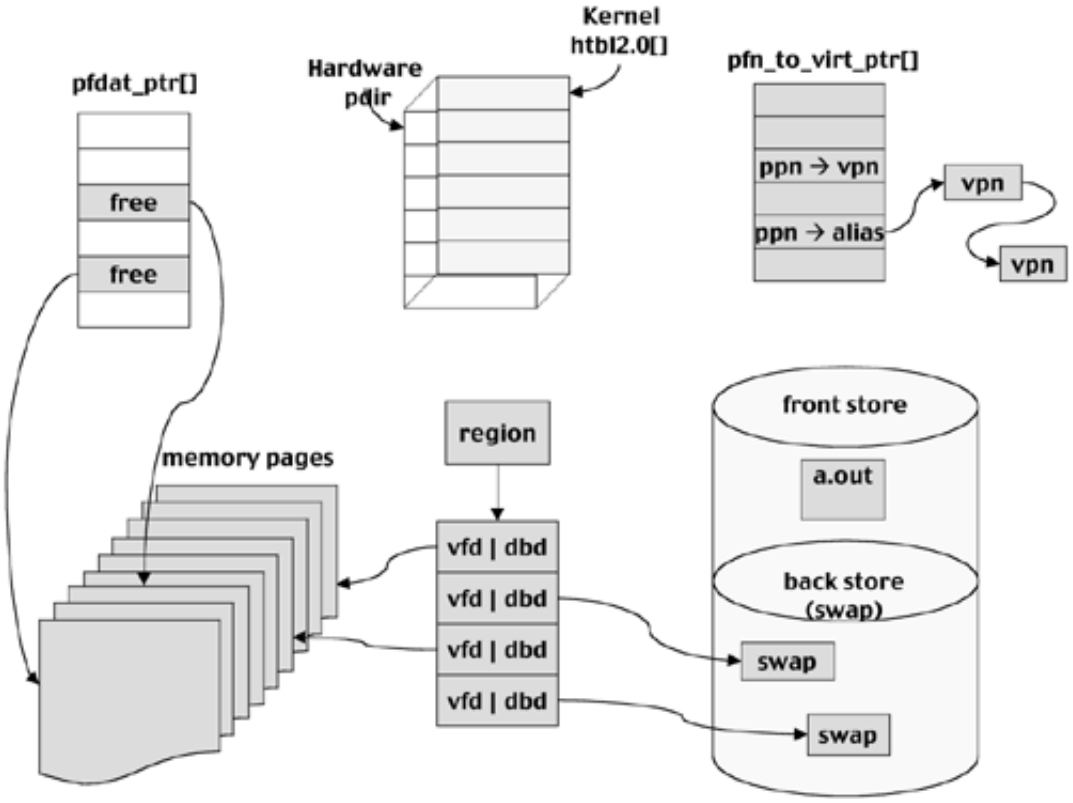
Fundamental Kernel Data Structures: A First Pass

HP-UX kernel data structures are a key focus of our discussion. We break them down into several key areas: kernel memory tables, process tables, disk space tables, file system tables, and input/output tables. As we explore these various areas, keep your eyes open for similarities in approach and design. Many teams of programmers work on the various modules and subsystems that make up the kernel. They frequently borrow methods and algorithms from one another, and there seems to be a never-ending attempt to tweak and tune them for improved performance. This type of crosspollination helps the kernel mature and improve.

Kernel Memory Tables

A prime concern of the kernel is the management of the system memory resources (see Figure 3-11). Memory comes in many flavors and types: physical, virtual, and logical.

Figure 3-11. Kernel Memory Tables



Physical Memory

HP-UX runs on processors that have a 32-bit instruction word size. The primary memory allocation size is

called a physical pageframe. On current HP-UX systems, a pageframe is 4096 bytes (or 1024 words); this has been constant for many years. While this reduces the number of on-demand page-in operations required for a process and its threads, it creates challenges for the memory management schemes. We explore this fully in Chapter 6.

Several primary data structures are required to track and map the system's physical memory. The `pfdat_ptr[x]` array is commonly called the *free-page table* and is used to keep track of which pages are currently in use by the kernel and which have been assigned to a process. This table is a partitioned table to allow for the mapping of physical memory around holes in the physical memory map. As a general rule, if a table name ends in `_ptr`, it is most likely a partitioned table.

With the release of HP-UX 11.i, a process may be assigned larger contiguous sets of physical page-frames under a newly introduced Variable Page Size (VPS) feature. This is also called Performance-Optimized Page size (POPs) in some sales and training literature. To accommodate these features, the `pfdat_ptr` table has been modified to allow the pooling of contiguous free pages into larger views, ponds, and pools of various colors.

Virtual Address Space

The VAS does not reference a physical system entity; instead it is the conceptual memory space onto which the underlying hardware platform (HP PA-RISC) and the kernel must map all potential regions of use. This phantom map is a key concept to master as we study the kernel's theory of operation.

The kernel memory management structures must allow the hardware to map virtual pageframes to the physical page-frames that contain current process code or data. The primary data structure for this task is the `htbl2.0[x]`. If the needed page-frame is not currently memory-resident, then it is up to the kernel to handle the resulting page fault and get it loaded as soon as possible.

The HP-PA RISC hardware as well as the HP-UX kernel requires this virtual-to-physical page-frame map. The hardware calls this table the page directory (or `pdir[x]`) and uses its entries, defined as page data entries (or `pde`s), to update the CPU translation lookaside buffer (TLB). The hardware and kernel names are different to illustrate that the hardware does not specify the use of all the various bits in this structure: the kernel designers use the undefined bits for their own purposes.

> **NOTE**
>
> On the older 32-bit HP PA-RISC–based systems (called narrow systems), this table is named **`htbl[x]`**.

The `htbl2.0[x]` only provides for the mapping of virtual pageframes to physical pageframes. While this is the direction of translation most frequently needed by kernel functions, occasionally there is a requirement to identify which virtual pageframe has been assigned to a particular physical pageframe. This requirement is fulfilled by the `pfn_to_virt_ptr[x]` table. In addition to this basic feature, it is also used to link `alias` data structures if they are required. An `alias` is used if more than one virtual pageframe has been mapped to a single physical page-frame, an important feature allowing copy-on-write semantics during the `fork()` system call.

Process Logical Memory Space

As a matter of concept, we need to consider a process's view of memory. Linking-loaders create executable image files (for C, the common name is *`a.out`*). These files and their headers contain information about which system resources will be needed for the program to run. For a program to run, its page images must be loaded into consecutive pages in the VAS. This is because when the image was created, all references to data and procedure calls were coded as absolute addresses within the process's logical address space.

To facilitate the sharing of process code, dynamic shared library code, shared memory-mapped files, and other shared objects and related consecutive pages in the program's image are said to occupy *regions* of address space. The mapping of these logical process address regions to kernel-managed virtual memory regions is the job of the kernel's many `region` data structures.

The `region` structure contains a database with a page-by-page description that indicates if a page is

currently in physical memory, stored as an image on a front-store (an executable program file), stored as an image on a back-store (a swap page), or still awaiting initialization (used for uninitialized data pages, BSS).

Managing Memory for Internal Kernel Usage

So far, we have discussed only the structures used for managing memory for use by the system's many processes. This type of memory management is done at the granularity of the page-frame. Additional structures are used inside the kernel for the allocation of smaller sized blocks of memory to be used by the kernel's many dynamic tables and linked lists. Until the 11.i release, HP-UX utilized the rather classic "kernel bucket" memory allocation scheme. This has been replaced with an "arena" allocation approach. This change was made to improve flexibility, reduce waste, and facilitate page reclamation.

All Together Now

It may seem at first glance that many structures are playing in the same sandbox. To some degree, this is an accurate assessment, and for it to work, all of the tables must play nice together! Each table has been optimized to provide support for a particular piece of the puzzle and must be meticulously managed to avoid system corruption. There are many levels of checks and balances used to maintain the memory management system's integrity.

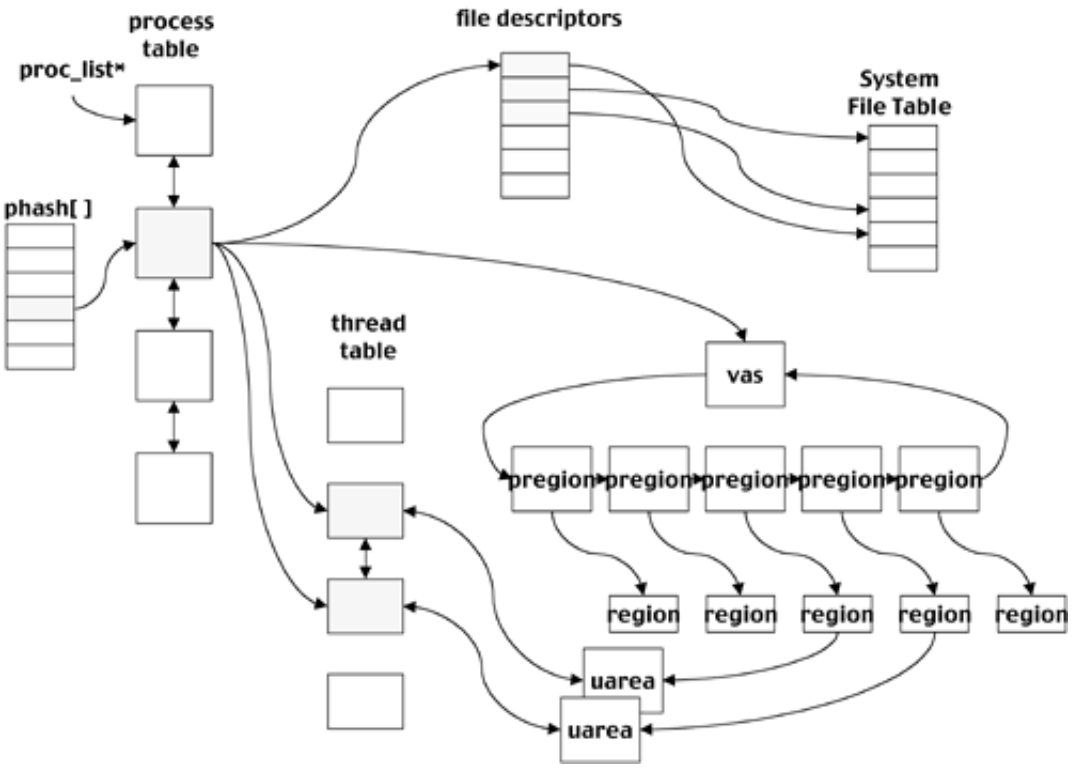
Kernel Process Tables

The process is truly the king of the HP-UX operating system, the primary entity managed by the kernel. In general, the prime directive for the HP-UX kernel is to level the playing field and make sure every process thread gets a chance to run.

The Process Table

Threads are the schedulable entity, but it is the venerable process that owns resources (at least as long as it is active). As such, the process table (see Figure 3-12) is the starting point for all process and thread management-related activities in the kernel. HP-UX 11.i features a dynamic process table. Entries are created as needed by allocating memory from a kernel memory arena and adding it to the linked list of active processes. The beginning of the process list is pointed to by the kernel pointer `proc_list*`.

Figure 3-12. Kernel Process Tables



Prior to the HP-UX 11.i release, the process table was a fixed-length table initialized at system boot, pointed to by the kernel pointer `proc*`, and defined in size by the tunable kernel parameter `nproc`. The original `nproc` parameter has been maintained through the current release for a couple of reasons. There are several static kernel tables that are sized directly in proportion to the process table. While the process table is now dynamic, not all related structures have made this change. It is reasonable to assume that in future releases this may not continue to be an issue, but for now `nproc` needs to be declared. This parameter also serves as a maximum limit for the growth of a system's process table.

Process management starts by using the phash[x] to locate a specific entry in the process table. Once we have located the appropriate proc structure, we follow its pointers to other key kernel tables.

The kthread Table

kthread structures are also dynamically allocated from a kernel memory arena and linked onto a list of active kthreads. In addition, all kthread structures belonging to a single process are linked together and to the parent process proc structure. Such threads are commonly called siblings. The process table has a pointer to the first and last of its sibling threads and maintains a count of how many it has spawned.

Prior to HP-UX 11.i, the kthread table was also a static array allocated at system boot and sized by the kernel parameter kthreadNTHREAD.

The vas and pregion Tables

In the grand scheme of things, a **pregion** contains the address offset and size of a process's logical region, access identifier information, and a type designator that specifies the usage mode. The pregion also maps a process's logical region into the system's VAS and creates a linked list of a process's virtual memory elements. When a process causes a page fault (requests access to a virtual page that is not currently memory-resident), the fault handler must search the process's linked list of pregions to determine which kernel region contains the necessary data for obtaining the faulting page. These structures and the underlying kernel region structures are the backbone of memory fault handling and have been optimized for function and efficiency. In addition to keeping track of its threads, a process table must also provide a map to each of the memory regions it will require for execution. To this end, a structure called a vas is created and linked to the proc of the process. The vas is the nexus of a linked list of pregion structures connected by a skip list. Each **pregion** links the process and its threads to a kernel-managed region structure. The **pregion**-to-**region** abstraction layer facilitates the mapping of shared objects by the kernel.

Process File Descriptors

All sibling threads share access to any file that has been opened in the name of its parent process. Each open() system call results in an entry in the process's file descriptor table. The size of this table is controlled by the kernel-tunable parameter nfile and prevents unlimited opens by a runaway thread.

All I/O is file I/O, and as such the file descriptor table represents the process's gateway to the "real world." File descriptors are required for each read and write operation. For example, consider the shell's default **STDIN**, **STDOUT**, and **STDERR** descriptors. As is the case with the shell, multiple file descriptors may point to the same file path, some for reading and some for writing.

The uarea Structure

An additional structure being introduced here is the uarea. The **uarea** is implemented as a kernel memory region, and unlike the other regions mapped in the parent process's pregion chain, access to the uarea is private to a single kthread. If there are multiple siblings, then there is an equal number of uarea pregions linked to the process's vas. An interesting note is that the uarea is mapped into a process's vas but never directly accessed by a user thread. The kernel has exclusive usage rights and stores the threads register image, known as a process control block (PCB) prior to a context switch-out. It then loads the next thread on the run queue's PCB from its uarea prior to switching it in.

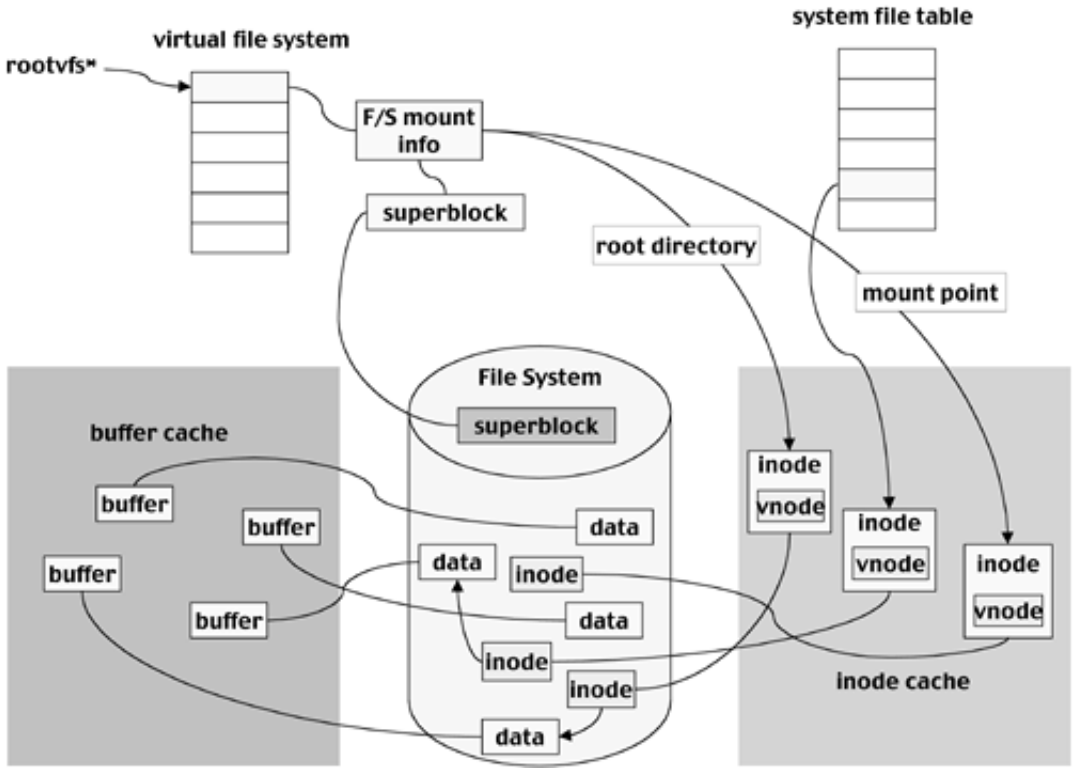
The Kernel File System Tables

The kernel must maintain a complete list of all opened files on the system, active mount points, and what is mounted there. Performance requires that many file system structures be maintained in core-resident `inode` caches and buffer caches as well as in a variety of tables.

The System File Table

Each discrete, open system call results in an entry to the system file table (see Figure 3-13). If the same file is opened 20 times, there will be 20 separate entries in the table. Discrete entries are required because this table keeps track of the type of open (read or write), the current offset into the file, and the number of linkages to it. As the various processes are terminated, their file descriptors are closed, and the linkage count in the system file table is decremented. If the linkage count goes to zero, then the entry is placed on a free list and may be reused.

Figure 3-13. Kernel File System Tables



The Virtual File System

The HP-UX kernel supports access to several different types of file systems. On earlier versions of the operating system, the specifics of each supported file system type was crafted into the kernel's core image.

Changing file system attributes was a major challenge and required patching the kernel. To move away from this dependency, a virtual file system interface was designed and implemented in the kernel.

The virtual file system treats all file system types the same. It is primarily concerned with their type, where they are mounted, a pointer to core-resident copies of any metadata that may be required to manage them, a cached copy of their respective root directories, and a pointer to an operational array of routines customized to handle their specific type.

The `Inode` Cache

The `inode` is the heart and soul of a UNIX file; it contains all attribute information with respect to a specific file. As the file is the basic object of all I/O operations, the `inode` attribute information is the key to access rights and data security within the system.

File attribute information is needed each time a thread requests access to file data or to modify the `inode` data itself. To speed this operation, an in-core copy is maintained of the `inode` data for each open file on the system. This is known as an `inode` cache.

As each file system type may define different `inode` data structures (different sizes, different block location schemes, different immediate data storage methods), it is the job of each configured file system type to define and build its own `inode` cache. To mask this difference from the higher levels of the kernel, an abstraction layer is added in which each file has assigned to it a systemwide unique virtual node (or `vnode`). Actions aimed at the `vnode` are translated through an operations array to file system type-specific routines in the kernel.

The Buffer Cache

Just as we keep copies of file attributes in an `inode` cache, the system maintains and manages a memory-resident buffer cache to hold recently requested copies of file data. When a process requests a read or write of file data, the buffer cache is checked first to see if a copy is present. If a cached copy is present, then the system merely needs to perform a memory-to-memory transfer of the requested buffer to or from the program's data space. Memory-to-memory transfers of this nature are called *logical reads* or *logical writes*. If a requested buffer is not present or the buffer is filled, a transfer must take place between the buffer cache and the physical disk. This constitutes a *physical read* or *physical write*.

When a read or write request results in an immediate physical action, it is said to be a *buffer cache miss*. The ratio between logical and physical reads and writes is called the *hit rate* and may be viewed using tools such as HP's `glance` or `gpm`.

The Kernel Input/Output Tables

The HP-UX kernel allows no direct access to physical devices. All I/O must be requested through the proper channel: the system call interface. A key component of this model is the kernel's extensive set of drivers, device maps, and the overall system's general input/output (GIO) system.

A large portion of the kernel memory space is occupied by these various components. The modern HP-UX kernel provides for the dynamic mapping of some of these should it be required.

The I/O Tree

A key table is appropriately named the I/O tree. This tree structure is actually a collection of linked objects used to identify and map system hardware devices to specific software drivers. The tree's roots stretch back to the system initialization, occurring even before the HP-UX kernel is loaded into memory. A subsection of this structure is the more elemental kernel I/O tree and is created in memory visible to the processor-dependent code (sometimes called the boot-rom) to be used in the event of a system crash.

The individual nodes of the I/O tree contain basic information about the hardware address: its type, which software has claimed it (driver association), and which context it belongs to. Additional properties, such as a kernel-assigned instance number and pointers to its parent, sibling, and children, are also present.

The I/O tree follows an object model; nodes lower down in the tree structure inherit attributes from their parents and pass them on to their children. We explore this fully in Chapter 10, "I/O and Device Management."

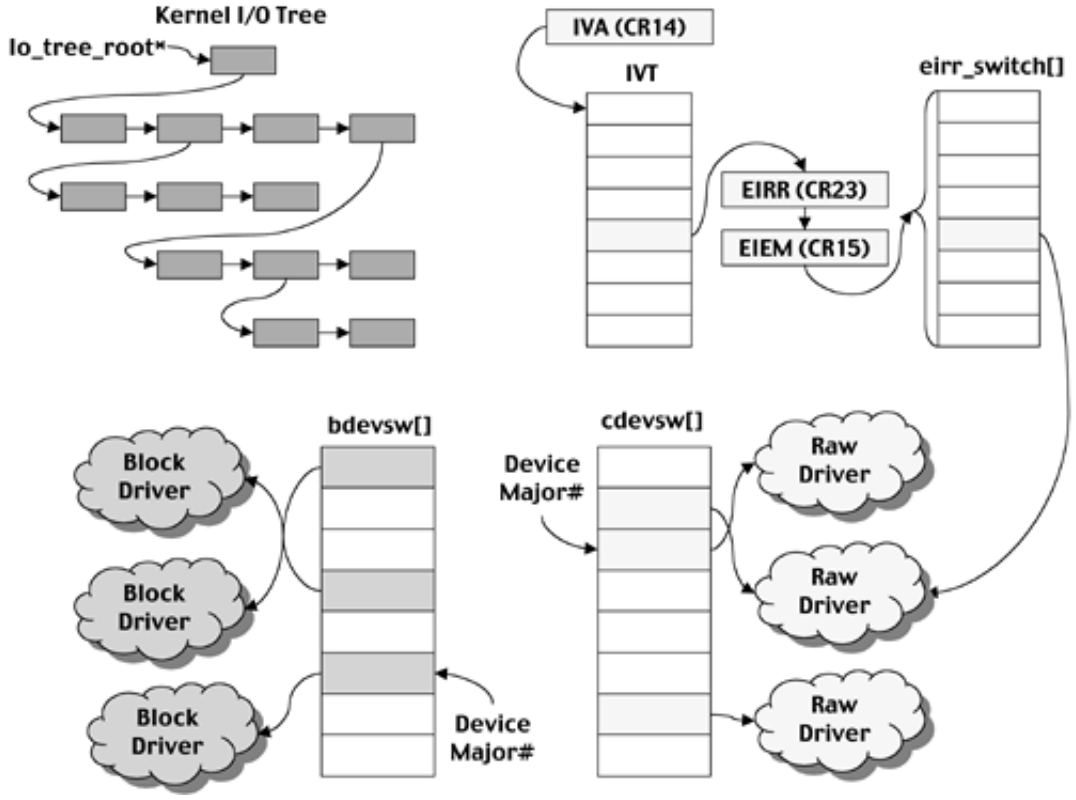
System Interruptions

In addition to the I/O tree, the I/O subsystem is responsible for the detection and response to interruptions. You may notice that we use the word *interruption*, not *interrupts*: HP-UX describes all categories of events affecting normal processor execution as being system interruptions. They are broken into four categories: faults, traps, interrupts, and checks.

To further complicate the topic, there are internal (CPU hardware, clocks, timers, etc.) and external (I/O devices, interface cards, physical device) interrupts, not to mention process signal handling!

As shown in Figure 3-14, internal interruptions are registered by a CPU and cause a vectored branch to memory-resident handling routines contained in an array called the interrupt vector table (IVT). Each entry in this table consists of a 32-byte block of code in which the initial portion of the interruption handler must be loaded. Also note that control register `CR14` contains the interrupt vector address (IVA), the starting address of the IVT. As each physical processor maintains its own register set, it would be possible to have interruption handlers specific to each processor. This is not generally the case, but it is an interesting option.

Figure 3-14. Kernel I/O Tables



An interesting interruption (registered as internal interrupt #4) is called the *external interrupt* and evokes routines vectored through an array named the *external interrupt receipt register switch table*, or simply the `eirr_switch[x]`. In most cases, this vectored switch table points toward the interrupt handling routine entry points associated with the kernel's various hardware drivers. Two additional control registers assist in registering and enabling the various external interrupts: `CR23` (the `eirr`) and `CR15` (the external interrupt enabling mask, or `eiem`).

Drivers and Switch Tables

The bottom portion of Figure 3-14 shows two switch tables. One is used for accessing block device drivers and the other for raw device drivers. A raw device is one to which data is transferred in byte streams of varying sizes.

A raw device may receive a single byte or many megabytes in a single transfer (or even gigabytes in today's computing climate). Raw device access is also used to talk to and configure device controllers and interface cards when necessary.

Block devices hold mountable file systems. All I/O to a block device is directed through the system's buffer cache to reduce physical requests to logical requests whenever possible for speed and convenience

As drivers are built into the kernel image or registered for future dynamic loading, they are assigned a driver number, often called their *major* number. Many of the major numbers are reserved and always assigned to a specific driver; for example, the logical volume manager pseudodriver is always assigned major number 64, while others are assigned from a dynamic pool of available numbers. This major number is visible through the `ll` or `ls -l` commands. To list all the currently used major numbers for an HP-UX kernel, enter the `lsdev` command. In general, many more devices have raw drivers than have block drivers.

The device switch tables are arrays of operational pointers. Each entry in the table consists of a subarray of jump points. The routines pointed to are entered in a documented order and cover actions such as open, close, read, write, and several more. If it sounds like we are talking about file I/O, you are absolutely correct—but remember that in the world of HP-UX (and UNIX in general), all types of I/O are treated like file I/O. No matter what type of physical device a driver was created to work with, to system process threads, first you open it, then you read or write to it, and when you are through you close it!

Summary

This concludes our first pass through the kernel designer's tool kit. Knowledge of these structures and tables and how they work will help prepare us for an in-depth examination of the HP-UX kernel. HP-UX and UNIX in general have many nooks and crannies to be explored and understood. Please reference the generic algorithm explanations offered in this chapter as we encounter specific examples of their usage in the chapters to come.

Chapter 4. Programs, Processes, and Threads

The study of UNIX internals may be approached from several points of view. One is to simply define all the individual kernel data structures and their basic functionality. While this presents a suitable framework for our study, we would like to also consider the kernel's construction and utilization from the perspective of an individual process or thread. The kernel perspective is macro in its scope, while the view from a process or thread is more closely related to the user or programmer's perspective.

The Players

There was a time when the process was the smallest schedulable entity of concern to the kernel; today, however, that honor belongs to the thread. Before we discuss HP-UX kernel structures in depth, we must define the relationship between programs, processes, and threads.

The Program

The term *program* refers to a file that stores an executable image. A program file contains a header record defining the architecture for which it was compiled, magic numbers, and a list of system resources it will require access to during its execution life cycle.

The first HP series 800 computers were based on the then newly developed 32-bit HP Precision Architecture (PA-RISC 1.x) chip set. This product line was called the Spectrum computing platform. The linking-loader for this architecture used a proprietary header format called the Spectrum Object Module, or SOM. With the advent of the 64-bit HP Precision Architecture (PA-RISC 2.0), the loader format adopted was the POSIX Executable Linking Format (ELF) definition. For compatibility reasons, all 32-bit executables on HP PA-RISC platforms continue to use the SOM format, and 64-bit executables use ELF.

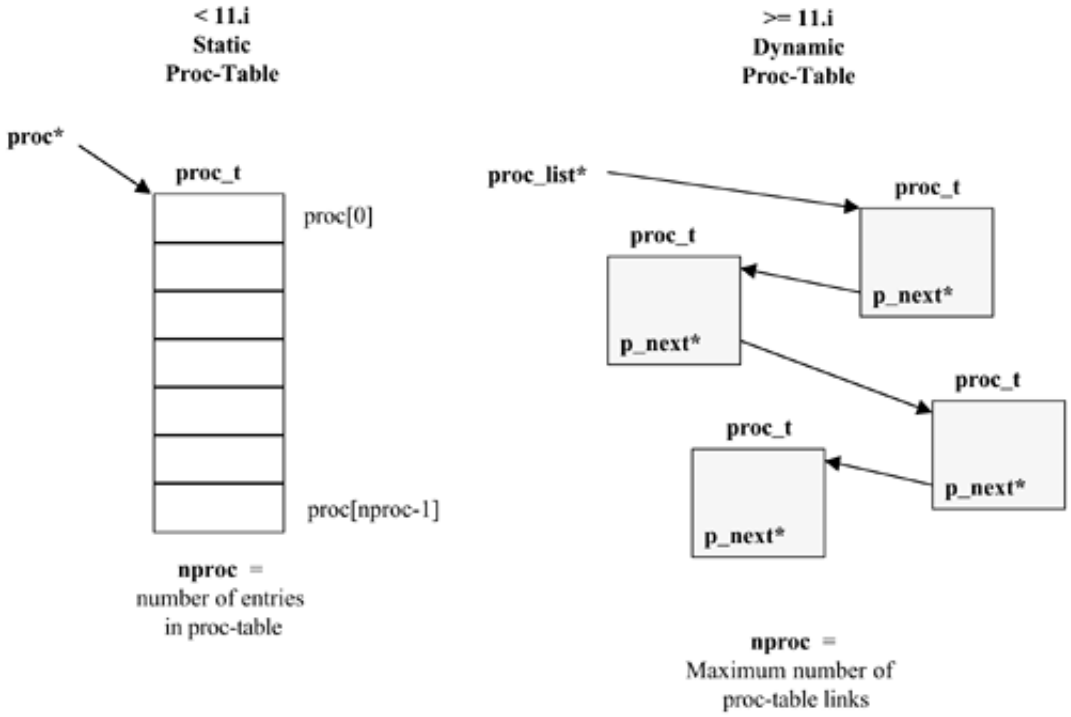
The memory management portion of the kernel also refers to program files as the *front store*. Since the program file contains static images of the program's text (or code) pages, the kernel paging system makes use of these if it needs to page-out any of the program's text once it has been loaded into physical memory. This negates the need to allocate a storage location from the system's swap space (a.k.a. *back store*) for a program's text pages. As the paging system relies on access to these page images as long as the process is active, the program file must be held in an open state for as long as the program is in use. Paging and swapping are discussed in greater detail in Chapter 6, "Managing Memory."

The Process

In the world of HP-UX, the term *process* is used to define all the parts that make up an active program on the system—that is, one that is currently being allowed to utilize system resources and perform work on the behalf of a user. Think of a process as a type of container used to organize and manage a task. It keeps track of what is being used, what is required to perform the task at hand, priority and scheduling information, and what permissions and authorizations have been granted to the process. In addition, the process keeps tally of processor time used, the amount of I/O performed at the request of the process, the state of pending signals sent to the process, and many other related operational parameters.

A process exists once there is a valid entry in the kernel process table (or `proc` table, as we call it in this book; see Figure 4-1). This table entry ties the process to all of its allocated system resources and is truly the nexus of the process's view of the kernel throughout its life cycle.

Figure 4-1. Static versus Dynamic `proc` Tables



Prior to the release of HP-UX 11.i, the `proc` table was a static table, and its size was defined by the system-tunable parameter `nproc`. The table was an array of data structures of type `proc` whose starting point was accessed through the kernel pointer `proc*`. When the table was full, no additional programs could be launched on the system. Increasing the number of allowable processes required a rebuild of the kernel. The current trend is to make many kernel structures dynamic in order to reduce the need to rebuild the kernel (and the number of reboots).

Starting with HP-UX 11.i, the `proc` table became a dynamic linked list (see Figure 4-1). This arrangement allows for `proc` table entries to be created on demand and dynamically linked into the list of active processes on the system. Access to the first `proc` table entry is now made by following the kernel pointer `proc_list*`. Each table structure is still defined as a `proc` (note that this structure definition has changed significantly since previous releases), which now contains the linkage pointer `p_next*` to the next `proc` table entry.

The kernel-tunable parameter `nproc` is still used, but its purpose has changed somewhat: it is now used to define a maximum number of allowable `proc` table entries that may be created on the system. The kernel contains many static lists, which need to be sized large enough to handle the kernel's worst-case needs. The `nproc` parameter has been used to help in this task during kernel initialization. Until these tables also become dynamic structures, a parameter like `nproc` will be required.

The process has long been the workhorse of the HP-UX runtime environment. With the release of HP-UX 10.0 came the introduction of the thread as the schedulable entity.

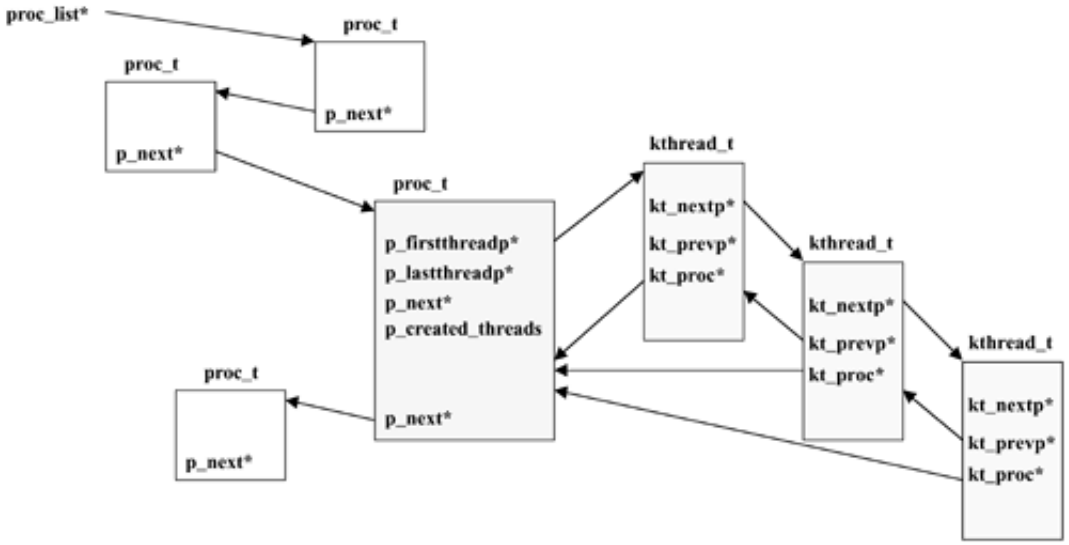
The Thread

A thread is an instance of execution of a process's code and is scheduled by the kernel. A single process may employ a single thread or several threads depending on its programming. A thread is running either in *kernel thread* or *user thread* mode depending on whether it is executing code from the process's text or from privileged areas of the kernel code known as system calls.

Each thread requires an active entry in the kernel thread table. Thread table entries are created on demand and are defined by the structure `kthread`. All the threads spawned for a process are said to be *siblings*, and their individual thread table entries are formed into a linked list using entries in their respective `kthread` structures. The parent process has linkage pointers in its `proc` structure to the first and last `kthread`

elements in this list (see Figure 4-2). The parent `proc` table entry also contains a count of threads it has created, `p_threads_created`; this counter is set to 0 with the creation of the process's initial thread and incremented for each additional sibling spawned.

Figure 4-2. Linking Threads to Their Process



In addition to the `proc`-to-`kthread` linkages, all the active threads on the system are linked together to form an active thread list.

As was the case with the `proc` table, in HP-UX 10.x releases the thread table was also a static table. The kernel pointer to the first `kthread` was `kthread*`, and the number of entries in the thread table was stored in `kthreadNKTHREAD`, which was set equal to `nproc`. As with the proc table at HP-UX 11.i the thread table has also been converted to a dynamic list.

A Process and Its Threads

On HP-UX operating system releases prior to 10.10, the kernel scheduled a process to run on the hardware to accomplish its programmatic tasks. By this model, a process lived a very solitary life and the programmer did not have to be concerned with the coordination of access to the process's private system resources. While the kernel provided many forms of sharing between process views (shared memory, shared file access, memory-mapped files, and shared libraries to name but a few), process-private data could only be used or modified by the process's single thread of execution.

A process could only do one thing at a time. If it was blocking (or sleeping) on a physical action, such as waiting for operator input or a tape drive to rewind, then it was said to be in a *wait-state* or *sleeping*, and only when the blocking action was completed could it perform its next instruction. In the early days of computing this model served quite well, since most systems had very limited resources, single processors, and comparatively slow response times.

System architectures progressed to include multiple processors under simultaneous control of a single kernel, referred to as symmetrical multiprocessing (SMP) systems (see Chapter 12, "Multiprocessing and HP-UX," for additional details on this aspect of the kernel). The next step was to devise a methodology whereby a single program could take advantage of SMP parallelism and execute tasks on several CPUs at the same time. This required a new breed of programmatic logic and gave rise to many innovations in operating system design. The challenge of parallelism with respect to the management of a process and its execution brought about the creation of multithreaded kernels.

Thread-based structures were first introduced into the HP-UX kernel with the 10.10 release (the initial threading of the kernel supported only a single thread per process, but it was a first step). With the release of HP-UX 11.0, the kernel became truly multithreaded; that is, a single process could have multiple user-space/kernel-space thread pairs performing tasks in parallel if it was on a SMP system. Even on a uniprocessor (UP) system, a process could spawn multiple threads, and they could compete for system resources: perhaps one could sleep waiting on user input while another processed existing data.

A thread-based kernel manages a client program by first creating a unique process container by which to track and manage the global attributes and allotted system resources of the program. Next, it creates one or more threads to implement the program's instructions. These threads are schedulable entities. In effect, the venerable process was promoted to a management position—instead of actually doing the work, it now owns the resources and supervises and coordinates the actions of its subordinate threads.

An operating system such as HP-UX (even before the implementation of kernel threads) was said to be *multiple-process* or *multiple-tasking* in reference to its ability to switch or timeslice between completely independent sets of process instructions. Each process had use of the system's resources and ran in a private context—without knowledge or interaction with other processes resident on the system. The kernel switched context from one to another, carefully managing the system resources so that each process had its own private view of "reality" and so that they did not disturb the view of their neighbors. The kernel maintained peace in the community and made each process feel as if it were king of the hill. This type of process management required heavyweight context-switching system calls.

A threaded process is one in which there may be several threads of execution simultaneously contending for system resources. Unlike in the older multiprocess environment, a single process may have one or more concurrent threads. Multiple threads of the same process (called siblings) share the same system resource view. They share the same code, most of the same data space (each thread does have its own stack space and user area), and the same libraries and system resources. Since all the threads of a process are playing in the same sandbox, it becomes the responsibility of the programmer to develop strategies to assure that sibling don't step on each other. To help in this task, many programming aids and libraries have been developed. The thread has become the schedulable entity and working incarnation of the program's instructions and logic.

Today many operating system kernels (including HP-UX) operate in a multiprocess and multithreaded environment. The kernel must manage many different process contexts and also must manage processes with multiple threads. To accomplish this, the kernel considers individual threads, not processes, for scheduling. The thread becomes the active, working, schedulable entity for kernel management. This change in behavior between the process-centric and the thread-centric view of running a program is a seminal point in the evolution of operating system design.

In HP-UX, the kernel must keep track of all active processes and each of their individual threads. This is accomplished by the creation of what many feel are the two most fundamental data structures in the operating system kernel: the process table and the kernel thread table.

Threading Models and HP-UX

Several threading models are utilized in modern operating systems (see Figure 4-3).

M x 1 threading allows for a single kernel-level thread per process. In the user space, a user-level scheduler may be employed to approximate the actions of a multithreaded environment. Programmers use user-space library routines to accomplish pseudo-scheduling and resource coordination. The library routines must monitor the individual user threads and context-switch if the current active thread is about to make a system call that would result in a blocking action. Once any of a process's user threads blocks on a call, the entire process is placed in a wait state. Keep in mind that from the perspective of the kernel, this is a single-threaded model (ergo the 1 in the M x 1 label). All of the scheduling is done by the user code, which results in a fairly low cost because the pseudo-scheduler does not encounter the additional overhead of system calls. While this is a plus, a big negative is that this model does not allow a process to take advantage of multiple CPU configurations.

1 x 1 threading does allow for multiple threads per process. In this model, there is a one-to-one relationship between kernel and user threads, and no user-space scheduler is allowed. The kernel scheduler works only with kernel threads and has no knowledge of what goes on in the user space. A process's user threads are bound one-on-one to a kernel thread, which in turn is scheduled on a specific CPU. The scheduler runs in kernel space and requires heavyweight system calls to manage each processor's run-queue. Processor run-queues are built by creating a linked list of kernel threads, *kthreads*, that share the same priority queue and processor affinity. The thread at the head of the list of the strongest active priority queue will be the next one scheduled. Once a kernel thread is scheduled, it may pass control to its user thread. We return to a kernel thread only when the user thread makes a system call or to the kernel proper upon receipt of a system interruption. While this model makes extensive use of system calls it allows a process to take advantage of parallel execution on SMP systems.

M x N threading brings together both of the previous models. In this environment, the programmer may choose to employe a user-space scheduler or simply rely on the kernel scheduler, or use a combination of both. On SMP systems, user threads may be multiplexed to several kernel threads or simply bound in a one-to-one fashion at the discretion of the application programmer. This gives the programmer the greatest degree of control when trying to optimize process performance.

Hewlett Packard is using a phased introduction of threading to the HP-UX kernel.

Phase 1 occurred with the release of HP-UX 10.10. This involved modifying the kernel's scheduling system to work with the newly created `kthread` table entries. For this release, only one thread of execution could exist per `proc` table entry. One could argue that this was not truly a threaded kernel: in effect, we were at the time merely "prepping the kernel for threads." A limited form of threading known as user threads, or the M x 1 model (making use of the DCE libcmathreads library), was also available to programmers at this time.

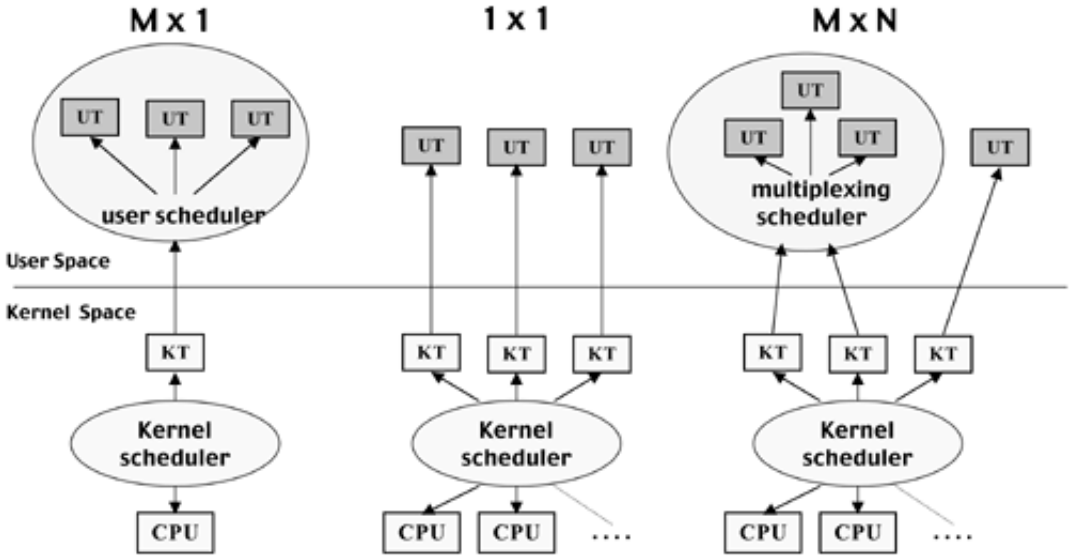
Phase 2 came with HP-UX release 10.30/11.00, which adopted a threading model called 1 x 1 threading: for each thread spawned by a user process, there is a single unique kernel thread. This new capability allowed a process to spawn multiple sibling threads and, for the first time on an HP-UX system, allowed a process to take full advantage of SMP system parallelism. To facilitate this model, the POSIX kernel threads library, `libpthread`, was implemented.

The release of HP-UX 10.30 (an early developers' release for workstations) provided for compliance with the POSIX 1003.1b API for multithreaded application development. With the full HP-UX 11.00 release, more thread features were added to make it POSIX 1003.1c compliant.

Phase 3 will be implemented in a future release. This phase will adopt the M x N threading model. In this model a single kernel thread may be associated with one or more individual user threads. This is a very flexible model, but along with flexibility comes an added cost in the complexity of the programming. M x N threads should be available on HP-UX in the near

future.

Figure 4-3. Threading Models

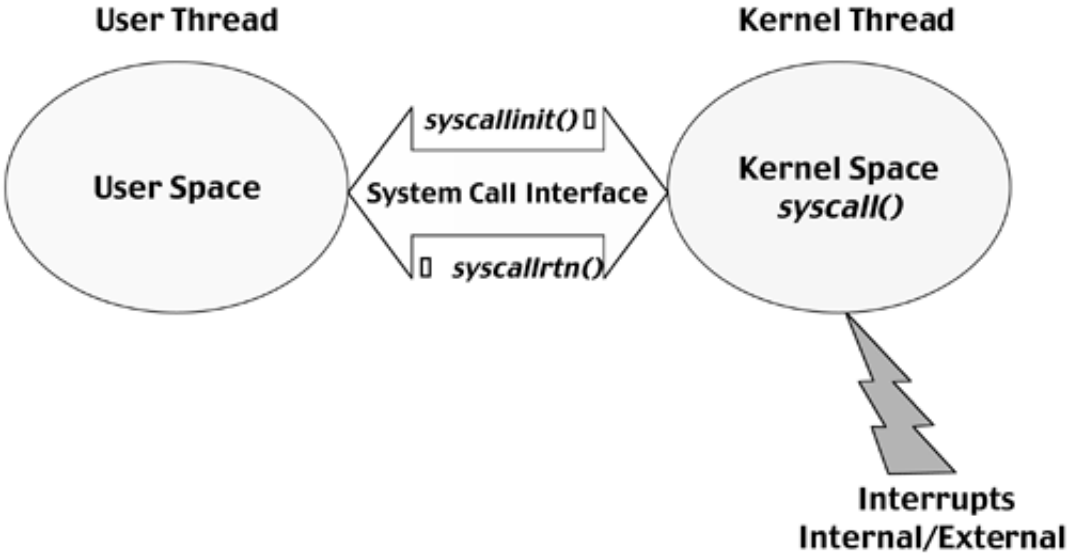


The System Call Interface

Each process thread consists of a bound user/kernel thread pair (unless a user-space scheduler is used). This thread is executing instructions from either the process's user memory space or from the privileged memory space of a legitimate kernel system call at any one instance of time. In addition to the many process-based threads, at times the system is required to halt the execution of the currently scheduled thread and switch to a specific kernel-only thread for the performance of interrupt handling, scheduling, or a host of other operating system-related tasks.

The transition between these various modes of operation is the study of the system call interface and the interrupt system (see Figure 4-4). We discussed the occurrence and handling of internal system-level interruptions in Chapter 1, "HP PA-RISC Architecture," and we explore external interrupts in detail in Chapter 10, "I/O and Device Management."

Figure 4-4. User-to-Kernel Thread Transitions

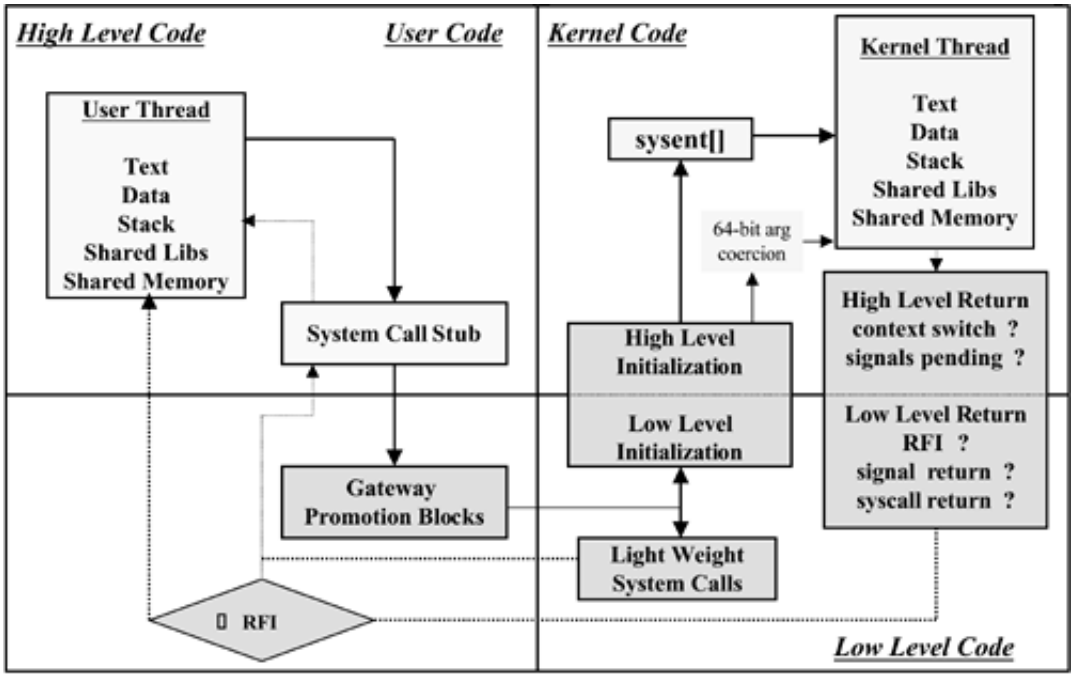


For now, consider that there are only two means by which a processor's current execution mode may be changed from user mode to kernel mode. The first is as the result of some type of system interruption that results in an immediate switch into the kernel context. The second is the direct result of the current user thread making a request to use one of the many provided kernel routines known as system calls, `syscall()`.

The return to the most deserving user thread is controlled by the same kernel-level mechanism, `syscall()`, regardless of which event caused the switch to kernel mode. This kernel routine is responsible for the delivery and notification of process-to-process signals and the handling of user-level context-switching, returning from a user-defined signal-handling routine, returning from a system-level interruption handler, and returning from a user thread-requested system call. It therefore must be able to deal with several different user and kernel stacks for the recovery of the user thread's context (a thread's run state context is stored in a process control block, or PCB, structure).

Reference Figure 4-5 as we describe the various tasks performed by a thread during the execution of a system call.

Figure 4-5. System Calls: The Big Picture



We start our discussion by considering that a process's user thread has been selected for execution on an available processor by the kernel scheduler. As execution continues in this user thread's memory space, it becomes necessary to request the services of an HP-UX system call.

The System Call Stub

System calls number in the hundreds and are made available to user threads by way of the system call library. These routines differ from conventional library routines in that they must first transition a user thread to a kernel thread in order to access kernel-resident subroutines. Execution access of code located on kernel-owned pages requires a privilege level of 0. User threads operate with a privilege level of 3; kernel threads operate at a level of 0. By necessity, the mechanism by which a thread's privilege level is set is tightly controlled by the hardware and the kernel to prevent unauthorized use. A listing of available system calls and their symbolic name and number is provided by the file `/usr/include/sys/scall_define.h` on HP-UX systems.

The Gateway Page

The system call stub is a very small piece of code that is linked into the thread's user-code space by the linking-loader at the time the executable image is built. This small stub appears as a simple intramodule procedure call. Each system call is assigned a unique system call number. It is the job of the compiler (or assembly language programmer) to map high-level language requests to the appropriate system call number and pass it to the stub (in GR22) along with the correct number and type of arguments for the specific system call being requested. This simple stub code blindly forwards the call number, calling register context, return pointer and passed arguments to a highly specialize routine called a *promotion block*. The return path to the requesting user thread's calling context is back through this stub when the call is completed.

The system call interface for narrow (32-bit) kernels has not changed with recent releases of the kernel; however, the interface for wide (64-bit) kernels follows the new procedure calling convention discussed earlier in this book. In addition, the gateway page layout for narrow and wide kernels differs somewhat. We

begin our discussion by examining the B,GATE instruction and the concept of promotion blocks.

The HP PA-RISC processor family (both narrow and wide) provides a highly specialized option to the branch instruction called the GATE completer (or simply the *gateway instruction*).

```
Mnemonic:

          B,GATE   target,t

Purpose:  Branch to target address, deposit current privilege

level into general register t, and set the privilege level

according to the rightmost two bits of the current page's TLB

access rights.

Example: B,GATE  .+8, r0
```

This instruction results in a branch eight bytes (two words) forward (.+8) relative to the B,GATE instruction's current address. The current privilege level (for a user thread, the value would be 3) is stored in GR0 (r0; in this example it is simply written to the system register equivalent of the bit bucket), and the privilege level is set equal to the rightmost two bits of the TLB access rights for the page from which the branch was made.

Security in this scheme is provided by the simple fact that only the kernel may set a page's TLB access rights. To enable the B,GATE completer, the TLB access rights on a gateway_page are set to PDE_AR_GATE (0x4C). The promotion takes place only if the processor status word C bit is set to 1 (this disallows gate promotions in real mode). To maximize security considerations, the kernel sets TLB access rights to PDE_AR_GATE on only the system call gateway pages during system initialization (reference pdapage() during kernel initialization).

Promotion Blocks

Individual system calls were originally branched to by means of a simple, four-instruction promotion block. Their basic function was to build an explicit virtual address pointing to the desired system call entry point. A long explicit pointer was needed, since kernel code was outside the scope of the calling thread's virtual memory map. The basic promotion block for a narrow kernel was as follows:

```
B,GATE   .+8,ro               # branch to current address+8
                              # and raise the privilege level
LDIL     L%target,r1          # start building a target address
                              # in r1
BE       R   %target,(sr7,r1)# complete the target address
                              # using sr7 and branch
DEPI     3,31,2,r31           # set the privilege bits to 3 in
                              # the stub's return pointer, note
                              # this occurs in the BE delay slot
```

On narrow systems, a thread's fourth quadrant was always mapped to virtual space 0 (see Chapter 6),and space register 7 (sr7) could always be assumed to have the value 0. This is no longer the case for wide kernels, so we must add an additional step to our code to assure that the explicit target address will lie in the first quadrant of virtual address space 0 (the kernel space). We assign and initialize sr2 for that task.

The promotion block on a wide HP-UX 11.0 kernel takes the following form:

```
MTSP            r0,sr2          # set up sr2 for later use (sr2=0)
B,GATE          .+8,ro          # branch to current address+8 and
                                # raise the privilege level
LDIL            L%target,r1     # start building a target address
                                # in r1
BE,N            R%target(sr2,r1) # complete the target address
                                # and branch
Note: The BE instruction uses nullification of the delay slot
to allow for tightly packing the four-instruction promotion
blocks. The task of setting the privilege bits in the stubs
return pointer (DEPI) has been moved to the first instruction
of the target routine (to preserve the four-instruction block
size).
```

A fundamental change to promotion block coding came about with the release of HP-UX 11.i. On wide kernels, the new promotion block requires five instructions. The DEPI instruction has been returned to the gateway page code for enhanced security. First, the new five-instruction block for a wide thread on a wide 11.i system:

```
B,GATE      .+8,ro          # branch to current address+8 and
                            # raise the privilege level
MTSP        r0,sr2          # set up sr2 for later use
LDIL        L%target,r1     # start building a target address
                            # in r1
BE          R% target (sr2,r1) # complete the target address
                            # and branch
DEPI        3,63,2,r31      # set the privilege bits to 3
                            # in the stub's return pointer
```

Now that the DEPI instruction has been moved back into the `gateway64_page`, the same is required of the `gateway_page`. To this end a small additional routine (named `gateway_generic`) was created in this page's space to keep the narrow promotion blocks at four instructions each. The promotion block for a narrow thread on a wide HP-UX 11.i system is as follows:

```
B,GATE      .+8,ro          # branch to current address+8
                            # and raise the privilege level
LDIL        L% target,r1    # start building a target address
                            # in r1
```

```
B(gateway_generic)            # branch to our new block of code
LDO         R% target (r1),rl# complete the target address
   .  .  .  .  .  .  .
(gateway_generic)
MTSP        r0,sr2            # set up sr2 for later use
BE          (sr2,r1)          # branch external to the target
                              # we have built
DEPI        3,63,2,r31        # set the privilege bits to 3 in
                              # the stub's return pointer.
```

Lightweight System Calls

To minimize system overhead cost, some system calls are implemented via simple assembly language routines called lightweight system calls. The lightweight calls simply set or return a value from a process's or thread's `proc_t`, `kthread_t`, or `user_t`. They never block, sleep, or have to worry about multiprocessor issues. Lightweight system calls include *`sigblock()`*, *`sigvector()`*, *`getpid()`*, *`getuid()`*, *`geteuid()`*, *`getgid()`*, *`getegid()`*, and *`umask()`* for an HP-UX 10.x kernel.

`gateway_page` on a Narrow Kernel

On early HP-UX systems, there were fewer system calls, which allowed a single page of system memory to contain a simple redirection routine and up to 253 individual promotion blocks. The `gateway_page` needed to be accessible from a user thread's memory map, so it was located at the beginning of quadrant 4, VAS 0 (explicit address of 0x0.C000000, implicit address of 0xC0000000). This address was hardcoded into the system call stub code for narrow applications and inserted into all SOM executables by the compilers.

As the number of system calls grew, exceeding 253, redirection code at the beginning of the `gateway_page` was modified so that all system call numbers greater than 253 would be sent to the same address as system call 0. On the kernel side a vector jump table (`sysent[]`) is maintained and indexed by the system call number (passed by the system call stub in `r22`) to direct the request to the correct target in the kernel.

Selected Portions of the `gateway_page` from a 32-Bit HP-UX 11.0 Kernel

Listing 4.1 was created using the adb utility:

```
#  adb -k /stand/vmunix /dev/kmem >> temp
   gateway_page,400?ia
```

The resulting temp file was then edited. Comments were added and redundant promotion blocks truncated. Please note that `adb` lists `B,GATE` as simply `GATE`.

Listing 4.1. `# adb -k /stand/vmunix /dev/kmem ; gateway_page,400?ia`

```
----------------------------------------
```

```
0xC0000000:   B,N ffffffffc0000018
# Default jump to syscall#0 promotion block
0xC0000004:   SUBI,>>=253,r22,r0
# is  r22 <= 253 ?
0xC0000008:    B,N ffffffffc0000018
# if not then go to syscall#0
0xC000000C:   ZDEPr22,30,30,r1
# else we need to calculate the jump
0xC0000010:   BLR,Nr1,r0
# to the correct promotion block
0xC0000014:   NOP
# these two instructions shift the value
# of r22 2 bits to the left (x4) this
# value is then used for a local branch
0xC0000018:   GATE ffffffffc0000020,r0
# Promotion block for syscall#0
0xC000001C:  LDILL%0x34800,r1
# This is the call to sysinit()
0xC0000020:   BE80(sr7,r1)
# (a heavyweight call example)
0xC0000024:   DEPI 3,31,2,r31
----------------------------------------
0xC0000158:   GATE ffffffffc0000160,r0
# Promotion Block for syscall #20
0xC000015C:   LDILL%0x385000,r1
# Notice the address is different
0xC0000160:   BE 648(sr7,r1)
# This is a lightweight call example
0xC0000164:   DEPI3,31,2,r31
----------------------------------------
0xC0000FE8:  GATE ffffffffc0000ff0,r0
# Promotion Block for syscall #253
0xC0000FEC:  LDILL%0x34800,r1
#  another heavyweight call
0xC0000FF0:   BE 80(sr7,r1)
0xC0000FF4:   DEPI 3,31,2,r31
0xC0000FF8:   BREAK
# End of promotion blocks!
```

```
0xC0000FFC:  BREAK

0xC0001000: # first address of the next page

----------------------------------------

Note: As shown in the listing, system calls 1 through 253 each

have a private promotion block defined, thus allowing them to

define a unique branch target. Most of the calls jump to

syscallinit, but some reference lightweight calls instead. All

the calls above 253 collectively use the promotion block for

syscall #0.

----------------------------------------
```

One restriction in this model was that no new lightweight calls could be defined with call numbers above 253. Another issue is that since the address of `gateway_page` is hardcoded into the system call stub it may not be relocated without breaking support of existing binaries.

The Wide Gateway: `gateway_page`, `sysvec`, and `gateway64_page` on a 64-bit Kernel

With the advent of the 64-bit HP-UX kernel, a new gateway page model was implemented. To address the issue of assigning lightweight system call numbers above 253, a user-space vector jump table (indexed by the system call number), `sysvec`, was created and accessed by a new system call stub for wide applications. The addresses contained in `sysvec` point to promotion blocks stored on a newly created `gateway64_page`. The kernel still contains a `sysent[]` vector jump array, so the `gateway64_page` needs to contain only a generic promotion block for all the heavyweight system calls and one additional promotion block for each distinct lightweight system call.

To provide continued support of existing narrow applications, a `gateway_page` is located at the virtual address 0xC0000000, and the `gateway64_page` and the `sysvec page` (or pages) are located immediately after. Unlike on narrow systems, these pages are not in the fourth quadrant of virtual space 0; instead, they are in the first quadrant of the space allocated for sharing objects between narrow and wide processes. This space number is stored in the kernel parameter `q1_64bit_spaceid`.

During kernel initialization, the `gateway_page` and `gateway64_page`s are configured with TLB access rights set to `PDE_AR_GATE`. The following kernel global variables help define the location and size of the gateway and `sysvec` pages for a wide kernel:

- `gate64_vaddr` – Virtual address of the `gateway64_page`
- `gate64_pages` – Number of pages needed for the 64-bit promotion blocks
- `sysvec_vaddr` – Virtual address of the 64-bit `sysvec` table
- `sysvec_pages` – Number of pages needed for the `sysvec` table
- `sysvec_entries` – Number of 64-bit promotion blocks
- `q1_64bit_spaceid` – Number of the virtual space holding the gateway pages

To make a listing of any of these pages on a wide kernel system, we must first discover their address. The q4 tool may help in this endeavor provided the kernel has been prepared to work with q4 (reference the **`q4pxdb`** command in Chapter 16).

Listing the `gateway_page` on a Wide System

The first step is to use q4 to find the address of `gateway_page`:

```
q4> &gateway_page
  0440000      147456   0x24000
```

The last value returned is the address of `gateway_page` in the kernel. It may be used with `adb` to create the listing:

```
$ adb -k /stand/vmunix /dev/mem > /tmp/gateway_page
  24000,400?ia
```

We did not include a copy of this output for a wide HP-UX 11.00 system because it is virtually the same as that for the narrow kernel, which we have already shown.

Selected Portions of the `sysvec` Page from a 64-Bit HP-UX 11.0 Kernel

Use q4 to find the address of the `sysvec` page:

```
q4> &sysvec
 0460000    1155648  0x26000
```

Next, use `adb` to create the listing:

```
$ adb -k /stand/vmunix /dev/mem > /tmp/sysvec
 26000,101?4X
```

The requested output format is four hex values per line. See Listing 4.2.

Listing 4.2. `# adb -k /stand/vmunix /dev/mem ; 26000,101?4X`

```
-------------------------------------------------------------
sysvec:
sysvec:     0          25000       0          25000
            0          25000       0          25000
            0          25000       0          25000
            0          25000       0          25000
            0          25000       0          25000
```

```
            0          25000       0          25000
            0          25000       0          25000
            0          25000       0          25000
            0          25000       0          25000
            0          25000       0          25000
(idx=20)    0(LW-Call) 25010       0          25000
            0          25000       0          25000
(idx=24)    0(LW-Call) 25020       0          25000
            0          25000       0          25000
(idx=28)    0(LW-Call) 25030       0          25000
            0          25000       0          25000
-------------------------------------------------------
(Note: Many entries have been truncated for brevity)
-------------------------------------------------------
            0          25000       0          25000
sysvec_end:8000240000
```

The `gateway64_page` from a 64-Bit HP-UX 11.0 Kernel

Use q4 to find the address of the `gateway64_page`:

```
q4> &gateway64_page
  0450000    151552   0x25000
```

Next, create the listing using `adb`:

```
$ adb -k /stand/vmunix /dev/mem > /tmp/gateway64_page
  25000,0x4c?ia
```

Listing 4.3 shows the individual promotion blocks for the 64-bit system calls. Note that there is only one block per specific call type.

Listing 4.3. `# adb -k /stand/vmunix /dev/mem ; 25000,0x4c?ia`

```
-------------------------------------------------
gateway64_page:
```

```
gateway64_page:                    MTSP r0,sr2
gateway64_page+4:                  GATE
gateway64_page+0x000c,r0
gateway64_page+8:                  LDIL L%0x34000,r1
gateway64_page+0xC:                BE,N440(sr2,r1)
lw_getpid_gate64:
lw_getpid_gate64:                  MTSP r0,sr2
lw_getpid_gate64+4:                GATE
lw_getpid_gate64+0x000c,r0
lw_getpid_gate64+8:                LDIL L%0x3b2000,r1
lw_getpid_gate64+0xC:              BE,N120(sr2,r1)
lw_getuid_gate64:
lw_getuid_gate64:                  MTSP r0,sr2
lw_getuid_gate64+4:                GATE
lw_getuid_gate64+0x000c,r0
lw_getuid_gate64+8:                LDIL L%0x3b2000,r1
lw_getuid_gate64+0xC:              BE,N420(sr2,r1)
cnx_lw_pmon_read_gate64:
cnx_lw_pmon_read_gate64:           MTSP r0,sr2
cnx_lw_pmon_read_gate64+4:         GATE
cnx_lw_pmon_read_gate64+0x000c,r0
cnx_lw_pmon_read_gate64+8:         LDIL L%0x3b3000,r1
cnx_lw_pmon_read_gate64+0xC:       BE,N 212(sr2,r1)
lw_getgid_gate64:
lw_getgid_gate64:                  MTSP r0,sr2
lw_getgid_gate64+4:                GATE
lw_getgid_gate64+0x000c,r0
lw_getgid_gate64+8:                LDIL L%0x3b2000,r1
lw_getgid_gate64+0xC:              BE,N728(sr2,r1)
lw_test_gate64:
lw_test_gate64:                    MTSP r0,sr2
lw_test_gate64+4:                  GATE
lw_test_gate64+0x000c,r0
lw_test_gate64+8:                  LDIL L%0x3b2000,r1
lw_test_gate64+0xC:                BE,N104(sr2,r1)
lw_mcas_util_gate64:
lw_mcas_util_gate64:               MTSP r0,sr2
lw_mcas_util_gate64+4:             GATE
```

```
lw_mcas_util_gate64+0x000c,r0
lw_mcas_util_gate64+8:              LDIL L%0x3b3000,r1
lw_mcas_util_gate64+0xC:            BE,N1800(sr2,r1)
lw_set_userthreadid_gate64:
lw_set_userthreadid_gate64:         MTSP r0,sr2
lw_set_userthreadid_gate64+4:       GATE
lw_set_userthreadid_gate64+0x000c,r0
lw_set_userthreadid_gate64+8:       LDIL L%0x3b2800,r1
lw_set_userthreadid_gate64+0xC:     BE,N996(sr2,r1)
lw_umask_gate64:
lw_umask_gate64:                    MTSP r0,sr2
lw_umask_gate64+4:                  GATE
lw_umask_gate64+0x000c,r0
lw_umask_gate64+8:LDILL%0x3b2800,r1
lw_umask_gate64+0xC:                BE,N 772(sr2,r1)
lw_lwp_getprivate_gate64:
lw_lwp_getprivate_gate64:           MTSP r0,sr2
lw_lwp_getprivate_gate64+4:         GATE
lw_lwp_getprivate_gate64+0x000c,r0
lw_lwp_getprivate_gate64+8:         LDIL L%0x3b2800,r1
lw_lwp_getprivate_gate64+0xC:       BE,N1656(sr2,r1)
lw_lwp_setprivate_gate64:
lw_lwp_setprivate_gate64:           MTSP r0,sr2
lw_lwp_setprivate_gate64+4:         GATE
lw_lwp_setprivate_gate64+0x000c,r0
lw_lwp_setprivate_gate64+8:         LDIL L%0x3b2800,r1
lw_lwp_setprivate_gate64+0xC:       BE,N1544(sr2,r1)
lw_lf_send_gate64:
lw_lf_send_gate64:                  MTSP r0,sr2
lw_lf_send_gate64+4:                GATE
lw_lf_send_gate64+0x000c,r0
lw_lf_send_gate64+8:                LDIL L%0x3b2800,r1
lw_lf_send_gate64+0xC:              BE,N 2044(sr2,r1)
lw_sigvec_gate64:
lw_sigvec_gate64:                   MTSP r0,sr2
lw_sigvec_gate64+4:                 GATE
lw_sigvec_gate64+0x000c,r0
```

```
lw_sigvec_gate64+8:                LDIL L%0x3b2000,r1
lw_sigvec_gate64+0xC:              BE,N 1364(sr2,r1)
lw_sigblock_gate64:
lw_sigblock_gate64:                MTSP r0,sr2
lw_sigblock_gate64+4:              GATE
lw_sigblock_gate64+0x000c,r0
lw_sigblock_gate64+8:              LDIL L%0x3b2800,r1
lw_sigblock_gate64+0xC:            BE,N 420(sr2,r1)
lw_sigsetmask_gate64:
lw_sigsetmask_gate64:              MTSP r0,sr2
lw_sigsetmask_gate64+4:            GATE
lw_sigsetmask_gate64+0x000c,r0
lw_sigsetmask_gate64+8:            LDIL L%0x3b2000,r1
lw_sigsetmask_gate64+0xC:          BE,N 1036(sr2,r1)
lw_lwp_self_gate64:
lw_lwp_self_gate64:                MTSP r0,sr2
lw_lwp_self_gate64+4:              GATE
lw_lwp_self_gate64+0x000c,r0
lw_lwp_self_gate64+8:              LDIL L%0x3b2800,r1
lw_lwp_self_gate64+0xC:            BE,N1244(sr2,r1)
lw_lf_next_scn_gate64:
lw_lf_next_scn_gate64:             MTSP r0,sr2
lw_lf_next_scn_gate64+4:           GATE
lw_lf_next_scn_gate64+0x000c,r0
lw_lf_next_scn_gate64+8:           LDIL L%0x3b3000,r1
lw_lf_next_scn_gate64+0xC:         BE,N104(sr2,r1)
lw_get_thread_times_gate64:
lw_get_thread_times_gate64:        MTSP r0,sr2
lw_get_thread_times_gate64+4:      GATE
lw_get_thread_times_gate64+0x000c,r0
lw_get_thread_times_gate64+8:      LDIL L%0x3b2800,r1
lw_get_thread_times_gate64+0xC:    BE,N 1768(sr2,r1)
lw_nosys_gate64:
lw_nosys_gate64:                   MTSP r0,sr2
lw_nosys_gate64+4:                 GATE
lw_nosys_gate64+0x000c,r0
lw_nosys_gate64+8:                 LDIL L%0x3ae800,r1
lw_nosys_gate64+0xC:               BE,N 304(sr2,r1)
```

```
lw_nosys_gate64+10:

(This last promotion block is used to trap unimplemented system calls)
```

Listing 4.4 is a partial listing of system calls and their numbers taken from the file */usr/include/sys/scall_define.h*. It is included for your consideration as you look at the listings of the promotion blocks:

Listing 4.4. `# more /usr/include/sys/scall_define.h`

```
misc.dsc:system call 0 is nosys;

pm.dsc:              system call 1 is exit;

pm.dsc:              system call 2 is fork;

fs.dsc:              system call 3 is read;

fs.dsc:              system call 4 is write;

fs.dsc:              system call 5 is open;

fs.dsc:              system call 6 is close;

pm.dsc:              system call 7 is wait;

fs.dsc:              system call 8 is creat;

fs.dsc:              system call 9 is link;

fs.dsc:              system call 10 is unlink;

vm.dsc:              system call 11 is execv;

fs.dsc:              system call 12 is chdir;

pm.dsc:              system call 13 is time;

fs.dsc:              system call 14 is mknod;

fs.dsc:              system call 15 is chmod;

fs.dsc:              system call 16 is chown;

vm.dsc:              system call 17 is brk;

fs.dsc:              system call 18 is lchmod;

fs.dsc:              system call 19 is lseek;

pm.dsc:              system call 20 is getpid;

fs.dsc:              system call 21 is mount;

fs.dsc:              system call 22 is umount;

pm.dsc:              system call 23 is setuid;

pm.dsc:              system call 24 is getuid;

pm.dsc:              system call 25 is stime;

pm.dsc:              system call 26 is ptrace;

signals.dsc:         system call 27 is alarm;
```

```
pm.dsc:              system call 28 is cnx_lw_pmon_read;

pm.dsc:              system call 29 is pause;

   ...
```

Low-Level Initialization

Once the appropriate promotion block is selected, the system will either transfer control to a lightweight system call (also known as a level-2 call) or it will enter the kernel `syscallinit()` function (a level-1 call) to begin low-level initialization.

In the case of a lightweight call, the action is performed and an immediate return is made to the system call stub. In reality, very little time is spent in the kernel code, so little in fact that the thread accounting system does not even attempt to charge any CPU utilization to the threads' system time bucket. Instead, it is simply considered as user time.

High-Level Initialization

`syscallinit()` contains an assembly-level initialization code routine designed to set up a save state and stack frame for use as we transition further into the kernel's higher level. As the same kernel code may be called simultaneously by different user threads on SMP systems, each thread brings with it a privately owned kernel stack area. This *`kstack`* (three pages for a narrow application and six pages for a wide application) immediately follows the thread's *`uarea`* (every thread has its own unique *`uarea`*/*`kstack`* memory region, discussed in greater detail in Chapter 6). The primary job of ***`syscallinit()`*** is to make sure that the stack pointer (*`sp`*), the stub return address (*`r31`*), the original caller's return address (*`rp`*), the user's data pointer (*`dp`*), and the user's space registers are safely stored away so that when all is said and done we will be able to return to the original calling context. The next step is to call the high-level initialization routine ***`syscall()`***.

The **`syscall()`** kernel routine is written in C code and starts by collecting performance metrics relating to this thread. The measurement system is initialized, and we now start charging CPU utilization tics to the thread's "system time" bucket. The current system call number is saved in the thread's `uarea` (at `u_syscall`) and is used as the index into the `sysent[]` vector jump table. The kernel also contains the array `sysaux[]` with specific 64-bit information about the pending call, such as the number of registers used by the call. Any extra arguments passed via the registers are copied to the `uarea` (`u_arg[]`), which satisfies the differences between the 32-bit and 64-bit calling conventions. Once the actual system call address is retrieved, a call is made to `setjmp()`.

Argument coercion may be required if the calling user thread is 32 bits and we are running a 64-bit kernel. This action is transparent for the most part, as the values stored in the thread's `u_arg[]` will have their sign bit extended to 64 bits. The argument coercion stub makes the actual call to the address index in `sysent[r22]`.

The System Call

Now we have arrived at the actual kernel code we requested. At this point, any of many paths may be followed. We may be asking for an I/O operation, in which case our thread will most likely be put to sleep and assigned a wait channel. We might simply perform the task requested and start on the return path, or an interruption may halt our progress. In any case, let's take stock of where we are and what we have working for us.

The thread is now in kernel mode, and required stack space is being provided by the thread's own private `kstack` area. A save state of the originating context is safely held in the `uarea`, and we are proceeding with the full authority and privilege of the kernel.

The next step is to begin the return trip to resume execution in our user thread context. We could find ourselves at this juncture as the result of several different scenarios. Perhaps an interrupt signifying the completion of an I/O operation was received, causing some other thread to be preempted; a user-defined signal handler may have completed and made a call to **`sig_return()`**; or a system call could be simply

finishing its job. The point is that the system call that initiates the return to a user thread is not necessarily the one that started the sequence: the kernel interrupt system, memory manager, I/O system, or any of a number of system services may have affected the sequence of events.

High-Level Return

The high-level return path starts in the midst of the kernel **`syscall()`** routine. The return trip start by checking to see if the thread we are about to return to has received any signals. This is the only time that a thread checks for signals, and the programmer may have chosen to ignore or block certain signals. We cover the signal-handling mechanism fully in a separate chapter of this book. For now, we simply want to mention that this is the point at which signals are checked for.

Next, the system call's return value is checked to see if an error occurred during the execution of the call or if the return was interrupted by the receipt of a signal. If all is well, the system call return value is passed back through the low-level return code by placing it into `ret` of the save state structure (which was created during the low-level initialization).

One last task before transferring control to the low-level return code is to check the processor's `runrun` flag. This flag is stored in a per-processor `mp_info_t` (this structure is covered in Chapter 12) and is set if an interrupt handler places a thread on a processor's run-queue and notices that it belongs to a higher priority run-queue than the one from which that processor's current thread is executing (remember on SMP systems, interrupts may be handled for an I/O operation requested from a different processor than the one on which the thread is running). If the `runrun` flag is set, the current thread is placed back on its run-queue by the `setrq()` kernel routine, and the kernel routine `swtch()`is called to make a context-switch to the `kthread` at the front of the processor's strongest run-queue. Once `syscall()` is satisfied that it has identified the most deserving thread, the measurement system is instructed to start charging CPU utilization to the user time bucket, and the low-level return routine is called.

Low-Level Return

`syscallrtn()` is an assembly code routine that basically restores the registers that were saved upon entry to the low-level initialization code: the stack pointer (`sp`), the stub return address (`r31`), the original caller's return address (`rp`), the data pointer (`dp`), and the user's space registers are all restored prior to branching back to the system call stub's return address (`r31`). The system call's return value is passed to the original calling thread by way of `r22`, the same register that was used by the system call stub to pass the system call number. While this describes the normal system call return path, there are a couple of other possibilities that could have brought us here.

If we have arrived here due to the trapping of a signal for which the programmer has defined a signal handler, we will check to see if a save state needs to be placed on a signal stack for future reference, and then we will return directly to the handler routine. If we are returning from a user-defined signal handler, then the save state will need to be recovered from the signal stack prior to completing the return.

If the return is the result of a context-switch, we will recover the save state from the thread's `uarea` and proceed directly back to the user thread's context. Flags are also checked to see if this thread belongs to a process that is being traced; if so, various actions are taken depending on the level and type of tracing being used.

The final step involves making a branch-external instruction to the stub's return address in `r31`. This completes the system call and in the process demotes the thread to the user privilege level of 3.

Summary

Knowledge of the relationship of processes to threads and the function of the system call interface will help us as we expand our study into the structural depths of the kernel. Remember that at any one instant in time a processor is simply executing a single instruction, and the processor is in either user mode or kernel mode. There are only two ways in which the kernel mode may be entered: a user-initiated system call or as the result of a system interruption (interrupt, fault, or check). The kernel always attempts to return to the user mode upon completion of its tasks. Along the way it evaluates which user thread is the most deserving to run.

Now that we understand the mechanics of moving between the two modes, let's examine in depth the kernel tables and data structures of key subsystems in the HP-UX kernel.

Chapter 5. Process and Thread Management from the Process's Viewpoint

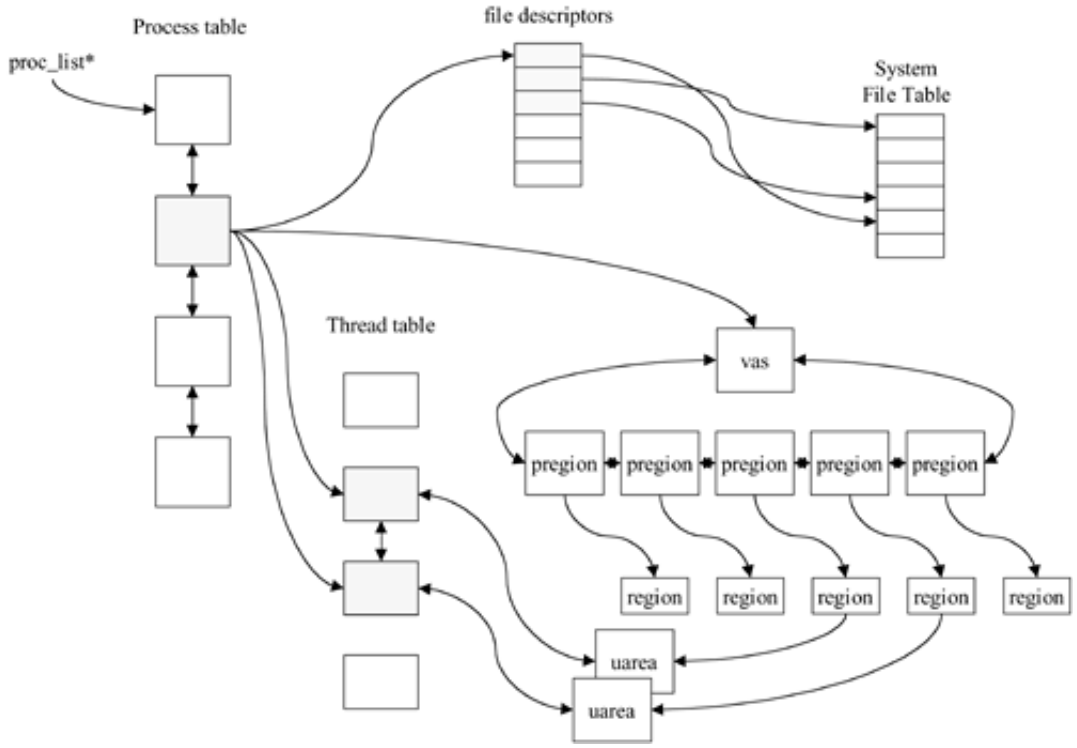
In this chapter, we explore kernel data structures used to manage resources allocated to a process and its threads. We see how a process's *logical address space* is mapped into the kernel's *virtual address space* (VAS) in preparation for translation to the hardware's *physical address space*. As the kernel is asked to host multiple processes, individual `kthread`s are placed in run queues organized according to processor, priority, and type. We also examine the overall implementation of the kernel scheduling system, its rules, and the concepts of *nice, real-time, timeshare,* and *priority decay*.

A Process and Its Resources

A process is a container used to link the system resources needed during the execution of its `kthread`s. A process must have a unique identity (its PID, or process identifier number), and the kernel needs a means of tracking system resources allocated for process use. To this end, the kernel employees several linked data structures to coordinate thread access to code, data, shared memory objects, files, and the I/O system.

From the process's point of view, the `proc` table (process table) is our starting point in the study of HP-UX internals. As we see in Figure 5-1, the `proc` table ties a process to its `kthread`s, virtual memory mappings, open file descriptors, and other system resources and services.

Figure 5-1. Kernel Process Tables

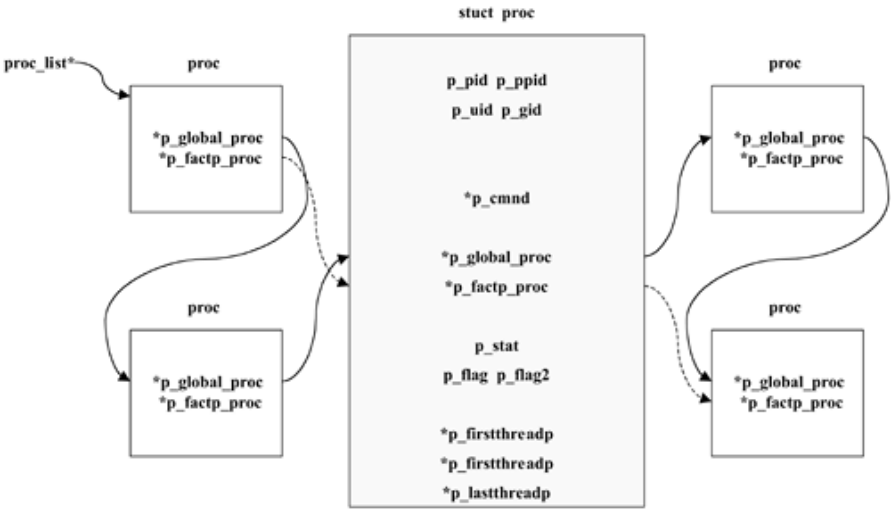


The proc Table

For a process to exist (at least from the point of view of the kernel), a proc data structure must be placed in the system proc table. Prior to HP-UX 11i, the proc table was a static array, the kernel pointer *proc defined its beginning, and the size was tuned via the kernel parameter nproc. The proc table is one of the largest tables in the kernel (at HP-UX 11.00, each proc structure was 700 bytes), and tuning nproc could significantly alter the size of the kernel. This effect was amplified, since many additional kernel data structures were sized in proportion to nproc.

With the release of HP-UX 11i, the proc table has become dynamic (see Figure 5-2). This means that individual proc structures are allocated from kernel memory arenas as they are needed. The beginning of this key system table is now pointed to by *proc_list, and linkage pointers have been added to the structure to create a dynamic linked list. At HP-UX 11i, a proc structure has grown to 828 bytes.

Figure 5-2. Kernel Process Tables



To date, not all kernel data structures have been converted from static to dynamic, which necessitates the maintenance of the nproc parameter to be used during the sizing of related tables. In addition, nproc acts as a tunable limit for the number of processes a kernel may be expected to manage.

In the current implementation, structures allocated to the proc table are not returned to the kernel memory pool when their process is terminated. Instead, the proc structure's status is set to SUNUSED, and it is placed on a free list. The current thinking is that once a proc entry has been allocated, it represents a type of high-water mark for system activity, and the system will simply place it on a free list so that it will be available next time the process count is high. Let's explore some additional linkages and parameters stored there.

q4 may be used to examine the process table on a live system.

First, launch q4, passing the kernel image file, */stand/vmunix*, and */dev/kmem* as arguments (if you have not set q4 up on your system, refer to Chapter 16, "Tools Overview," for an explanation of the process).

```
#  q4  /stand/vmunix  /dev/mem
```

Now you can examine the fields of the proc structure:

```
q4>  fields  struct  proc  |  more
```

To load the contents of the proc table on an HP-UX 11i system,

```
q4> load struct proc from proc_list next p_global_proc max nproc (to load all
➥ entries)or

q4> load struct proc from proc_list next p_factp_proc max nproc (to load active
➥ entries only)

For an HP-UX 11.0 (or earlier) system,

q4> load struct proc from proc max nproc
```

The following q4 fields listing (Listing 5.1) has been condensed and annotated for our discussion. (A q4 field listing produces six fields per line: byte-offset, bit-offset, size-in-whole-bytes, number of additional bits, field-type, and field-label.)

Listing 5.1. `q4> fields struct proc`

```
Forward and previous pointers linking all the active processes

  0 0  4 0 *         p_factp
  4 0  4 0 *         p_pactp
Various pointers to the process's thread(s)
  8 0  4 0 *         p_firstthreadp
 12 0  4 0 *         p_lastthreadp
 32 0  4 0 int       p_created_threads
Spinlock structures to protect access to proc and thread data
 24 0  4 0 *         thread_lock
 28 0  4 0 *         p_lock
The process flags (see enum proc_flag and proc_flag2 in
proc_private.h)
The process states are:
SUNUSED    =  0   unused, proc available
SWAIT      =  1   abandoned
SIDL       =  2   in process creation( kernel routine
                  new_proc() )
SZOMB      =  3   in process termination ( waiting for signal
from parent process)
SSTOP      =  4   process is currently stopped
SINUSE     =  5   proc entry in use
 36 0  4 0 enum4     p_flag2
 40 0  4 0 enum4     p_flag
 44 0  4 0 enum4     p_stat
A partitioned pointer to the process's file descriptor(s)
 56 0  4 0 *         p_ofilep
 This process's default priority and nice values and a
 reference count (used by the kernel to determine when to call
```

freeproc() to cleanup the process)

72 0 2 0 u_short   p_pri

74 0 1 0 char   p_nice

76 0 4 0 int   p_refcnt

A forward link connecting all proc structures both active and free

92 0 4 0 *   p_global_proc

The process's credentials; user id, su id, process group id, process id, parent process id, maximum number of physical page frames (known at the resident set size), forward and reverse hash chain links (there is a hash table used to speed up location of a proc structure entry when the process id is known), forward and reverse links connecting all proc table entries belonging to the same user id, and a pointer to this process's virtual address space header (structure **vas**)

176 0 4 0 int   p_uid

180 0 4 0 int   p_suid

188 0 4 0 int   p_pgrp

192 0 4 0 int   p_pid

196 0 4 0 int   p_ppid

200 0 4 0 u_long   p_maxrss

204 0 4 0 *   p_idhash

208 0 4 0 *   p_ridhash

220 0 4 0 *   p_uidlist

224 0 4 0 *   p_ruidlist

248 0 4 0 *   p_vas

Memory and swap reserved for this process

252 0 2 0 short   p_memresv

254 0 2 0 short   p_swpresv

Deactivation time for this process (if it has been deactivated) and a forward linkage connecting all deactivated processes

264 0 4 0 int   p_deactime

304 0 4 0 *   p_nextdeact

Pointer to temporary vforkinfo data structure used during a **vfork()** call

320 0 4 0 *   p_vforkbuf

The type of scheduling policy for this process's threads

324 0 4 0 u_int   p_schedpolicy

Pointer to the process's command line arguments and the process's base name

456 0 4 0 *   p_cmnd

460 0 15 0 char[15] p_comm

Number of clock ticks charged to this process

504 0 4 0 int   p_ticks

In addition, there are many fields used by processor affinity calls, the signal system, System-V IPC services, and many more.

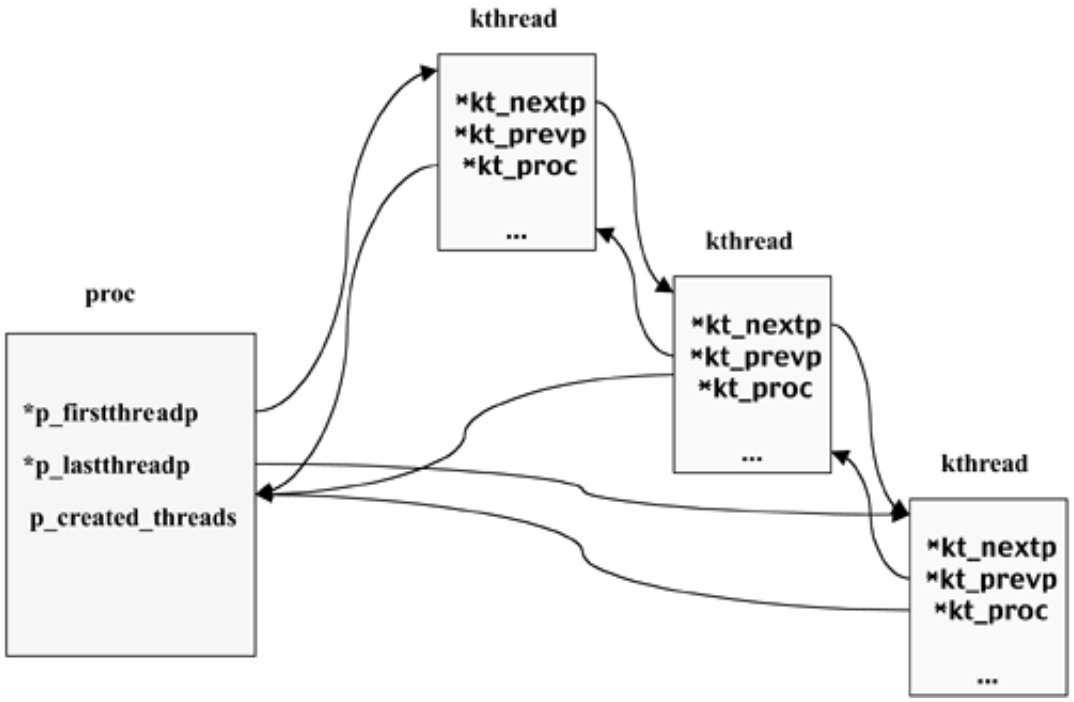
We return to many of these parameters as we continue our exploration of the kernel. The next stop on our trip is the `kthread` structure.

The kthread Table

Originally, HP-UX managed jobs only at the process level, and the proc table contained all information relating to the execution of a process by the kernel. HP-UX 10.0 introduced the kthread structure as a first step toward implementation of a fully threaded kernel environment. Although this was only a first step toward a true multithreaded environment, it initiated the division of static process-related parameters and dynamic thread–related values into separate structures. The proc table contains static values and pointers to resources shared equally by all of its threads, while the kthread structure(s) contain dynamic information relating to their execution and runtime environment.

First, let's examine the manner in which a proc table entry and its kthreads are linked together. In Figure 5-3, we see this relationship mapped out. The proc table maintains a pointer to the first kthread and last (most recently created) kthread. In addition, a count is kept of all the created threads. Please note that this count is not incremented for the original thread, so a process with a single original thread will show p_created_threads = 0 until the first sibling thread is spawned.

Figure 5-3. The kthread Table



The kthread structure maintains next/previous links and each sibling kthread has a direct pointer to its parent proc structure entry for quick access. In addition, each kthread has an indirect pointer to a uarea structure via a pregion structure (both of which are discussed later in this chapter).

Beginning with HP-UX 11.i, the kthread table is now a dynamic linked list. As with the proc table, entries no longer needed are placed on a free list (as opposed to having their space returned to the kernel memory allocation arena), and new entries are allocated as needed, up to a tunable limit, when the free list is empty.

Let's examine the fields of interest in the kthread structure, shown in Listing 5.2.

Listing 5.2. `q4> fields struct kthread`

```
 These forward and reverse links are used to place a thread
into an appropriate run or sleep queue
  0 0  4 0 *        kt_link
  4 0  4 0 *        kt_rlink
As we mentioned, each kthread has a direct pointer to its proc
structure and its uarea
  8 0  4 0 *        kt_procp
 52 0  4 0 *        kt_upreg
SMP systems make use of this spinlock to synchronize access to
this structure
 12 0  4 0 *        kt_lock
Next we see forward and previous linkage pointers connecting
all the active kthreads
 16 0  4 0 *        kt_factp
 20 0  4 0 *        kt_pactp
All the sibling threads of a single process are linked through
these two pointers.
 24 0  4 0 *        kt_nextp
 28 0  4 0 *        kt_prevp
All kthread structures are linked to a global kthread list
 64 0  4 0 *        kt_global_kthread
Thread status and flag values for the thread (see enum
kthread_flag in kthread_private.h)
 32 0  4 0 enum4    kt_flag
 36 0  4 0 enum4    kt_cntxt_flags
 44 0  4 0 enum4    kt_stat
180 0  4 0 enum4    kt_flag2
184 0  4 0 enum4    kt_flag_tl
Waiting threads maintain a sleep wait channel pointer and a
forward and reverse pointer into a wait list (used by I/O
drivers)
 48 0  4 0 *        kt_wchan
236 0  4 0 *        kt_wait_list
240 0  4 0 *        kt_rwait_list
In addition, a reference count, the most recent user mode
```

```
priority, the current internal priority, a thread id number,
the current scheduling policy, number of ticks left before a
voluntary timeslice will be requested, and the mysterious
kt_cpu weighting factor (covered later) are maintained
 72 0  4 0 int      kt_refcnt
 76 0  4 0 int      kt_deactime
 80 0  4 0 int      kt_sleeptime
 84 0  2 0 u_short  kt_usrpri
 86 0  2 0 u_short  kt_pri
124 0  4 0 int      kt_tid
196 0  4 0 u_int    kt_schedpolicy
200 0  4 0 int      kt_ticksleft
 88 0  1 0 u_char   kt_cpu

There are additional parameters that are used during MP load
balancing, signal handling, and IPC services.
```

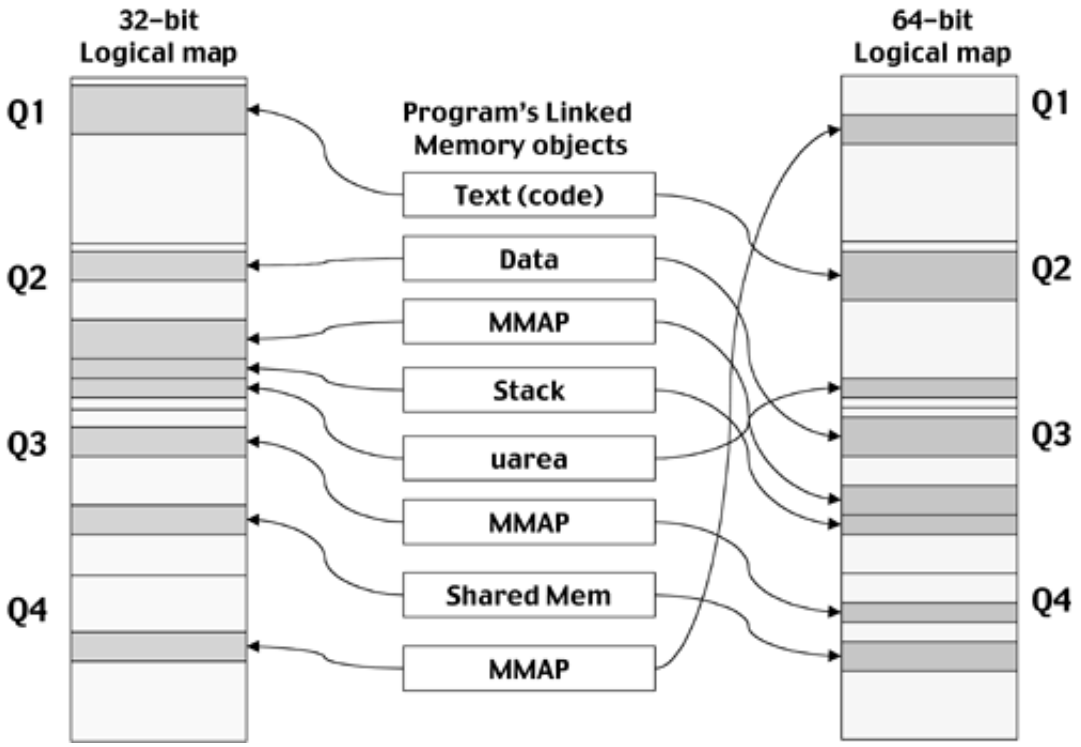
The next stop on our tour is the `vas` structure.

The Process's Logical View

In order to facilitate the concept of relocatable code, compiled programs are linked against a logical memory image map. For 32-bit narrow applications, the logical size is 2^32, or 4 GB; in the case of 64-bit wide applications, the theoretical size is limited to 2^64, or 16 exabytes! In either case, the logical map is divided into four quadrants of equal size. This is not an arbitrary configuration. It is dictated by the underlying design of the HP PA-RISC hardware platform and its virtual memory system.

We discuss the virtual memory-to-physical memory mapping in the next chapter, but first we must look at the rules and options that control the placement of a program's various memory objects into a logical space. Figure 5-4 illustrates this concept for both narrow and wide program images.

Figure 5-4. Mapping a Program to a Logical Space



The mapping of individual program object modules is accomplished by the infamous linking-loader. This program is usually coupled with the compiler and must understand which modules will be needed, and where to find them. Frequently, the build of the runnable image is controlled by the `make` utility and a corresponding *`Makefile`*.

Prior to the final linking of the executable image, the compiler builds relocatable object modules and a skeletal symbol table. The linking-loader must make sure that following the link there are no undefined external references to procedures not linked into the image (or to be made available via shared libraries at runtime).

The linking-loader must understand the system architecture for which it is building the executable image. Each hardware platform has its own unique way of dealing with the assignment of sections of virtual memory. In effect, the program's objects, or code and data blocks, are separated according to their function

and type, and level of security they required.

Most modern compilers divide the text (instruction code) of a program into one object and program data into one or more additional objects. HP PA-RISC systems follow certain defaults for the setting of access rights and permissions depending upon which quadrant a page frame is mapped into.

Narrow executables and wide executables map objects in a different manner. This change was necessitated by the desire to allow narrow and wide processes running on the same machine to share data objects. To accomplish this goal, it was necessary for wide processes to have quadrant **Q1** be allocated for shared objects (after all, the entire logical space of a narrow process fits easily into the first.01% of a wide process's **Q1**).

For a narrow application, the defaults are as follows:

Q1 is shared read-only access,

Q2 is private read-write,

Q3 is shared read-write, and

Q4 is shared read-write.

For a wide application,

Q1 is shared read-write access,

Q2 is shared read-only,

Q3 is private read-write, and

Q4 is shared read-write.

As we mentioned, the compilers must be aware of these rules, but as with all rules, there may be exceptions!

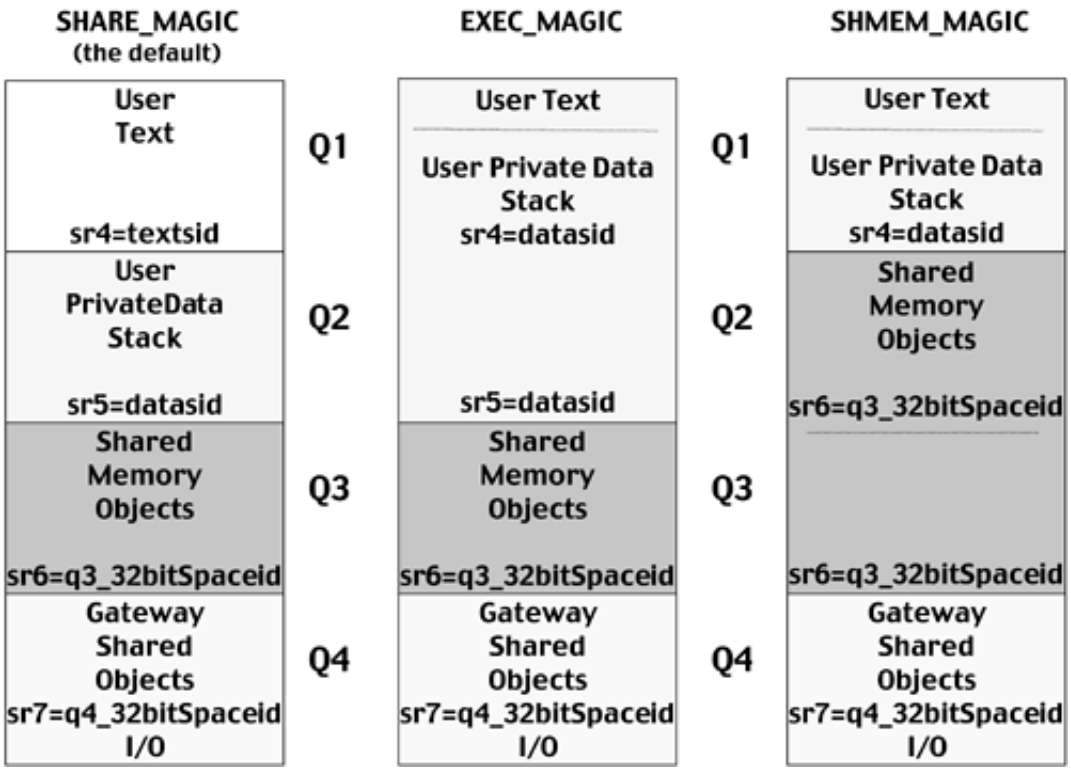
You may have noted that because of the implied restrictions controlling which program objects may be placed in which quadrants, there are systemwide limits to how large a program memory object may be. In general, a single object may not be larger than the quadrant to which it is mapped, and in many cases it must share the space with other objects. On a 64-bit system, this is not currently much of a concern, considering its current 4-TB hardware-implemented quadrant size, but for 32-bit applications with a 1 GB quadrant size, this has become a serious limiting factor to some applications.

The size of a process's private data quadrant may be a concern, and in some cases there are only 1 3/4 quadrants available for shared objects on the system, which has been a limiting factor to the applications we wish to run (remember that in **Q4** on a narrow memory map, the last 256 MB are reserved for memory-mapped I/O). There are several tricks of the trade that may be implemented to address some of the limits. Let's make some magic!

`SHARE_MAGIC`, `EXEC_MAGIC`, and `SHMEM_MAGIC`

The first compiler directives we consider are `SHARE_MAGIC`, `EXEC_MAGIC`. The first is the default for HP-UX compilers, and the second allows a program to increase the amount of private memory space it may use. Figure 5-5 gives us a visual comparison of the logical mapping for these options.

Figure 5-5. Magic for 32-Bit Applications



These options are only for use by narrow programs (running on either a 32-bit or a 64-bit kernel and hardware platform).

SHARE_MAGIC

The SHARE_MAGIC directive is the system default and requires no special parameter be passed to the compiler or linking-loader. The term *share* as used here refers to the fact the executable image should be built with the code and data separated into different objects. This is necessary if you want to allow multiple copies of the same program to be resident on the system and have them share access to a common text object, reducing overall system memory utilization. This is often called a multiprocess shared-code access model and is implemented on most modern operating systems.

In this model, the various memory objects are assigned to the quadrants according to the default access settings. A program's data and possibly a null-dereference page are placed in **Q1**.

Q2 gets, starting from the lowest offset, the process's private data, comprised of the program's initialized data, uninitialized data (or BSS, an old mainframe term: Block Store by Symbol), and the beginning of the process's heap space (this is where additional data storage may be allocated to a running process using calls such as malloc()). The uarea is located near the highest offset into **Q2**, and working back toward the private data area, we have the process's *stack*, which contains a "red zone" and room for the process's private stack area. Next, we find any private memory mapped objects. The *heap* grows up, and the *mmap*s grow down until there is no space left. The red zone we mentioned is a type of buffer area at the high-address end of the stack; this address space is off limits to the process. In the case of a stack overflow, the kernel must kill the process, and the red zone space is used by the signal code during the kill procedure.

Several tunable kernel parameters control the size of objects located in **Q1** and **Q2**.

maxtsiz limits a 32-bit process's text object; defaults to 16384 pages.

maxtsiz_64bit limits a 64-bit process's text object; defaults to 16384 pages.

maxdsiz limits a 32-bit process's private data object; defaults to 65536 pages.

maxdsiz_64bit limits a 64-bit process's private data object; defaults to 65536 pages.

maxssiz limits a 32-bit process's stack object; defaults to 2048 pages.

maxssiz_64bit limits a 64-bit process's stack object; defaults to 2048 pages.

As you can see, these default values are fairly conservative. The reason for setting these limits along with other tunables is to give the kernel a best guesstimate as to the amount of space it will need to allocate for its internal data tables. It can always allocate more pages as needed, but it is nice to start with a reasonable amount at system boot.

The **Q4** and **Q3** quadrants are used for shared library text, shared memory, shared memory-mapped files, and other shared objects. The **Q4** space is allocated first, then **Q3** (this order was established with the release HP-UX 10.0; in previous versions **Q3** was used first, then **Q4**).

Alright, you can't fit your process data in the single quadrant allocated, then perhaps you need a little executive magic!

EXEC_MAGIC

The EXEC_MAGIC directive changes things a bit. The first thing we notice is that the process's private data starts immediately after the end of its text. In most cases, a program does not need an entire quadrant for its text object (and I certainly wouldn't want to be around when you tried to print out the source code for a program that produced a gigabyte of machine code, or when you tried to debug it!). As a result, most processes have a considerable amount of unused space in their first quadrant. The EXEC_MAGIC directive allows a process to make use of this space. In general, this increases the amount of private data space to approximately 1.9 GB(depending on the size of the text object, stack, and uarea). An interesting side effect of this option is that the process's text is now writable; if you want to try your hand at self-modifying code (definitely not for the casual programmer!), this is one method to explore.

To solve this dilemma, Hewlett-Packard introduced the practice of virtual address aliasing with HP-UX 10.0. Physical text page frames of an EXEC_MAGIC process are mapped to two unique virtual addresses by the kernel. This allows multiple instances of the same program to share access to common physical page frames. Because these pages are mapped to separate "private" virtual quadrants, there must be a means to avoid corruption if one of the process threads tries to write to the page frame. This issue is handled with a technique called *read-only aliasing*. If any of the processes sharing the physical page frame attempts a write, then a kernel mechanism called *copy-on-write* simply creates a duplicate copy of the physical page frame and relinks the writer's virtual page to it. In this manner, as long as you are only reading from the page, data corruption is avoided and we can enjoy the advantage of sharing text pages, thus reducing overall memory pressure (always a good thing!). There is a price to pay for this action. We are now placing private data in **Q1**, and we will need to change access permissions to private-read-write. Prior to HP-UX 10.0, this meant that programs compiled with this directive were not allowed to run in a shared text mode: two instances of the same program required two copies of the text to be placed in memory.

In the case of EXEC_MAGIC, the four quadrants could be mapped to four different spaces (their access controlled by the four space registers, **sr4** through **sr7**). For an EXEC_MAGIC process, **Q1** and **Q2** must reside in the same virtual space (**sr4** and **sr5** will have the same value), allowing the data object to cross the **Q1**/**Q2** boundary with no discontinuity. Quadrants **Q3** and **Q4** may each be mapped to different spaces.

What if your size problem isn't with the private data space; instead, you simply need more shareable memory. SHMEM_MAGIC may be just the trick you are looking for.

SHMEM_MAGIC

The SHMEM_MAGIC directive is very similar to the EXEC_MAGIC in that it combines text and data objects in the process's **Q1**. The big difference is that the **Q2** does not contain private data but instead has been turned over for use for additional shared objects. It would do very little good to run only one process compiled with this option—after all, who could it share this new space with?

Again noting Figure 5-5, we see that with this option all four quadrants may be mapped to different virtual spaces. In general this option is used in conjunction with another relatively new option, Memory Windows. We examine memory windows in just a moment.

More Magic

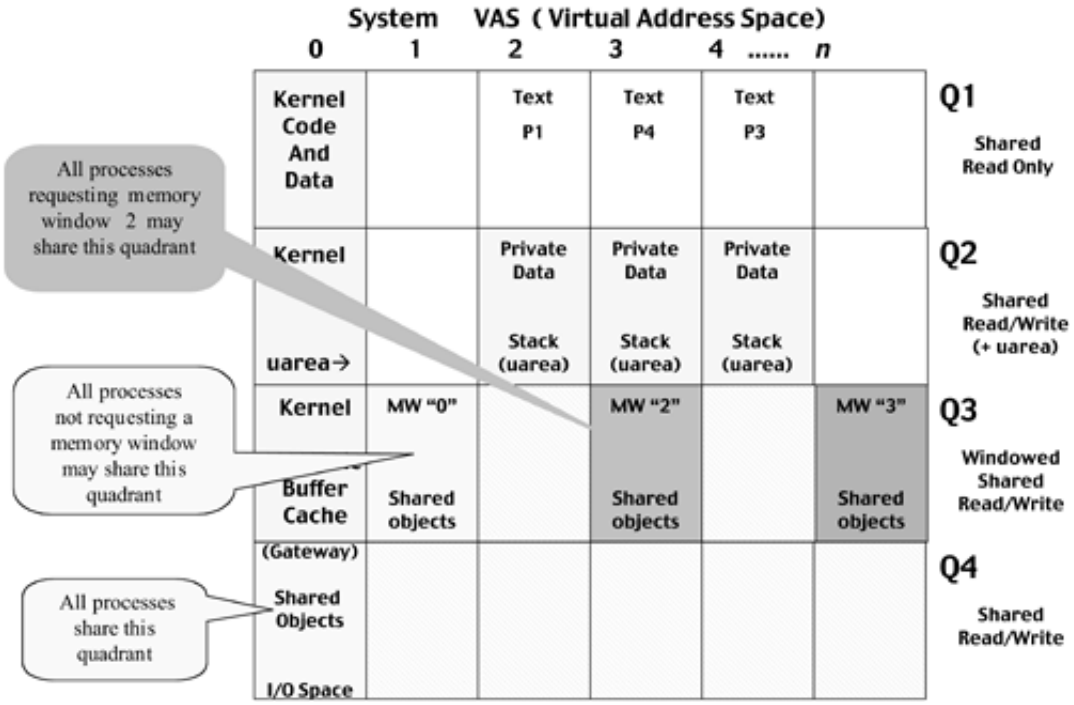
In addition to the three magic directives we have mentioned, there are several others. Compilers designed for HP-UX also understand `DEMAND_MAGIC` and `SHL_MAGIC` directives. Both of these are enabled by default and deal with the issue of demand page loading and the use of shared libraries located and linked at runtime. We also have `RELOC_MAGIC` for loading a program in relocatable-only mode, `AR_MAGIC` for linking in archive libraries, and `DL_MAGIC` for the inclusion of dynamic libraries. This concludes our magic lesson for now. Let's move on to memory windows.

PREV < Day Day Up > NEXT

Memory Windows

The memory windowing system was introduced to HP-UX 11.0 as a patch (PHKL_13810 and PHCO_13811) and is part of the HP-UX 11.i release. Figure 5-6 shows a greatly simplified example of a system VAS with memory windows implemented. To enable memory windows on a system, the kernel-tunable parameter `max_mem_window` must be set to the desired number of user windows. The kernel adds 1 to this value and initializes a memory-window array structure during system boot.

Figure 5-6. `SHARE_MAGIC` and Memory Windows



NOTE

Memory windows may only be implemented for 32-bit narrow application memory mapping.

The kernel pointer `*memWindows` directs us to an array of individual window data structures of type `vas_window`. Let's take a look at this structure in Listing 5.3.

Listing 5.3. `q4> fields struct vas_window`

```
The first three fields are used to map quadrant Q2
```

```
(in the case of SHMEM_MAGIC)

 0 0 4 0 u_int q2_32bitSpaceid

 4 0 4 0 u_int q2_32bitSpaceidInitial

 8 0 4 0 *     q2_32bitMap

The next two fields are used to map quadrant Q3

12 0 4 0 u_int q3_32bitSpaceid

16 0 4 0 *     q3_32bitMap

The "key" is required to join a window

20 0 4 0 int   key

This keeps track of how many processes are currently using the

window.

24 0 4 0 int   refCnt

This value keeps track of the last process to join the window

(useful in debugging)

28 0 4 0 int   lastSetpid
```

The first window in the array is reserved as the global window. Its **Q3** (and **Q2**) space IDs are the system defaults used by all processes that do not request a specific memory window assignment.

The second window is reserved for `q3_private` process usage (coming up next in our discussion). The rest of the windows may be requested by system call or user command, up to the configured limit.

There are two mechanisms by which a process may join a window: *pure inheritance* refers to a child of a process that has inherited its window association from its parent; *key associative* refers to a process that has either made the `_set_mem_window()` system call or has had the `setmemwindow` user command issued on its behalf.

Fundamentally, when a process joins a memory window, its **Q3** (and **Q2** for `SHMEM_MAGIC` processes) space ID is obtained from the appropriate `vas_window` array element. You may have noticed the `*q2_32bitMap` and `*q3_32bitMap` pointers in the `vas_window` structure. These point to resource maps used to managed availability of address ranges within the window quadrants. The kernel must keep track of which areas of a quadrant are in use and which are free so that space may be allocated to handle requests from new processes. The resource map approach was covered in Chapter 3, "The Kernel: Basic Organization."

One thing to note is that on a system with memory windows enabled, the basic algorithm for allocation of shared objects is modified slightly. Since there is only one truly global quadrant now (**Q4**) and it is slightly less than 75 percent available (remember the gateway page and the 256 MB of I/O space mapped here), shared library space comes at a slightly higher premium.

For a non-memory windowed system, the allocation routine for shared objects is as follows:

1. Try finding space for the object in **Q4** (regardless of object type).
2. If step 1 fails, then look in **Q3** (and then in **Q2** for `SHMEM_MAGIC`).
3. If step 2 fails, then return an error.

For a memory window-enabled system, the new routine is as follows:

1. If the object is a shared library, try to put it in **Q4**; if no room in **Q4**, then try **Q3** (or **Q2** for `SHMEM_MAGIC`).
2. If it isn't a shared library, then place it in **Q3** (or in **Q2** for `SHMEM_MAGIC`).

3. If step 2 fails, then place it in **Q4**.

4. If step 3 fails, then return an error.

The policy used by the kernel is controlled by `allocAddrPolicy`, which is in turn set according to the value of `max_mem_window`. (If `max_mem_window = 0`, memory windows are disabled and we use the first policy; if `max_mem_window != 0`, memory windows are enabled and we use the second policy.) In addition to the tunable parameter, the kernel parameter `maxMemWindow` is maintained and set to `max_mem_window` plus 1 (this allows space for the second `q3_private` window).

To use q4 to monitor memory window usage,

q4> load struct vas_window from memWindows max maxMemWindow

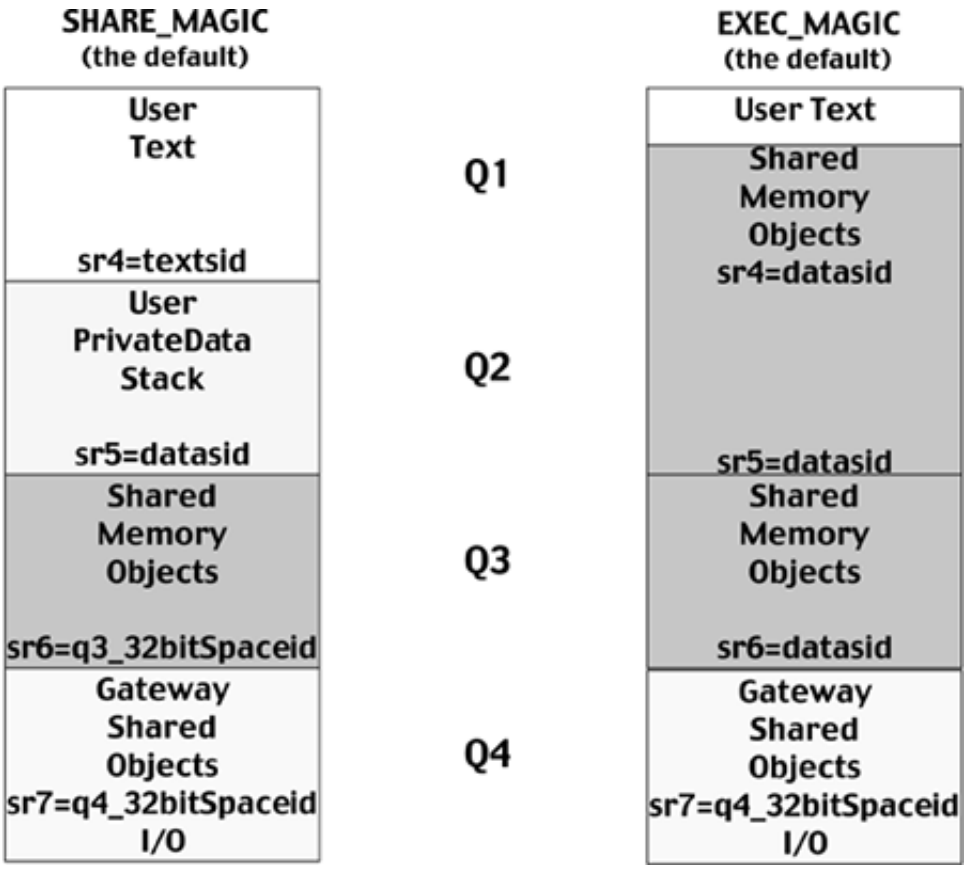
q4> prinrfrt -tx key refCnt

We have mentioned the term `q3_private` a couple of times—now we look at this new option.

Q3 Private Address Space

With the release of patch PHKL_20224 for HP-UX 11.0 and on subsequent releases, 32-bit narrow applications running under control of a 64-bit wide kernel on 64-bit PA-RISC 2.0 hardware have an additional mapping option available to them (a 32-bit kernel will simply ignore this request for large private data). Figure 5-7 illustrates the basics of this option.

Figure 5-7. Q3 Private Address Space



The classic narrow process map reserves **Q3** and **Q4** for shared objects, but what if the process needs an extraordinary amount of private data space? As there is a hard limit of four quadrants, the only way to increase a process's private data space comes at the expense of its ability to access system-shared space.

To implement this new feature requires the use of the `chattr` command-line utility. Two new options have been added to `chattr`: `+q3p <enable/disable> <process-name>` and `+q4p <enable/disable> <process-name>`. Before either of these commands may be issued, the kernel must be tuned to allow the new, larger private data size, which is accomplished by tuning the age-old parameter `maxdsiz`.

Several other stipulations apply to a `q3_private` process. Such processes cannot share objects not created by another `q3_private` process even in **Q4** unless the objects were created by a non-`q3-private` process using the `IPC_GLOBAL` or `MAP_GLOBAL` flags, which may require a recompile. If a recompile is not an option, then all processes needing to share objects in **Q4** will simply need to be made into `q3_private` processes (`chattr +q3p enable <process-name>`).

In a limited number of extreme cases, you may wish to extend the process's private data space even farther. This may be done (but only with extreme caution and need!) by invoking the `+q4p` option with `chattr`. The use of `+q4p` presupposes the implementation of the `+q3p` rules. This is an extreme decision, as a process running in this mode will have access to no shared objects whatsoever (with exception to the I/O space and the gateway page).

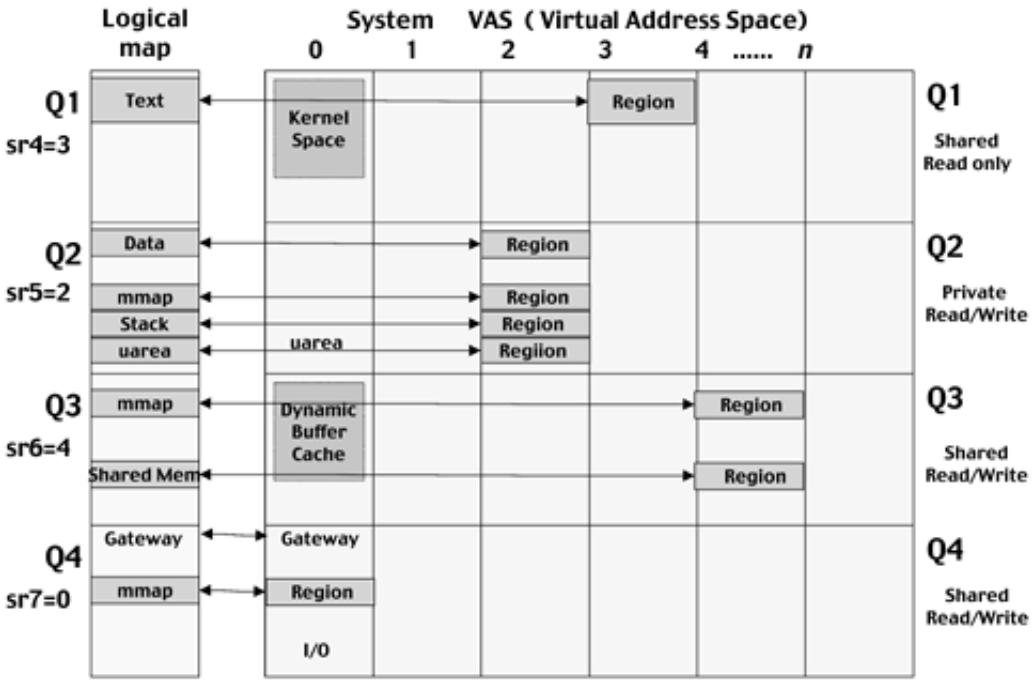
Note that the space IDs for the extended data spaces are identical. In effect, this means that the logical data size may exceed the previous limits. While this does bring some relief, all is not skittles and beer here. The fixed placement of the gateway page means that even with `+q4P` invoked, a singular data object may not extend past the beginning of the fourth quadrant. In order to use the additional space in **Q4**, the process must make private memory map calls using `mmap(MAP_PRIVATE)` or `malloc()`, which has been modified to address this limit.

As an observation, if you need to utilize these types of options to make your process fit into the 32-bit narrow environment, perhaps your process is trying to tell you something—like, "I need to be a 64-bit wide process!"

Building the Logical Map

Now that we have muddied the waters of a process's view of memory a bit, we need to stop and consider how these various mapping strategies may be tied to a specific process. The header in a program file contains information regarding the various magic options used during the compile; the kernel takes these into consideration as it sets up the process and `kthread` environment. Figure 5-8 presents a model of the logical and virtual address spaces.

Figure 5-8. The Process Logical Address Space

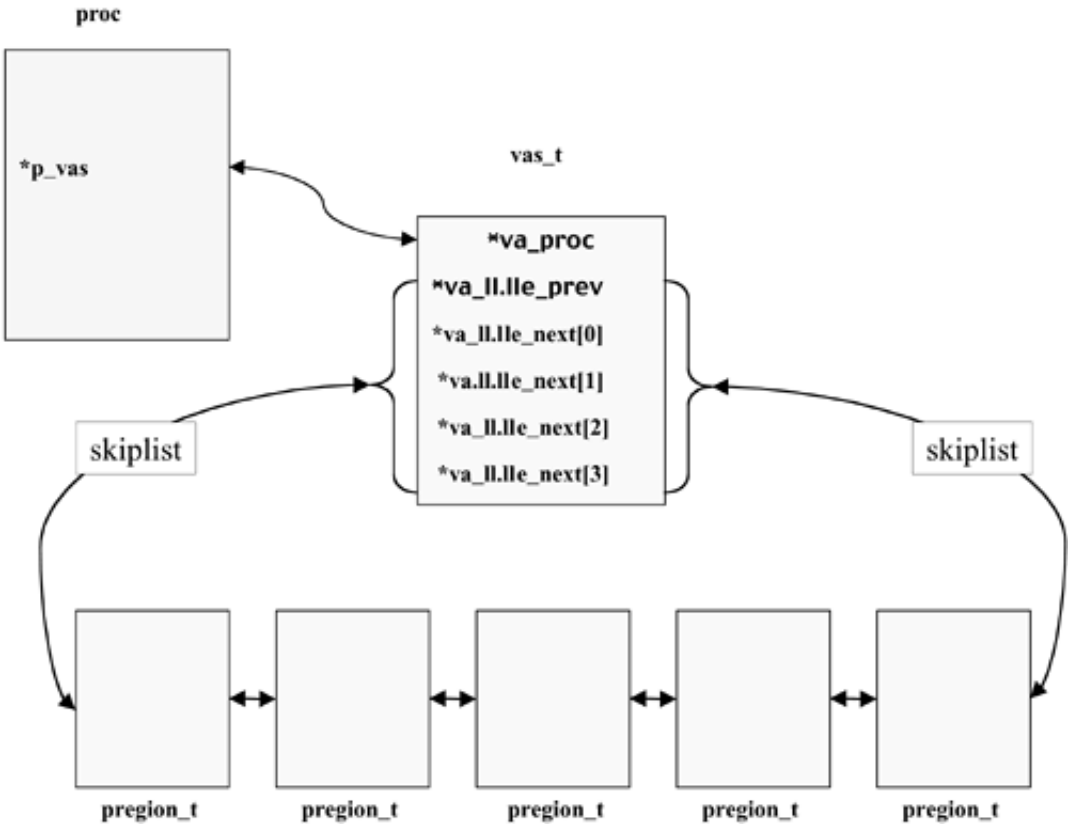


The various options we have discussed dictate where in the virtual map the various memory objects of a process may be allocated. It is up to the kernel to handle issues such as sharing and privacy. The kernel also must track what is where and who it belongs to; these issues are covered in Chapter 6. For now, our attention should be turned to the issue of mapping specific virtual memory objects to the logical map of a process. To understand how this is done, we must look to the `vas` and `pregion` structures created by the kernel and linked to a process's `proc` and `kthread` structures when a process is created (by the **`fork()`** or **`vfork()`** system calls). On to the `vas`!

The `vas`

The `vas` (virtual address space) structure is merely the head of a list of `pregion` structures, which in turn define the type and location of the various memory resources required to build the process's logical address space (see Figure 5-9). Modern compilers build an executable image by loading discrete components of the program into contiguous address blocks and then linking them together. This approach is necessary if you want to implement shared libraries, memory-mapped files, shared text, and shared memory—all features of the HP-UX kernel.

Figure 5-9. The `vas/pregion` List



To clarify and equate a couple of terms, contiguously addressed logical memory blocks are mapped to virtual page objects. These are also called memory regions. The term *region* goes back a long way in the history of UNIX kernels and is present in virtually all UNIX and Linux implementations.

One example of the use of the `vas` structure occurs when there is a physical page fault. The kernel's fault handler must locate the specific region of memory in which the missing page is managed (this is discussed in greater detail in Chapter 6). The kernel knows which `kthread` was executing when the fault occurred. The `kthread` structure provides a direct pointer to the associated `proc` structure, and it in turn provides a pointer to the `vas` structure at the head of the linked list of `pregion`s that define the process's logical address space. Next, the fault handler must search the chain of `pregion` structures to find the one containing the faulted page address. Once the correct `pregion` is found, page frame–specific information is contained in the kernel-managed `region` structure pointed to by the process's `pregion`. To speed up the searching of these linked structures, the kernel uses a four-level skip list.

As `pregion`s are added to a process's `vas` chain, they are assigned a skip level between 0 and 3. Level determination is pseudo-random in nature; the general rule is that we are four times as likely to be assigned level 0 as level 1, and four times as likely to be assigned level 1 as level 2, and four times as likely to be assigned level 2 as level 3. Even given this rule of thumb, you can't predict which level any specific `pregion` will have until it is allocated by the kernel. We discussed the mechanics of skip lists earlier in this book.

Let's start our examination of the `vas` structure (Listing 5.4) with the skip list pointers, note there are four forward pointers and a single `prev` or reverse pointer.

Listing 5.4. `q4 fields struct vas`

The **va_hilevel** records the highest skip level in use for this specific **pregion**.

```
  0 0 4 0 *       va_ll.lle_next[0]
  4 0 4 0 *       va_ll.lle_next[1]
  8 0 4 0 *       va_ll.lle_next[2]
 12 0 4 0 *       va_ll.lle_next[3]
 16 0 4 0 *       va_ll.lle_prev
 20 0 4 0 int     va_hilevel
```

Next comes a reference count, an approximate count of shared resident page frames, an approximate count of private resident page frames, and an approximate count of private page frames in swap.

```
 40 0 4 0 int     va_refcnt
 44 0 4 0 int     va_rss
 48 0 4 0 int     va_prss
 52 0 4 0 int     va_dprss
```

A pointer to the **proc** structure or a pointer to a memory-mapped file's **vnode** and a pointer to the process's credentials is kept for quick access

```
 56 0 4 0 *       va_proc
 64 0 4 0 *       va_fp
 96 0 4 0 *       va_cred
```

Hardware-dependent information is next

```
100 0 4 0 u_int   va_hdl.hdl_textsid
104 0 4 0 u_int   va_hdl.hdl_textpid
108 0 4 0 u_int   va_hdl.hdl_datasid
112 0 4 0 u_int   va_hdl.hdl_datapid
116 0 2 0 u_short va_hdl.v_hdlflags
120 0 4 0 int     va_ki_vss
124 0 1 0 u_char  va_ki_flag
125 0 1 0 u_char  va_stlocked
```

Total user space virtual memory count

```
128 0 4 0 int     va_ucount
```

Run environment information

```
132 0 4 0 enum4   va_runenv.r_machine
136 0 4 0 enum4   va_runenv.r_arch
140 0 4 0 enum4   va_runenv.r_os
```

```
144 0 4 0 enum4   va_runenv.r_asmodel

148 0 4 0 enum4   va_runenv.r_magic

Large page-specific information

152 0 4 0 int     va_lgpg_env.lgpg_data_size

156 0 4 0 int     va_lgpg_env.lgpg_text_size

160 0 4 0 int     va_lgpg_env.lgpg_next_brk_inc

164 0 4 0 int     va_lgpg_env.lgpg_user_brk_cnt

168 0 4 0 int     va_lgpg_env.lgpg_next_stk_inc

Memory window information and Memory Reference Group info for

private and shared page frames (this is where a process may

point to a specific memory window for the values for its Q3

and possibly Q2 space registers)

172 0 4 0 u_int   va_winId

176 0 4 0 *       va_winMap

180 0 4 0 *       va_priv_mrg

184 0 4 0 *       va_shar_mrg
```

Let's move our discussion on to the `pregion`.

The `pregion` (pseudo-region)

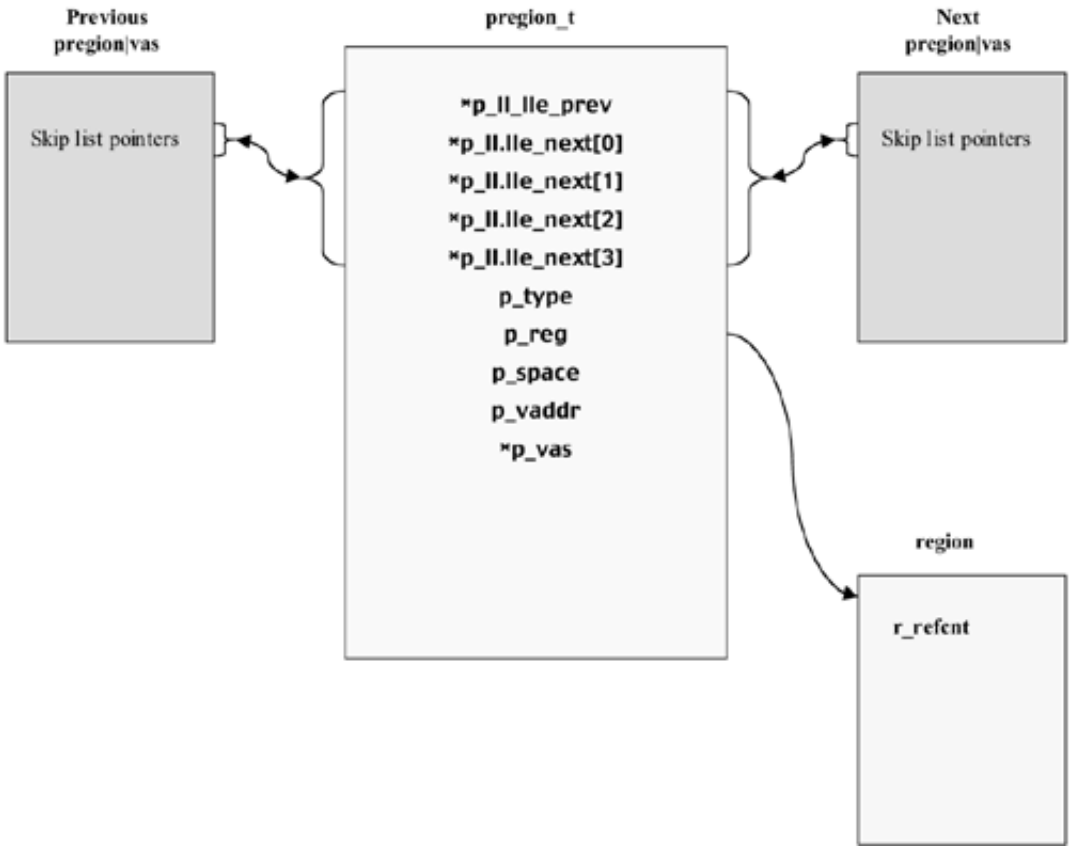
This is a good point to discuss what is meant by a *pseudo-structure*. As mentioned earlier, memory management may be approached from two perspectives. A process could care less if a page is private or shared—it simply wants it to be made available when its data or text is needed during the execution of its code. From the process's viewpoint, the `vas` should provide a direct connection to individual page-frame management data, but in practice it is not that simple!

The kernel also has a perspective when it comes to memory management. We fully explore this point of view in Chapter 6, but for now must consider the basic idea of memory regions. A region is a contiguous range of virtual page frames to be used in a consistent manner with common access rights and privileges. To increase memory utilization efficiencies and provide various types of shared access to memory objects, a concept of private and shared regions has been developed.

A reference count is kept in the `region` structure, tracking the number of `pregions` that are currently pointing to it. If this reference count is reduced to zero, then the kernel deallocates the `region` structure and all the memory resources it controls.

Region sharing is accomplished through the creation of an additional abstraction layer in the kernel. For a process to claim access to a region, a structure known as a `pregion` (or pseudo-region) is linked into the process's `vas` chain. Each `pregion` in the process's chain points to a specific region mapped to the kernel's VAS (see Figure 5-10).

Figure 5-10. The `pregion` Structure



Pregions from multiple processes may point to the same kernel region, thus creating a shared memory object under kernel control. A region only knows the total number of page frames it contains, where they have been placed in the kernel's VAS map, whether it is private or shared, and the specific status and location of each page frame. The actual purpose of these pages (whether it is text, data, a memory-map or shared memory) is not known at this level.

Conversely, the pregion maintains a detailed list of attributes from the perspective of the process, where it has been linked into the process's logical memory, its usage type, size, and current status.

Let's examine the pregion fields (Listing 5.5).

Listing 5.5. q4 fields struct pregion

```
As with the vas, the first entries in the pregion structure

hold the skip list pointers

  0 0 4 0 *       p_ll.lle_next[0]

  4 0 4 0 *       p_ll.lle_next[1]

  8 0 4 0 *       p_ll.lle_next[2]

 12 0 4 0 *       p_ll.lle_next[3]

 16 0 4 0 *       p_ll.lle_prev

Enumerated flags are next
```

```
 20 0 4 0 enum4   p_flags
Each region is assigned a type number depending on its usage
 24 0 4 0 enum4   p_type
The type numbers and enumerations are:
1=PT_UAREA per-thread kernel management area
2=PT_TEXT shared, read-only instruction text
3=PT_DATA  private, read-write process data area
4=PT_STACK private, read-write process stack area
5=PT_SHMEM shared, read-write memory region
6=PT_NULLDREF shared, read-only null pointer dereference
7=PT_IO shared, read-write I/O memory-mapped area
8=PT_MMAP shared or private, read-write memory-mapped file
9=PT_GRAFLOCKPG Frame buffer lock page (used by some graphics devices)
10=PT_ANON_SHMEM shared, read-write anonymous shared memory
11=PT_GRAFDMA graphics DMA region
12=PT_RSESTACK RSE stack (HP emulation on IA64 only)
13=PT_GATEWAY shared, read-only system call gateway page
Next, a pointer to the kernel region structure, the virtual
space for the region, and the starting virtual address for the
region
 28 0 4 0 *       p_reg
 32 0 4 0 u_int   p_space
 36 0 4 0 *       p_vaddr
These next 5 parameters are used by the paging system (more
later)
 40 0 4 0 int     p_off
 44 0 4 0 int     p_count
 48 0 4 0 int     p_ageremain
 52 0 4 0 int     p_agescan
 56 0 4 0 int     p_stealscan
A direct pointer to the vas this pregion is chained to
 60 0 4 0 *       p_vas
A direct pointer to the owner kthread if this is a PT_UAREA
 64 0 4 0 *       p_thread
All pregions on the system are dual linked by these two
pointers; this master list will be searched when memory
pressure triggers the paging system to become active.
 68 0 4 0 *       p_forw
```

```
72 0 4 0 *       p_back
```

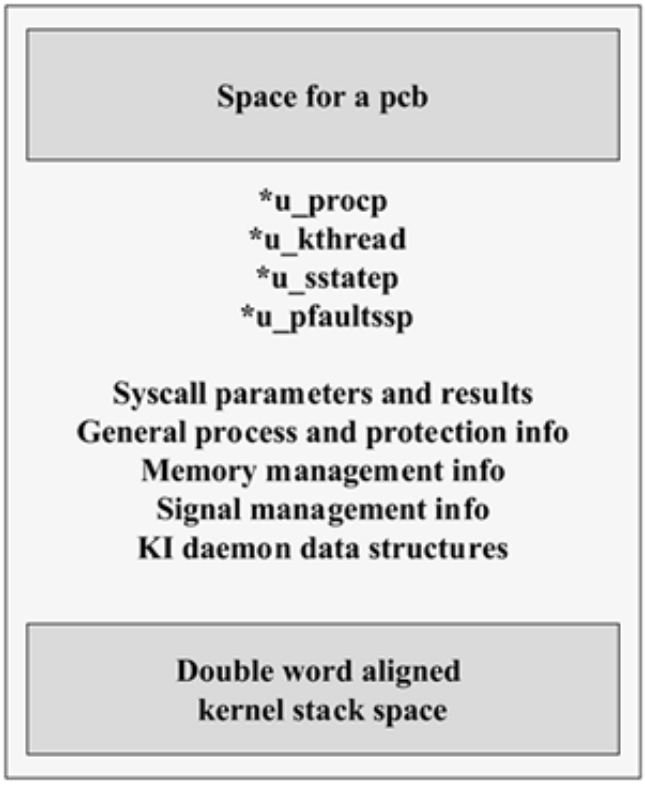
Next, we examine the `uarea`.

The `uarea`

This snapshot includes a copy of the general registers, control registers, program counters (space and offset queues in the case of PA-RISC), the processor status word, and space registers. The snapshot is called a Process Control Block, or `pcb`. A key element of any multitasking operating system is the ability to switch between the various threads of execution. This basic function is called a context switch. The term *context* refers to the state of the machine when a specific thread is executing. If we are to halt execution of a specific thread and return to it at a later time, we need to take a snapshot of the processor's registers on which the thread is running at the instant we suspend execution and return them to this exact state when we want to resume.

The kernel must have a place to safely store the `pcb` of a sleeping or blocked process until it is required for a context switch back into the processor's registers. To this end, each process thread has created for it a memory region known as a `uarea` (see Figure 5-11).

Figure 5-11. The `uarea` Structure



The `uarea` is not visible from the process's threads and is mapped into the kernel space whenever a thread is activated (made the running thread on a specific processor). When a thread makes a system call or the kernel preempts a thread to handle an interruption, it already has the current thread's `uarea` mapped into

the kernel's memory view. If the kernel determines that it needs to perform a context switch prior to returning to the user space, then the current thread's `pcb` is stored in its `uarea`. Next, the kernel determines which thread to switch in and restores a `pcb` from its `uarea` prior to the transition to user mode. The kernel routine that manages this black magic is called `swtch()`and is one of the more expensive, but very necessary, actions the kernel may undertake!

The `uarea` comprise several key sections (see Figure 5-11). The first is the `user` structure, which contains room for the `pcb` and other thread-specific kernel management data, and the second is a kernel stack space. Each thread must provide a limited amount of stack space for use by kernel routines, drivers, and kernel services requested through the system call interface.

Consider for a moment the scenario where two independent threads make a request to the same kernel service, possibly a disk read or write request. As both threads enter the gateway page and transition from user space to kernel space, they are in effect running the same exact set of instructions in the kernel but from two different program contexts.

If the service being requested needs to make a procedure call (and the kernel employs an extensive set of library routines), it must allocate a new stack pointer to accommodate local data and register spill areas. If we could be guaranteed that all calls to the kernel would complete in a first-in last-out order, then a simple kernel stack could be used, but this may not be the case. If the requested disk transactions complete in the wrong order, the unwinding of a common stack could cause corruption!

To avoid this situation, each thread requesting kernel services (system calls) is expected to provide its own localized kernel stack, or `kstack`, space. Since a thread's `uarea` is already mapped into the kernel's space while a thread is running, it is a simple matter to add a minimal `kstack` to the end of this region. For a narrow kernel (32-bit), the `uarea` is sized to four page frames (one for the `user` structure and three for the `kstack`). For wide kernels (64-bit), it is eight pages frames (two for the `user` structure and six for the `kstack`).

Let's examine the `u_pcb` portion of the `user` structure (Listing 5.6).

Listing 5.6. `q4 fields structure uarea`

```
The u_pcb starts with storage space for the general registers,
r1 through r31 (there is no need to store the value of  r0
considering that it is the permanent zero source register!)
   0 0  4 0 u_long             u_pcb.pcb_r1
     -  -  -
 120 0  4 0 u_long             u_pcb.pcb_r31
Next, the 8 space registers, sr0 - sr7, are saved
 124 0  4 0 u_long             u_pcb.pcb_sr0
    -  -  -
 152 0  4 0 u_long             u_pcb.pcb_sr7
Control register cr1 and registers cr8 through cr31 are saved
here. Control registers cr2 - cr7 are reserved for use by the
hardware and not included as part of a thread's context
 156 0  4 0 u_long             u_pcb.pcb_cr0
 160 0  4 0 u_long             u_pcb.pcb_cr8
    -  -  -
 252 0  4 0 u_long             u_pcb.pcb_cr31
Room is also allocated for the threads pc, psw, ipsw, pcsqe,
```

```
pcoqe, and ksp
 256 0  4 0 u_long             u_pcb.pcb_pc
 260 0  4 0 u_long             u_pcb.pcb_psw
 264 0  4 0 u_long             u_pcb.pcb_ipsw
 268 0  4 0 u_long             u_pcb.pcb_pcsqe
 272 0  4 0 u_long             u_pcb.pcb_pcoqe
 276 0  4 0 u_long             u_pcb.pcb_ksp
The floating-point registers fr0 through fr31 come next
 280 0  8 0 double             u_pcb.pcb_fr0
    -  -  -
 528 0  8 0 double             u_pcb.pcb_fr31
The pcb ends with several miscellaneous parameters dealing
with the signal system and swap. After the pcb, the user
structure has pointers to the thread's proc structure,
kthread, a saved state, the most recent saved state
 584 0  4 0 *                  u_procp
 588 0  4 0 *                  u_kthreadp
 592 0  4 0 *                  u_sstatep
 596 0  4 0 *                  u_pfaultssp
The rest of the user structure contains an assortment of
syscall parameters (including the parameters passed to the
most recent system call made by the thread, very useful during
debugging a crash dump), process and protection information,
miscellaneous memory management information, various signal
management parameters, KI daemon metrics, and last but not
least a union with the provided kernel stack space.
```

To use q4 to examine a specific thread's **uarea**, first make your way to its **kthread** structure:

q4> load struct preg from kt_upreg ← load the uarea's pregion structure

q4> print –x p_space p_vaddr ← print out the virtual address information

q4> load struct user from <*p_space.p_vaddr*> from the previous print statement

---

Additional Resource Pointers

In addition to the various structures we have mentioned in this chapter, the `proc` structure provides pointers to a process's file descriptor and kernel signal structures. These are covered later in this book.

Process/Thread Scheduling

Now that we have seen the fundamental structures used to define a process's logical space and assign system resources, we must consider the task of scheduling threads for execution. This is one of the kernel's primary responsibilities and requires a considerable number of system resources. Traditionally, UNIX is designed to be a load-leveling operating system; that is, the kernel attempts to distribute access to system resources in an equitable fashion to all active threads. While this is a noble cause, at times you might like certain threads to receive a larger (or possibly smaller) share of the kernel's attention. Many operating systems accomplish this by assigning a priority to a job. UNIX has priorities but they don't function exactly the way you might suspect.

HP-UX uses priorities to place threads in an appropriate run queue. When a processor needs to determine what to do next, it simply goes to its strongest active run queue and context switches to the first thread in the queue. A thread's position in a run queue is subject to possible kernel-induced priority decay, "voluntary" time slicing (round-robin queuing), and preemption by threads from a stronger queue being awakened. In this section, we explain the various schedulers and introduce the "mystic formula" by which HP-UX adjusts a thread's priority.

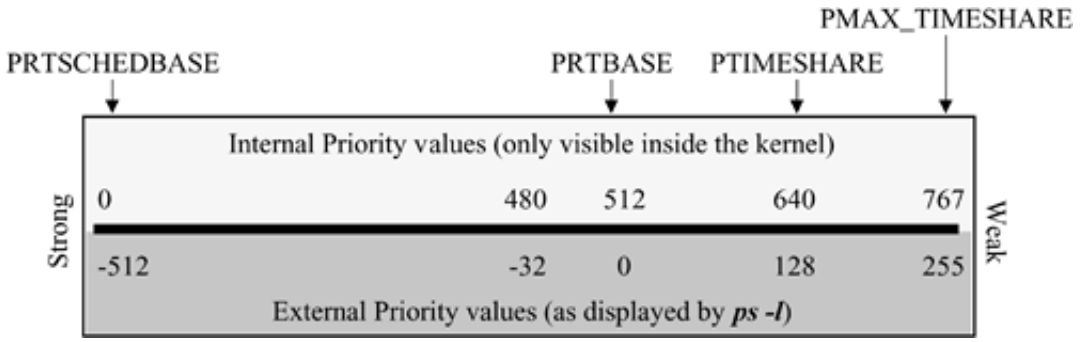
A natural tendency in the discussion of priorities is to think in terms of numbers. We encourage you to instead consider only relative strength: is one priority stronger or weaker than another? There are several different priority schemes in use in the kernel. For some, zero is the strongest possible priority, and for others it is the weakest! (see Table 5-1.)

Table 5-1. Priority policies

Scheduler	Policy #	Policy Label
HP-UX Timeshare	2	SCHED_HPUX
		SCHED_TIMESHARE
		SCHED_OTHER
	8	SCHED_NOAGE
HP-UX Real Time	6	SCHED_RTPRIO
	0	SCHED_FIFO
POSIX Real Time	1	SCHED_RR
	5	SCHED_RR2

The challenge for modern UNIX systems is to incorporate the various priority schemes into one contiguous model. HP-UX uses a combination of three primary scheduling schemes: the traditional UNIX timeshare, HP real-time, and POSIX real-time schemes. Each scheme maps a range of priorities to a series of run queues; Figure 5-12 illustrates how the three schemes have been merged into a single priority continuum.

Figure 5-12. Thread Priorities



Each run queue has both internal and external priority values. The POSIX priorities presented a couple of challenges for the kernel designers. First, the number of POSIX priorities is a tunable parameter; 1 is the weakest priority, and the strength progresses as the value grows to the maximum of 512. For the other existing priority schemes the more positive the integer value the weaker the priority! This conundrum was addressed by adding the maximum number of POSIX priorities (512) to existing number of HP real-time and timeshare priorities (256) for a new total of 768 possible values. For internal use a process/thread priority is assigned a value between 0 and 767. An equivalent external priority is calculated using kernel macros and will fall in the range of –512 to +255,

To understand the external number, we need to look back into the HP-UX history books a bit. Earlier versions of HP-UX had only 256 priority values—0 to 255, strongest to weakest—which were reported via commands such as `ps`, `top`, and `glance`. Legacy scripts and programs expect timeshare processes to have priorities between 128 and 255. In order to maintain this standard and protect these scripts and utilities, an external number scheme needed to be developed.

It was decided to report POSIX priorities as negative integer values starting with –1 and decrementing to the tunable maximum of –512. This approach meant that what used to be reported externally as the weakest priority of 255 stills comes out as 255. The strongest HP real time was and still is reported as 0, and the new strongest POSIX priority is reported as –512. Implementation of this model is really quite simple, as the POSIX real time priorities are simply inverted and added to this reporting continuum in the range of –1 to –512. This also maintains the concept that the more positive the value, the weaker the priority.

There are several kernel parameters that relate to the priority system:

`PRTSCHEDBASE = 0`: Strongest internal priority.

`RTSCHED_NUMPRI_FLOOR = 32`: Minimum number of POSIX queues.

`MAX_RTSCHED_PRI = 512`: Maximum number of POSIX queues.

`RTSCHED_NUMPRI_CIELING = ?`: Tunable number of POSIX queues (defaults to 32).

`PRTBASE = 512`: Strongest HP Real time priority.

`PTIMESHARE = 640`: Strongest timeshare priority.

`PMAX_TIMESHARE = 767`: Weakest timeshare.

`pzero = 153`: Uninterruptible sleep upper limit.

`puser = 178`: Start of the user-level priorities.

`timeslice = ?` (default is 10): Number of 10-ms ticks a thread is allowed before a voluntary context switch is called by the kernel.

As priority values are stored in the `kthread` structure (`kt_pri`) according to their internal value, you may

need to convert them when looking at dumps or q4 printouts. Here are the equations to convert between the two schemes.

To convert an internal priority to an external priority:

External-Priority = Internal-Priority – MAX_RTSCHED_PRI

To convert an internal priority to an external POSIX priority (the priority number you would pass as an argument in the **rtsched()** system call or **rtsched** line-mode command):

External-POSIX-Priority = (MAX_RTSCHED_PRI – 1) – Internal-priority

To convert an external POSIX priority to an internal priority:

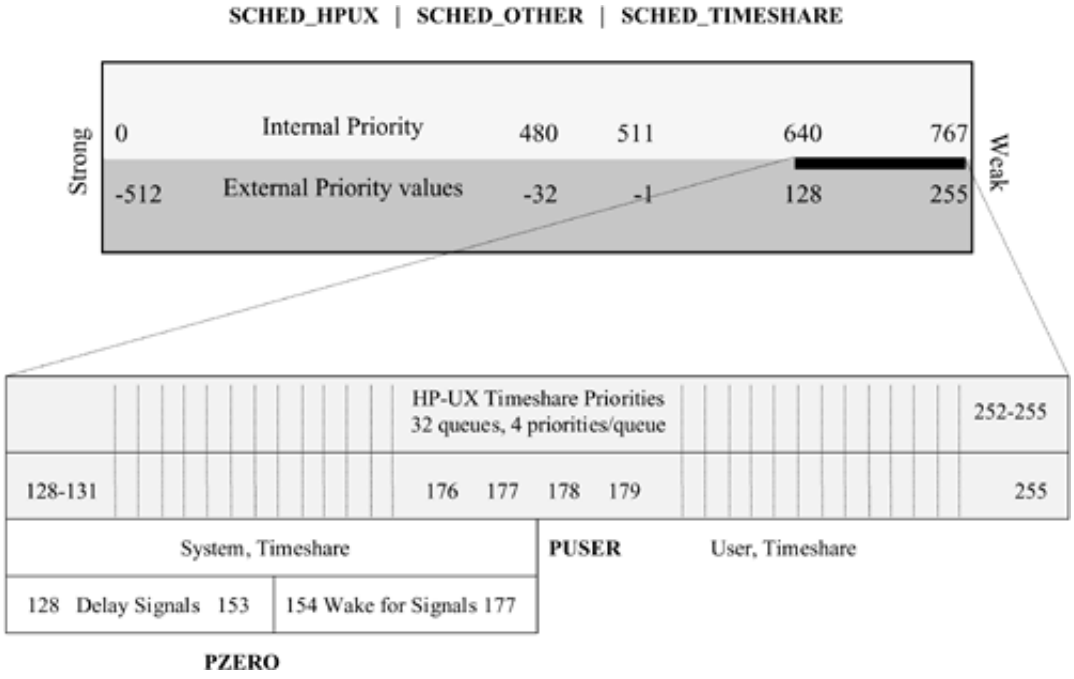
Internal-Priority = (MAX_RTSCHED_PRI – 1) – External-POSIX-priority

Next, let's examine the scheduling schemes.

SCHED_TIMESHARE and SCHED_NOAGE

The classic UNIX scheduler is implemented in HP-UX as SCHED_TIMESHARE. This policy has two additional aliases: SCHED_OTHER and SCHED_HPUX. In Figure 5-13, we illustrate the basics of the timeshare scheduler and position it with respect to the overall priority system.

Figure 5-13. Thread Priorities



This scheduling policy consists of 32 run queues. Each queue contains four sequential priority values. The strongest queue handles threads with priorities between 128 and 131, while the weakest handles those between 252 and 255. The four priorities per queue is part of the legacy left by the early UNIX development work done on an older Digital Equipment Corp. hardware platform. The system was set up to handle 32

unique queues; this number was carried through in the UNIX kernel design. When it was decided to increase the priority range to an 8-bit value (0 to 255), the existing 32 queues were divided into the upper half of the number range, allowing for the later introduction of additional higher priority schemes.

System-Level and User-Level Priorities

Overall, the **SCHED_TIMESHARE** queues are divided into two groups. External priorities 128 to 177 are system-level priorities; 178 to 255 are user-level priorities. The system parameter `puser` sets the base priority for the user level (this should not be changed!). The system-level priorities are reserved for privileged kernel routines and system call code running on behalf of a user thread.

Threads assigned one of the user-level priorities may have their value adjusted by the kernel depending on current system resource utilization levels or the thread's behavioral patterns (more on this later). This adjustment by the kernel is called *priority decay*. System-level priorities are said to be nondecaying; that is, their priority is not adjusted by the kernel.

Threads in the system levels may fall into one of two additional subdivisions: those that will be awakened from a sleep queue to handle an incoming signal and those that will delay signal handling until whatever they are blocking on wakes them. Have you ever tried to halt a process with a kill-9 only to see it still there when you run `ps` immediately afterwards? If so, then you may be looking at a thread sleeping in the un-interruptible range (128 to `pzero`).

Consider a kernel driver that has set up an interface card to perform a direct memory access (DMA) transfer between a data block on a disk and a physical address range in memory currently mapped to a thread's data space. If you were allowed to signal this thread to exit before the DMA completed, the kernel might attempt to return a physical page frame to the free list while the I/O interface was affecting its data transfer —not a comforting thought! For such reasons, a sleeping thread may be placed at a priority below `pzero` to assure that the DMA transaction completes and the associated external interrupt is posted by the interface to the kernel I/O system to wake the sleeping thread (also called un-blocking the thread). When the thread wakes, the signal will be caught prior to the transition from the kernel space back to the user space, which in most cases results in the termination of the process and its threads. The kernel parameter `pzero` is the upper limit of the group, which delays signal delivery until wake up.

Time Basics in HP-UX Scheduling: The Tick

Time in HP-UX is measured by the kernel in terms of *ticks*. A tick is defined to be 10 ms. The CPU's hardware clock signal is monitored by a configurable counter circuit. The counter is set such that an internal processor interrupt occurs every 10 ms. This interrupt is handled by a kernel routine known as `hardclock()`. As a thread runs, the number of ticks that occur while it is running is stored in the thread's `proc` structure in the field `p_ticks`. This counter circuit is also the basis for the network time protocol service in which a system compares its time of day to that of another "server" system. Systems either are peers or belong to various strata; a system of lower strata will make fine adjustments to its counter interface to effectively speed up or slow down its tick. These adjustments are continually refined until the system at the lower time strata is synchronized with the higher level server.

Time Slicing or Round-Robin Queuing

When more than one thread exists in the same run queue, a round-robin scheduling scheme is usually observed. The kernel-tunable parameter `timeslice` determines the number of ticks a thread is allowed to run before being rethreaded to the end of its queue. The default value of `timeslice` is 10 ticks, or 100 ms. Each thread maintains a count of its remaining ticks until the next timeslice in `kt_ticsleft` (in its `kthread` structure).

Every time the hard-clock interrupt occurs, the running thread on each processor has its `kt_ticsleft` decremented by 1 and compared to 0. If the thread's timeslice has expired, then the current thread has `kt_ticsleft` reset to `timeslice`, and it is moved from the front to the end of its run queue. In effect, threads of the same run queue take turns in a round-robin manner. If only one thread exists in a queue when its `timeslice` expires, it is simply rethreaded to the front of its own queue. The kernel realizes this is the case and stops short of actually performing a full context-switch operation and simply continues execution of the thread. The thread's `kt_ticsleft` is reset, and the countdown starts all over again in case new threads are added to the queue.

Priority Decay, `nice`, and User-Level Priorities

The HP-UX source code includes an interesting statement in the comments; it goes something like this:

> ***A thread's priority is adjusted according to a mystic formula handed down by kernel hack to seed as part of the oral UNIX tradition.***

We don't claim to be sage kernel hacks, but we would like to demystify this a bit. Each thread keeps a parameter in its `kthread` known as `kt_cpu`. In older UNIX incarnations there was something called the process *hog factor*. This newer, per-thread `kt_cpu`, to some degree, is a modern extension of that idea, although implemented in a somewhat different manner.

The `kt_cpu` may range in value between 0 and 255 and is used whenever the kernel needs to reevaluate a thread's priority. Another factor in the recalculation of a thread's priority is its `nice` value. The `nice` value is a user-provided parameter and may be set either programmatically via a system call or by the user from the command line, using `nice` when the process is first launched or `renice` to adjust the value of a running process.

The system assigns a default `nice` value of 20 to all processes; this value is stored in the kernel parameter `pnice`. The `nice` command has an overall range of 0 to 39. A regular user may only increase the `nice` value between 20 and 39, while a superuser may "nasty"—that is, negative nice—a process all the way to 0. Okay, so each thread has a `nice` value—so what? Here is the way the `nice` value and the `kt_cpu` come into play.

Periodically, the kernel stops to reevaluate the priority of all threads actively waiting to run on the system. This happens once a second, or every 100 tick**s** of the `hardclock()`. A kernel routine named `statdaemon()` calls `schedcpu()`, which calls its helper `setpri()`, which in turn calls `calcusrpri()` to assess and adjust user-level thread priorities. The basic formula looks something like this (remember that `puser` = 178 and `pnice` is 20):

> The new priority = (`puser` + `kt_cpu`/4) – 2(`pnice` – `p_nice`)

Upon examination of this pseudo-formula, we immediately see that if the thread's `p_nice` value is set to the default of 20, then the niceness has no effect on the calculation.

The `nice` value is stored in a thread's `proc` table structure in the `p_nice` field. As a result when you `nice` a process, you `nice` all of its threads to the same level. The `nice` parameter predates the introduction of threads to the HP-UX kernel. Table 5-2 shows the contribution of the nice value to a thread's calculated priority:

Table 5-2. `nice` Contribution to Priority Calculations

`nice` =	0	5	10	15	20	25	30	35	39
Priority adjustment	-40	-30	-20	-10	0	10	20	30	38

The contribution of `kt_cpu` may also be calculated; it is simply `kt_cpu`/4 and added to the base user-level priority `puser` (178). To complete our total understanding of this phenomena, we simply need to ask how `kt_cpu` is set. Simple, right? If only it were so. Enter the mystic formula!

A thread's `kt_cpu` is adjusted at the whim of the kernel. When a process requests kernel services, I/O, systems calls, and so on, the requested kernel routine may adjust the requesting thread's `kt_cpu` depending upon its mood. Seriously, if a resource is currently in heavy demand, then `kt_cpu` may be increased. If there is average traffic, it may be left alone, or if the kernel notes light usage, it may reward the thread and reduce it. During the `statdaemon()`, waiting threads are checked to see if they are being starved for CPU time—how long it has been since they had a turn as the running thread. If CPU utilization is low, this time should be very short; adjustments to `kt_cpu` may be made at this time as well. The kernel uses this as a means to avoid potential bottlenecks in various system resources. It is the sum total of the kernel's current activity level and the behavioral patterns of the thread which in the end determines how this parameter is set.

Our story is not quite over yet. When these two factors have been assessed and totaled, they are limit-checked against the weakest priority (255) and the base of the user-level priorities `puser` (178) and adjusted within these bounds.

There is another way that a thread's priority may decay: each time the total number of ticks a thread has used since it started is evenly divisible by 4, its current user-level priority is incremented by 1. This means that a user-level thread locked in a CPU loop would have its priority decay to the next weaker run queue after racking up 16 clock ticks (160 ms) assuming that it had started at the strongest priority in its run queue (remember there are four priorities per user-level run queue).

When both mechanisms—the "four ticks and you lose" and the once-a-second `statdaemon()` recalculation of all runnable thread priorities—are combined, the basic personality of punishment and reward appears: if you are a CPU-intensive thread, you will be caught by many clock ticks and priority decay will move you out of the active queue. If you stop and ask the CPU to perform lots of I/O, your chances of being the running thread at the hard-clock interrupt is drastically reduced and you will be dealt with in a much gentler manner.

There is another vague statement that bears repeating here:

> ***In general a thread will recover 90 percent of the priority points it has lost in four times the current percentage of CPU utilization, in seconds, once it has been pushed out of an active run queue.***

In other words, if the system is currently running at 80 percent CPU utilization, a thread that has suffered a loss of 10 priority points due to decay will recover 9 of them in 3.2 seconds. This is a very rough characterization but is surprisingly accurate.

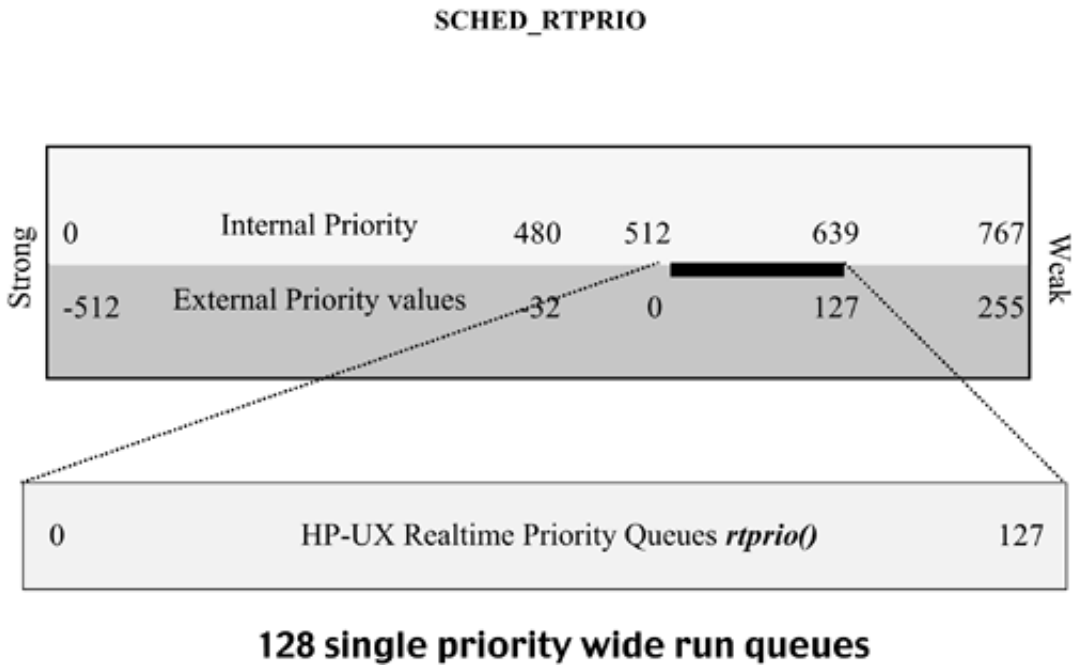
SCHED_NOAGE

Now that you are beginning to see what priority decay is all about, let's introduce a relatively new kid on the block. The policy `SCHED_NOAGE` allows threads in the user-level run queues to be assigned a fixed timeshare priority. These threads are linked to the same run queue as other threads in `SCHED_TIMESHARE` and compete alongside them for system attention, but their priorities are not changed by any of the mechanisms we just discussed. This feature was originally introduced to support database operations. Prior to this option, you had to use one of the real-time policies if you wanted to have a fixed (or nondecaying) priority assigned to a process or its threads. `SCHED_NOAGE` offers a thread many of the same benefits as real-time run queues but without the inherent danger of a system hang which could result if a real-time thread behaves badly.

SCHED_RTPRIO

A number of years back, Hewlett-Packard decided to implement a real-time scheduling policy in addition to the classic timeshare scheme. Hewlett-Packard has a strong history in real-time computing, specifically with its HP-1000 family of computers running the proprietary Real Time Executive operating system on A-series minicomputers (referred to as RTE-A). These machines were used in manufacturing and process-control systems, and as HP-UX entered into this space and was determined to be Hewlett-Packard's strategic future path, real-time extensions to the HP-UX kernel had to be developed. Hewlett-Packard addressed this challenge with the introduction of the `rtprio()` system call and `rtprio` line-mode command in the HP-UX 7.0 release. In Figure 5-14, we see the relationship of this scheduler to the overall priority scheme.

Figure 5-14. SCHED_RTPRIO



The real-time scheduler was named `SCHED_RTPRIO` and made use of the previously unused priority numbers 1 through 127. As Hewlett-Packard was writing its own rules at this point, the decision was to have a unique run queue for each priority number. These queues would use a round-robin scheme for threads in the same queue in the same manner as the `SCHED_TIMESHARE` scheduler and utilize the systemwide timeslice quantum value.

As **`SCHED_RTPRIO`** run queues are stronger than the system-level queues when a real-time thread requests kernel services and needs to be put to sleep (possibly to wait on an I/O operation to complete), the sleeping kernel routine is promoted to the requesting thread's real-time priority. This allows the blocking routine's interrupt handler to respond at the stronger real-time priority when it is time to finish the request.

Another advantage to real-time priority queues, at least from the kernel's viewpoint, is that they don't need to be considered by the **`statdaemon`** during the 100-tick priority readjustment cycle. Here is a hint: On a development system it is a good idea to always have at least one shell session running at a stronger real-time priority than the code being developed; if a real-time application becomes a runaway, the stronger shell could be used to send it a kill signal as long as the **`tty`** (keyboard) interrupt handler is also running at a stronger priority.

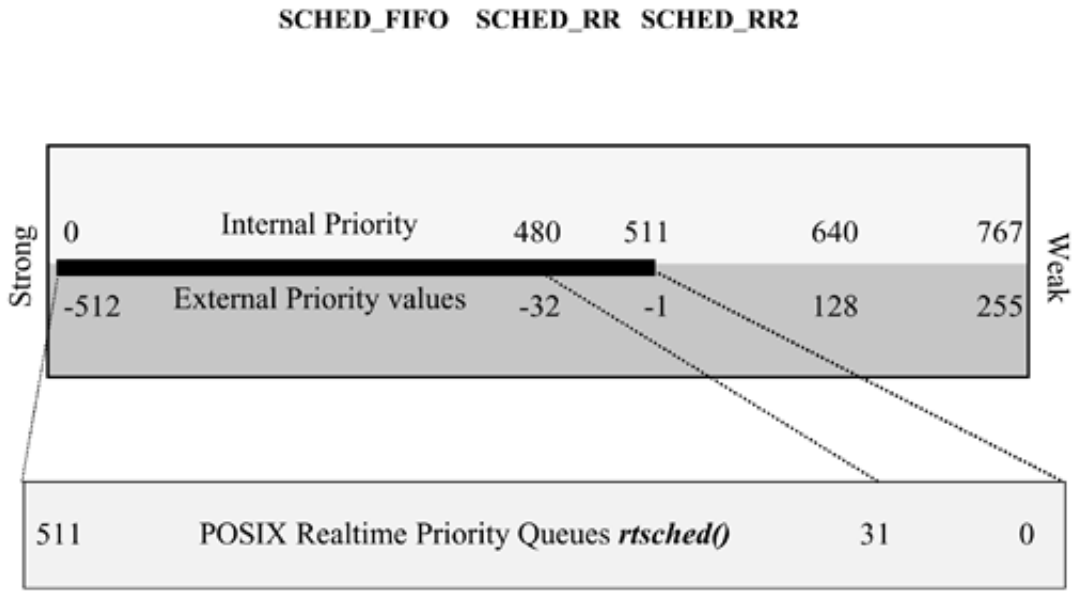
Take a look on an HP-UX 11i system: what process runs at the absolute strongest priority? It should show a process named `ttisr` running at a priority of –32. On most systems this is the strongest configured POSIX real-time priority. This routine is the teletype interrupt service routine—no matter what, we want to be able to have our keyboard input detected!

Speaking of POSIX real-time priorities, they are next on our list.

POSIX Real-Time Scheduler

The last of the three main schedulers—and the newest—is the POSIX real-time scheduler, Figure 5-15 illustrates its position in the priority continuum. This scheduler features three different policies: **`SCHED_FIFO`**, **`SCHED_RR`**, and **`SCHED_RR2`**.

Figure 5-15. SCHED_RTPRIO



When the POSIX committee first approached the creation of a UNIX real-time scheduler specification, it requested several key features:

- The POSIX scheduler would feature up to 512 single-priority run queues.
- The number of queues would be configurable between 32 and 512.
- The weakest priority would be 1 and the strongest 512.
- Each thread assigned to a run queue would have a choice of three scheduling policies among other threads with which it shared a queue: FIFO, RR, and RR2.
- The POSIX run queues would be systemwide and not tied to a specific processor. When a POSIX queue had an active, runnable thread and it was the strongest queue, it would schedule the thread to run on the first available processor (unless the thread itself had expressed a specific processor affinity).

The implementation of these design goals presented its own set of challenges, as we previously mentioned. To understand how these features have been implemented in the HP-UX kernel, let's break them down.

Sizing the `POSIX` Queues

The sizing of the POSIX queue is controlled by a tunable kernel parameter, `rtsched_numpri`. The minimum supported value is `RTSCHED_NUMPRI_FLOOR`, or 32, and the maximum is `RTSCHED_NUMPRI_CEILING`, or 512 . Most systems simply leave this parameter at its default value of 32. Once this count is set, the kernel builds the POSIX run queues. The POSIX run queue is considered to be global, but it should be noted that on large systems with multiple processor sets (`pset`s), there is a unique POSIX run queue created for each `pset`.

`SCHED_FIFO`

This policy schedules threads in a queue strictly according to time, when they were placed into the queue. First-in, first-out is the name of the game here. Once a thread gets the CPU, it keeps it until it requests a system call resulting in its blocking (being placed or a sleep queue) or until a thread of stronger priority preempts it. When a running thread is preempted, it is placed at the head of its queue. If it blocks on a

system call upon wakeup, it is placed at the end of its queue.

SCHED_RR

The **SCHED_RR**, or round-robin queuing policy, uses the system's **timeslice** quantum and moves a thread to the end of its queue when its **timeslice** expires. If a thread operating under this policy is preempted, it resumes operation at the head of its queue.

SCHED_RR2

This policy represents a round-robin scheduler with the difference that the timeslice may be adjusted differently for each queue—at least that was the intention. Current implementation allows this policy to be selected, but there is no kernel code to implement the per-queue timeslice quantum value; it simply uses the system timeslice quantum value. In effect, SCHED-RR and SCHED_RR2 both function in an identical manner.

Some general housekeeping items need to be mentioned at this time. With the introduction of the POSIX real-time extension came the rtsched() system call and the command-line rtsched. This call, just like the rtprio() call, requires privileged group capabilities in order to be used (see set_priv_grp and get_priv_grp commands in the manual pages). To make things a bit easier, the rtsched call may be passed any of the valid scheduling policy names as an argument, and it will implement the requested policy.

The timeslice value kernel parameter may be set to 0, in which case the default value of 10 ticks is used, or the desired number of ticks may be set (use caution: in most cases this parameter should not be set lower than 5 or greater than 30—these are both extreme cases). Another interesting option is to set timeslice to −1; this will effectively turn round-robin scheduling off for the system. Again, we don't recommend this for most configurations.

Run Queues

We have used the term *run queue* frequently in this chapter. Let's take a moment to understand what a run queue is, where it is located, and how it is implemented.

The run queue does not exist as a single structure. It is built by linking together all runnable threads on the system to appropriate queue headers. Which queue header a runnable `kthread` is linked to depends on its processor affinity, its priority, `kt_pri` and its scheduling policy stored in `kt_schedpolicy` (stored in the thread's `kthread` structure). To enter a processor's run queue, we start with the `mp_threadhd` structures.

The structures are stored in a number of arrays (dependent on the number of processors and processor sets). Each of the per-processor run queues has 160 `mp_threadhd_t` elements, while the global POSIX run queue size is controlled by the kernel-tunable `rtsched_numpri` (default is 32). In either case each run queue header contains two words `th_link` and `tn_rlink`. The `th_link` points to the first thread in this specific run queue, and the `th_rlink` points to the last thread currently in the queue. Maintaining a dual linkage facilitates the linking of a `kthread` to either the front or back of its perspective queue. Threads are normally added to the back of a queue except in the case of preemption.

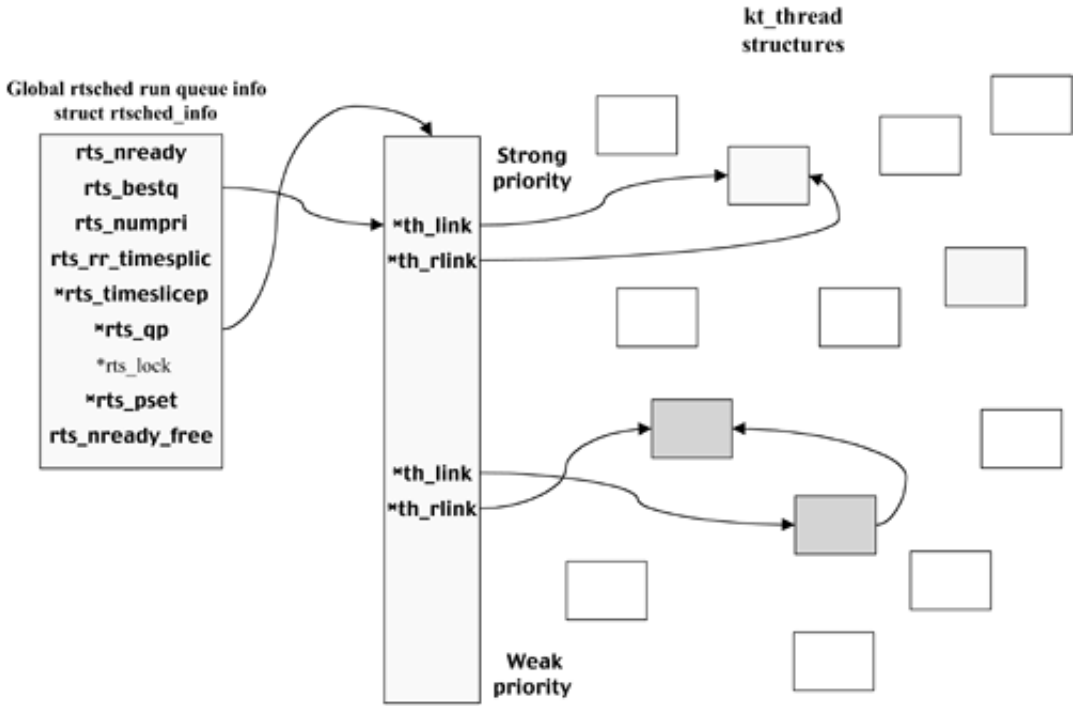
The bulk of the run queue consists of the `kt_link` and `kt_rlink` pointers maintained in each individual `kthread` structure As an individual thread's state changes from run-able to blocked (sleeping), these linkage pointers are updated by the kernel as part of the overall scheduling process. These same pointers serve a dual purpose: they link a `kthread` to other `kthread`s in the same run queue or to other `kthread`s that are blocking on the same kernel service or I/O routine. `kthread`s blocking on the same kernel routine are said to be on a sleep queue. The `kthread`'s present state (`kt_stat`) notes whether it is on a run queue or a sleep queue.

Next, we explore the two types of run queues: global and per-processor.

The Systemwide POSIX Run Queues

As we have stated, the POSIX run queue is a global entity (one per `pset`). Each processor starts its search for runnable threads by first checking the POSIX run queue to see if it has any waiting threads. If not, it checks its own per-processor run queue. Figure 5-16 shows the basic principals of the POSIX real-time run queue.

Figure 5-16. `POSIX` Global Real-Time Run Queue



The beginning of the POSIX run queue from the kernel's point of view is a data structure named `rtsched_info`:

Listing 5.7. `q4 fields structure mp_threadhd`

```
This is the total number of runnable threads waiting on all
queue's headers in this POSIX queue
  0 0 4 0 int rts_nready
This index points into the qp[] header array and directs the
kernel to the best queue to search for a runnable ktrhead
  4 0 4 0 int rts_bestq
This is where the tunable number of POSIX queues is stored
  8 0 4 0 int rts_numpri
Timeslice value for the SCHED_RR policy (set to system
timeslice value)
  12 0 4 0 int rts_rr_timeslice
Per priority timeslice for SCHED_RR2 policy (currently also
set to system timeslice value)
  16 0 4 0 *   rts_timeslicep
Pointer to the mp_threadhd array structure (the head of the
header array)
```

```
  20 0 4 0 *   rts_qp
A pointer to the spinlock used to make this queue mp-safe
  24 0 4 0 *   rts_lock
Points to the pset to which this run queue is attached
  28 0 4 0 *   rts_pset
A count of the number of waiting POSIX threads that have not
specified an MPC_MANDATORY binding preference
  32 0 4 0 int rts_nready_free
```

To examine the POSIX rtsched run queues using q4, launch a q4 session against the running kernel:

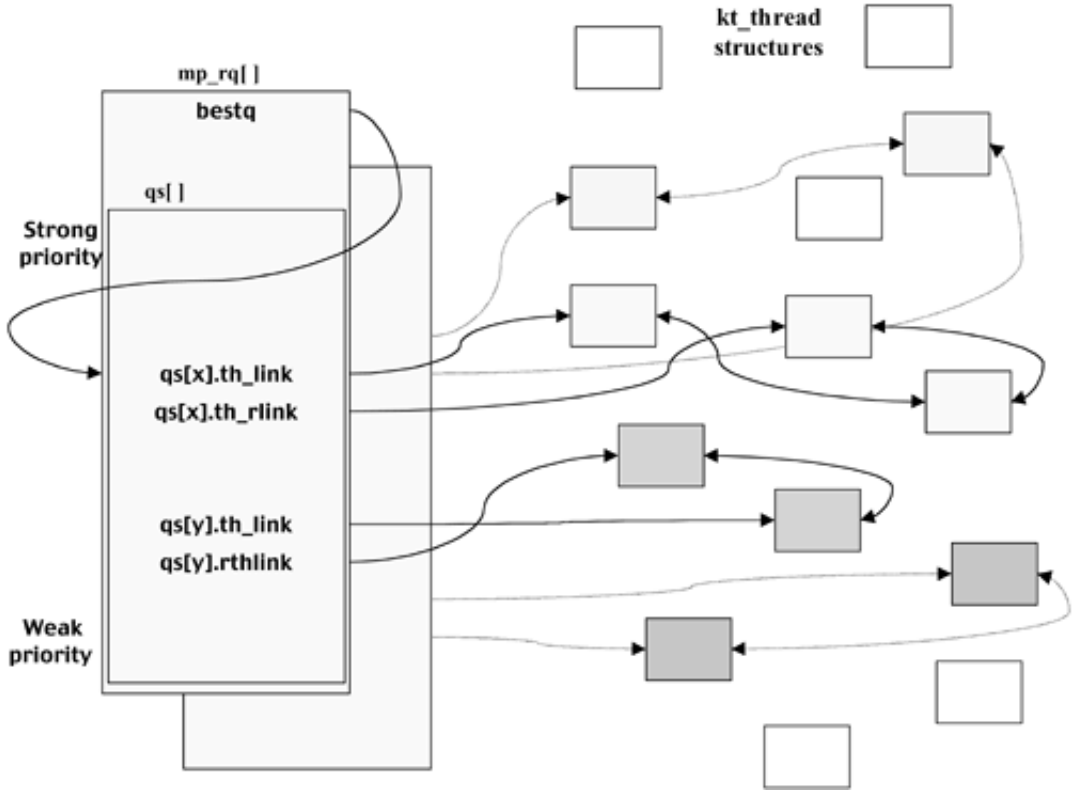
```
# load all of the POSIX run queues
    q4> load struct rtsched_info from &rtsched_info
    q4> load struct mp_threadhd from rts_qp max rts_numpri
    q4> print –x addrof th_link th_rlink
# next make a pile of all the kthreads on nonempty run queues
# (hint: empty queues point to themselves.)
    q4> keep th_thlink != addrof
    q4> load dthread_t from th_link next kt_link until addrof max 100
# to see much you should try this on a multiprocessor system with running
# POSIX real-time processes
```

Next, we will take a look at the per-processor run queues.

Per-Processor Run Queues

As with the POSIX queues, individual `kthread`s are linked together and to an appropriate `mp_threadhd` header structure. The headers are contained in an array within the `mpinfo` structure. Figure 5-17 shows the layout of the per-processor run queues.

Figure 5-17. Per-Processor Run Queues



For our purpose here, we are only interested in a small portion of the overall structure, that which deals with the run queue, the structure `mp_rq`.

Listing 5.8. `q4 fields structure mp_rp`

```
This index points into the qs[] header array and directs the
kernel to the best queue to search for a runnable kthread
   0 0 4 0 int    bestq
This is effectively the run queue average for the processor
   4 0 4 0 int    neavg_on_rq
Number of waiting run-able kthreads not locked to any spu
   8 0 4 0 int   nready_free
alpha semaphore-related information
  12 0 4 0 int    nready_free_alpha
Number of waiting threads locked to this spu
  16 0 4 0 int    nready_locked
alpha semaphore-related information used for mp synchronization
  20 0 4 0 int    nready_locked_alpha
  24 0 4 0 int    asema_ignored
```

```
Thread migration information used by the mp load balancer
  28 0 4 0 u_int  ticks_last_migration
  32 0 4 0 u_long cycles_last_migration
Pointer to next horse in the FSS carousel (for use by PRM)
  42 0 4 0 *      nexthorse
Spinlock for this processor's run queue
  40 0 4 0 *      run_queue_lock
  44 0 4 0 *      qs[0].th_link
  48 0 4 0 *      qs[0].th_rlink
    -  -  -  -  -  -  -  -  -  -  -
1316 0 4 0 *      qs[159].th_link
1320 0 4 0 *      qs[159].th_rlink
```

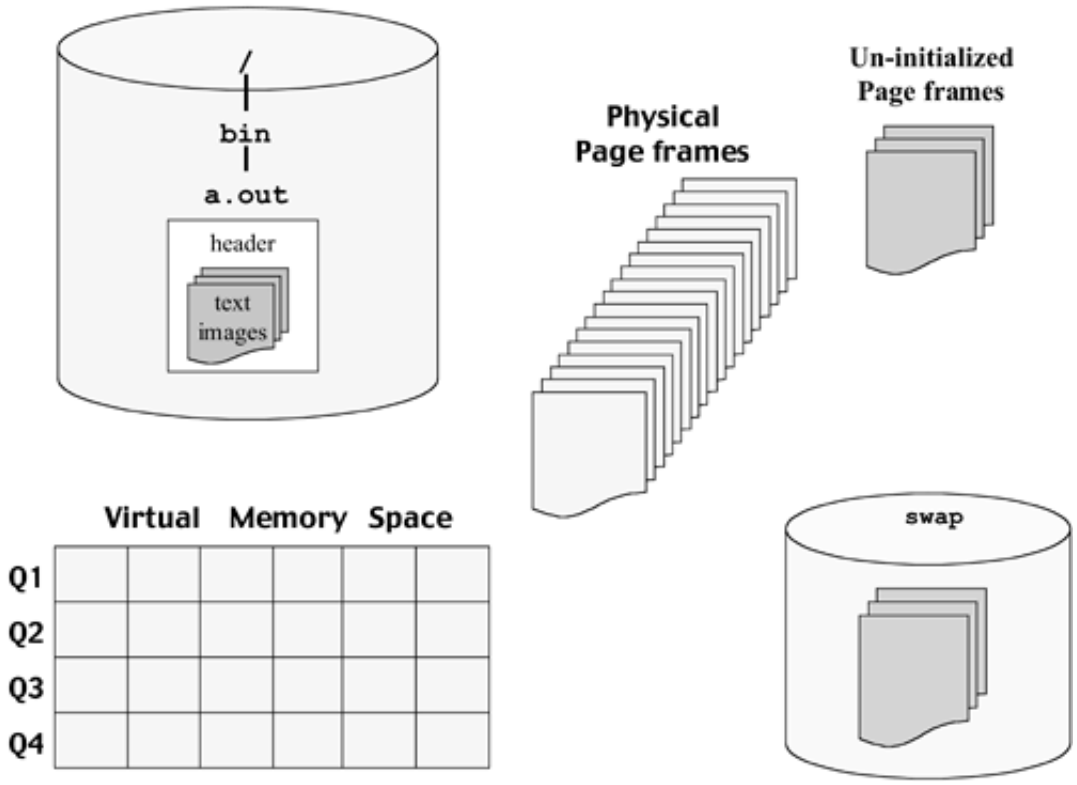
Summary

Now that we have a fair knowledge of the structures and tables that provide the process its logical address space view, we will explore the way the kernel views and manages these memory resources. In Chapter 6, we start at the bottom with a single page frame and build the management structures required to implement entire regions of page frames; then we can join the two perspectives and start to appreciate the intricacies of the overall scheme.

Chapter 6. Managing Memory

In this chapter,we examine HP-UX memory management by looking at systemwide memory resources. Figure 6-1 shows the various forms of memory the kernel must manage. We examine how the per-process memory areas are allocated from systemwide memory and how the kernel's virtual memory is mapped to physical memory. We also look at how free memory and swap space are allocated and managed.

Figure 6-1. Types of Memory



Now we can begin to see the challenge faced by the kernel architects. Memory management is not just a simple case of "use a page, free a page."

Types of Memory

Various levels of abstraction have been introduced to facilitate the use of relocatable code, shared memory objects, and private data areas. To support these abstraction layers, memory must be managed from the logical, virtual, and physical page levels. In addition, memory page images may be stored on a disk device as part of a "front store" or in a "back store" when the kernel needs to temporarily move them out of the way.

Logical Memory

As we discussed in Chapter 5, "Process and Thread Management from the Process's Viewpoint," a process takes a "logical" view of memory. As far as the process is concerned, it is only interested in its own memory requirements: Where is my code? Where is my data? Where are the shared library routines I requested?

The process doesn't really know whether memory objects are shared or private. That is not to say that the programmer doesn't need to know the difference, but from the individual process's point of view as long as the next instruction is available when it is fetched, its data is present and accounted for, and its system calls are fulfilled, then all is right with the world. The kernel must be much more pragmatic and play the part of the "man behind the curtain," pulling the levers and turning the cranks to keep up the appearance that the process is the end-all and be-all for when its threads are selected to take their turn at running.

The **proc** table's **vas** and **pregion** list provides the mapping of the process perspective, but the hardware has no understanding of this point of view. The processor only understands virtual and physical memory addressing. As we learned in our discussion of the PA-RISC processor, the hardware is specifically designed to allow virtual addressing (remember the **TLB** and the **PDIR**). The hardware view of virtual memory must be supported by the kernel if it is to be allowed the privilege of scheduling various threads for execution. Let's take a look at the kernel's view of virtual memory.

Virtual Memory

Of all the memory perspectives, this one encompasses the largest scope. When the thing you are managing is "make believe," size is not a limiting factor. This is not to say that virtual memory is unlimited; after all, we need to establish an addressing scheme and it must fit within the capabilities of the underlying architecture.

During our discussion of the HP PA-RISC architecture, we learned that the range of the virtual address space (VAS) depends upon the specific processor we are using. In the case of narrow 32-bit processors, the theoretical virtual address range is 2^32 spaces, each containing 2^32 bytes. In actual practice, most narrow PA-RISC processors implement 2^16 spaces (65,534), each containing 2^32 bytes (4 GB).

For wide 64-bit processors, the scale is much larger. Each space consists of 2^64 bytes (16 EB), and there could be 2^64 spaces. As is the case with the narrow mode, the current processors do not implement the entire range of virtual addresses available. Each space is currently limited to 2^42 bytes (16 TB), and there are 2^22 (4 million) spaces—this still provides a very large virtual memory perspective.

Remember that each "space" contains four quadrants of equal size. On narrow systems, each quadrant is 1 GB in size, while on wide systems each quadrant is 4 TB in size. This configuration dictates that the two most significant bits of an offset address be used to determine which quadrant we are referencing. These are sometimes called the *space register selection* bits.

It is the kernel's job to manage and map this VAS, providing both private and shared regions of virtual memory. Pointers in a process **pregion** connect the process to kernel **regions**; this is where the two perspectives meet.

Physical Memory

Physical memory is one of the system's most precious commodities, and the kernel devotes much of its time and effort to the efficient management of this resource.

Prior to the introduction of the V-Class family of computers (and the Super-Dome, which closely followed), all the physical memory on an HP-UX system was managed by a single memory controller and was mapped to a contiguous range of physical addresses. This early model allowed for a fairly simple data structure to track the utilization of the physical pages.

With the release of HP-UX 11.0, this simple model became more complex, as the newer system configurations demanded ways to track noncontiguous blocks of physical memory while also allowing a great deal more physical memory to be configured for the system. On a narrow system, the maximum physical memory was limited to 4 GB, while the newer wide systems could map 512 GB (most models cannot physically hold this amount of RAM, at least not yet).

> **NOTE**
>
> The current wide system size limitation is due to constraints imposed by some of the kernel data structures and could be expanded greatly with the re-sizing of these structures. Try to spot them as we continue our discussion in this chapter.

To facilitate these growing requirements, the physical memory management structures have been partitioned to increase their scope and flexibility.

Front Store and Back Store

These two terms refer to carbon copies of a physical memory page frame stored as part of a program file, the ***front store***, or temporarily placed on a swap device, the ***back store***.

The front store is commonly known as an executable file, the product of a compiler and a linking-loader (generically a file named *a.out* by default). An executable program file also contains reference in its header to other types of memory which may be require to run, such as initialized data, uninitialized data and shared library routines. Initialized data pages are loaded from a copy in the front store while uninitialized pages must be produced out of thin air by the kernel as they are. In reality, the kernel isn't a magician and simply finds an unused physical page to fill with "0's" when one is requested.

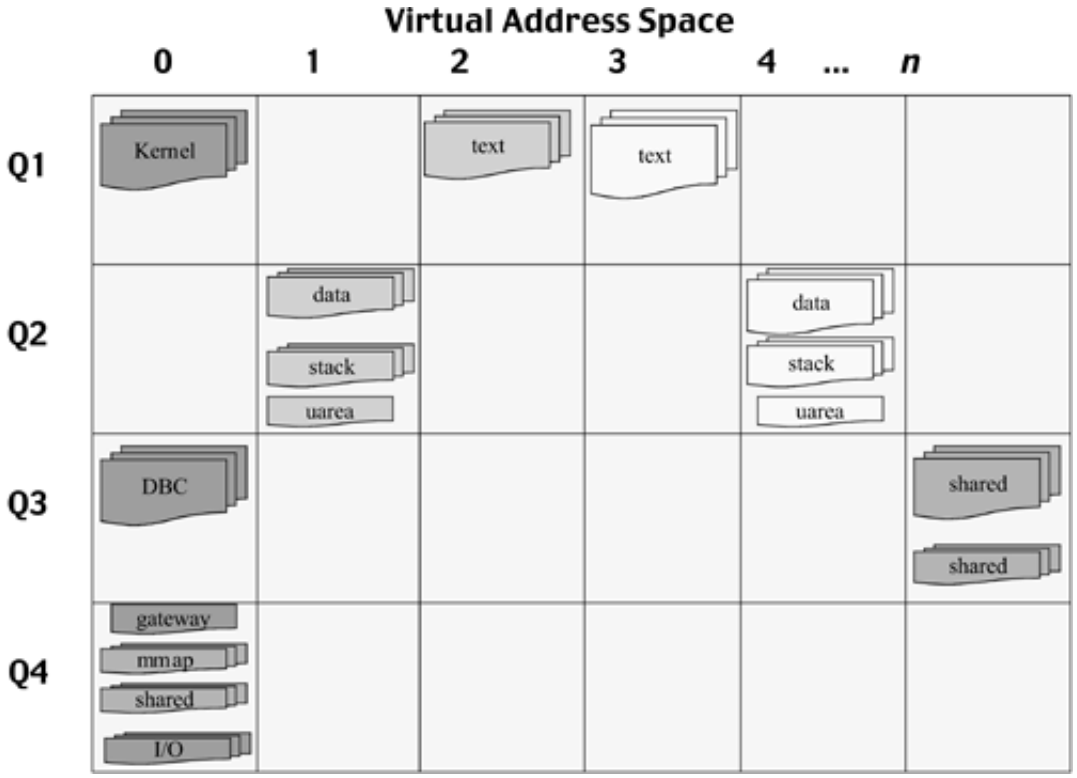
When system memory pressure is high (the number of unallocated or free memory pages is relatively low), the kernel employs a paging system to free up space. Allocated pages that are not in frequent use are identified and copied to a back-store image on available swap space. The kernel's paging system is responsible for the reservation and allocation of the back-store pages. We discuss paging in detail later, but next we will look at the specific kernel structures created to manage each memory perspective.

The System's Virtual Address Space

The VAS is an organizational concept that allows the implementation of shared and private memory objects through a single level of address abstraction. This approach is part of the PA-RISC processor design and is supported in the HP-UX kernel code as part of its hardware-dependent layer. As we learned in our discussion of PA-RISC computing platforms, the VAS is conceptually arranged as a large grid, and each column represents a unique virtual space containing four quadrants (represented by four rows).

In Figure 6-2, we see the practical application of this VAS model. As a process uses its **vas** and **pregions** to map various memory objects to specific offsets within its logical memory view, these objects are mapped to corresponding regions of virtual pages in the system VAS. Physical pages are in turn mapped to specific virtual pages by hardware functionality (**pdir** and **tlb**) supported by kernel fault handlers and resource managers.

Figure 6-2. Populating the VAS with Regions



During execution, the PA-RISC hardware utilizes the translation lookaside buffer (**tlb**) and the page directory (**pdir**) to convert implicit virtual addresses to physical addresses. When a requested virtual address is not resident, or *in-core*, the hardware suffers a *page fault*. It is at this point that the kernel fault-handling code must come into play, find (or create) the missing page image, load it in an available physical page frame, update associated hardware-dependent layer data structures, update kernel hardware-independent layer data structures, and then return from the fault handler so that the hardware may request the address again.

The kernel must keep track of which regions occupy which quadrants in which space. This is accomplished

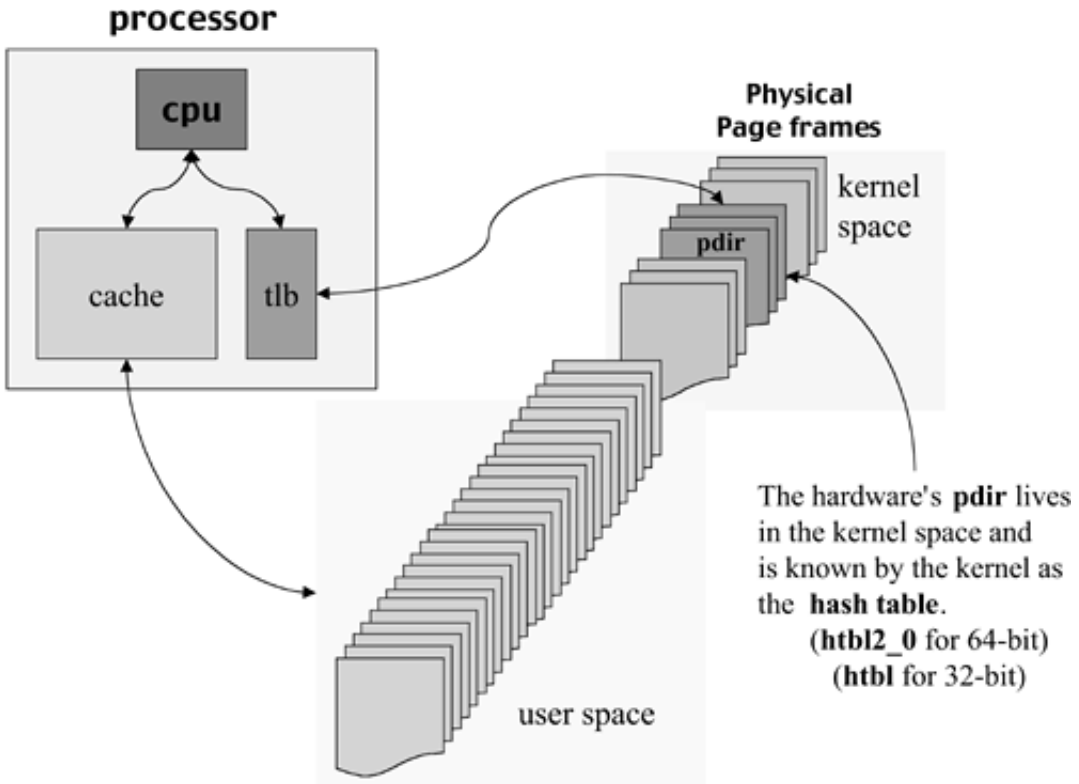
by a host of bit maps and resource maps maintained by the kernel. A key feature of this approach has been that even though the scope of virtual addressing increased many orders of magnitude when the underlying hardware grew from 32 bit to a 64 bit, the VAS concept has changed very little. True, the quadrants are much larger and there are many more potential spaces, but the concept remains consistent. The core issue for both the processor hardware and the kernel is the translation of virtual-to-physical translation and, to a lesser degree, physical-to-virtual mapping.

PREV < Day Day Up > NEXT

Virtual-to-Physical Page Tables

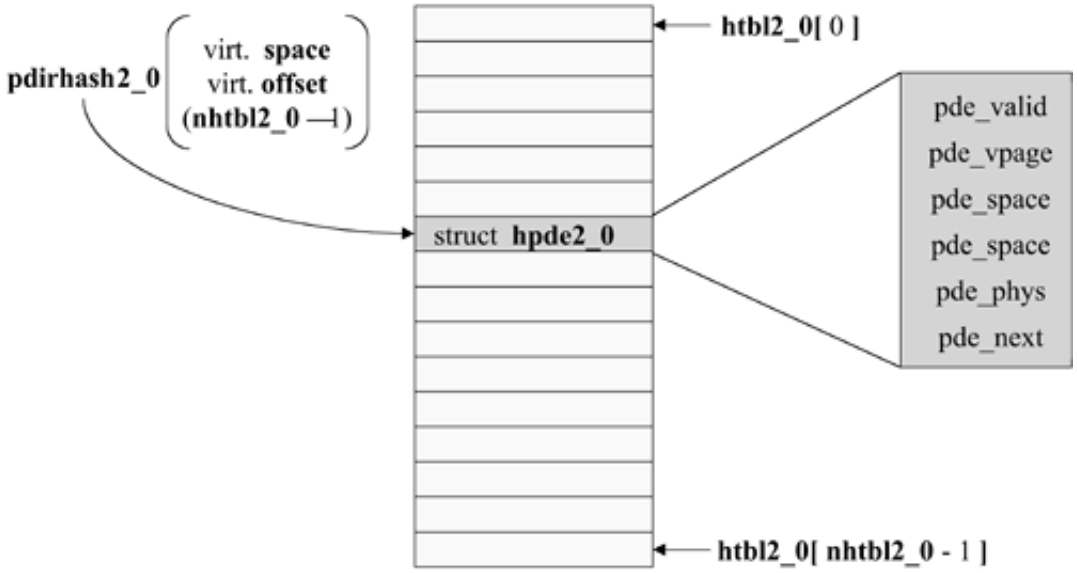
While it is true that virtual-to-physical translation takes place in the processor hardware, the table that supports it resides in the kernel's memory space, and if the kernel wishes to move processes in and out of play, it must adjust entries in this table accordingly. Figure 6-3 demonstrates how the hardware's page directory, or `pdir`, is also known to the kernel as the hashtable.

Figure 6-3. How the Kernel Views the `pdir`



The processor populates the hashtable **(htbl/`pdir`)** with page data entries (`pde`), which contain a variety of status bits, the current virtual address, and the current physical address of a page (see Figure 6-4). The kernel references and maintains this structure as a table of hashed page data entries (`hpde` on narrow systems and `hpde2_0` on wide systems).

Figure 6-4. The Hashtable



As the virtual map is huge in comparison to the physical map, it would be impractical to size it large enough for each virtual page to have its own **pde**. That's one of the reasons we utilize a hashtable. The trick then is to size the table large enough to minimize the occurrence of two virtual addresses hashing to the same entry yet small enough to avoid wasting physical memory.

The basic approach is to size the hashtable in relationship to the number of translations the system is expected to manage at any one time. The simple answer would be to set it to the number of physical pages configured into the system, but this doesn't factor in all the addresses for which translations may be needed. Don't forget that PA-RISC processors use a memory-mapped I/O approach; these pseudo-memory pages may also require the creation of virtual translations. The size is determined in the following way:

- The total number of pages in physical memory is stored in the kernel parameter **npdir**.
- Each I/O module (buss controllers, graphics devices, …) is polled to determine its mapped memory requirements, and this value is stored in the kernel parameter **niopdir**.
- These two parameters are totaled and stored in the parameter **nhtbl**.
- We aren't quite done yet: in order to simplify the hashtable index masking, **nhtbl** is adjusted to a power of two. When the calculated `nhtbl` is an exact power of two then that value is used. In most cases the calculated value falls somewhere between two powers of two; if it less than 25% greater than the difference between them, the lower power of two value will be used. If is greater than 25% of the differential, then the larger power of two value will be used.

Note the annotated study of the **hpde** and **hpde2_0** kernel structures in Listings 6.1 and 6.2.

Listing 6.1. `q4> fields struct hpde`

```
The first word of the structure contains a valid bit, the 15

highest bits of the virtual page address, and 16 bits for the

virtual space number

 0 0 0 1 u_int pde_valid

 0 1 1 7 u_int pde_vpage
```

2 0 2 0 u_int pde_space

The second word contains status, protection and access bits.

The reference, accessed, reference trap, and executed bits are

used to indicate that the page is "in play" and to facilitate

a "stingy cache" algorithm

4 0 0 1 u_int pde_ref

4 1 0 1 u_int pde_accessed

4 2 0 1 u_int pde_rtrap

The dirty bit is set to indicate that the page's content in

cache has been modified and is not in sync with its memory-

resident copy.

4 3 0 1 u_int pde_dirty

4 4 0 1 u_int pde_dbrk

The access rights consist of a 7-bit field and have the

following translation:

0x00 PDE_AR_KR kernel read-only

0x10 PDE_AR_KRW kernel read-write

0x20 PDE_AR_KRX, PDE_AR_KXR kernel read-execute, kernel read-only

0x30 PDE_AR_KRWX, PDE_AR_KXRW kernel read-write-execute

0x0f PDE_AR_UR user read-only

0x7f PDE_AR_UX user execute-only

0x1f PDE_AR_URW user read-write

0x2f PDE_AR_URX, PDE_AR_URXKR, PDE_AR_CWX user read-execute,

kernel read-only, and copy-on-write

0x3f PDE_AR_URWX user read-write-execute

0x4c PDE_AR_GATE promote on execute-only (**gateway** page)

0x73 PDE_AR_NOACC no access allowed

4 5 0 7 u_int pde_ar

Next we can mark uncacheable pages

5 4 0 1 u_int pde_uncache

The protection ID is a pseudorandom number between 0 and

32767. A bitmap is maintained to keep track of which values

are in use (**protid_map**). If a unique protection ID cannot be

found, the kernel will panic!

5 5 2 2 u_int pde_protid

The execute bit is part of the stingy cache flush algorithm

7 7 0 1 u_int pde_executed

```
The first bit of the third word is used to indicate when an
update to the pde data is in progress (this alleviates the
necessity of using a spin-lock during hashtable updates)while
the physical page number occupies the next 20 bits
 8 0 0 1 u_int pde_uip
 8 7 2 4 u_int pde_phys
The modified bit is used to let the kernel know the page's
contents have been modified since the last time it was written
to the back store (swap spaced)
11 4 0 1 u_int pde_modified
The trickle and block bits facilitate operations on specific
PA-RISC processors with hardware walkers and block TLB entries
(only available on a limited number of older 32-bit
processors)
11 5 0 1 u_int pde_ref_trickle
11 6 0 1 u_int pde_block_mapped
If the alias bit is set, this structure was created as a
result of a virtual page aliasing
11 7 0 1 u_int pde_alias
The last word of the structure is a forward pointer to a
sparse hpde entry if one is required.
12 0 4 0 *     pde_next
```

Listing 6.2. `q4> fields struct hpde2_0`

```
On wide hardware the hpde2_0 structure consists of four double
words. Again, the first double word starts off with a "valid"
bit (actually an "invalid" bit) followed by the high 20 bits
of the virtual page and 32 bits containing the virtual space
number
 0 0 0 1 u_int pde_invalid
 1 4 2 4 u_int pde_vpage
 4 0 4 0 u_int pde_space
The second double word contains various status bits, the
access rights, and protection ID key
 8 2 0 1 u_int pde_rtrap
```

```
 8 3 0 1 u_int pde_dirty
 8 4 0 1 u_int pde_dbrk
 8 5 0 7 u_int pde_ar
 9 4 0 1 u_int pde_uncache
The PA-RISC 2.0 processor allows instruction ordering and
branch prediction hints to be assigned to a page
 9 5 0 1 u_int pde_order
 9 6 0 1 u_int pde_br_predict
10 2 0 1 u_int pde_ref_trickle
10 3 0 1 u_int pde_block_mapped
10 4 0 1 u_int pde_executed
10 5 0 1 u_int pde_ref
10 6 0 1 u_int pde_accessed
10 7 0 1 u_int pde_modified
11 7 0 1 u_int pde_uip
12 0 3 7 u_int pde_protid
15 7 0 1 u_int pde_os
The third double word begins with the virtual alias flag, has
52 bits for the physical page number (the first 25 are
currently not used and labeled pde_phys_u, for "unused")
16 0 0 1 u_int pde_alias
16 7 3 1 u_int pde_phys_u
20 0 3 3 u_int pde_phys
The next 5 bits are used in conjunction with performance-
optimized page sizing
23 3 0 5 u_int var_page
The fourth double word contains the pointer to the next sparse
entry.
24 0 4 0 u_int unused_upper
28 0 4 0 u_int pde_next
```

Let's check to see how observant you were of the annotations. Did you notice the size (in bits) of the virtual page number stored in the **hpde**? It is only 15 bits wide, which presents a challenge, since there are 2^20 virtual pages per space. The bits are noted to be the high-order 15 bits of the virtual page number. With this model, it seems that only one out of every 32 pages will be translatable by this table! There must be more to this story than meets the eye—and there is. To understand the secret of the missing five virtual page address bits, we need to examine the hashing algorithm being used by the hardware (and understood by the kernel).

The Kernel View: The Hashtable

The name **pdir** is a bit of a throwback. Originally, HP-UX used an inverse page directory that mapped physical page numbers to virtual page numbers. This meant that while the size of the table was easy to determine (simply one entry for each page), searching for a specific virtual address was a lengthy process. To reduce search times, a hashed table of virtual addresses was created. In this format, a virtual page address is used to create a *hashtable index* value. The index is then tested against a mask value to make sure the index doesn't exceed the size of the hashtable, and the result is used as an offset into the **htbl** (or **htbl2_0**).

The hashing algorithm varies according to the hardware width. For a narrow system, the routine is as follows:

- **pdirhash1_1(space,offset) & (mask)**
- We left-shift the space value by 5 bits.
- The offset is right-shifted by 12 bits (this is the byte offset into a page and as such is not part of the virtual page number).
- These two values are exclusively ORed to create the initial hashtable index.
- To assure that the index does not exceed the size of the hashtable, an AND operation is performed between the initial index value and the mask (the size of the table, a power of two).

This is where we answer the question asked previously. By shifting the space value by 5 bits to the left, the last 5 bits of the virtual page number translate directly to the hashed index. Effectively this means that 32 sequential virtual page numbers are mapped in 32 consecutive `htbl` entries.

For a wide system,

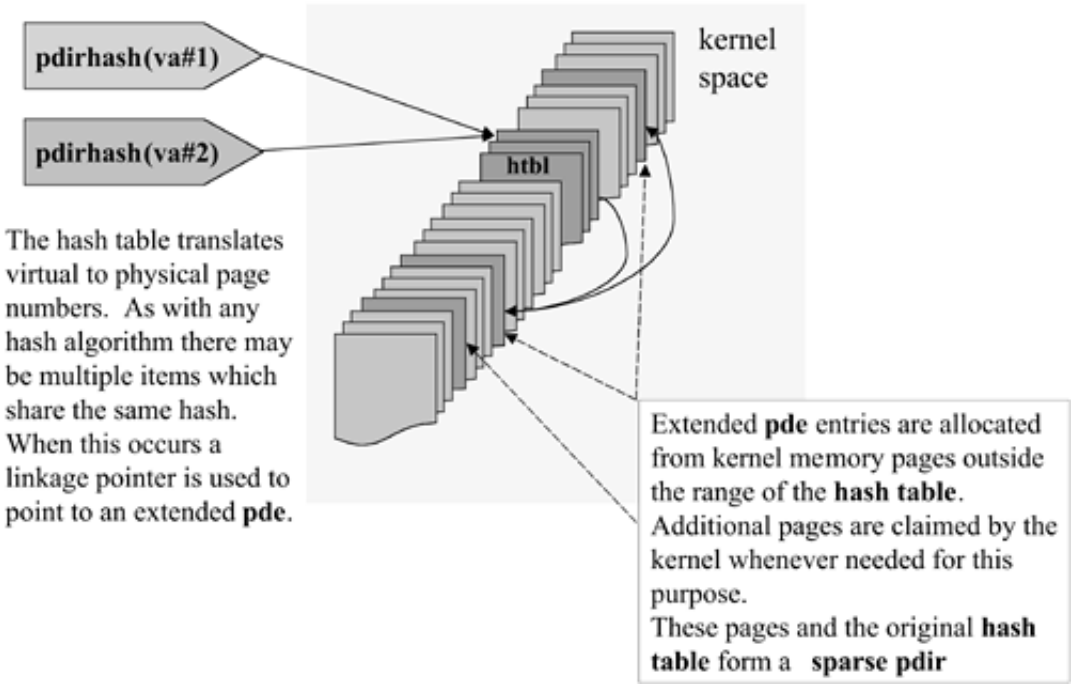
- **pdirhash2_0(space,offset) & (mask)**
- **pdirhash2_0** bit shifts the space value 10 bits to the right (recall from our discussion of the Global Virtual Address (GVA) formation in Chapter 2, "PA-Risc 2.0 Architecture," that the lowest 10 bits of the space number are always set to 0).
- Depending on whether this is a narrow or wide kernel, additional bit operations are performed on the offset value (again from our GVA discussion, there are only 30 actual virtual page address bits implemented by the current processors: 64 – 2 space reg. selection bits – 20 unimplemented bits – 12 page offset bits = 30 bits).
- As with the narrow version, to assure that the index does not exceed the size of the hashtable, an AND operation is performed between the initial index value and the mask.

For the most part, the index calculation involves bit shifts, exclusive ORs, and ANDs. The indexing functions are quick and dirty, but keep in mind that the hardware performs a parallel to these operations: each time an implicit address is requested, the hardware calculates the index as an atomic operation and its value is temporarily stored in CR28. If a **tlb** fault occurs the hardware will use **CR28** as an index into the kernel hashtable in an attempt to find the correct translation. The only reason the kernel needs to be able to duplicate this effort is so that it can move virtual pages in and out of the physical memory map and keep the **htbl/pdir** in sync with the changes it makes.

Creating a Sparse Table You may be wondering what happens if two virtual page numbers hash to the same index, which is a very real possibility. When this happens a forward linkage pointer will direct the fault handler to an extension ***pde***. Extended ***pde***s are allocated from pages of ***pde*** structures managed by the kernel. The hashtable pages and pages allocated for the extended ***pde*** entries are collectively called the sparse *pdir*.

When an extended **hpde** entry needs to be stored, the kernel allocates the space from the head of **pdir_free_list** (see Figure 6-5). In order to maintain a supply of these structures for use when needed, several kernel parameters are utilized. The **pdir_free_list** and **pdir_free_list_tail** point to the front and back of a linked list of available structures. Whenever the number of free entries begins to run low (drops below 256), the kernel allocates an additional page to the effort, carves it up into **hpde**s, and links them to the free list.

Figure 6-5. Sparse Tables



For wide systems, the free lists are maintained on a per-spu (system processor unit) basis and the pointers are **pd_fl2_0[spu].head** and **pd_fl2_0[spu].tail**.

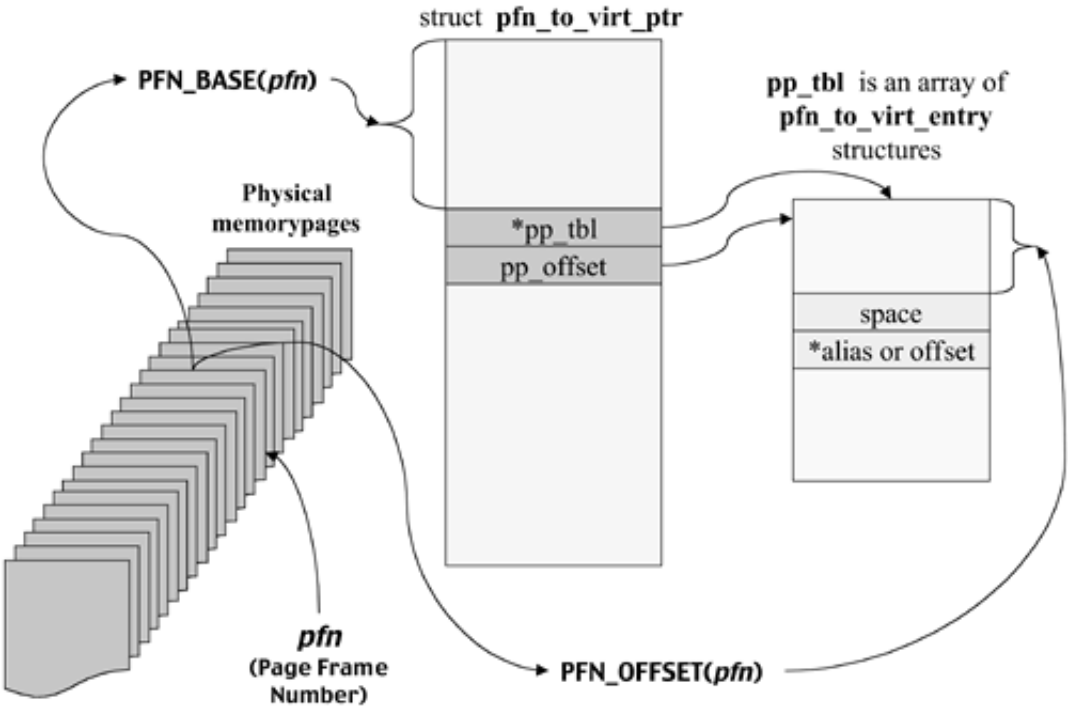
Once memory pages have been allocated by the kernel for sparse structures, they will not be returned to the memory free list. In actual practice, the virtual-to-physical mapping system rarely sees more than three virtual addresses hashed to the same index. The hashed sparse table structure serves our needs very well, but what if you know the physical address and need to determine the virtual address (or addresses) currently assigned to it?

Page Frame to Virtual Page Frame Tables

Let's address a structure that is completely independent of the underlying hardware architecture. The kernel may know the physical page number and need to determine if there are any virtual pages currently mapped to it. This is a scenario that occurs at various points when the kernel is trying to clean up resources following a process termination or panic.

In Figure 6-6, we see the page frame number to virtual address table, or **pfn_to_virt_ptr**, as its current incarnation is named. In early HP-UX systems, this was a very simple table; it had one entry for each physical page configured on the system. The physical page number was the index to get the associated virtual address if there was one. As the complexity of the Hewlett Packard computer hardware platforms grew, so did the complexity of physical-to-virtual mapping. The V-class introduced partitioned memory designs to the HP-UX kernel, and in order to avoid waste in the **pfn_to_virt** table, it was converted to a partitioned design, thus the new name **pfn_to_virt_ptr**.

Figure 6-6. Physical to Virtual Address Translation



The **pfn_to_virt_ptr** structure merely directs us to the appropriate page partition table that contains the actual **pfn_to_virt_entry** structures. The granularity of the partitioning is determined by the kernel parameter **PFN_CONTIGUOUS_PAGES** (currently defaulted to 4096). To find the correct **pfn_to_virt_entry**, we need to divide the physical page number by 4096 to find an index and offset (the remainder). First, we index into **pfn_to_virt_ptr** and find the pointer to the **pp_tbl** structure containing our **pfn_to_virt_entry**. The remainder is our index into this **pp_tbl**. Let's take a look at an annotated listing of these two structures (Listings 6.3 and 6.4).

Listing 6.3. `q4> fields struct pfn_to_virt_ptr`

```
The pp_tbl points to a block of "page frame number to virtual"

entries and the pp_offset directs us to the first valid entry

in the block. Each block maps 4096 physical page translations

(determined by the kernel parameter PFN_CONTIGUOUS_PAGES)

0 0 4 0 *   pp_tbl

4 0 4 0 int pp_offset
```

Listing 6.4. `q4> fields struct pfn_to_virt_entry`

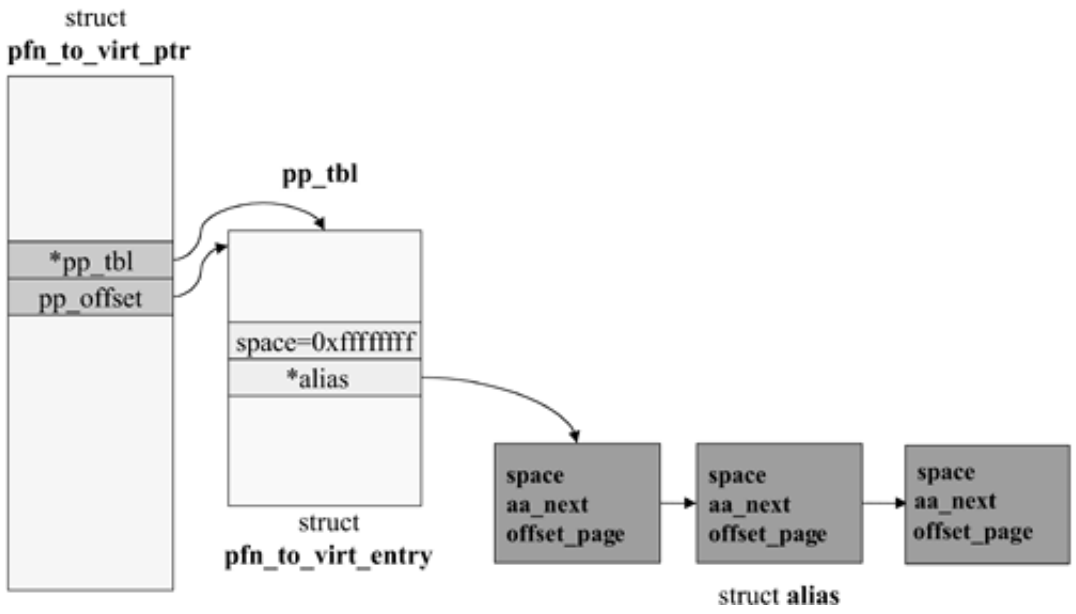
```
The first word of each entry contains either the virtual space
```

```
number or INVALID_SPACE_ENTRY (0xffffffff). The second word is

either the page offset or a pointer to an alias structure

0 0 4 0 u_int  space

4 0 4 0 u_long alias_or_offset.offset_page

4 0 4 0 *      alias_or_offset.alias
```

Virtual Address Aliasing

Virtual addressing aliasing was introduced to the HP-UX kernel with the release of HP-UX 10.0. This feature was incorporated to facilitate a "copy-on-write" scenario in the process creation **`fork()`** system call. Figure 6-7 demonstrates the kernel data structure changes this entailed.

Figure 6-7. Virtual Address Aliasing



In the case of a single virtual page number being assigned to a specific physical page, no **`alias`** structure is required. The physical to virtual translation follows the previously discussed model. If more than one virtual address needs to be associated with a physical page, then its **`pfn_to_virt_entry space`** value will be set to **`INVALID_SPACE_ENTRY`** (0xffffffff), and the **`offset`** value will be a pointer to an **`alias`** data structure. The **`alias`** structure will contain a virtual space and offset and a pointer to the next **`alias`** structure if one exists (Listing 6.5).

Listing 6.5. `q4> fields struct alias`

```
 0 0 4 0 *      aa_next
```

```
 4 0 4 0 u_int  space

 8 0 0 7 u_int  aa_savear

 8 7 3 1 u_int  aa_saveprot

12 0 4 0 u_long offset_page
```

A free list of unused alias structures is maintained by the kernel and pointed to by **aa_entfreelist** (the **alias** structure free list). Another kernel parameter, **min_alias_entries**, sets a low-water mark for free **alias** structures. When the available count, **aa_entcnt**, falls below this limit, the kernel allocates a new page of **alias** structures.

When an aliased virtual address is created, an entry must also be added to the hashtable **(htbl/pdir)**. If the new virtual address maps to a used hashtable entry, a sparse entry must be allocated. Space for one of these alias pdes is allocated from the **aa_pdirfreelist** instead of the system's **pdir_free_list**. The **aa_pdirfreelist** is maintained to avoid overtaxing the **pdir_free_list**. The system has a pointer to both the head and the tail of this free list, **aa_pdirfreelist** and **aa_pdiefreelist_tail** respectively. For a wide system, the pointers are **aa_pdirfreelist2_0** and **a_pdirfreelist_tail2_0**.

As with the alias structures, the kernel monitors their use and availability. The parameter **aa_pdircnt** is monitored; if it falls below **min_alias_pdirs** (256 by default), then a new page of alias PDEs is allocated. The total number of kernel alias PDEs is stored in **max_aapdir**; this value may grow over time but will never shrink.

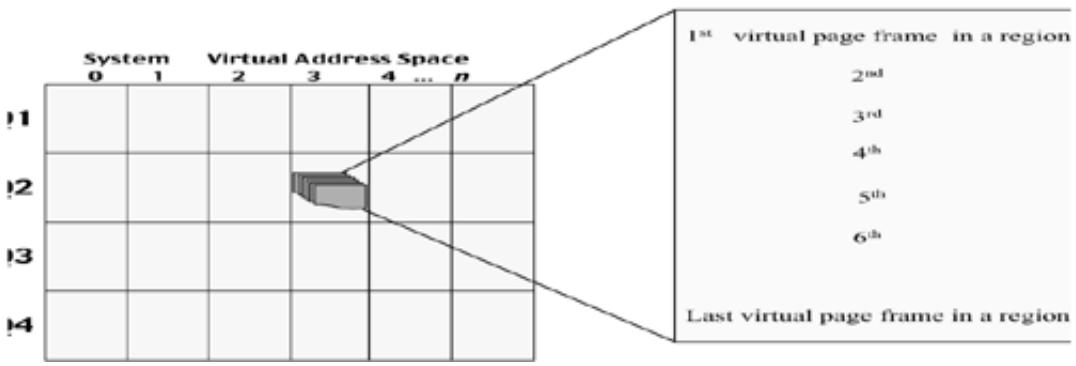
Pages allocated to the kernel for use as alias PDEs or alias structures are never returned to the free page list. They represent a type of high-water mark, and the current thought is that if you need them once, you may need them again. Tying up a few pages is less costly than the overhead of reallocating the pages.

Now that we can convert from virtual to physical and back, let's consider how the kernel manages its many regions of virtual pages.

Regions of Pages

The **region** is the workhorse of the kernel memory management subsystem. It represents the highest level of memory resource management available to a process (via pointers from the process's **pregion** structure). As we see in Figure 6-8, the **region** is simply a collection of consecutive virtual pages. Each region occupies a unique space in the system's VAS. The kernel must make sure the regions are assigned to the correct quadrant type. As far as the **region** structure is concerned, there are only two types of regions: shared and private.

Figure 6-8. Region of Page Frames



The **region** structure contains the organizational pointers and status flags used to manage it. Let's look at an annotated listing of the **region** structure (Listing 6.6).

Listing 6.6. q4> fields struct region

```
The region flags indicate the current region state:
0x00000004 RF_ALLOC This region structure is allocated
0x00000020 RF_UNALIGNED For support of unaligned code pages
0x00000080 RF_WANTLOCK lock requestor waiting
0x00000200 RF_EVERSWP set if the b-tree was ever swapped
0x00000400 RF_NOWSWP set if b-tree is currently swapped
0x00002000 RF_IOMAP region created by iomap() call
0x00004000 RF_LOCAL front-store is NFS but local swap has been
allocated for text
0x00008000 RF_EXCLUSIVE memory mapped MAP_EXCLUSIVE
0x00200000 RF_SUPERPAGE_TEXT variable-page-size in use
0x02000000 RF_PGSIZE_FIXED variable-page-size not in use
0x01000000 RF_MPROTECTED indicates the mprotect() has been
called for this region
  0 0 4 0 enum4   r_flags
A region is either RT_PRIVATE (0x1) or RT_SHARED (0x2)
  4 0 4 0 enum4   r_type
Number of pages in this region
  8 0 4 0 int     r_pgsz
Current number of valid vfd entries in the region's b-tree
 12 0 4 0 int     r_nvalid
Number of valid pages at time of deactivations used by the swapper
 16 0 4 0 int     r_dnvalid
If RF_SWLAZY flag is set (controlled by the chart command)
this is the number pages with an actual swap page allocated.
If lazy swap is not enabled, then this is the total number of
allocated and reserved pages for the region
 20 0 4 0 int     r_swalloc
The number of pages allocated and reserved in pseudo-swap
 24 0 4 0 int     r_swapmem
 28 0 4 0 int     r_vfd_swapmem
Used with the mlock() call
```

```
 32 0 4 0 int     r_lockmem
Forward and backward pointers linking all regions using
pseudo-swap (head of this list is pointed to by the kernel
pointer pswaplist)
 36 0 4 0 *       r_pswapf
 40 0 4 0 *       r_pswapb
A reference count of all pregions pointing to this region
 44 0 2 0 u_short r_refcnt
A region zombie is one whose a.out file is remote (nfs
mounted) and has had its contents modified
 46 0 2 0 short   r_zomb
For page-aligned regions being mapped from a vnode, this is
the offset into the file for the first page of the region.
 48 0 4 0 int     r_off
Number of pregions sharing this region which are currently
"in-core"
 52 0 2 0 u_short r_incore
If a b-tree has been moved to swap, this will point to the
location of the first page in its page list (each subsequent
page's swap location is contained in a pointer at the end of
the previous page)
 56 0 4 0 u_int   r_dbd
 60 0 4 0 int     r_scan
The front and back store values point to the preferred paging
location for this region's pages in the event of high memory
pressure
 64 0 4 0 *       r_fstore
 68 0 4 0 *       r_bstore
All active regions are linked via the next two pointers (the
head of this list is regactive)
 72 0 4 0 *       r_forw
 76 0 4 0 *       r_back
Unaligned regions may be accessed by the TEXTHASH algorithm
(there are 32 hash headers in the kernel structure
texts[TEXTHSHSZ]) and linked by this pointer
 80 0 4 0 *       r_hchain
Next we locate the text image in the front store by its offset
```

```
and length in bytes
 84 0 4 0 u_long  r_byte
 88 0 4 0 u_long  r_bytelen
Additional locking structures and flags come next.
 92 0 4 0 *       r_lock.interlock
 96 0 4 0 u_int   r_lock.delay
100 0 4 0 u_int   r_lock.write_waiters
104 0 4 0 int     r_lock.read_count
108 0 1 0 char    r_lock.want_write
109 0 1 0 char    r_lock.want_upgrade
110 0 1 0 char    r_lock.waiting
111 0 1 0 char    r_lock.rwl_flags
112 0 4 0 *       r_lock.l_kthread
116 0 1 0 u_char  r_mlock.b_lock
118 0 2 0 u_short r_mlock.order
120 0 4 0 *       r_mlock.owner
Number of page I/O's currently in process for this region
124 0 4 0 int     r_poip
Pointer to the root of the b-tree (if a b-tree is being used)
128 0 4 0 *       r_root
The key value may be UNUSED_IDX (0x7fffffff) if we are not
using a b-tree, or DONTUSE_IDX (0x7ffffffe) if we are using
the b-tree pointed to by b_root. Any other value in r_key is a
key associated with r_chunk
132 0 4 0 int     r_key
136 0 4 0 *       r_chunk
All regions associated with a front-store vnode are linked
using the next two pointers
140 0 4 0 *       r_next
144 0 4 0 *       r_prev
148 0 4 0 *       r_preg_un.r_un_pregskl
148 0 4 0 *       r_preg_un.r_un_pregion
152 0 4 0 *       r_psklh.l_header.n_next[0]
156 0 4 0 *       r_psklh.l_header.n_next[1]
160 0 4 0 *       r_psklh.l_header.n_next[2]
164 0 4 0 *       r_psklh.l_header.n_next[3]
168 0 4 0 *       r_psklh.l_header.n_prev
172 0 4 0 u_long  r_psklh.l_header.n_key
```

```
176 0 4 0 u_long  r_psklh.l_header.n_value
180 0 1 0 char    r_psklh.l_header.n_flags
181 0 1 0 char    r_psklh.l_header.n_cookie
184 0 4 0 *       r_psklh.l_tail
188 0 4 0 *       r_psklh.l_cache
192 0 4 0 *       r_psklh.l_cmpf
196 0 4 0 int     r_psklh.l_level
200 0 1 0 char    r_psklh.l_cookie
204 0 4 0 *       r_excproc
208 0 4 0 *       r_lchain
212 0 4 0 int     r_mlockswap
This contains the performance-optimized page sizing hint
216 0 4 0 int     r_pgszhint
The following hardware-dependent layer (structure hdlregion)
contain hints used when attaching the region to a pregion
220 0 4 0 u_int   r_hdl.r_space
224 0 4 0 u_int   r_hdl.r_prot
228 0 4 0 *       r_hdl.r_vaddr
232 0 2 0 u_short r_hdl.r_hdlflags
236 0 4 0 *       r_mrg
240 0 4 0 int     r_mrg_status
244 0 4 0 int     r_dbd_asyncinflight
a pointer to the spinlock used to protect this structure
248 0 4 0 *       r_spinlock
```

As we study the fields of the **region**, you may have noticed that it doesn't include a spot for the virtual address, just the number of page frames it manages. From the kernel's point of view, it isn't really necessary to know the virtual address—only the number of pages in the region and how they are to be used (you may recall that earlier we describe the VAS as a conceptual device for memory management).

We need to examine the **pregion** to **region** linkages in order to explain how regions may be shared.

Shared Objects

To understand the mechanics of region sharing, let's revisit the virtual addressing. The virtual address is really only of concern from the process's point of view. In order for a process thread to run, the kernel must first set up the processor space registers to point to the process's four quadrants. Once the environment has been initialized for a thread, future context switches for the thread will simply save and restore these as part or the thread's process control block (**pcb**) in its **uarea**. The representation of the VAS as a large checkerboard helps us visualize and discuss memory management concepts.

As the process maps its logical `pregions` to the VAS, the process's pregions must be ordered so that no two overlap within the process logical view. A simple allocation map is maintained by the kernel to guarantee that each process gets unique space numbers for its private quadrants.

System-shared quadrants have resource maps to make sure that each shared object mapped to them occupies a unique address range. Resource maps exist for each of the quadrants in the kernel space, and special maps are created for the first shared quadrants 2 and 3 (if memory windows are enabled, additional maps will be created). All other quadrant mappings are controlled by the address assignments made in the process **pregion** chains.

If different processes need to share a **region**, the second one simply duplicates first one's **pregion**, inserts it in its **pregion** list, and increments the **r_refcnt** in the **region** structure.

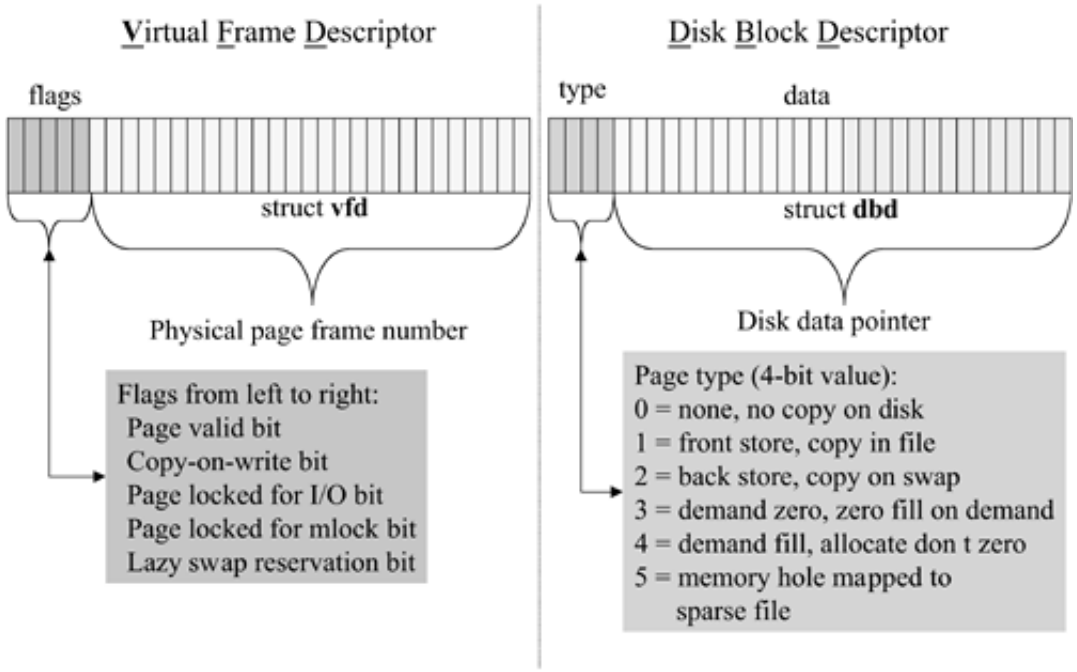
It's easy to get lost in the details of the **region** structure, but we need to remember what the purpose of this structure is: to map a contiguous block of virtual pages to specific physical front-store or back-store locations. To handle this task, the **region** structure is merely a header that leads us to pairs of virtual frame descriptors and disk block descriptors, one for each page frame the **region** manages.

Virtual Page Locator, VFD, and DBD

There are many different answers to the question, Where is my page?

Page faults occur when a virtual page's location cannot be determined by the hardware or the first-level fault handler. The kernel must then find the **region** responsible for the management of the missing page. The **region** structure is simply a collection of per-page objects that define the current state and location of each page frame in the region set. Figure 6-9 demonstrates these per-page structures.

Figure 6-9. VFD|DBD



Each and every page in a **`region`** has a **`vfd|dbd`** structure pair; these define the current status of the page. Let's look at an annotated listing of these structures (Listings 6.7 and 6.8).

Listing 6.7. `q4> fields struct vfd`

```
The first bit is the valid bit; it indicates that the data in
the vfd is valid
0 0 0 1 u_int pg_v
The next bit sets copy-on-write access mode for a page
0 1 0 1 u_int pg_cw
When a page is locked for an I/O operation, this bit is set
0 2 0 1 u_int pg_lock
If an mlock() call has been made for a region, this bit is set
0 3 0 1 u_int pg_mlock
If lazy swap has been activated and a swap page has been
reserved, this bit is set
0 4 0 1 u_int pg_swresv
On narrow systems the physical page frame number may only use
21 bits (due to the memory size limitations of 32-bit
hardware). For a wide kernel, this will be a single 27-bit
field
```

```
0 5 0 6 u_int pg_fill

1 3 2 5 u_int pg_pfn
```

Listing 6.8. `q4> fields struct dbd`

```
The first four bits of this structure denote the page type:

0000 DBD_NONE There is no copy of this page on the disk

0001 DBD_FSTORE A copy of the page is on the front store

0010 DBD_BSTORE A copy of the page is on the back store

0011 DBD_DZERO This will be a "demand zero" page and will be

filled with zeros when it is first requested

0100 DBD_DFILL The page is of type "demand fill." When first

allocated the page will not be initially filled with zeros

(this is only used during the initialization of pages

allocated for use by a UAREA)

0101 DBD_HOLE This page will be used to map a sparse memory-

mapped file. A read request will return a zero, but a write

will cause a fresh page to be allocated and zero-filled

0 0 0 3 u_int dbd_type

The remaining 28 bits contain pointer(s) to the page's image

on either a front store or back store.

0 3 3 5 u_int dbd_data
```

The primary job of the **`region`** is to organize the **`vfd|dbd`** data in an easy-to-search manner. It would be very easy to place the **`vfd|dbd`**'s in an array and use the relative page number as an index into the array. This would be a suitable solution if all regions were of a similar size, but this is not the case.

Previously, we learned that there are several types of process `pregion`s, and in a case of guilt by association, there are several varieties of regions. The size of these regions varies greatly, from a mere 4 to 8 pages for a **`uarea`** to literally millions of pages in a large wide process data region. Another challenge is that some regions may need to grow as the process's threads execute: think of the *`malloc()`* call or a memory-mapped file being written to.

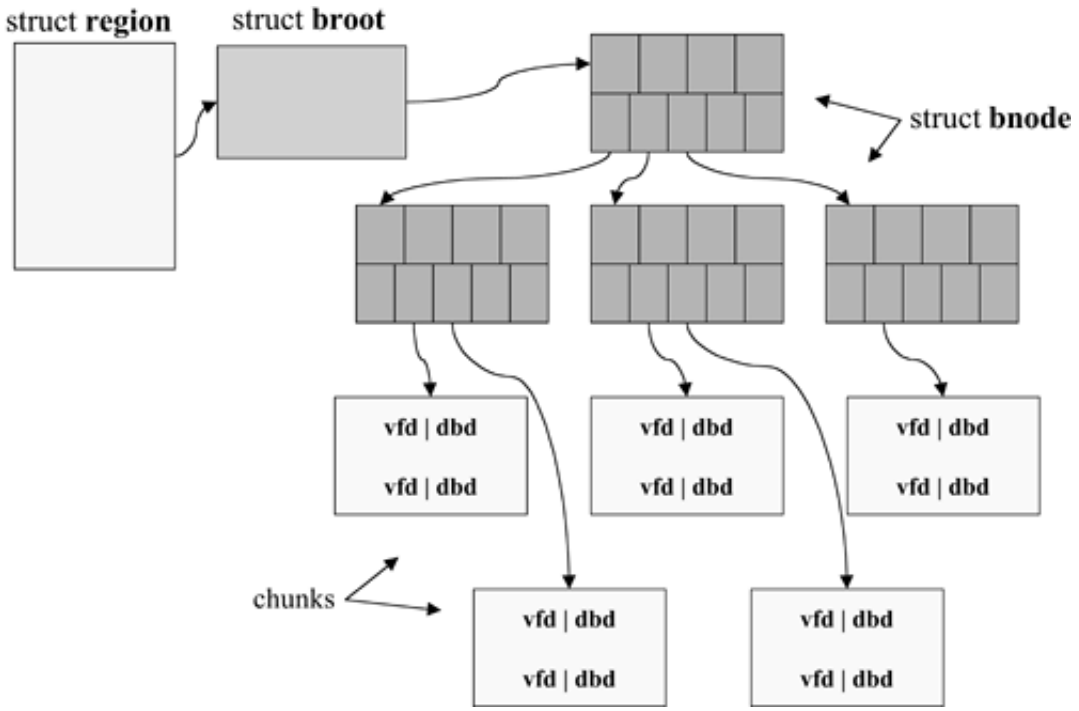
Now we see the challenge: design a data base that may be easily allocated and deallocated from kernel memory space, that handles very small and very large data sets, that may be expanded on the fly, and that provides a low overhead searching algorithm. Easy, right?

The HP-UX kernel employs a design called the `b-tree`. Let's take a look.

The b-tree

We discussed the generic b-tree in Chapter 3, "The Kernel: Basic Organization," and now we need to apply the specifics used to map the **vfd|dbd** data structures. Figure 6-10 shows the overall design of the memory management b-tree and its relationship to a **region** structure.

Figure 6-10. The b-tree



The base of the tree is its **broot**. It sets the rules for the tree's usage and directs us to the first level **bnode** in the tree. This first **bnode** is sometimes called the root **bnode**—try not to confuse the two structure names.

The **bnode** structures are ranked and ordered according to the tree's depth (defined in the **broot**) and form the tree's branches. To continue with the tree model, the **chunk** structures could be thought of as the leaves on the branches. Each chunk contains **vfd|dbd** structure pairs; these **vfd|dbd**s are the fruit of a successful search of the tree.

Broot, Bnodes, and Chunks

Let's examine the b-tree structures in Listings 6.9 and 6.10.

Listing 6.9. q4> fields struct broot

```
We start off with a pointer to the root (first) bnode and
record the depth of the b-tree
 0 0 4 0 *     b_root
 4 0 4 0 int   b_depth
Next we find the number of pages mapped and the number of swap
pages reserved for this tree
 8 0 4 0 int   b_npages
12 0 4 0 int   b_rpages
The page list pointer directs us to this region's page list,
and nfrag points to the next available chunk (required if we
need to grow the tree)
16 0 4 0 *     b_list
20 0 4 0 int   b_nfrag
This pointer gets us back to the region structure this b-tree
serves
24 0 4 0 *     b_rp
28 0 4 0 int   b_protoidx
There are two sets of prototype dbd structures used when the
b-tree is being initialized or grown; the protoidx gives an
index into the tree where we switch from using the first
prototype to the second
32 0 0 3 u_int b_proto1.dbd_type
32 3 3 5 u_int b_proto1.dbd_data
36 0 0 3 u_int b_proto2.dbd_type
36 3 3 5 u_int b_proto2.dbd_data
This array stores information about ranges of pages that need
to have their copy-on-write bit set
40 0 4 0 int   b_vproto.v_start[0]
44 0 4 0 int   b_vproto.v_start[1]
48 0 4 0 int   b_vproto.v_start[2]
```

```
52 0 4 0 int   b_vproto.v_start[3]
56 0 4 0 int   b_vproto.v_start[4]
60 0 4 0 int   b_vproto.v_end[0]
64 0 4 0 int   b_vproto.v_end[1]
68 0 4 0 int   b_vproto.v_end[2]
72 0 4 0 int   b_vproto.v_end[3]
76 0 4 0 int   b_vproto.v_end[4]
To speed up repetitive references, the two most recent key/
value data pairs are cached here
80 0 4 0 int   b_key_cache[0]
84 0 4 0 int   b_key_cache[1]
88 0 4 0 *     b_val_cache[0]
92 0 4 0 *     b_val_cache[1]
```

Listing 6.10. `q4> fields struct bnode`

```
For a 29th order b-tree each bnode contains 30 keys and 31
values. The value array contains either pointers to actual
chunk structures or pointers to the next lower level of the b-
tree structure
  0 0 4 0 int b_key[0]
  4 0 4 0 int b_key[1]
  --------------------
112 0 4 0 int b_key[28]
116 0 4 0 int b_key[29]
number of valid key/values in this bnode
120 0 4 0 int b_nelem
124 0 4 0 *   b_down[0]
128 0 4 0 *   b_down[1]
  --------------------
240 0 4 0 *   b_down[29]
244 0 4 0 *   b_down[30]
pads to align the structure
248 0 4 0 int pad[0]
252 0 4 0 int pad[1]
```

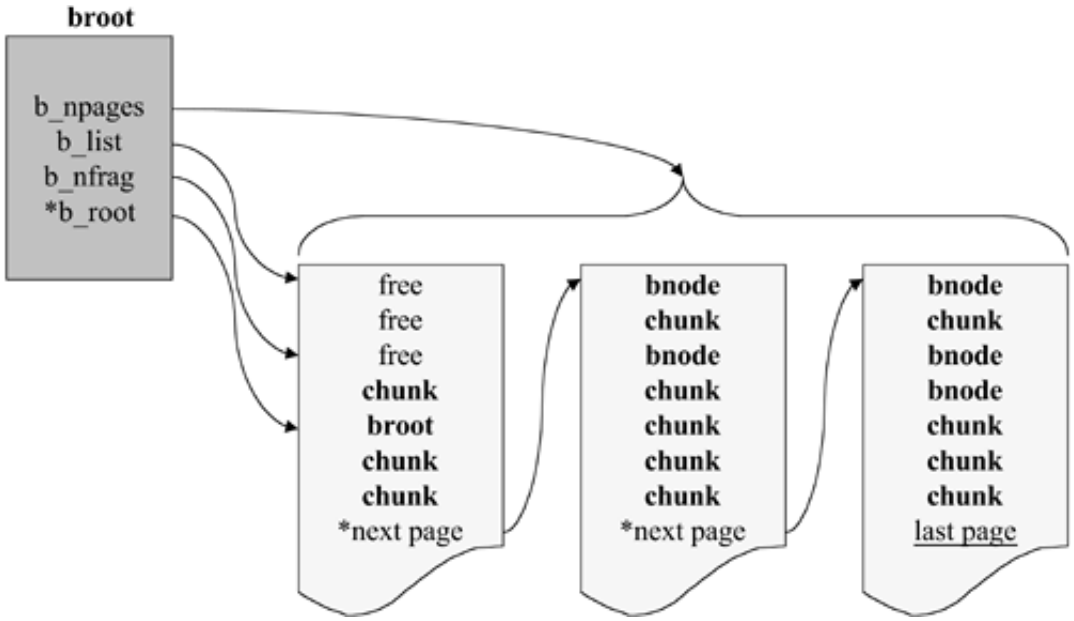
Now that we have defined the various parts of the **b-tree**, we need to assemble them.

Page Lists

The **chunk**, **bnode**, and **broot** structures are of a common size: 256 bytes on narrow systems or 512 bytes for wide systems. As a **region** is initialized, the total number of pages it will map (or at least the number it will start with) is used to calculate the depth of its b-tree. Once this is known, space for the **chunks**, **bnodes**, and the **broot** will need to be allocated. As we noted, all three structures are sized the same. This allows room for a ***29th-order*** **bnode**. Each **chunk** will hold either 32 or 64 **vfd|dbd** structure pairs, depending on the architecture.

These uniformly sized structures may be conveniently packed into whole memory pages. Once the total space requirements are determined, the kernel allocates enough pages to hold them (see Figure 6-11). This group of allocated page frames is reffered to as a page list. In most instances not all the space in the page list is filled with chunks or bnodes; if the tree grows, these unused fragments are utilized. If we run out of page fragments, the system simply allocates another page and adds it to the region's page list.

Figure 6-11. Growing the Tree



An interesting feature of this model is that pages for the b-trees are allocated from the system's free memory pools the same as pages allocated to a user process. These page allocations do not impact the kernel's memory arenas, and as some b-trees must be very large, this is a good thing. Another feature is that if a process is deactivated, its page list may be moved to swap. As process deactivation occurs only in cases of extreme memory pressure, being able to reclaim these extra pages helps **vhand** perform its task.

If a process has been deactivated and its b-tree has been paged out, we must be able to find these pages and restore them to memory when the process is reactivated. As these pages are not defined by a disk block descriptor entry, the back-store location needs to be stored in the region structure. Since the region structure has a static size, and the number of pages in the page list may vary according to the overall region page count, an interesting method was implemented to find the swapped page list pages. The **region r_dbd** contains a disk block descriptor to locate the first page of the page list in swap, the disk block descriptor of the second page is stored at the end of the first page, and so on.

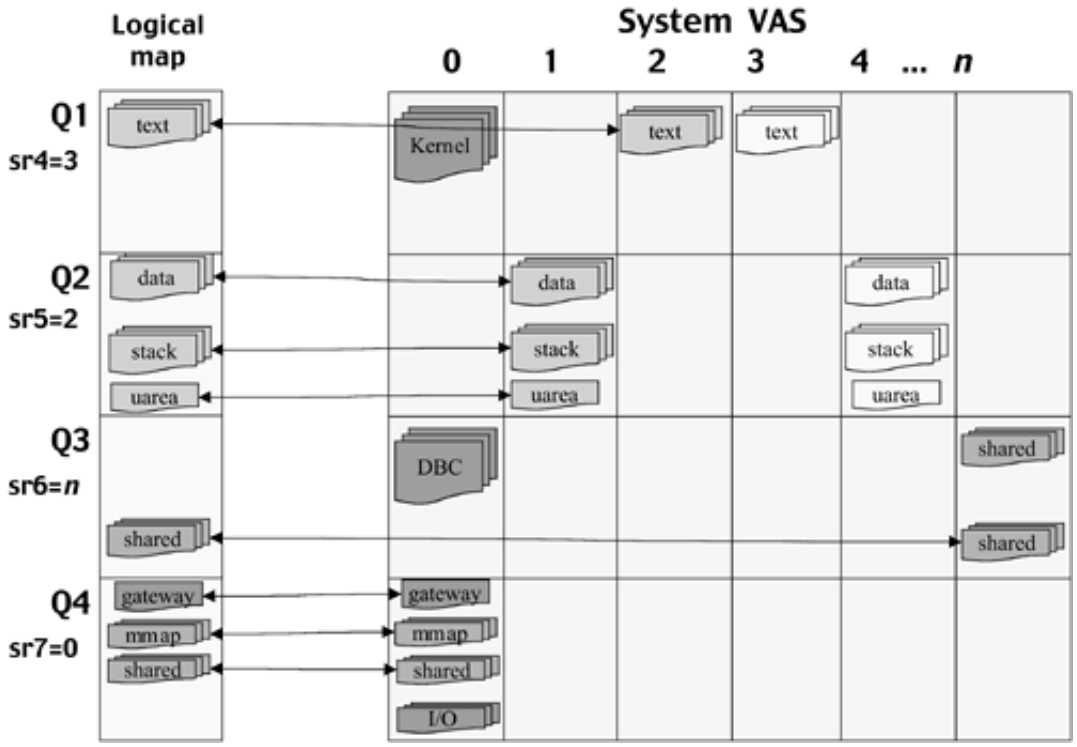
This presents a bit of an anomaly. It is said that HP-UX kernel pages are never swapped out, that the kernel is always in-core. Now we see that a `region`'s `b-tree` may be paged out if its process is deactivated. As a `page list` is allocated to hold a region `b-tree` and the `region` is a kernel structure these pages are kernel pages. So, is the HP-UX kernel ever paged out? The answer is still no, but with the following clarification. Pages initially assigned to the kernel at boot time are considered static kernel pages; those allocated by the kernel later are considered dynamic kernel pages. The kernel never swaps its static pages, but under certain circumstances, such as deactivation, some dynamic pages may be paged out.

Another benefit of this design is that when a process terminates its page list, pages are returned to the system memory pools.

Connecting the Kernel View to the Process View

We are finally ready to tie the process logical view to the kernel virtual view. To review what we have seen so far, as a process is initialized, the system creates a linked list of **`pregion`** structures in accordance with header information stored in the program file. This information was created and placed in the executable file header by the linking-loader that built the program file. As Figure 6-12 shows, each program's logical `pregion`s are linked to an underlying kernel region structure.

Figure 6-12. The Process View Meets the Kernel View

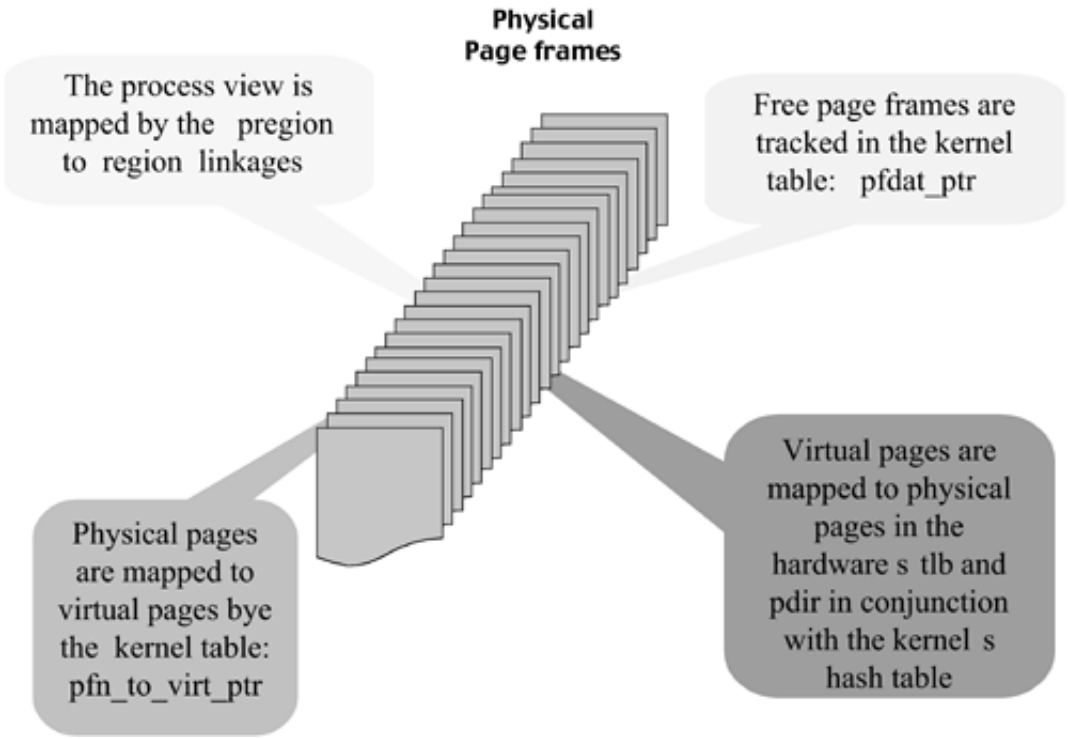


It is at this level of addressing abstraction that the first level of memory sharing may occur. If two instances of the same program are given process status on the system, they may both share pointers to the same text region. In addition, shared library objects may be mapped by many simultaneous processes. Another level of sharing may occur at the physical level; this is called *virtual address aliasing*. To understand this feature, we turn our attention to the physical page management features of the HP-UX kernel.

Managing Page Frames: Many Points of View

As we see in Figure 6-13, there are many ways to view a page frame. It all depends upon whether you are approaching memory from the kernel or a process point of view. So far in this chapter, we have concentrated on the logical and virtual aspects of memory management. Now we get down to the nitty-gritty of managing physical memory.

Figure 6-13. Managing Page Frames



Our first challenge is simply to know how many physical pages there are, which pages are free, and which have been allocated for use either by a user process or the kernel. To serve this need, HP-UX uses a modified version of an age-old structure called the page free data table, or **`pfdat`**. The **`pfdat`** structure has been around in concept since very early UNIX versions, even as far back as Bell Lab System (BLS) releases.

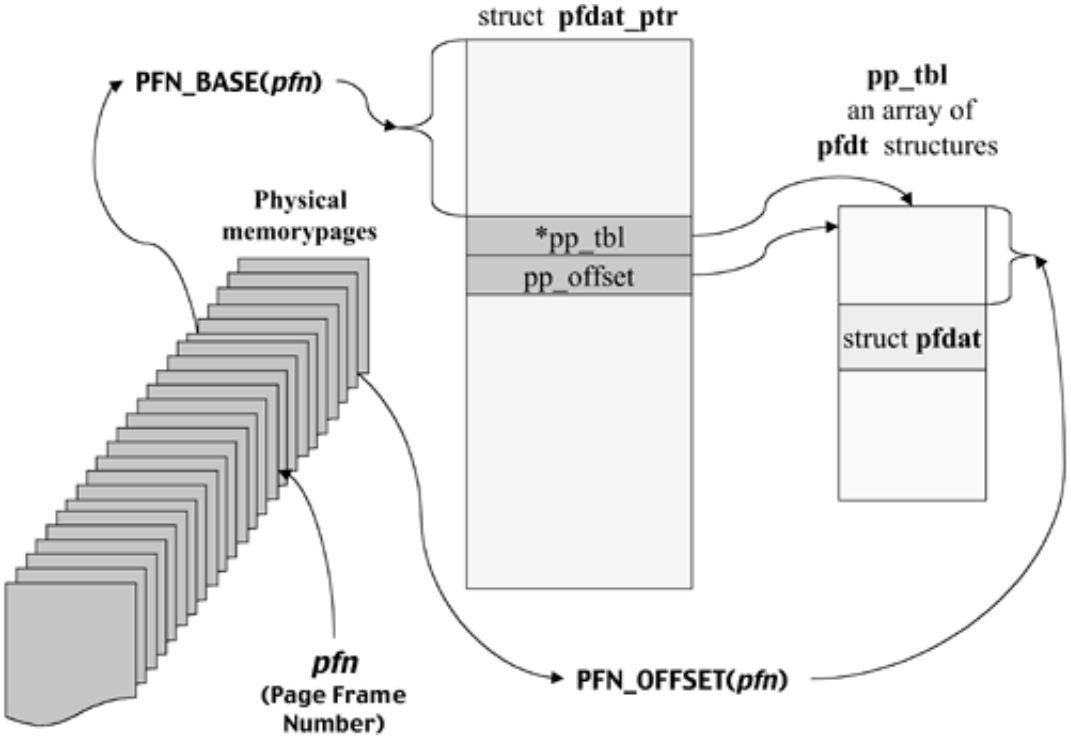
This table is part of the kernel's hardware-independent layer and is not directly connected to the virtual memory translation system. The kernel must manage physical pages and map them to the virtual memory space in alignment with the requirements of the kernel and its processes.

PREV < Day Day Up > NEXT

Keeping Track of Free Physical Page Frames

The original incarnation of **pfdat** was quite simple. Each page had a **pfdat** structure, and the entries were arranged in a simple array. The physical page frame number served as the index into the array. As we mentioned in our discussion of the **pfn_to_virt_ptr** structure, modern HP-UX systems must be able to handle partitioned memory configurations. With the advent of today's cell-based systems with their physical and virtual partitioning schemes, this requirement is coming under a new focus. Figure 6-14 illustrates the current incarnation of the page free data table.

Figure 6-14. Page Free Data, pfdat



The **pfdat_ptr** partition size is set by **PFN_CONTIGUOUS_PAGES** and currently defaults to 4096 pages, or 16 MB, of contiguous memory. Only those physical page frames that may be dynamically paged in or out by the kernel have **pfdat** structures. The kernel's static pages do not require inclusion in this table, as they will never be part of the dynamic memory pool.

The two-tier nature of this structure makes it easy to skip blocks of memory pages such as kernel static memory pages or holes in the physical hardware map. In **pfdat_ptr**, a **pp_tbl** set to zero simply means that the corresponding 16-MB block of memory is not dynamically mapped. Note, this does not mean that the memory does not physically exist, just that it is not to be managed by this structure. (See Listing 6.11.)

Listing 6.11. q4> fields struct pfdat_ptr

```
The pp_tbl points to a block of "page free data" entries and
```

```
the pp_offset directs us to the first valid entry in the

block. Each block maps 4096 physical page translations

(determined by the kernel parameter PFN_CONTIGUOUS_PAGES)

0 0 4 0 *   pp_tbl

4 0 4 0 int pp_offset
```

Once we have used the physical page number and have followed the appropriate **`*pp_tbl`** pointer, we will find the page's **`pfdat`** data structure. Let's examine an annotated listing of this structure (Listing 6.12).

Listing 6.12. `q4> fields struct pfdat`

```
Hash chain linkage pointer

 0 0 4 0 *       pf_hchain

Pointer to the device file vnode (devvp)from which this page

was mapped (if applicable)

 4 0 4 0 *       pf_devvp

Double-linked list of free page frames

 8 0 4 0 *       pf_next

12 0 4 0 *       pf_prev

Double-linked list connecting all page frames sharing a vnode

16 0 4 0 *       pf_vnext

20 0 4 0 *       pf_vprev

Locking structures used during table maintenance

24 0 1 0 u_char  pf_lock.b_lock

26 0 2 0 u_short pf_lock.order

28 0 4 0 *       pf_lock.owner

Process-dependent information

32 0 2 5 u_int   pf_pdk.pf_pfn

34 5 1 3 u_int   pf_pdk.pf_fill

Number of regions using this page (0 means the page should be

on a free list)

36 0 2 0 short   pf_use

Incremented prior to requesting a page frame lock

38 0 2 0 u_short pf_cache_waiting

Disk block number page is mapped from (if applicable)

40 0 4 0 u_int   pf_data

VPS system data for this page and its page group association
```

```
44 0 0 6 u_int   pf_sizeidx
44 6 0 5 u_int   pf_size
45 3 0 5 u_int   pf_size_fill
46 0 2 0 u_short pf_group_id
Flags showing the page frame's current status:
0x001 P_FREE or P_QUEUE Page is on free list
0x002 P_BAD Page has been marked as bad (do not use!)
0x004 P_HASH Page is on a hash queue
0x008 P_CLIC Page is being used by the Cluster InterConnect
0x010 P_SYS This page belongs to the kernel
0x040 P_DMEM Page is locked by the memory deallocation
subsystem
0x080 P_LCOW Page is being remapped by copy-on-write subsystem
0x100 P_UAREA Page is mapped to a kthread's UAREA
0x200 P_KERN_DYNAMIC Page is allocated as dynamic kernel
memory (it may be returned by the kernel later)
48 0 2 0 short   pf_flags
50 0 1 0 u_char  pf_fs_priv_data
The pf_hdl structure contains hardware dependent layer data
The hdl flag values are:
0x01 HDLPF_TRANS A virtual address translation exists
0x02 HDLPF_NOTIFY Virtual DMA, not used by HP-UX
0x04 HDLPF_PROTECT User access no allowed
0x08 HDLPF_STEAL Remove translation when I/O completes
0x10 HDLPF_ONULLPG Used for null d-reference emulation
0x20 HDLPF_MOD A short cut to setting pde_mod flag
0x40 HDLPF_REF A short cut to setting pde_ref flag
0x80 HDLPF_READA Read ahead page in transit
52 0 2 0 u_short pf_hdl.hdlpf_flags
During an alias operation we save the original page's access
rights and protection id bits here
54 0 2 0 u_short pf_hdl.hdlpf_savear
56 0 4 0 u_int   pf_hdl.hdlpf_saveprot
```

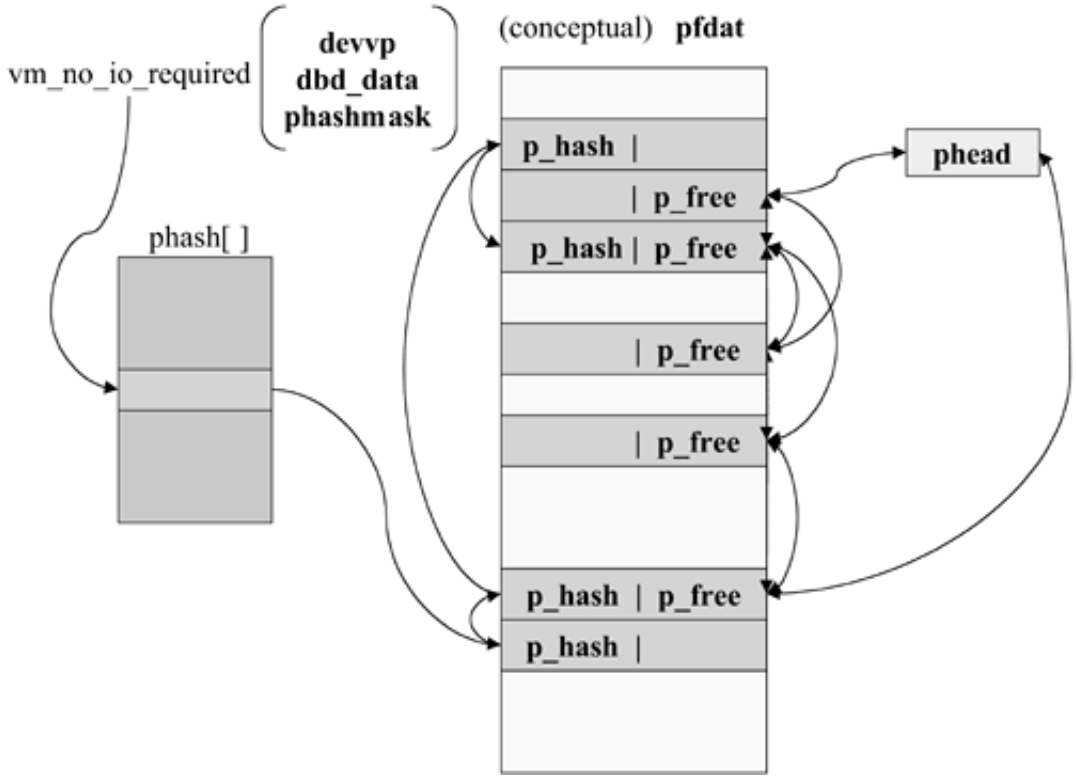
As we see in this structure, page frames are linked to several lists. Of course, there is a free list to facilitate page allocation schemes, and all pages related to a specific open file are linked together as well. Another linkage is the **pf_hchain** at the top of the structure; let's take a look at how this hash is utilized.

Is a Page I/O Really Needed? (`phash` and `phead`)

To assure that there are always free pages available when one is requested, the kernel invests a portion of its resources into the proactive identification of stale pages and the transfer of these pages to a swap location or back store. Moving pages between a disk and physical memory is a very costly operation in terms of computer instruction cycle times and I/O bandwidth. However, not being able to find a free page when one is needed can really throw a wrench into the works! As part of the page-out operation, a page's **`pfdat`** structure is added to a hashed list (see Figure 6-15). The pages are hashed according to the disk block descriptor and the device pointer of the device they have been written to. The actual hash algorithm is

```
((dbd_data >> 5) + dbd_data) xor (devvp >> 6) & phashmask
```

Figure 6-15. `phash` and `phead`



To minimize the cost of paging, the **`phash`** is searched before a page-in operation is called just in case the page frame has not been reallocated since it was paged out. If the required page is found on the **`phash`**, it is simply removed from its free list and reinstated in the **`region`**, **`htbl`**, and **`pfn_to_virt`** tables.

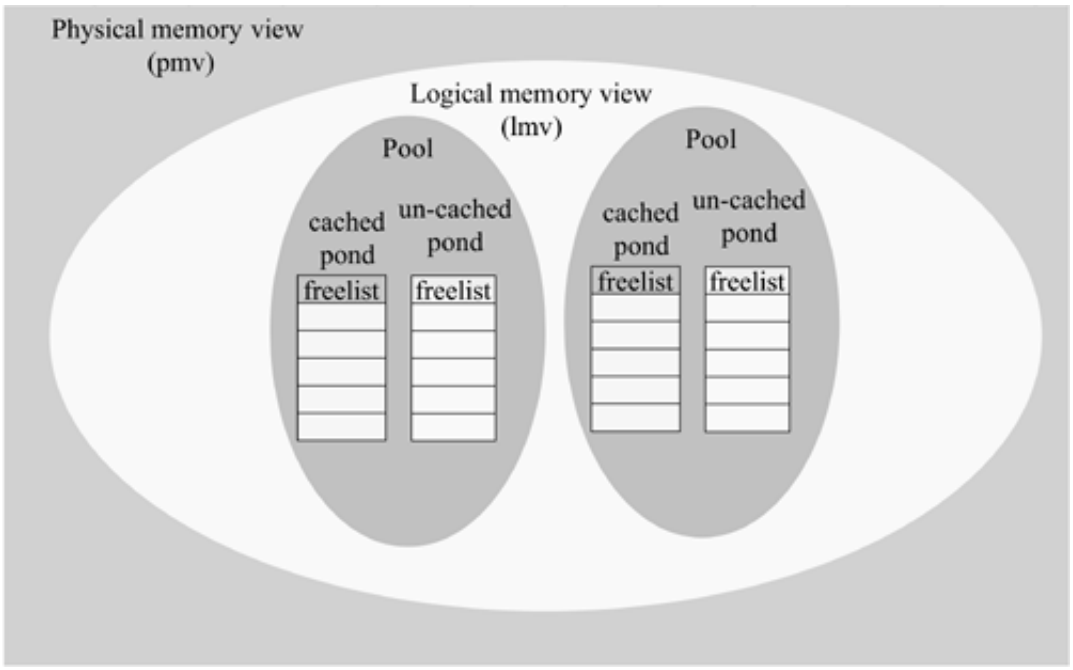
If you are starting to feel comfortable with the logical to virtual to physical to front store and back store relationships, great! That's the goal of this work, but we need to take a deep breath and press on. Recent developments have introduced another level of complexity to our strategy for managing this precious resource.

Views, Ponds, Pools, Pages

An operating system's physical page allocator must be able to work in harmony with the underlying physical hardware. The kernel claims physical pages for its own memory arena scheme and for dynamic assignment to meet its process's needs. As HP-UX is migrating from a purely proprietary architecture, PA-RISC to the

new IA-64, the memory allocator must be modified to handle architectural differences. In preparation for this change and to facilitate partitioning schemes recently introduced, the kernel memory allocator has been modified. Figure 6-16 illustrates an abstraction of the new model.

Figure 6-16. Pools, Ponds, Views



This new model includes a combination of views, ponds, pools, and groups. Let's try to define these terms and concepts.

Physical Memory View

The highest level of organization is the physical memory view, or **pmv**, which allows a system's physical memory to be divided into sets depending on specific properties or aligned according to physical configurations. One use would be to divide memory according to its access latency. On an SMP (symmetrical multi-processor) or UMA (uniform memory access) machine there would be only one **pmv**, while on a multinode design or ccNUMA (non-uniform memory access) platform, each node (or cell) might have its own **pmv**. To date, this level has not been coded into the HP-UX kernel and is presented here only for future considerations.

Logical Memory Views

The logical memory view, or **lmv**, is the next level of division within a **pmv**. The **lmv** maintains its own free lists and lock structures. Currently, the `lmv`s are aligned with the system's processor sets. By having each processor set allocate memory from its own free list and use its own locking mechanism, overall system performance is increased due to the reduction in lock contentions. This is currently the highest abstraction level incorporated in the HP-UX kernel. The **lmv** is managed by an array of **lmv** structures allocated at boot time. The number of `lmv`s is currently calculated as the total number of processors divided by four.

Memory Pools

Each **lmv** is made of one or more pools of memory. A memory **pool** boundary is set in alignment with intrinsic hardware restrictions. For instance, on an HP-UX 32-bit kernel the first gigabyte of physical memory belongs to one **pool**, and the memory above 1 GB belongs to another. To allow equivalent memory mapping of kernel pages, it is required that they be in the first physical gigabyte of memory. By having the two pools, the memory allocator may be instructed to always allocate kernel pages from the first pool and to allocate user pages from the second pool (unless none are available, in which case the first pool is used).

On current PA-RISC systems, there are only two pools, one for equivalently mapped memory and one for nonequivalently mapped memory. On a wide system, the entire physical memory may be equivalently mapped, as the addressing range for the 64-bit system is quite large (2^42).

The page pool data is stored in an array of **page_pool** structures, which reside inside their respective **lmv**.

Memory Ponds

As mentioned, if a page is swapped out to increase the supply of free pages, its **pfdat** structure is linked to the **phash**. These pages are called *cached* free pages. As pages are freed, they are divided into one of two ponds of memory: one for cached free pages and the other for uncached. If the system uses variable page sizing, then a separate free list is maintained for each page size. The number of free list headers created per **pond** is determined by the kernel tunable parameter **pf_offset_idx** and set at boot time. The **pond** structure is contained as an array in its corresponding pool structure.

The Nested Structures

In Listing 6.13, we see that the pond array is nested in the pool structure, which in turn is nested in the **lmv** structure.

Listing 6.13. `q4> fields lmv_t`

```
Number of pools in this logical memory view

  0 0 4 0 int   lmv_npools

Index of this logical memory view

  4 0 4 0 int   lmv_id

Amount of free memory in this view

  8 0 4 0 int   lmv_freemem

Pointer to spinlock for protection of this view's free lists

 12 0 4 0 *     lmv_fl_lock

Individual page pool structures (for PA-RISC there are two

pools per logical memory view). We start with a linked list of

all page groups in the pool, a free list spinlock for this

pool, a linked list of the pools in this logical view, a

pointer to the parent lmv structure, and a "used" count for

this pool

 16 0 4 0 *     lmv_pools[0].pp_next

 20 0 4 0 *     lmv_pools[0].pp_fl_lock

 24 0 4 0 *     lmv_pools[0].pp_next_pool

 28 0 4 0 *     lmv_pools[0].pp_lmv
```

 32 0 4 0 *   lmv_pools[0].pp_n_used

Within the pool structure we have two pond structures. Each pond starts with a pointer to its free list headers, the fee count for this pond, a pointer to the parent pool structure

 36 0 4 0 *   lmv_pools[0].pp_ponds[0].pp_phead

 40 0 4 0 *   lmv_pools[0].pp_ponds[0].pp_n_free

 44 0 4 0 *   lmv_pools[0].pp_ponds[0].pp_pool

In addition there is an index count to the number of active entries in the pp_phead[] (number of page sizes allowed), the index number of this pond, and low- and high-water marks used by the lazy coalesce routines

 48 0 4 0 u_int lmv_pools[0].pp_ponds[0].pp_max_idx

 52 0 4 0 u_int lmv_pools[0].pp_ponds[0].pp_pond_index

 56 0 4 0 u_int lmv_pools[0].pp_ponds[0].pp_lb_shift

 60 0 4 0 u_int lmv_pools[0].pp_ponds[0].pp_ub_shift

 64 0 4 0 *   lmv_pools[0].pp_ponds[1].pp_phead

 68 0 4 0 *   lmv_pools[0].pp_ponds[1].pp_n_free

 72 0 4 0 *   lmv_pools[0].pp_ponds[1].pp_pool

 76 0 4 0 u_int lmv_pools[0].pp_ponds[1].pp_max_idx

 80 0 4 0 u_int lmv_pools[0].pp_ponds[1].pp_pond_index

 84 0 4 0 u_int lmv_pools[0].pp_ponds[1].pp_lb_shift

 88 0 4 0 u_int lmv_pools[0].pp_ponds[1].pp_ub_shift

Each pool maintains a total count of available page frames and their status flags

 92 0 4 0 int   lmv_pools[0].pp_pfdatnumentries

 96 0 4 0 u_int lmv_pools[0].pp_flags

100 0 4 0 *   lmv_pools[1].pp_next

104 0 4 0 *   lmv_pools[1].pp_fl_lock

108 0 4 0 *   lmv_pools[1].pp_next_pool

112 0 4 0 *   lmv_pools[1].pp_lmv

116 0 4 0 *   lmv_pools[1].pp_n_used

120 0 4 0 *   lmv_pools[1].pp_ponds[0].pp_phead

124 0 4 0 *   lmv_pools[1].pp_ponds[0].pp_n_free

128 0 4 0 *   lmv_pools[1].pp_ponds[0].pp_pool

132 0 4 0 u_int lmv_pools[1].pp_ponds[0].pp_max_idx

136 0 4 0 u_int lmv_pools[1].pp_ponds[0].pp_pond_index

140 0 4 0 u_int lmv_pools[1].pp_ponds[0].pp_lb_shift

```
144 0 4 0 u_int lmv_pools[1].pp_ponds[0].pp_ub_shift

148 0 4 0 *     lmv_pools[1].pp_ponds[1].pp_phead

152 0 4 0 *     lmv_pools[1].pp_ponds[1].pp_n_free

156 0 4 0 *     lmv_pools[1].pp_ponds[1].pp_pool

160 0 4 0 u_int lmv_pools[1].pp_ponds[1].pp_max_idx

164 0 4 0 u_int lmv_pools[1].pp_ponds[1].pp_pond_index

168 0 4 0 u_int lmv_pools[1].pp_ponds[1].pp_lb_shift

172 0 4 0 u_int lmv_pools[1].pp_ponds[1].pp_ub_shift

180 0 4 0 u_int lmv_pools[1].pp_flags

176 0 4 0 int   lmv_pools[1].pp_pfdatnumentries
```

The **pp_head[]** contains the actual free list pointers for each **pond**. The first element in this array has the next and previous pointer for the one-page frame free list, the second entry has the linkage pointers to the four-page frame free list, the third to the 16-page frame free list, and so on up to the maximum configured page size. To facilitate management of multiple page sizes, an additional structure called the memory **page group** was created.

Variable Page Size

The HP-UX 11.00 release introduced variable page sizes (vps) also known as performance optimized page sizing (pops). This feature allowed the kernel to take advantage of new hardware features introduces in PA-RISC 2.0 processors. On the new wide processors the tlb has an additional field introduced to allow the mapping of a variety of virtual-to-physical page translation sizes. The basic page frame remains 4Kbytes but the vps parameter allows a single tlb entry to map from 4Kbytes to a potential maximum of 1048576 Kbytes of contiguous physical memory with a single entry. The page size is determined by a 5-bit field in the `htbl/pdir` entry. The current implemented sizes range from 1 to 256K page frames (each progressively larger size is 4X the previous small size). The kernel tracks a process's memory useage and may adjust the page size or a program or a programmer may use the `chatr` command to adjust the page size. There are several kernel tunable parameters used to control this feature:

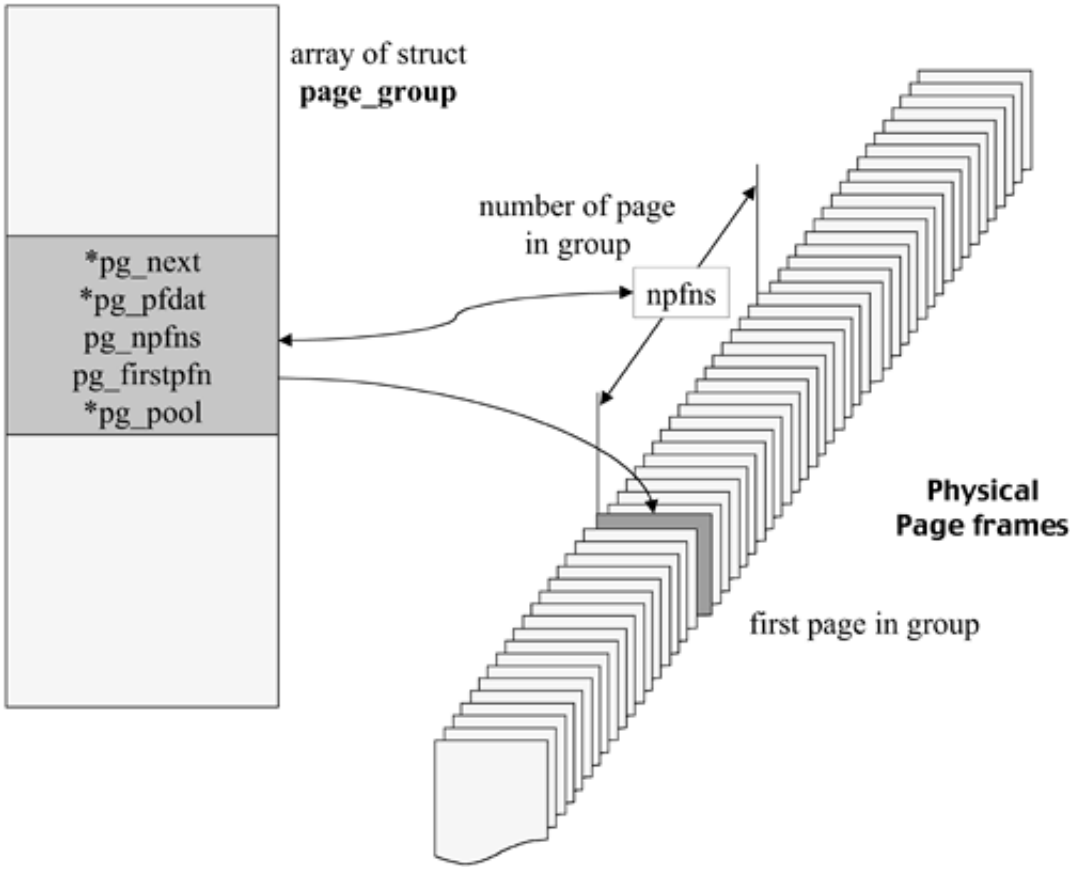
`vps_pagesize` This parameter sets the base page size and defaults to 4Kbytes

`vps_cieling` This parameter sets the maximum page size the kernel may select and is currently limited to 65536KBytes

`vps_chatr_cieling` This is the maximum page size which may be requested by the `chatr()`and is limited to 1048576Kbytes

With respect to kernel memory management, the various sizes are referred to as a *page group*. As we see in Figure 6-17, a **`page_group`** structure helps the kernel identify and manage consecutive blocks of free pages.

Figure 6-17. Page Groups



The **page_group** structure contains information to help the kernel decide if this group may be combined, coalesced with its "buddies" to form a larger group. Let's examine the structure in Listing 6.14.

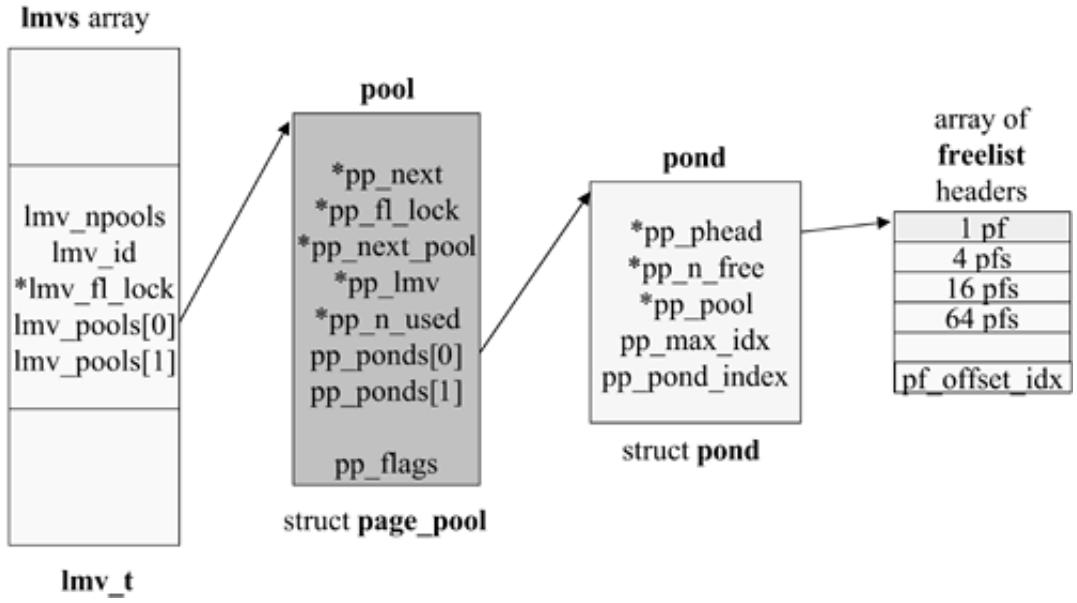
Listing 6.14. q4> fields struct page_group

```
A linkage pointer connecting all groups in a pool
 0 0 4 0 *     pg_next
A pointer to the start of this group's pfdat array
 4 0 4 0 *     pg_pfdat
Number of page frames in this group
 8 0 4 0 int   pg_npfns
First page frame in this group
12 0 4 0 u_int pg_firstpfn
Pointer to the pool this group belongs to
16 0 4 0 *     pg_pool
```

Physical Memory Allocator at HP-UX 11.0

Now that we have examined the various components of the **lmv**, we must put it all together and consider the two tasks of memory allocation and memory freeing. Figure 6-18 ties the structures into a single view.

Figure 6-18. Physical Memory Allocator



Allocating Memory

The kernel routine **allocate_page()** is responsible for the allocation of physical pages of a specific requested size. It works with the views, pools, and ponds. The basic algorithm works as follows:

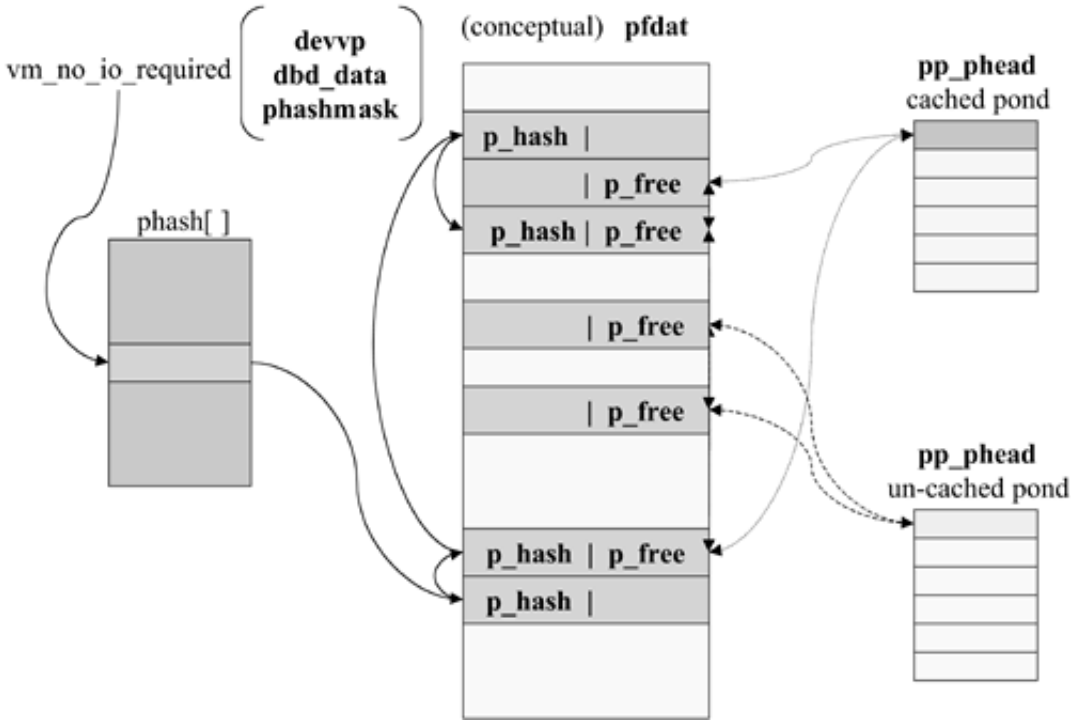
- An allocation is attempted from the uncached ponds of all pools for the desired size (or larger). The initial pool is determined by **CHOOSE_VIEW_AND_POOL()**, which is keyed to the underlying system architecture.
- If the initial pool can not fill the request, **NEXT_POOL_TO_TRY()** is called to select the next victim.
- If the request is for an equivalently mapped page, the processor's own **lmv EQUIV_MAPPED_POOL** is tried first; then the other **lmv EQUIV_MAPPED_POOL**s are tried.
- If the request is for a nonequivalently mapped page, we first try our own **lmv NON_EQUIV_MAPPED_POOL** followed by the other **lmv NON_EQUIV_MAPPED_POOLs**, and then the **EQUIV_MAPPED_POOL**s are tried.
- If all uncached ponds of all pools have been tried, then we try the cached ponds of all the pools.

- If we still can't find the needed page size, we step the request down to the next smaller page size and repeat the above steps.
- If we still can't make the requested allocation and the call doesn't permit blocking, then we return a `NULL` and let the thread handle the failure.
- If blocking is allowed, the thread will be put to sleep after tickling the paging daemon and the unhash daemon.

Freeing Memory

Freeing memory when you have multiple free lists and the added challenge of coalescing pages to form larger **`page_group`**s has complicated this part of the process. Figure 6-19 illustrates the concept of multiple free lists.

Figure 6-19. Free Lists Revisited



The kernel routine used for this daunting task is `freepfd()`.

- The first rule for **`freepfd()`** is to decrement the **`pf_use`** in the page's **`pfdat`** structure.
- If the **`pf_use`** is now a zero, the physical page is ready for assignment to a free list following the cleanup of its **`hash`**, **`region`**, and **`pfn_to_virt`** table entries.
- The routine **`freeup_page()`** is called to assign the page to the appropriate free pool.

- Finally, the global free memory count, **`freemem`**, is incremented following any coalescing that may occur.

Coalescing Free Pages

As pages are placed on free lists, we must check if their neighbors are also free and if they can be combined to form larger sized free pages. This process is called *page coalescing*. Pages next to each other in the physical memory map are called buddies, and the routine responsible for coalescing pages is **`place_on_freelist()`**. The **`freeup_page()`** routine calls **`place_on_freelist()`** and passes it the physical page frame number it is trying to free up.

- Find the page's buddy. This is a simple task of changing the bit corresponding to the page's current size and checking whether it is free.
- Confirm that the page to be freed and its buddy are in the same pond.
- Coalescence is only attempted for pages in the cached pond if at least one-third of the pond's pages are currently free. In the case of uncached pages, coalescence is always attempted.
- The resulting page (coalesced if possible) is then placed at the end of the appropriate free list.

Summary

We have seen the four primary views of memory managed by the kernel and utilized by the process threads and PA-RISC hardware. All of the various levels of abstraction would be pointless if we couldn't allocate physical memory when it was time to allow a thread to run.

Having an ample supply of free pages is very important to overall system performance. To this end the kernel employs a swapping or paging system. We examine it next.

Chapter 7. The HP-UX Paging System

Classic UNIX featured a program swapping system. When the kernel needed to make room in-core memory for a process to run, it would select an inactive process or one of a lesser priority and move it from physical memory to a temporary storage location on disk. The program was then said to be *swapped out*. Before the process could run again, it would have to be swapped back into memory. This could tie up a large portion of the system's I/O bandwidth. Whole process swapping has been relegated to the history books: modern kernels utilize a *demand paging system*.

Pages on Demand

In a demand paging system, instead of loading all of a program's text and data at the time the program is dispatched, the system waits for a page to fault and then loads only the required page. This means that at any specific point only a small portion of a process's pages need be resident in physical memory. All first references to a page result in a page fault, but if a system has ample memory then eventually all of a process's pages will be mapped to physical pages and performance will be optimal. If the demand for pages exceeds the system's configured or available memory, then pages will have to be paged out before others may be paged in.

The HP-UX paging system is responsible for loading pages when they are first requested and monitoring and maintaining a reasonable supply of free pages. If the amount of free memory drops below tunable system thresholds, the paging system must identify pages to be paged out. Page-ins will always occur; at least as long as new programs are dispatched, but significant numbers of page-outs indicate memory pressure has increased and overall system performance will begin to diminish. This is not necessarily bad; it is just a case of cause and effect that the system administrator must understand.

When a page fault occurs, the kernel must be able to identify the location of the requested page and take corrective action. As we saw in Chapter 6, "Managing Memory," in our discussion of the **dbd**, each noncore resident page is typed. The paging system understands the various types and how to recover them. Let's look at the types in detail.

Faulting in a Page from Front Store

A requested page may be part of a process's text image or an initialized data page. If so, the appropriate page is found in the program file, and it is loaded to a free page frame. In addition to scheduling the page-in, the kernel must update the **hash**, **region**, **pfdat**, and **pfn_to_virt** table entries as well.

The routine handles read-ahead page requests as well; the **p_pagein** in the **pregion** is adjusted according to whether the file is being accessed in a random or sequential manner, which is determined by monitoring the address of faulting pages. Each time a fault is handled for a region, its **p_nextfault** is set to the next sequential page address following the one that caused the current fault (the current faulting page number plus **p_pagein**). When the next fault for this **pregion** occurs, the faulting page address is compared to this value. If it matches, we are performing sequential reads, in which case **p_pagein** is multiplied by 2 (the value is limit-checked against the kernel's **maxpagein_size**, default is 64). If **p_strength** is less than 100 and we have detected a sequential operation, it is incremented. This gives us a sense of how sequential our access is.

If the next fault is not sequential, **p_pagein** is divided by 2 (limit-checked against **minpagein_size**, defaults to 1), and the **p_strength** is decremented unless it is currently equal to –100.

From the above description we can see that **p_strength** may vary between –100 (defined as **PURELY_RANDOM**) and +100 (defined as **PURELY_SEQUENTIAL**), and **p_pagein** may vary by powers of two between 1 and 64.

Faulting in a Page from Back Store (Swap)

A page may currently reside in swap. Before the page is scheduled for a page-in, the system first calls **vm_no_io_required()** to check the **phash** in case the page was swapped but never reallocated. If this is the case, the page is still in memory and all the kernel has to do is update the tables—no actual page-in is required. This is called a *soft page fault*. If the fault is not soft, then a page-in is scheduled and the four tables are updated as in the previous case.

Faulting in a Zero-Fill Page

If the page requested is a uninitialized data page, the kernel simply has to find a free page, fill it with zeros, and update the four tables. Uninitialized data pages are also known as block store by symbol (BSS) pages and constitute the majority of a process's private data area.

Faulting in a No-Fill Page

In addition to these three types of pages, the kernel may be asked to fault in a no-fill page. This is similar to the BSS page fault except that we don't fill the page with zeros. The page will still contain whatever residue was left behind by its previous owner. For this reason, this option is available only for pages being allocated for kernel use and, in particular, only for use as **`uarea`** pages. The `uarea` is not visible to the process, only to the kernel so residue data is not subject to examination by a user thread.

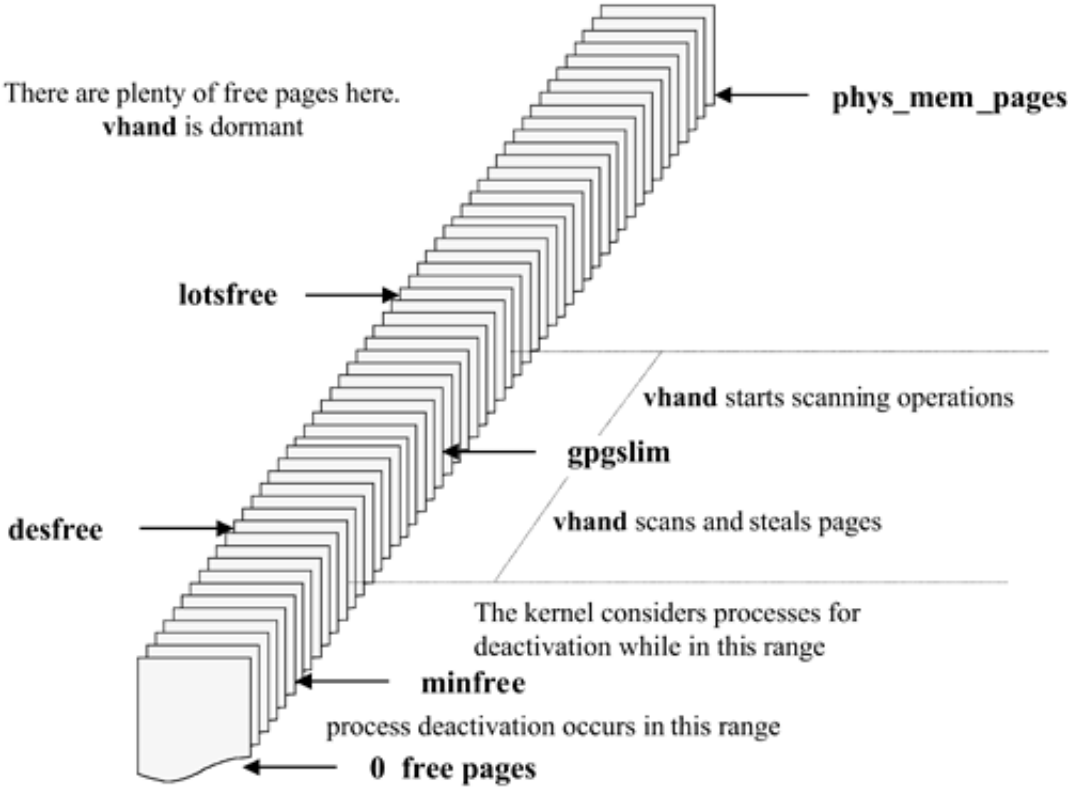
Digging a Memory Hole

The final type is a page called a *memory hole*, used for linkage to sparse memory-mapped files. When a read is scheduled against one of these pages, a requestor receives a 0 or null value; when a write is attempted, the semantics of the zero-fill page are followed.

Monitoring Free Memory

Maintaining an ample supply of free pages for the system is a key duty of the kernel; to this end, the kernel symbol `freemem` contains the total number of page frames currently on a free list. When memory pressure is high, there are relatively few free pages on the free lists, and the kernel must take corrective action. Figure 7-1 gives us an overall view of the free-page monitoring system.

Figure 7-1. `Lotsfree`



The total amount of memory available to the operating system is determined at boot and stored in the kernel parameter **`phys_mem_pages`**. Additional setpoints are calculated and continuously monitored by the kernel.

`lotsfree`

The tunable parameter **`lotsfree`** sets the point at which the paging system is first called into action. If the number of free pages, `freemem`, is above **`lotsfree`**, then the paging system has nothing to do. The default value for **`lotsfree`** depends on the amount memory configured for the system.

For systems with 32 MB to 2 GB of physical memory, **`lotsfree`** will be 1/16 of the non-kernel memory, not to exceed 8192 pages (32 MB). For systems with more than 2 GB, **`lotsfree`** is set to 16,384 pages (64 MB).

desfree

The tunable parameter desfree sets the minimum desired number of free system pages. When the number of free pages drop below this limit, the **vhand** agents are already working at their maximum rate. The swapper starts evaluating processes for potential deactivation, but no deactivations take place yet.

For systems with 32 MB to 2 GB of memory, **desfree** is set to 1/64 of non-kernel memory, not to exceed 1024 pages (4 MB). For systems with more than 2 GB of memory, **desfree** is set to 3072 pages (12 MB).

minfree

When the tunable parameter minfree's limit is hit, things are really tight, supply is not keeping up with demand by any stretch of the imagination. In terms of memory availability, we are in the 11th hour, so the kernel asks the swapper to start process deactivations. Hopefully, this action will get **freemem** back into an acceptable range.

For systems with 32 MB to 2 GB of memory, **minfree** is set to 1/4 of **desfree**, not to exceed 256 pages. For systems with more than 2 GB of memory, **minfree** is set to 1280 pages (5 MB).

gpgslim

At boot time, this parameter is set to 1/4 of the way between **desfree** and **lotsfree**. When **freemem** drops below **lotsfree**, the **vhand** daemon is started, but at this time it only ages pages—no stealing is done. If **freemem** drops below **gpgslim**, then the other half of **vhand** is activated, and it steals stale pages. The kernel adjusts **gpgslim** between **lotsfree** and **desfree** in accordance with recent current trends in memory pressure. (In earlier versions of HP-UX this parameter was tunable.)

Other Kernel Paging Parameters

There are several other kernel parameters we need to mention:

- **freemem** (calculated): This is the current free page count for the system, the number of page frames on all the free lists.
- **parolemem** (calculated): This is the count of pages pending page-out operations at this time.
- **vhandrunrate** (tunable): This is the number of times per second the kernel will kick off the **vhand** daemon if the free page count (**freemem** plus **parolemem**) has dropped below **lotsfree**. The default is 8.
- **memzeroperiod** (tunable): This parameter, in .1 second increments, determines how often **gpgslim** is adjusted by the kernel. Every **2 * memzeroperiod** seconds or if **freemem** has reached zero more than two times, **gpgslim** is adjusted. The formula for this adjustment is [3 * **gpgslim** + (**lotsfree**)/4] if **freemem** has reached its target limit within **memzeroperiod** and [3 * **gpgslim** + (**desfree**)/4] if **freemem** has not reached its target limit within **memzeroperiod**. The default value is 30, or every 3 seconds.
- **hitzerofreemem** (kernel): The number of times **freemem** has hit zero in the last **memzeroperiod**.
- **targetcpu** (tunable): This represents the maximum number of CPU cycles we are willing to allocate to the **vhand** daemon. The default is 100 ticks, or 10 percent of the total CPU time.
- **maxqueuetime** (tunable): This number controls the pending **parolemem** queue depth. The value is in .1 seconds and defaults to 10. This means that the system will attempt to keep the page-out queue to no more than the system can handle in 1 second.
- **pageoutcnt** (counter): This is a count of recent kernel- and user-requested page-outs.

- **kpageoutcnt** (counter): This is the number of recently completed kernel-requested page-outs (**vhand**).
- **pageoutrate** (calculated): This is the current total page-out rate.
- **kpageoutrate** (calculated): This is the current **vhand** page-out rate.
- **maxpendpageouts** (calculated): **pageoutrate** * (**maxqueuetime**/10).
- **pgrateupdate** (tunable): Number of seconds to copy **curr_pgrate** to **max_pgrate**; defaults to 60.
- **curr_pgrate** (counter): Maximum **kpageoutrate** seen in the current period.
- **max_pgrate** (calculated): Maximum **kpageoutrate** seen in the last **pgrateupdate** period. This is an indicator of possible page thrashing if it is high and the CPU utilization rate is low.
- **agehand** (kernel): This is a pointer to the next **pregion** in which **vhand** may age (scan) pages.
- **stealhand** (kernel): This is a pointer to the next **pregion** in which **vhand** may steal pages.
- **agerate** (kernel): This is the age quota assigned by the swapper to **vhand** for its next run.
- **stealrate** (kernel): This is the steal quota assigned by the swapper to **vhand** for its next run.
- **handlaps** (kernel): This it the desired amount of lag between the scanning operation and the stealing operation in a **pregion**.
- **targetlaps** (kernel): This it the actual amount of lag between the scanning operation and the stealing operation in a **pregion**.
- **vhandinfoticks** (adb tunable): This parameter should only be set using **adb** (by a very knowledgeable user). If it is a nonzero value, then every **vhandinfoticks vhand** prints a status report out to the system console, including the current values of **handlaps**, **targetlaps**, **agerate**, **gpgslim**, **parolemem**, **stealrate**, and **freemem**.

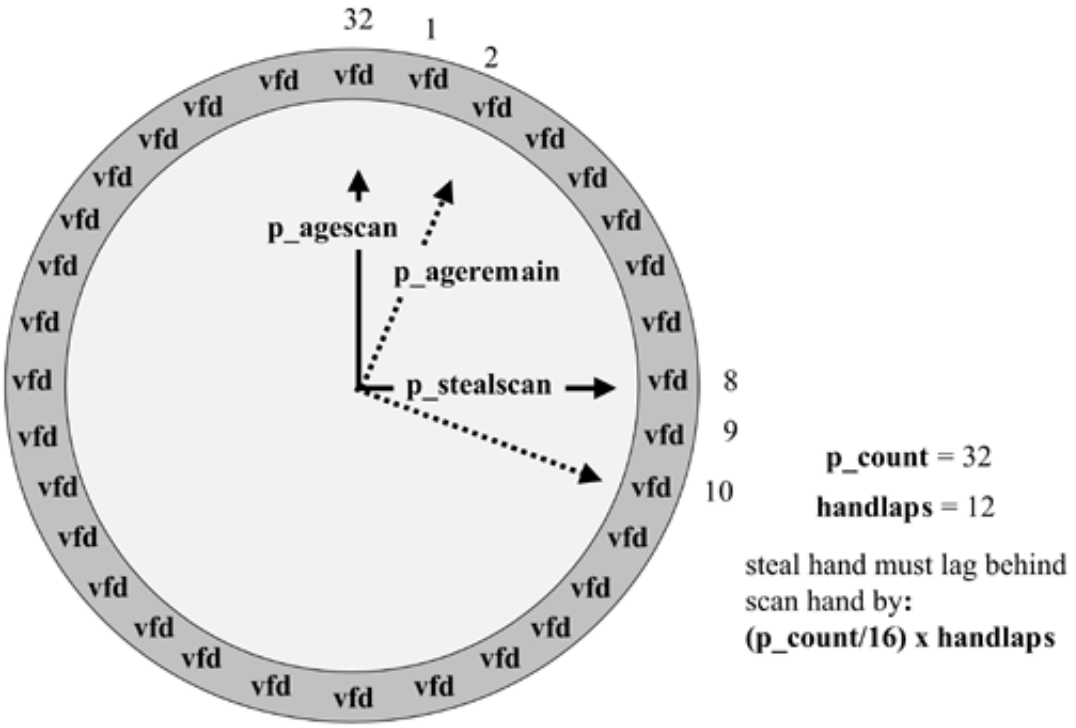
The **vhandrunrate** sets flags in the kernel to call **schedpaging()** at the appropriate rate. This routines checks the current value of **freemem**, and if it has fallen below **lotsfree** (or **gpgslim** in an extreme case), the **vhand** daemon is called. Let's examine **vhand** in detail.

A Thief in the Night: vhand

The vhand kernel daemon is actually two operations in a single package. The first scan targets pages and, if their pde_accessed bits are still clean, steals them; the second scan ages targeted pages. To *age* a page means to invalidate its current virtual address translation(s) in all processor TLBs and clear the access bit for the page in its hashtable entry (pde_accessed = 0). vhand *steals* a page when it discovers that the page has not been referenced since it was last scanned. It then requests a page-out and allocates a swap location if one does not already exist for the page.

Pages are aged and stolen through their pregion → region → vfd linkages. Figure 7-2 shows vhand's view of a pregion.

Figure 7-2. Inside Each pregion



In this simplified model we see that all the pages of the pregion are arranged in a circle, and their respective vfds appear as the numerals on the face of a clock. In our simple example, the pregion has 32 pages, sufficient to explain vhand operation. In the pregion structure, p_stealscan points to the next page in this pregion, which may be checked for parole. The p_agescan points to the next page vhand may age. When vhand enters a pregion, it is only allowed to scan 1/16 of the available pages.

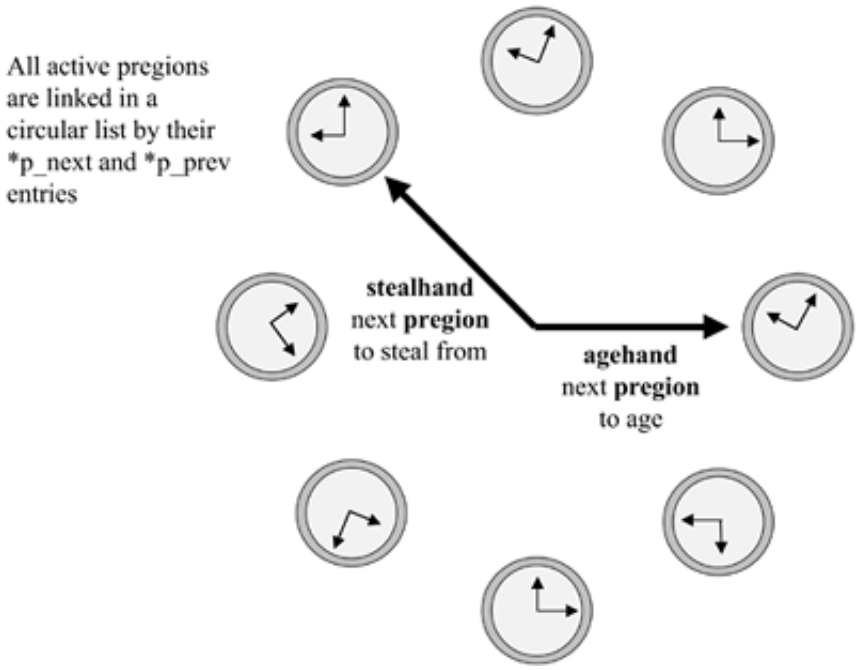
The p_stealscan must lag the position of the p_agescan by [(p_count/16) * handlaps]. Maintaining this lag assures that a process that still needs the aged page may access it before it is stolen. The larger the lag, the less likely vhand is to steal a page in current use.

The next time the system points to this pregion, the next 1/16 of its pages may be scanned. Individual pregions are dual-linked in a systemwide list of all active pregions by their p_forw and p_back pointers

The Linked `pregion` List

The kernel follows this dual-linked list to find the next **`pregion`** to scan Figure 7-3 illustrates this concept. You can think of the operation as a two-level clock. The hands on the main clock are the system's **`agehand`** and **`stealhand`**. Inside each **`pregion`** are corresponding **`p_agescan`** and **`p_stealscan`** pointers. This is where the paging routine's name came from: **`vhand`** stands for the *virtual hand* pager.

Figure 7-3. Linking All the `pregions`



In order for all the pages of a **`region`** to be aged, the system's **`agehand`** would have to make 16 complete laps of the systemwide **`pregion`** list—hence the term **`handlap`**. The larger the **`handlap`** value (maximum is 15), the greater the lag time between the **`vhand`** aging a page and being able to steal it. As memory pressure falls below **`gpgslim`**, the `handlap`'s value is reduced until the **`p_stealscan`** is bumping into the **`p_agescan`** (it is never allowed to overtake it).

Each time **`vhand`** is scheduled, it is given a quota of pages to age and steal: **`stealrate`** and **`agerate`** respectively. As it enters the **`pregion`**, it attempts to meet its quota. if the **`pregion`** doesn't have enough pages to meet the quota, **`stealhand`** simply moves on to the next **`pregion`**. Once the quota is reached, it stops. The current position of **`p_stealscan`** is where it starts next time this **`pregion`** is scanned. The same is true for the aging process

The quotas are also adjusted as **`freemem`** loses ground with respect to **`gpgslim`** and **`desfree`**. By the time we reach **`desfree`**, `handlap`s are at a minimum, the quota is at its maximum (determined by the setting of **`targetcpu`**), and the paging is as aggressive as it can be. If this isn't enough to get us back into the good graces of the memory management system, then deactivation may do the trick.

Programs begin with no pages in memory. As page faults occur, the requested pages are loaded; this is the essence of demand paging. As time goes by, a process may end up with a substantial number of its pages in-core. As other processes are scheduled, memory pressure may start to increase. When a program is deactivated, all of its **`pregions`** are marked as deactivated and moved to the front of the **`stealhand`** list. All of its pages may be paroled the next time **`vhand`** runs.

As soon as we get above the **`desfree`** limit, we consider deactivated processes for reactivation. Even if we reactivate a large process, we may be able to stay above the limit, since pages are only reloaded on an on-

demand basis.

Reservation Versus Allocation

In operating system design, there are many variations of swap management. Some systems dynamically assign swap space from available file system space and return it when it's not needed. Others configure static, dedicated swap at boot time and force the system to work within its boundaries. Some allow swap space to be allocated on the fly from either dedicated volumes or by creating a directory under a mounted file system and creating swap files there.

An important design consideration is how the system responds if it runs out of swap space. Will the kernel crash or simply terminate the process that made the swap request? This could have severe impact on running applications and wreak havoc on system uptime.

A final issue is when to allocate a swap page for a process page. Some systems make an entire copy of a program to swap prior to running the first command. These allocated swap pages belong to the process as long as it is active and are reclaimed by the swap system only at process termination. Another approach is to hand out swap locations as they are needed and return them to a free list as soon as their pages are reloaded into memory. In this model, a page may be swapped several times during its life cycle, and it might use a different swap location each time. This approach means we have to go through the overhead of allocation each time the page is to be swapped. We could also wait to assign a page until it is needed and then keep the allocation until the process terminates. In this last model, we would have to pay the price of allocation only once. Each approach has its pros and cons.

HP-UX uses bits and pieces of the two approaches and adds a few twists of its own. The HP-UX paging strategy keeps track of the number of reserved, allocated, and free swap pages under its control.

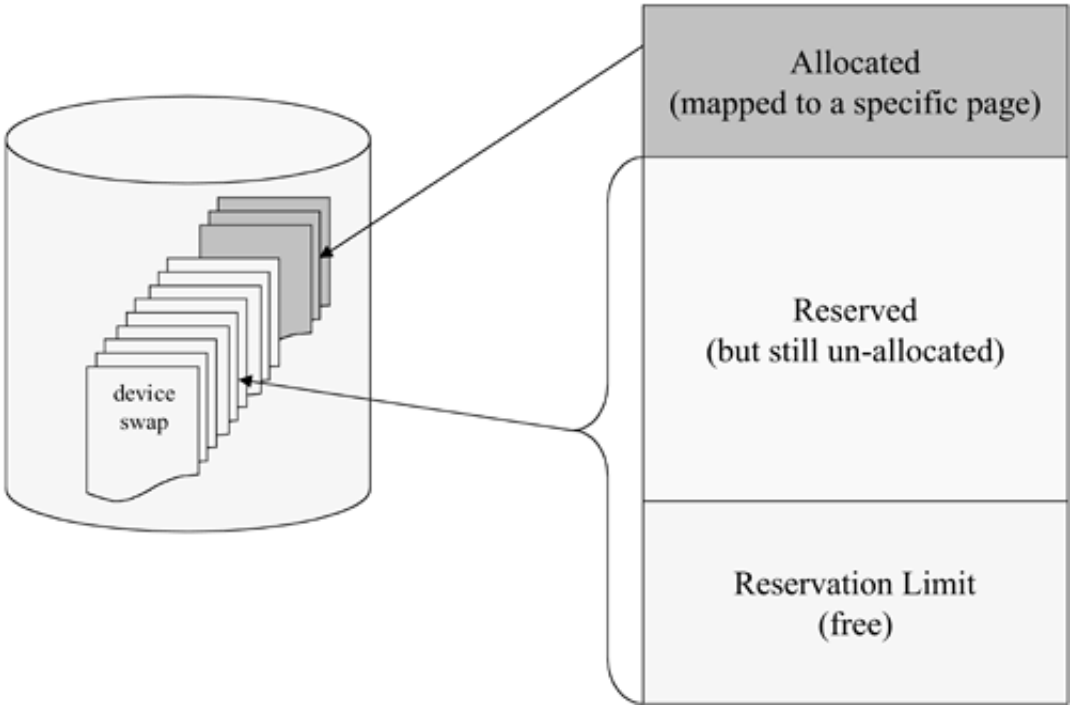
When a process requests permission to become active via the **`fork()`** or **`vfork()`** system calls, the system calculates the total number of swap pages this process might need. It uses a worst-case calculation and includes room for all of the process's data pages, **`region`** page lists, **`uarea`**, and any other process-specific writable memory pages. This number of pages is then subtracted from the system's swap reservation limit. In this way, HP-UX performs prereservation and postallocation of swap.

To understand this approach, think of a fancy restaurant. You plan to bring a party of 12 to the restaurant for dinner on Friday night at 7:00 p.m. If you wait until Friday and simply show up with your group, you may have to wait a long time for seating. Although you may eventually get to eat, your enjoyment of the experience may be adversely affected by the long wait. To avoid customer dissatisfaction, the restaurant takes seating reservations ahead of time. You call the restaurant on Tuesday to make a reservation for you and your party on Friday. When you call the maitre d' and request seating for Friday, he or she checks a list and lets you know if the restaurant can accommodate you at that specific time. If it can, the maitre d' takes your name and confirms the reservation.

Now let's think about this process a bit. Does the maitre d' know exactly where you will be seated? Does he or she know what you will order or how long you plan to stay? The only thing the maitre d' can assume is that the restaurant's resources—tables and chairs, capacity of wait staff and kitchen—will need to accommodate your party. The establishment determines a reservation limit dependent upon its capacity. When you actually arrive at the restaurant, the decision is made to allocate specific resources to you as they are required. Once you are seated, the space remains yours until your evening is finished. If you get up to have a spin on the dance floor or go to the powder room, the resources assigned to you and your party are maintained. When you pay your check and leave, the resources are collected and made available for the next party. If you hadn't shown up on Friday, or if your party ended up being only 8 instead of 12, no resources were wasted; they simply remained available to accommodate other customers as required.

This is a fairly decent analogy of the HP-UX swap reservation policy. Once a page has been allocated to hold a specific page, it is retained for this purpose until the process exits, then it is returned to the swap map as a free page. In Figure 7-4, we see the relationship between swap space, reservation, and allocation.

Figure 7-4. Reserved Versus Allocated



At boot time, the reservation limit is set to the total amount of configured swap space. When the system is asked to `fork()` a new process, its potential swap requirements are subtracted from the swap reservation. If this causes the limit to go negative, the `fork()` fails and the error message returned will indicate the failure was due to insufficient swap.

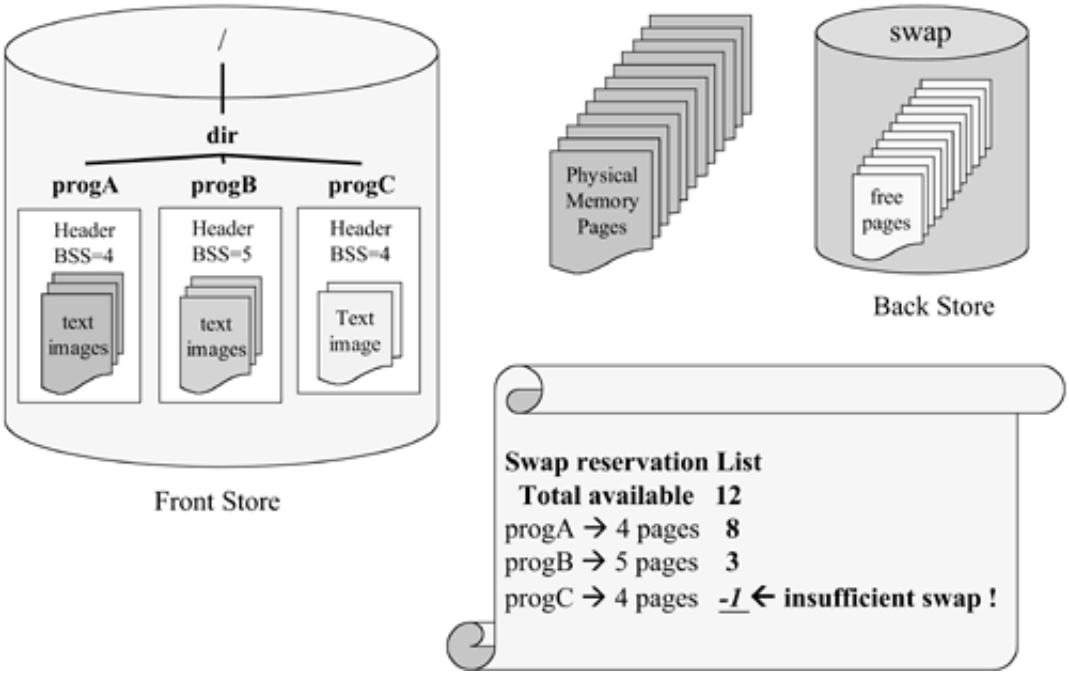
If the `fork()` succeeds, the reservation limit is adjusted and the process is initialized. At this point, no pages have actually been allocated, only reserved. On a system with ample physical memory, it is quite possible that the process will never have a swap page allocated. This is the ideal, as it means system memory pressure never triggered the paging system.

When a page needs to be swapped for the first time, the kernel allocates a page from the swap map and pages the in-core page to the newly allocated swap page. When this occurs, the allocated count is incremented and the reserved count is decremented; the reservation limit remains unchanged. The swap reservation limit plus the number of reserved and allocated pages should always add up to the total amount of swap configured on the system. Once a page allocation has been made, it is maintained until the process is terminated.

A Simple Swap Example

Figure 7-5 demonstrates a simple swap scenario: three programs would like to run in our model. The model is configured with 12 pages of physical memory and 12 pages of swap space.

Figure 7-5. A Simple Swap Example



The swap reservation limit is 12 when **progA** asks to run. Its swap requirement is four pages for data; 12 – 4 = 8, so the reservation limit is adjusted and the process is given the green light.

When **progB** asks to run, its swap requirement is determined to be five pages. The reservation limit is checked: 8 – 5 = 3, so this process is also given the green light.

When **progC** asks to join the club, it is denied permission to run. Its swap requirement is four pages. Since 3 – 4 = –1, this would result in a negative swap reservation limit, so the `fork()` fails.

You might ask why we don't just let all three run. The total memory requirement appears to be 21 pages, and we have a total of 24 pages counting physical memory and swap. If we returned freed pages to the swap map as they are paged back into core, we could get by. The approach HP-UX takes is on the conservative side but assures reasonable performance. Stopping to allocate a swap page is a timely operation, and we don't want to do this more than once for a page. If your kernel is scheduling page-outs, HP-UX chooses a strategy that allows us to minimize the overhead of the paging system at a time when performance is already being compromised by high memory pressure.

Another issue is that we don't want to find ourselves in the situation where a page needs to be swapped and the system is simply out of swap. When this happens, a kernel either has to panic or at the very least kill the process that is causing the memory pressure. The HP-UX approach is to never allow this scenario by simply assuring that sufficient swap space is configured before allowing a process to start. With the current cost of disk storage, large swap space should not be considered a major issue.

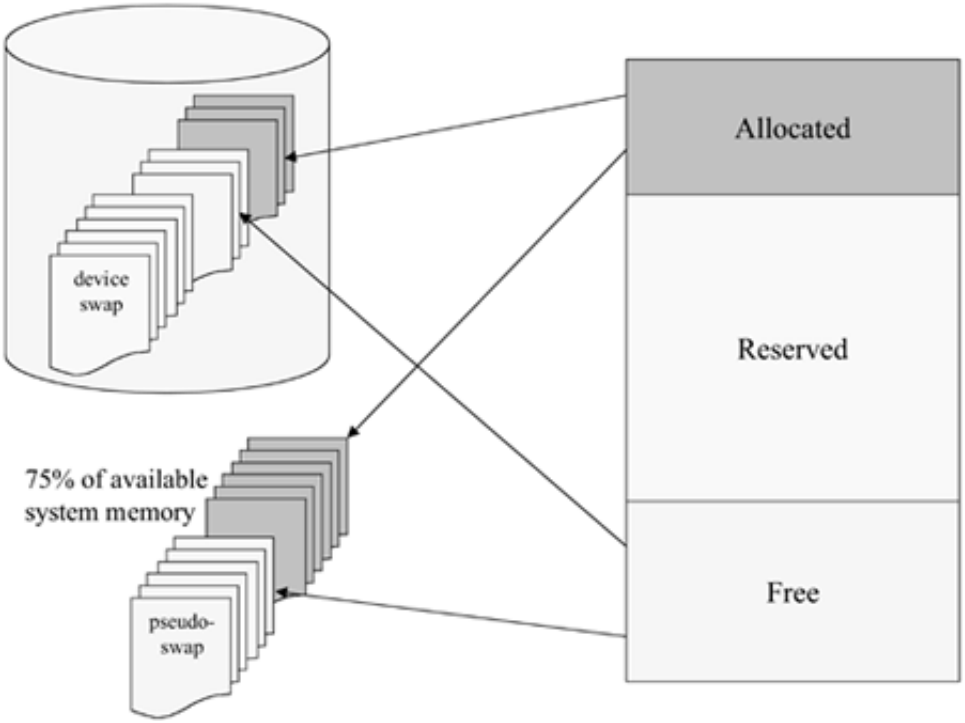
Let's revisit our simple model. Suppose you decided to double the amount of physical memory on the system so that you would never have to swap. This sounds like a good idea and should guarantee optimum performance as far as memory pressure is concerned. The problem arises when we work through the model: if we don't also increase our swap space we won't be allowed to schedule all three processes, since we will still run out of swap reservation space before we can launch **progC**. In our model is seems simple to adjust the swap space, but in the real world and with the large configurable memory options available today, this isn't always practical.

On early versions of HP-UX, when you wanted to increase the physical memory to eliminate the need for swap, you still had to provide the swap space to satisfy the swap reservation scheme. Customers did not want to buy swap space that might not even be used, and who could blame them? In those days, disk space was a much more valuable commodity. One approach would have been to simply turn off swap prereservation, but this would lead to the problems discussed earlier. The answer arrived at by the HP-UX kernel designers is a feature called *pseudo-swap*.

Pseudo-Swap

The pseudo-swap approach is one of the least understood features of the HP-UX kernel; hopefully, we can make you comfortable with its purpose and function. Pseudo-swap is enabled in the HP-UX kernel by setting the kernel parameter **swapmemon** to 1 (this is the default for freshly installed systems). With pseudo-swap enabled, the swap reservation limit is adjusted to include all physical swap plus 75 percent of the system's available memory (87 percent on the latest release). This has been described as "using physical memory for swap space." While there is a basic truth to this statement, it doesn't tell the whole story. Although pseudo-swap may sound a bit strange at first, it provides several benefits over simply turning swap reservation off. Figure 7-6 illustrates the addition of pseudo-swap to the mix.

Figure 7-6. Adding Pseudo-Swap to the Mix



The addition of pseudo swap allows the actual swap space to be reduced to as little as 25 percent of the physical memory. That's not to say that you *should* reduce it—just that you *can*. If the system is configured with ample physical memory, then there is no danger in reducing the size of the swap map. If, however, memory pressure increases and we don't have actual swap space to handle it, things can get a bit interesting. Consider the following scenarios:

First scenario: We have enough physical memory for all the system and user needs. Pseudo-swap is turned on, and device swap is reduced to 25 percent of the available memory. No problem—this should work just fine.

Second scenario: You have maximized your physical memory, but your processes simply want more! It's easy: configure the amount of swap your processes need. In this scenario, we recommend that you leave pseudo-swap disabled, as you would really prefer to have

device swap fulfill the system's paging requirements.

Third scenario: Originally we had sufficient memory, turned on pseudo-swap, and reduced the physical swap to a minimum. The challenge is that we have since increased our demand on memory by adding more processes, but we haven't upgraded the memory or the swap space. What happens when we start using the pseudo-swap to satisfy our system needs?

When a process starts, its swap space requirements are calculated and reserved. As long as there is device or file system swap space available to cover this reservation, all is fine. Once we exceed the physical swap limit, we begin to allocate from the pseudo-swap area. As swap is reserved for a new `region`, if it falls within pseudo-swap, then its pages are not reserved; instead they are allocated right form the start. In either case, this has the same net effect on the reservation limit but the dynamics of this **`region`** with respect to **`vhand`** are quite different.

The mechanics of pseudo-swap are that if a swap space cannot be reserved against the available physical swap during the creation of a memory region, then it is flagged as being preallocated in pseudo-swap (this is actually noted in the **`pregion`**(s) pointing to the **`region`**). On the surface, this almost sounds logical! We need to make room in memory, so we move a page to swap, but if the swap page we move it to is in memory (pseudo-swap), we haven't gained anything—we haven't increased the **`freemem`** count. This is the case, but the important issue is that swap didn't fail. We just didn't accomplish the desired goal this time, so hopefully `vhand` will continue and try to steal a page with an actual reserved or allocated back-store page.

Once we start allocating from pseudo-swap, in effect the number of pages available for paging out is reduced. If a page is mapped by a `region` that has been marked as a pseudo-swap `region`, then all attempts to swap one of its pages simply result in the page remaining where it is. Conceptually, the swap would be successful but at a net-sum-gain of zero as far as increasing the value of **`freemem`** is concerned. To avoid wasting **`vhand`**'s time, `regions` identified as containing pseudo-swap pages are not eligible for inclusion in the **`vhand`** algorithms. When **`agehand`** or **`stealhand`** points to the **`pregion`** of a pseudo-swap **`region`**, **`r_swapmem`**, it simply moves on to the next **`pregion`** in the chain.

As pseudo-swap is allocated, it has the effect of reducing the number of physical pages that may be paged out in times of high memory pressure. A process lucky enough to have its pages in pseudo-swap regions will continue to operate with little impact because its pages are, in effect, locked in memory. But those processes resident in the remaining pages will suffer a performance hit because their pages must carry the extra burden of being at risk of being aged and stolen by the paging system.

Interpreting the Output of **`swapinfo`**

If you run the **`swapinfo`** command and have pseudo-swap enabled, you will see a line that reports the amount of memory swap available and the amount currently in use. Don't be surprised if pseudo-swap usage is reported even when no physical swap has been allocated. Since pseudo-swap is letting us cheat the limit, the system must be able to adjust this cheat as the kernel takes dynamic pages out of circulation, effectively reducing the available memory count. Therefore, the system reports these kernel-allocated pages as allocated pseudo-swap pages. This effectively reduces the swap reservation limit available and keeps the system from overcommitting its memory resource.

Pseudo-swap works well for its intended purpose, but if you push it past the limits, you will have slower overall system performance—but everything will still run. Don't be afraid of pseudo-swap, but make sure you understand its purpose.

Device Swap

Device swap is dedicated physical or logical disk volumes. These raw volumes are configured for exclusive use by the swap system. If logical volumes are used, you should attempt to have only one swap space per physical disk controller. It is much better to have one large volume than two smaller ones sharing the same spindle.

It is a common practice to also configure swap devices as dump devices; this may be done either through inclusion in the kernel's "system" file when the kernel is being built or through lines placed in the */etc/fstab* file. The use of the */etc/fstab* file gives you more control and makes it easier to modify your configuration. If a swap area is added to */etc/fstab* it may be activated by issuing the swapon –a command. If a swap area definition is being removed from */etc/fstab* a system reboot is required to halt its use. There is no *swapoff* command for HP-UX.

File System Swap

As the name suggests, we may also allocate swap space from swap files reserved on mounted file systems. The swap files are sized in alignment with the swap chunk size (the swap chunk is the basic unit used to track swap in kernel data structures, we will discuss it in detail shortly) and may be dynamically allocated as needed. When a file system swap space is initialized, a minimum and maximum amount of space may be configured. A reserved free space limit for the file system may also be specified.

Initialized file system swap appears as a directory named *paging*, which contains individual swap files. The swap file naming convention is somewhat interesting. The root of the name is the system's hostname followed by a period (.) and the swap chunk number this file represents in the system's swap table. While file system swap spaces are fully functional, they do present a couple of concerns. When a file system is 100 percent full, its performance becomes sluggish, and a file system with a paging directory and swap files may not be unmounted. If the host file system employs features such as VxFS logging, this could further impact swap performance. Keep these issues in mind before deciding to use this feature.

It is entirely possible to allocate file system swap on a remotely mounted Network File System (NFS), but this is generally not a good idea. The extra overhead incurred in the network combined with the stateless nature of an NFS mount makes for very bad chemistry! Besides, the NFS server's administrator may not appreciate your chewing up their disk space to satisfy your swap requirements (such connections are readily apparent, and with the naming convention used, it is a simple matter to determine which remote host has configured the remote swap).

Swap Priority

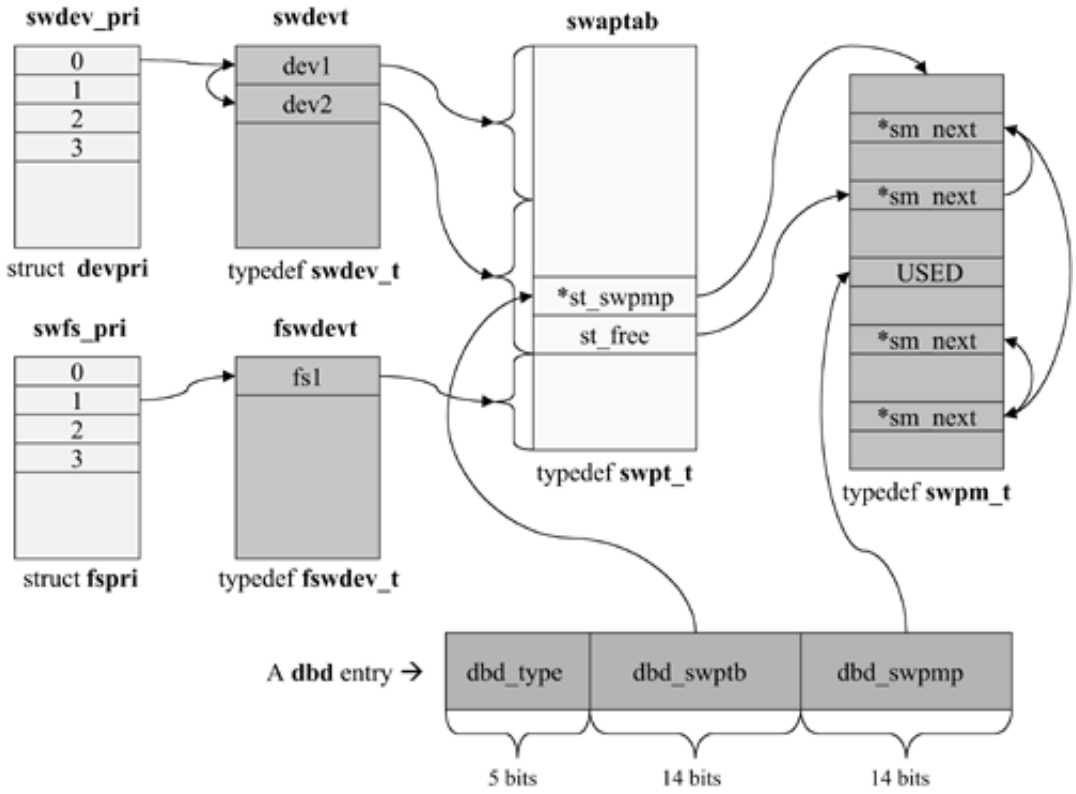
Both types of swap, device and file system, may be assigned a priority in the range of 0 to 10. Priority 0 devices are used first, and 10 are used last. The primary boot swap device is assigned a priority of 1, and if the priority is not specified to the `swapon` command, it assumes a default priority value of 1 as well. Device swap takes precedence over file system swap at the same priority, but file system swap at a stronger priority takes precedence over device swap at a weaker priority.

When two or more device or file system swap areas share the same priority, space is allocate from them in a round-robin fashion. This round-robin scheme is employed at the swap chunk level, not at the individual page level.

Tracking Swap in the Kernel Structures

In Figure 7-7, we see the relationship between the various kernel-resident data structures required to manage the swap space.

Figure 7-7. Swap Kernel Structures



As we see in the diagram, the hambone is connected to the knee bone. Swap management starts with entries in the swap device table, **swdevt**, and the file system device table, **fswdevt**. Pointers from the two priority arrays are used to order the searching of the swap devices. A swap device is broken down into individual swap chunks. The kernel tunable **maxswapchunks** sets the systemwide number and determines the size of the swap table. Each swap chunk is sized by the kernel-tunable **swchunk**, which defaults to 2 MB, enough room for 512 page-outs.

Individual pages have entries in the chunk's swap map. This map also contains a linked free list for its pages. The swap maps are pointed to by entries in the system swap table, **swaptab**. The swap table is filled in the order the swap devices were enabled but is searched according to the order implied by the priority table pointers. If two devices share a priority, their chunks are searched in round-robin fashion.

Let's examine these structures in Listings 7.1 through 7.6.

Listing 7.1. `q4> fields struct devpri`

```
These structures are maintained in the array swdev_pri[NSWPRI]
(default value is 11)
Pointer to the first swap device at this priority
 0 0 4 0 * first
Pointer to next device at this priority to allocate from
 4 0 4 0 * curr
```

Listing 7.2. `q4> fields struct fspri`

```
These structures are maintained in the array swfs_pri[NSWPRI]
(default value is 11)
Pointer to first file system swap at this priority
 0 0 4 0 * first
Pointer to next swap area at this priority to allocate from
 4 0 4 0 * curr
```

Listing 7.3. `q4> fields swdev_t`

```
The swap device number
  0 0 4 0 int  sw_dev
The swap device flags (i.e. SW_ENABLE)
 4 0 4 0 int  sw_flags
The Kbyte (DEV_BSIZE) offset to the beginning of the swap area
on the disk device
 8 0 4 0 long sw_start
Number of blocks on the device
12 0 4 0 long sw_nblksavail
Number of blocks enabled for swap
16 0 4 0 long sw_nblksenabled
Number of free pages
20 0 4 0 int  sw_nfpgs
Swap priority for this device
24 0 4 0 int  sw_priority
First swap table entry for this device
```

```
28 0 4 0 int  sw_head
Last swap table entry for this device
32 0 4 0 int  sw_tail
Pointer to next swap device sharing the same priority
36 0 4 0 *    sw_next
```

Listing 7.4. `q4> fields fswdev_t`

```
Pointer to next file system swap area with the same priority
 0 0   4 0 *         fsw_next
The status flags
 4 0   4 0 int       fsw_flags
Number of free swap pages
 8 0   4 0 int       fsw_nfpgs
Number of blocks allocated
12 0   4 0 long      fsw_allocated
Minimum number of preallocated blocks
16 0   4 0 u_long    fsw_min
The block allocation limit
20 0   4 0 u_long    fsw_limit
The block reservation limit (File System swap equivalent to
minimum free space)
24 0   4 0 u_long    fsw_reserve
Priority for this file system swap space
28 0   4 0 int       fsw_priority
Pointer to the vnode for the file system's mount point
32 0   4 0 *         fsw_vnode
The underlying file system's block size
36 0   4 0 u_int     fsw_bsize
This swap space's first swap table entry
40 0   2 0 short     fsw_head
This swap space's last swap table entry
42 0   2 0 short     fsw_tail
The directory path name for the underlying file system's mount
point
44 0 256 0 char[256] fsw_mntpoint
```

Listing 7.5. `q4> fields swpt_t`

```
Index to the first free swapmap array entry

 0 0 2 0 short st_free

Index of next chunk for same device or file system swap area

 2 0 2 0 short st_next

Status flags (ST_INDEL|ST_FREE|ST_INUSE)

 4 0 4 0 int   st_flags

Pointer to swap device

 8 0 4 0 *     st_dev

Pointer to swap file system

12 0 4 0 *     st_fsp

Device of file system chunk vnode

16 0 4 0 *     st_vnode

Number of free pages on the device

20 0 4 0 int   st_nfpgs

Pointer to a swap maps starting address

24 0 4 0 *     st_swpmp
```

Listing 7.6. `q4> fields swpm_t`

```
Number of kthreads using this page

0 0 2 0 u_short sm_ucnt

Index to first free entry in this swap map

2 0 2 0 short   sm_next
```

As a final bit of discussion, do you see the slight-of-hand trick played by the disk block descriptor data? How can the 28-bit field in the **`dbd`** point to a specific device and an offset on the device. Device numbers are 32 bits long by themselves, and the block address on a modern disk may be quite large. The smoke and mirrors employed here are several levels of indirection. The upper half of the **`dbd`** data, **`dbd_swptb`**, points to the appropriate swap table entry. Here we pick up **`st_dev`**, the device number, and **`st_swpmp`**, the pointer to this chunk's swap map. Next, the **`dbd dbd_swpmp`** is the page offset into the swap chunk.

This means that currently no more than 2^14 swap chunks may be configured on a system and that each chunk may hold only 2^14 pages at most. If we do the math, this limits the maximum device swap space to:

2^14 * 2^14 * 4096 or 2^40 or 1 TB

Summary

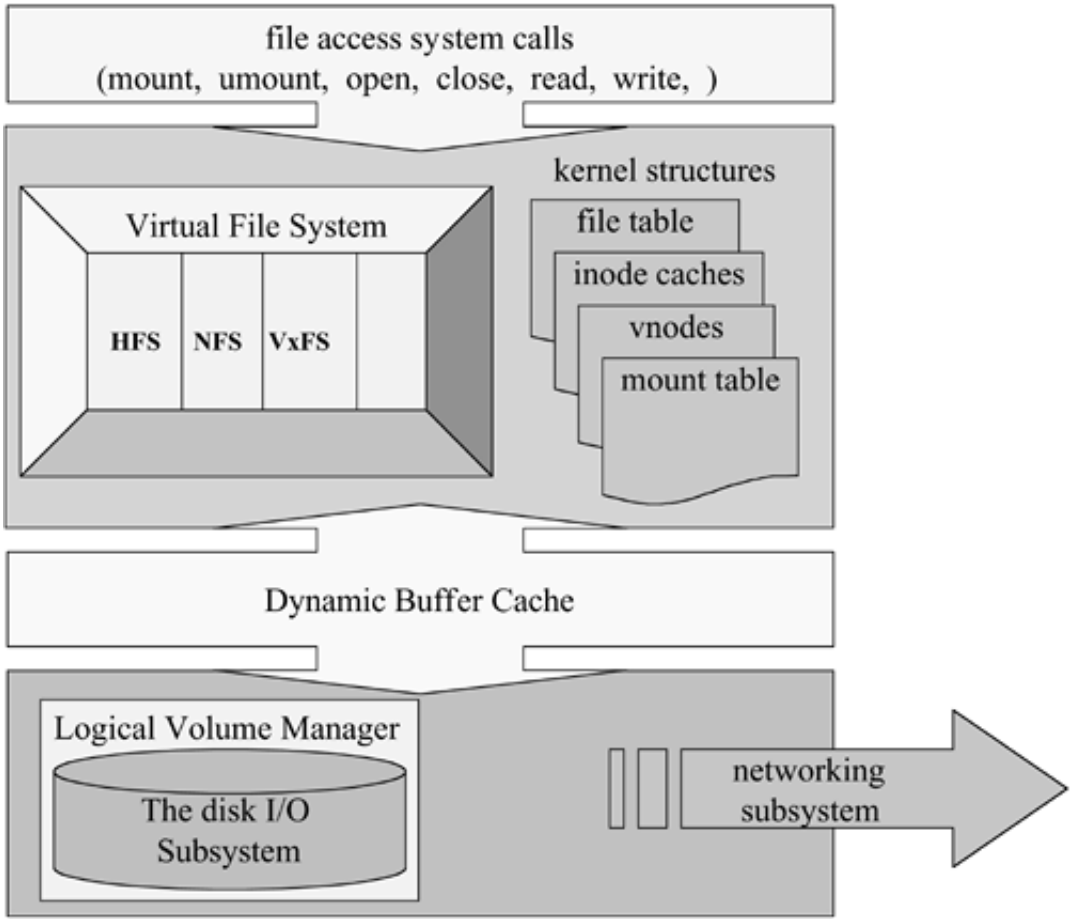
As we have seen, the HP-UX swap reservation policy in combination with the kernel-tunable `swapmemon` feature allows the administrator control of the paging dynamics of the system. Swap is a necessary evil in many cases; it should be configured in alignment with the available memory resources and the anticipated needs of the system's process loads.

There is another type of memory that we haven't discussed yet: *secondary* or *non-core* memory. We are referring to our old friend, rotating magnetic disk memory. Disks provide nonvolatile memory for short-term and long-term storage. The next stop on our tour of HP-UX internals is an in-depth look into the kernel file management subsystem.

Chapter 8. Files and File Systems

The file system is a basic workhorse component of an operating system. Figure 8-1 shows the big picture of the file management subsystem in the HP-UX kernel. Our study is divided into three phases: first we examine basic file system concepts, then we focus on the design and implementation of the classic UNIX File System (UFS), and finally we examine the kernel-resident structures utilized to keep track of mounted file systems.

Figure 8-1. The File Management Subsystem



Not all applications store data of the same type or follow the same data utilization modes. Over the years, a variety of file system types have been created to meet the needs of programmers, improve dependability, and increase performance. To simplify the interaction between a specific file system type and the rest of the kernel, an abstraction layer known as the *virtual file system* (VFS) has been implemented. As we study the file management subsystem, be on the lookout for transition points between VFS structures and implementation-specific structures.

File System Concepts

Before we begin our study of the specifics of file systems and HP-UX, we need to spend a little time thinking about what a file system really is.

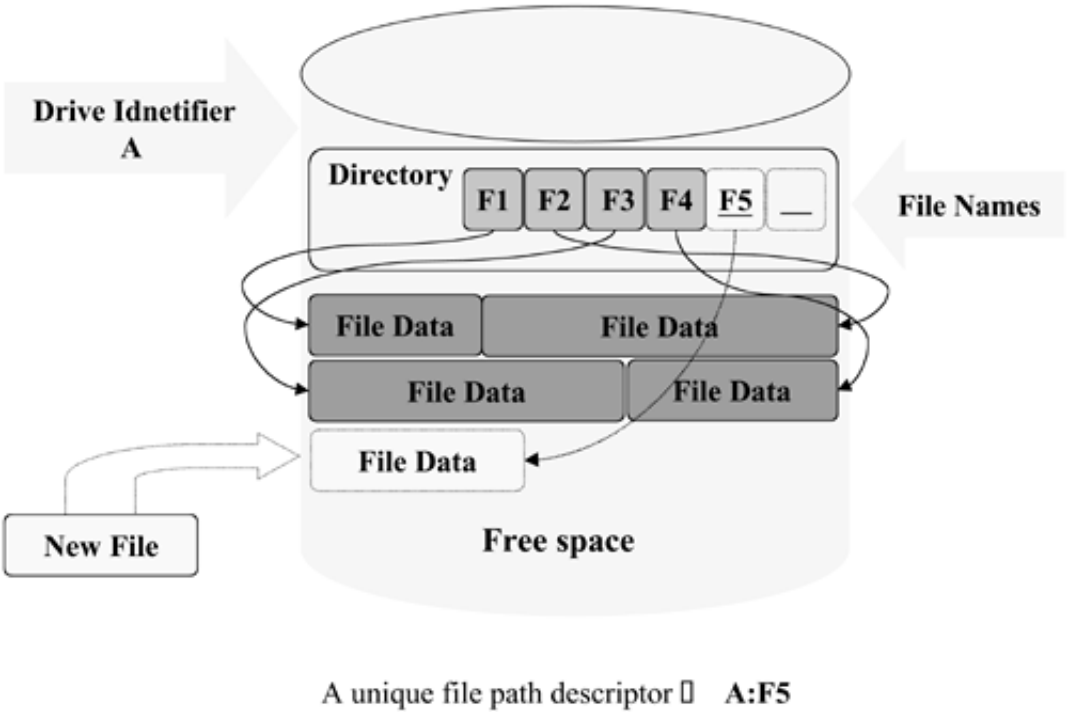
The Challenge

Since the beginning of computer operating systems, there has been a need to store program instructions and data in some type of nonvolatile memory so that they may be recalled and loaded into the system's RAM when needed. Very early systems stored information by punching patterns into paper tape or "keypunch" cards. This worked but was very slow and added the tasks of storage, collation, and labeling to the growing list of duties for the computer operator. When rotating magnetic storage mechanisms were first created (early cylindrical rotating drums evolved into flat disks with multiple platters), basic storage algorithms were developed.

File Systems: A Simple Approach

The first disk drives had precious little storage space (some of the first floppy drives held only 140 KB of data!), and the storage algorithms employed were quite simplistic by necessity. Figure 8-2 outlines a basic approach.

Figure 8-2. File System Basics: 101



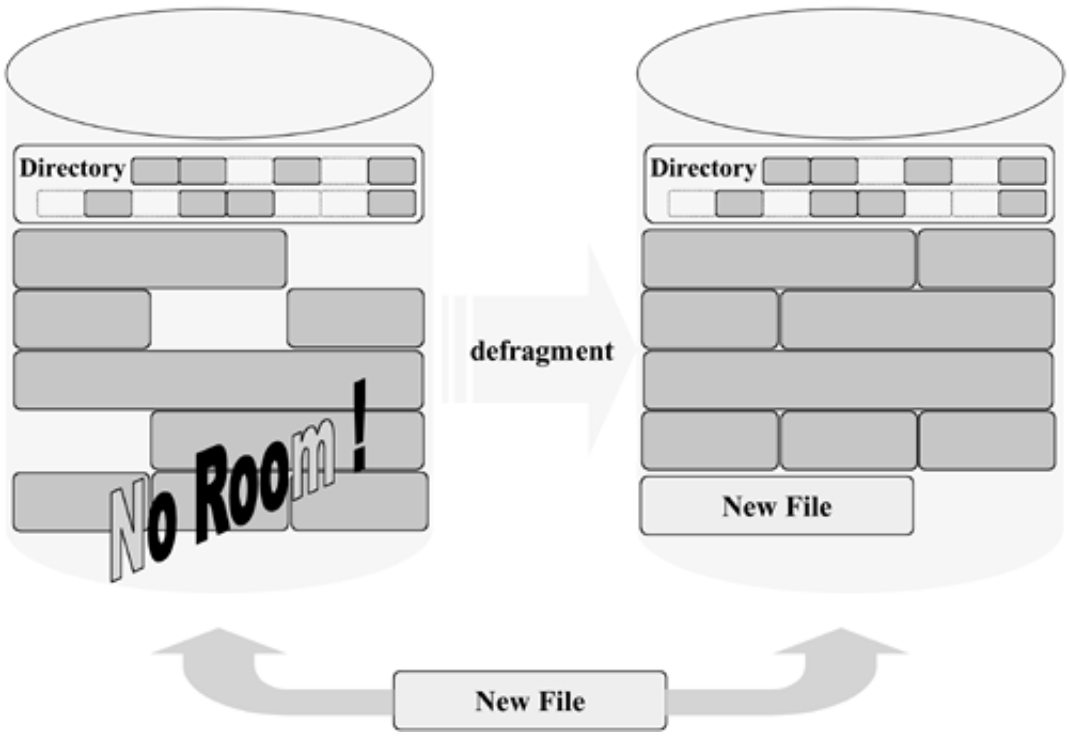
The first step was to come up with a way to identify specific drives under the control of the operating system. One approach was to assign drive letters to each physical drive. Once we can identify a specific drive, we need to be able to locate directory information for the files stored on it. Most early file system designs placed a pointer to the disk's directory data on the first physical sector of the mechanism (regardless of the overall size of a disk drive, they all have a first sector).

These early directory structures were very sketchy in their details. Basically, there was an entry for each file consisting of its name, a starting offset, and the file's overall size. This approach is called a *flat file system*, so named because the file system had a single directory and all of its files were managed by this directory. In addition, all the data making up a specific file was stored in contiguous sectors on the disk. Once you found the beginning of a file, you simply read its data from the disk in a sequential manner until you reached its end. Although this approach is very dated, it still has certain strengths. If a file is to be accessed in a sequential manner or its contents are needed in their entirety, then the contiguous nature of storage in a flat file system provided good performance. A major weakness was the dynamics of what happened when a file was removed from the storage matrix or when an existing file needed to grow.

When a file was removed, it left a hole. New files could be located in the hole as long as their size was equal to or smaller than that of the hole. New files larger than the hole needed to find available free space elsewhere on the disk. If an existing file needed to grow, it could simply extend its size if there was adjacent free space or be moved to an area of larger free space if one was available.

This juggling of data resulted in fragmentation of the available free space within the overall disk. An approach to minimize this problem involved periodic disk maintenance to defragment the disk (see Figure 8-3).

Figure 8-3. Fragmentation of Available Space

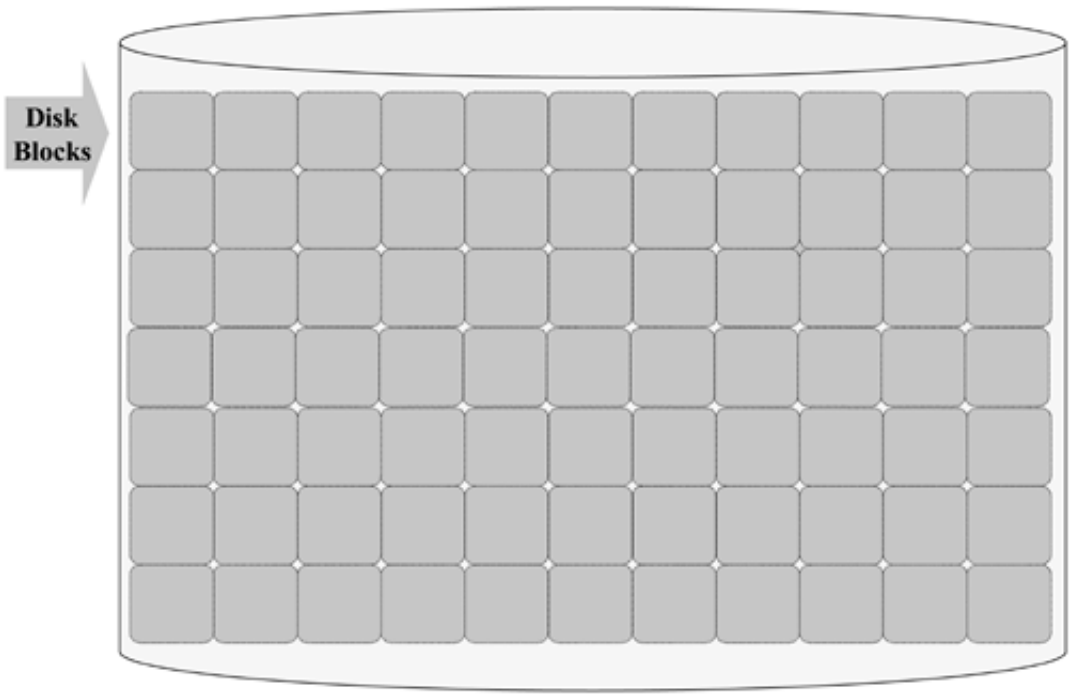


While defragmentation allowed the administrator to coalesce the free space holes into a contiguous free space at the end of the drive, it had to be performed while all the files on the drive were quiescent. This may not seem like much of an issue, but in today's world with 7-by-24 system utilization and +99.999% uptime requirements, defragmentation is a costly option.

UNIX Gets a File System

In the early 1970s, the developers of what was to become known as UNIX faced the need to design and implement a file system for their fledgling operating system. The file system designs in use at the time were mostly flat and required frequent defragmentation. Instead of following the crowd, our fearless UNIX developers chose to think outside the box and make fragmentation their friend. Figure 8-4 shows the first step to this new approach.

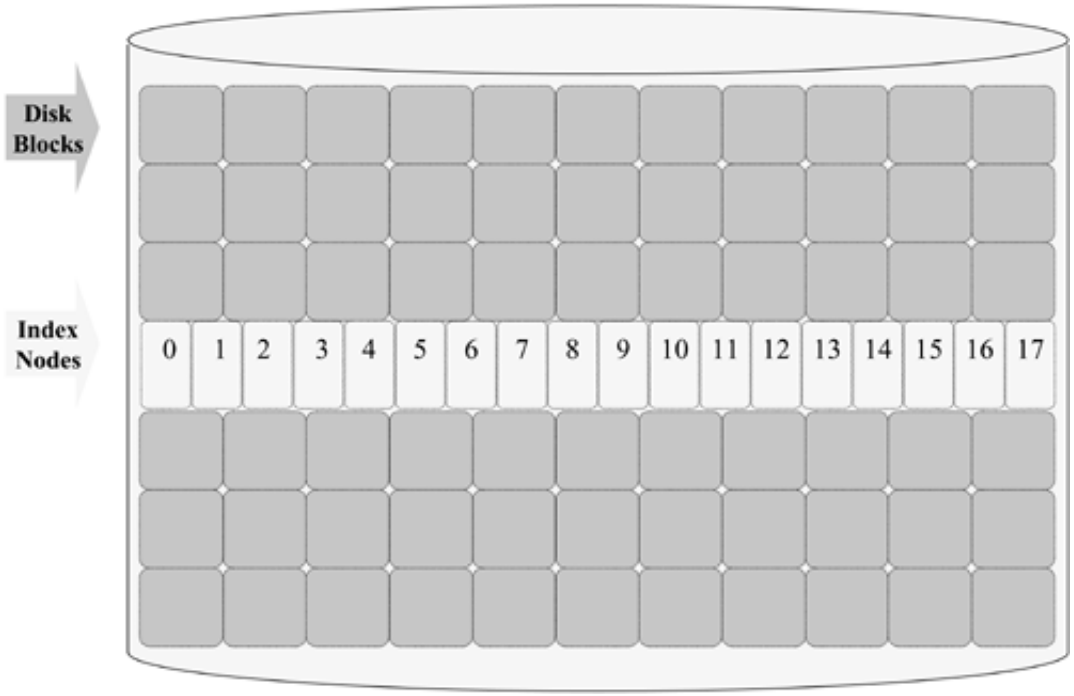
Figure 8-4. Fragmentation Is Our Friend



First, the entire disk is subdivided into uniform *disk blocks* (for the original UNIX file system, the block size was 512 bytes). A file's data is stored in any available disk block. There is no requirement that they be sequential or contiguous. The disk's free blocks are easily managed by the use of a simple bit allocation map. Each block is represented by a bit in the bitmap. If the bit is set, the block is in use; if it is not set, the block is available.

The challenge with this design is to track the specific blocks that belong to a file and the order in which they are allocated. To this end, a section of the disk space is reserved (at the time of file system creation) to hold information about each file (see Figure 8-5).

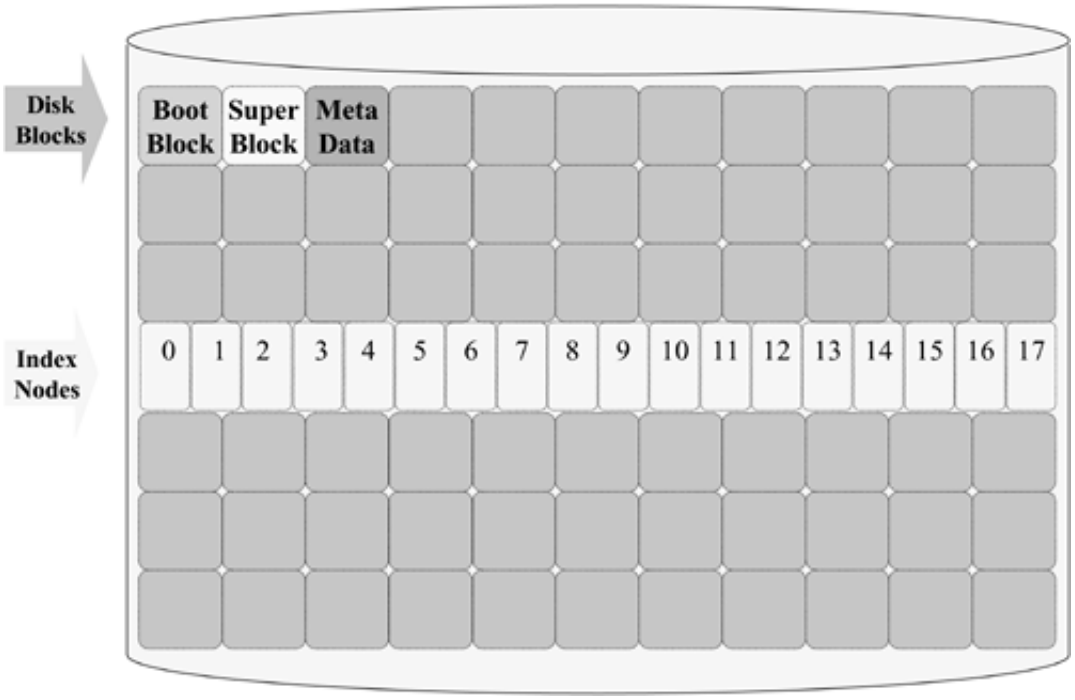
Figure 8-5. Creating an Index



These indexed information nodes (`inode`s) have a fixed size and are arranged in an array located in the middle of the disk's storage space. Since each file reference required accessing the file's `inode` by locating the `inode`s in the middle of the disk, overall disk access times were optimized.

The `inode` array is a prime example of what we call metadata, the administrative overhead required to manage the file system's user data. Additional metadata structures are shown in Figure 8-6.

Figure 8-6. Metadata



As we mentioned, the `inode` table was located in the middle of the disk to improve overall access time; but as not all disk drives are the same size, there needed to be a metadata structure in a well-known location on all disks containing potential file systems. To meet this need, several additional structures were created.

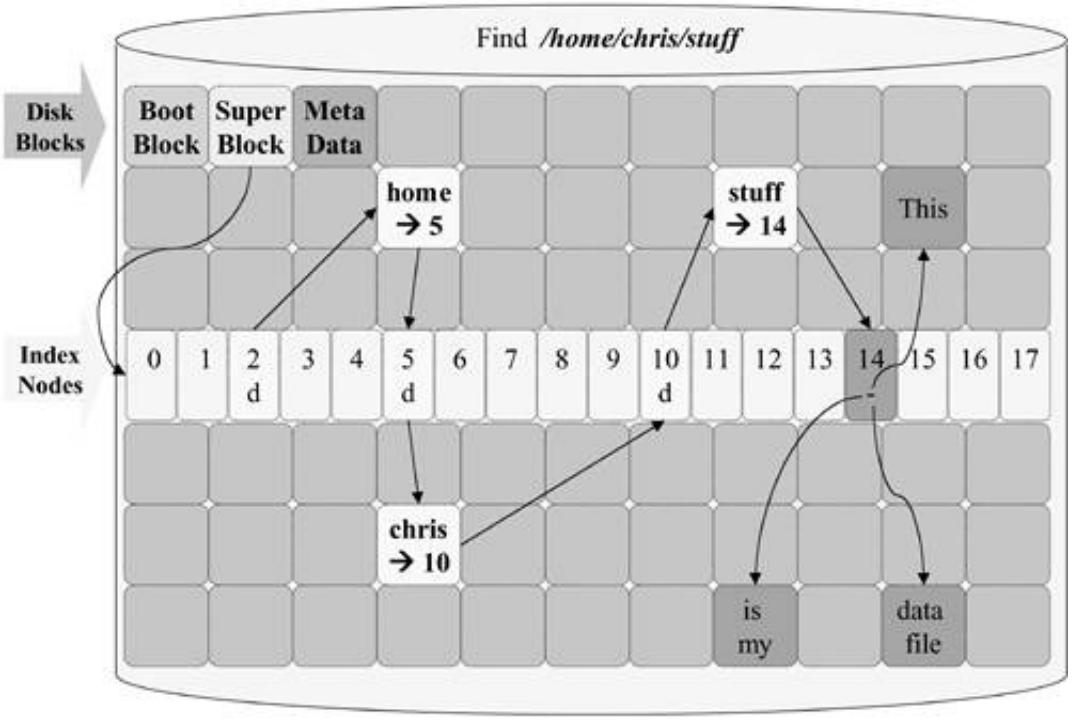
When a machine is booting, it is running from simplified code in its boot ROM. A boot ROM usually has very minimalist driver code and can read only the first block of a device. For this reason, the first block of the disk is reserved as a boot block. Next, a block called the *superblock* is allocated. The superblock contains static and dynamic information about the specifics of the file system on the disk. One of the most important components is a pointer to the starting address (a byte offset) of the `inode` table. Additional structures, such as bit allocation maps, are also assigned and mapped by the superblock. Each type of file system may have its own specific additions to the contents of the superblock but certain key values are located at fixed offsets within the superblock regardless of the file system type. These values are used by the kernel to identify the file system type, version, and basic static configuration data.

Layering a Directory Hierarchy on a Flat File System

Another feature desired by the UNIX developers was the implementation of a hierarchical file system. The file structure was based on a basically flat model (there is only one `inode` table per file system), so a method needed to be developed to facilitate the use of directories and subdirectories. This brought about the creation of a special type of file known as a *directory*. This is an important point: in UNIX, a directory is simply a special type of data file.

Every file on a UNIX system is known by its file system's device number and its `inode` number. Directory files contain a pairing of file names with their `inode` numbers on the local file system. Let's consider what is required to locate the contents of a file with the pathname of `/home/chris/stuff` (see Figure 8-7).

Figure 8-7. Super Blocks, Index Nodes, Directories, and User Data



We start our search with the superblock of the `/` (root) file system. The superblock gives us the offset for the start of the `inode` table. As a standard, all file systems use `inode` 2 to point to their root directory data blocks. The size of an `inode` is fixed, and the first `inode` has an index of 0. Therefore, if you know an `inode` number, you may easily calculate its location on the disk volume. To find `inode` 2, you simply take the `inode` table offset from the superblock and add *2 * inode-size-in-bytes* and you have the byte offset from the beginning of the file system.

Examination of the root directory reveals that the *home* directory file's `inode` is 5. We return to the `inode` table and access node 5 and follow the block pointer to its data block. This reveals the `inode` number for the *`chris`* subdirectory. We then access node 10 and locate the *`stuff`* directory block pointer. From this block we learn that `inode` 14 contains the attributes of the *`/home/chris/stuff`* file. Now we simply have to follow the various block pointers in this `inode` to access the file's data.

You may have noticed that the `inode` contains pointers to disk data blocks and a type flag to let us know whether the next step of our scavenger hunt will lead us to a directory file or a regular file. The actual specifics of the `inode` data structure differ from one file system type to another. Next, we study the specifics of the Hierarchical File System (HFS) as implemented by the HP-UX operating system.

PREV < Day Day Up > NEXT

The New and Improved UNIX File System

The early history of UNIX development consisted of two competing flavors of UNIX: the original AT&T/Bell Labs release and the Berkeley System Definition. The original code from AT&T could not be sold, as antitrust laws didn't allow AT&T to compete in this arena, but the company was allowed to provide copies of its operating system code for free, and many universities took them up on the offer. The University of California, Berkeley, became a hotbed of UNIX development efforts, and one of the first areas it addressed was the file system.

The original UNIX file system was unique in its approach to file data management but had several weak points in its design. Among them was its dependency on the all-important superblock. If this key structure was corrupted, then all access to data on the file system was lost. In addition, the design allowed the blocks of a single file to be spread across the entire disk space with no attempt to keep related blocks in any type of order. This played well when the kernel was trying to figure out where to put the next block of data, but the price had to be paid in performance when a file had to be accessed in its entirety (file copies, backups, etc.).

As disks got bigger, the tendency was to increase the file system block size to keep the management overhead to an acceptable limit. This was fine as long as you didn't have lots of very small files. A file with only a small number of bytes would require an entire block for its storage. This could result in an inefficient utilization of disk space.

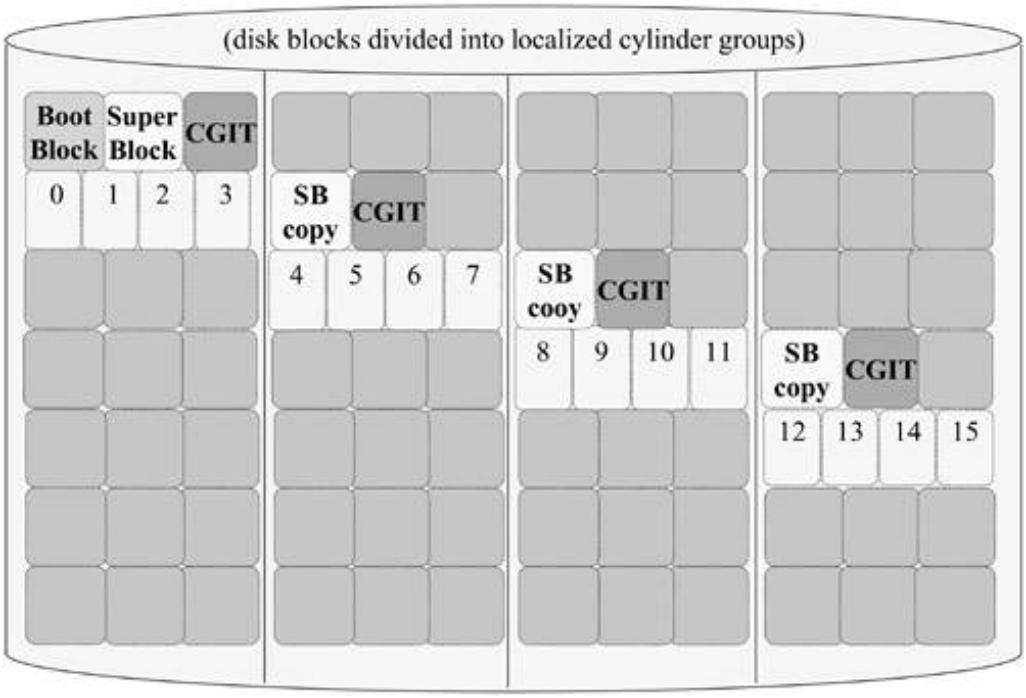
The Berkeley File System (BFS) attempted to address these and other performance issues. For the record, the following names all refer to the same file system implementation:

- High-performance File System or Hierarchical File System (HFS in HP-UX)
- Berkeley File System
- Fast File System (FFS in SYS V.4)
- McKusick File System
- UNIX File System (UFS)

HP-UX's HFS

The HFS has long been the main file system for the HP-UX kernel, and indeed for PA-RISC-based systems, the kernel image may be loaded only from HFS. Although recently there are contenders to HFS's position as the HP-UX default file system, we study it here, as it sets the baseline for HP-UX file system features and performance comparisons. Figure 8-8 illustrates a high-level view of this model.

Figure 8-8. The HFS File System Layout



File *access time* depends on a couple of basic factors: first, the average *seek time*. Assume that a disk has 1,000 distinct tracks. The amount of time it takes to move the disk head from one track to another is the seek time. Sometimes the data you are seeking is one track away and sometimes it might be 1,000 tracks away. The average seek time is the time it takes to reposition the head across ½ of its total track count (500 in our example).

The other factor affecting disk access time is *rotational latency*. Consider a rotating disk platter with a number of sectors located around each track. The sector you require might be the next to come under the head, or it might have just passed by, in which case you will have to wait for a complete revolution of the disk. As such, the average rotational latency is the time it takes for half a revolution of the disk media.

From a designer's point of view, the only way to reduce the rotational latency is to get a faster disk (you may have noticed the trend from 5400 rpm disks to 10,000 rpm or higher in the PC market recently). The disadvantages of speeding up a drive is that the tolerances of the heads must be increased and that speed equals heat—the faster the drive spins, the more heat dissipation the hardware designer has to deal with. These two issues add up to increased cost. As far as the seek time is concerned, the trend is to increase the number of tracks, not decrease them, as hardware designers are constantly trying to increase the capacity of their drives.

This is where the file system designer chose to implement a "soft" solution to the challenge. By dividing the disk into multiple cylinder groups and allocating `inode`s and their related data blocks within a cylinder group whenever possible, the average seek time may be greatly improved. Each cylinder group contains a portion of the `inode` table, a redundant copy of the superblock (the `fsck` utility can use one of these copies to recover a corrupted super block), and localized metadata to manage and monitor utilization of the local cylinder group space.

Reconsider our example, assume that the whole disk was divided into 100 cylinder groups. Each cylinder group (CG) would be mapped to 10 adjacent tracks. While the rotational latency would be a constant the all the individual blocks containing a file's data would be located within a 10 track area. For access to this single file the average seek time would be reduced from the time to move 500 tracks to the time to move 5 tracks! Simply by following these conventions for the placement of a files data we could see a vast improvement in access time. We realize that this is an over-simplified explanation but it points our thinking in the right direction.

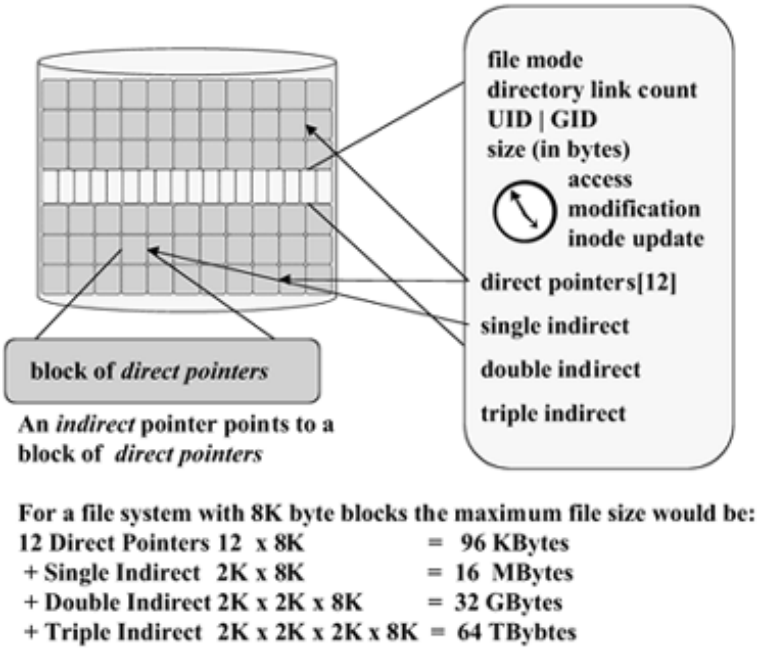
Direct and Indirect Block Pointers

You may be wondering how a fixed-size `inode` can point to all the data blocks of files of varying sizes. If the block size is fixed and the `inode` size is fixed, it seems that there would be an implied maximum file size. To work around this issue, the HFS `inode` employs four types of block pointers:

- Direct pointers (12 per `inode`)
- Single indirect pointers (one per `inode`)
- Double indirect pointers (one per `inode`)
- Triple indirect pointers (one per `inode`)

The original AT&T file system had a 512-byte block, and the `inode` was 128 bytes and had room for 15 block pointers (each pointer was 32 bits). Let's examine the limits of this model (Figure 8-9).

Figure 8-9. The `Inode`



There are 12 direct pointers that locate the first 12 blocks containing file data, allowing for 12 * 512, or 6 KB of storage. If the file grew beyond this size, then the 13th pointer (the single indirect pointer) was used to point to a data block that could hold additional direct pointers (a block size of 512 meant that an indirect block would hold an additional 128 pointers). The 14th pointer (the double indirect) could point to a block containing additional single indirect pointers, and the 15th pointer (the triple indirect) could point to a block containing additional double indirect pointers. In this manner, a fixed-size `inode` could map a very large number of file data blocks.

With the original 512-byte block size, we get the following capabilities:

Direct pointers →	12 ∞ 512	= 6 KB
Single indirect →	128 ∞ 512	= 64 KB
Double indirect →	128 ∞ 128 ∞ 512	= 8 MB
Triple indirect →	128 ∞ 128 ∞ 128 ∞ 512	= 2 B

With the more modern 8-KB block size, we get the following:

Direct pointers →	12 ∞ 8 KB	= 96 KB
Single indirect →	2 KB ∞ 8 KB	= 16 MB
Double indirect →	2 KB ∞ 2 KB ∞ 8 KB	= 32 GB
Triple indirect →	2 KB ∞ 2 KB ∞ 2 KB ∞ 8 KB	= 64 TB

The HFS `inode`

The `inode` is the end-all, be-all metadata structure when it comes to defining a file. All the attributes of a file—its permissions, type, size, credentials, timestamps, linkage counts, and data block pointers—are held in this structure. The only attribute of the file that isn't held in the `inode` is its name! In actuality, a name is an abstraction used to locate a specific file's data and exists to make it easier for humans to reference a file.

By maintaining this degree of separation between a file's attributes (its `inode` data) and its name (directory entry), it is a simple task to create multiple file names pointing to the same `inode`. This is the basis for the creation of hard links (reference the `ln` command).

To study the structure of a HFS `inode`, let's examine the `icommon` structure in the HP-UX kernel (Listing 8.1).

Listing 8.1. `q4> fields struct icommon`

```
inode type (upper 4 bits) and access mode (lower 12 bits)
0 0 2 0 u_short ic_mode
```

0x1000 IFIFO	FIFO or named pipe
0x4000 IFDIR	Directory
0x6000 IFBLK	Block special file
0x7000 IFCONT	Continuation inode
0x8000 IFREG	Regular file
0x9000 IFNWK	Network special file (retired)
0xA000 IFLKN	Symbolic Link
0xC000 IFSOCK	Socket

```
current number of directory links (hard links)to this inode
  2 0  2 0 short     ic_nlink
low 16 bits of the owner's UID and GID
  4 0  2 0 u_short   ic_uid_lsb
  6 0  2 0 u_short   ic_gid_lsb
file size in bytes
  8 0  8 0 long long ic_size
```

```
timestamps (access, inode modification, and creation)
Note: the modification timestamp refers to the last time the
contents of the inode were modified and does not necessarily
mean that the file's contents were changed
 16 0  4 0 u_int     ic_atime_tv_sec
 20 0  4 0 u_int     ic_atime_tv_usec
 24 0  4 0 u_int     ic_mtime_tv_sec
 28 0  4 0 u_int     ic_mtime_tv_usec
 32 0  4 0 u_int     ic_ctime_tv_sec
 36 0  4 0 u_int     ic_ctime_tv_usec
next are the 12 direct block pointers
 40 0  4 0 int       ic_un2.ic_reg.ic_db[0]
 44 0  4 0 int       ic_un2.ic_reg.ic_db[1]
 48 0  4 0 int       ic_un2.ic_reg.ic_db[2]
 52 0  4 0 int       ic_un2.ic_reg.ic_db[3]
 56 0  4 0 int       ic_un2.ic_reg.ic_db[4]
 60 0  4 0 int       ic_un2.ic_reg.ic_db[5]
 64 0  4 0 int       ic_un2.ic_reg.ic_db[6]
 68 0  4 0 int       ic_un2.ic_reg.ic_db[7]
 72 0  4 0 int       ic_un2.ic_reg.ic_db[8]
 76 0  4 0 int       ic_un2.ic_reg.ic_db[9]
 80 0  4 0 int       ic_un2.ic_reg.ic_db[10]
 84 0  4 0 int       ic_un2.ic_reg.ic_db[11]
followed by the single indirect, double indirect, and triple
indirect block pointers
 88 0  4 0 int       ic_un2.ic_reg.ic_un.ic_ib[0]
 92 0  4 0 int       ic_un2.ic_reg.ic_un.ic_ib[1]
 96 0  4 0 int       ic_un2.ic_reg.ic_un.ic_ib[2]
```

If this is a fast symbolic link, the block pointers are replaced with

```
40 0 60 0 char[60]   ic_un2.ic_symlink
```

A fast symbolic link allows the symbolic or "soft link" pathname to be stored in the inode instead of a data block. This is possible as long as the linkage path doesn't exceed 59 characters in length.

```
the status flags
100 0  4 0 int       ic_flags
```

0x01IC_FASTLINK	enable fast symbolic links
0x02IC_LARGEUIDS	enable use of large UIDs

```
blocks held
104 0  4 0 int       ic_blocks
108 0  4 0 int       ic_gen
upper 16 bits of the owner's UID and GID if large UIDs have
been enabled
112 0  2 0 u_short   ic_uid_msb
114 0  2 0 u_short   ic_gid_msb
116 0  4 0 int       ic_spare[0]
120 0  4 0 int       ic_spare[1]
continuation inode number (if needed)
124 0  4 0 u_int     ic_contin
```

As we see from this listing, the size of the HFS `inode` is 128 bytes. If access control lists (ACLs) are being used, the last four bytes in an `inode` point to a *continuation* `inode`, which holds up to 13 additional access control settings. For more on ACLs, reference the manual pages on `chacl` and `lsacl`.

Following Metadata Using `fsdb`

Another useful tool for studying the internals of file systems is `fsdb`. It is similar to the classic `adb` debugging utility except that it is designed to allow the root user to examine disk-resident metadata structures. To reinforce the information about HFS `inode`s and the basic organization of file system metadata, we use `fsdb` to examine an HFS file system mounted on `/dev/vg00/lvol10` (Listing 8.2). The file system's mount point is `/chris`, and it has a data file stored under it named `/chris/stuff` containing the following three lines of text:

```
This
is my
data file
```

Listing 8.2. `# fsdb -F hfs /dev/vg00/lvol10 session output`

```
First we will point fsdb (actually fsdb_hfs) toward the file
system mount point and then examine the contents of inode #2

fsdb -F hfs /dev/vg00/lvol10  ← This opens the file system on
```

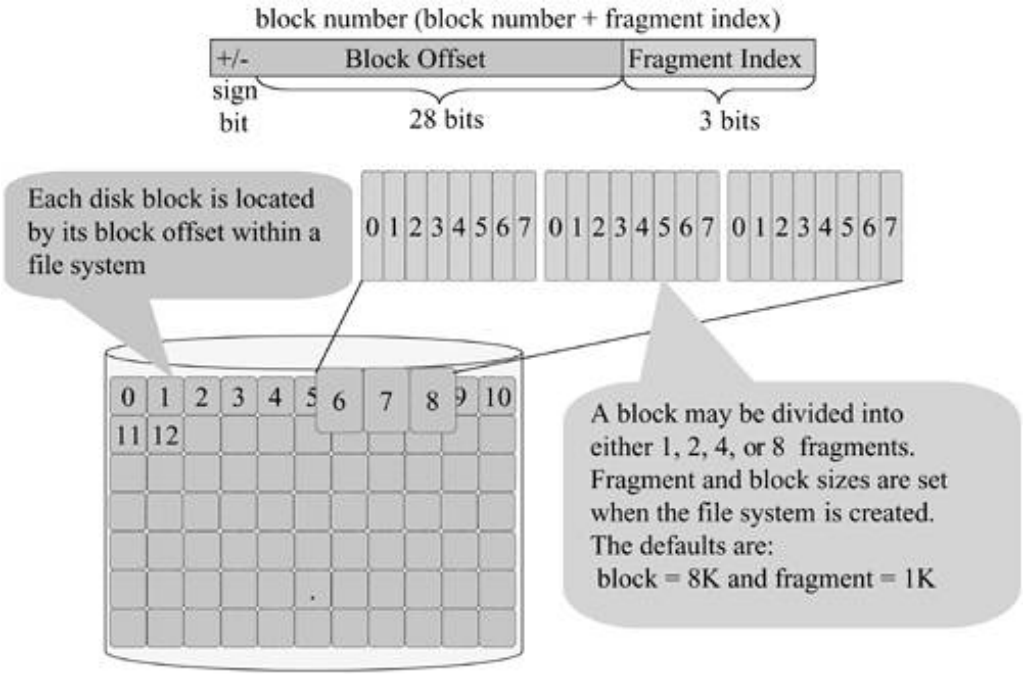
```
            /dev/vg00/lvol10 for examination by fsdb.
            Note: fsdb has no promp string

file system size = 53248(frags)
isize/cyl group=160(Kbyte blocks)
primary block size=8192(bytes)
fragment size=1024 (bytes)

2i  <-- This command list the contents of inode #2
no. of cyl groups = 7
i#:2  md: d---rwxr-xr-x ln:    4 uid:    0 gid:    0 sz: 1024 ci:0
a0 :   208  a1 :     0  a2 :     0  a3 :     0  a4 :     0  a5 :     0
a6 :     0  a7 :     0  a8 :     0  a9 :     0  a10:     0  a11:     0
a12:     0  a13:     0  a14:     0
at: Fri Aug  8 12:56:54 2003
mt: Fri Aug  8 12:46:22 2003
ct: Fri Aug  8 12:46:22 2003


a0b.p0d  <-- Next we list the block pointed to by "a0"
and format it as directory data
d0: 2      .
d1: 2      .  .
d2: 3      l  o  s  t  +  f  o  u  n  d
d3: 1280   c  h  r  i  s

1280i  <-- We see that inode #1280 is the "chris"
       directory and list its contents

i#:1280  md: d---rwxrwxrwx ln:    2 uid:    0 gid:    3 sz: 1024 ci:0
a0 :  8200  a1 :     0  a2 :     0  a3 :     0  a4 :     0  a5 :     0
a6 :     0  a7 :     0  a8 :     0  a9 :     0  a10:     0  a11:     0
a12:     0  a13:     0  a14:     0
at: Fri Aug  8 12:54:28 2003
mt: Fri Aug  8 12:54:35 2003
ct: Fri Aug  8 12:54:35 2003
```

```
a0b.p0d  <-- We again list the directories data block
d0: 1280   .
d1: 2      .  .
d2: 1281   s  t  u  f  f
d3: 1283   y  y  y

1281i  <-- Next we list inode #1281, the "stuff" file metadata
i#:1281  md: f---rw-rw-rw- ln:    1 uid:    0 gid:    3 sz: 22 ci:0
a0 :  8201  a1 :     0  a2 :     0  a3 :     0  a4 :     0  a5 :     0
a6 :     0  a7 :     0  a8 :     0  a9 :     0  a10:     0  a11:     0
a12:     0  a13:     0  a14:     0
at: Fri Aug  8 12:47:06 2003
mt: Fri Aug  8 12:46:55 2003
ct: Fri Aug  8 12:46:55 2003

8201b.p0c  <-- follow the pointer to the file's data block
 and dump it in character format
40022000    : T  h  i  s    \n  i  s     m  y \n  d  a  t  a     f  i  l  e \n \0 \0 \0
➥ \0 \0 \0 \0 \0 \0 \0
40022040    : \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
➥ \0 \0 \0 \0 \0 \0 \0
...
40023740    : \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
➥ \0 \0 \0 \0 \0 \0 \0
```

Blocks and Fragments

As we noted, the block size of the file system has increased over the years from its original 512 bytes to a typical 8 KB. On HP-UX, the block size is tunable on a per-file system basis between 2 KB and 64 KB. With this flexibility comes the potential to waste file space when storing small files, so HFS file systems support the allocation of file system block fragments. Like the block size, the fragment size is also a tunable parameter at the time of file system creation. The fragment size may be set to the same as the block size: one-half, one-quarter, or one-eighth the block size (with a hard minimum of 1 KB). To accommodate the fragment size variations, the last three bits of the block pointer are reserved to address a fragment offset within a block. Figure 8-10 illustrates this basic concept.

Figure 8-10. Blocks and Fragments



The `inode` points to specific disk blocks using a 32-bit signed integer. The last three bits are the fragment offset, which leaves 28 bits for the actual block number and one sign bit. Note that these bits are reserved for the fragment offset regardless of the actual file system configuration. Only the last assigned direct pointer in an `inode` may point to a fragment.

Consider a file 27 KB in size. The first three direct pointers in its `inode` would point to full blocks (this would hold the first 24 KB, leaving 3 KB of space to be allocated). The fourth direct pointer could locate the remaining 3 KB of data in any fragmented block with three contiguous fragments available. As a file grows, whole free blocks are allocated; when the file is closed, the last direct pointer used is checked to see if its data may be relocated to an existing fragmented block or if it is a good candidate to allow remaining fragments to be used by other files. Each cylinder group maintains a list of fragmented blocks within its boundary.

Fragment allocation and management requires extra overhead within the kernel, and its payback is diminished once indirect pointers are required to map data. Once a file has grown to the point where all 12 direct pointers in the `inode` are used, all future allocations are of whole disk blocks, and no fragmentation is attempted for the file.

Allocation Policy

Specialized allocation policies are used by the kernel when an `inode` or data blocks are requested. An `inode` is allocated as follows:

- Allocate `inode`s for files in the same cylinder group as their parent directory.
- Allocate `inode`s for new directories in the cylinder group with a higher than average number of free `inode`s and the smallest number of directory entries.

Data block allocation follows these rules:

- If we are allocating a direct block, allocate the block in the same cylinder group as the `inode` that describes the file.
- If the block is not available or if allocating an indirect block, allocate the block in a group with a greater than average number of free blocks.

A secondary set of allocation policies follows:

- Allocate the requested block.
- Allocate a rotationally equivalent block in the same cylinder group.
- Allocate any block in the same cylinder group.
- If no data blocks are available in the requested cylinder group, then the try another cylinder group with a greater than average number of free blocks.

Exceptions to these policies are made if there is insufficient space within the cylinder group. In this manner, a single file may still occupy the entire available disk space (within tunable limits), but in general all the blocks of a file tend to exist within the narrow boundary of a cylinder group. This improves the locality of the data and greatly speeds up file access during operations such as copies and backups, or any other time the entire file needs to be accessed. Another tunable file system parameter in the kernel is `maxbpg`, which is set to 25 percent by default. This parameter states the maximum percentage of space in a cylinder group that may be allocated to a single file. This is an advisory limit and may be exceeded if a file can find no other available blocks.

Other File System Types

There are other file system types that may be utilized with the HP-UX operating system. The most common is the Veritas file system (VxFS). It is a third-party offering, so we will not go into the internal specifics of VxFS in this book. In general, the features offered by VxFS are transactional journaling (or intent logging) to speed up file system recovery following a system crash and a variety of data storage algorithms based on the allocation of large contiguous blocks of disk space for the storage of large files (it is also called an extent-based file system).

Following the recent merger of Hewlett-Packard and Compaq, features of the Compaq Tru-64 operating system, particularly the Advanced File System, are being considered for porting to HP-UX.

File System Utility Wrappers

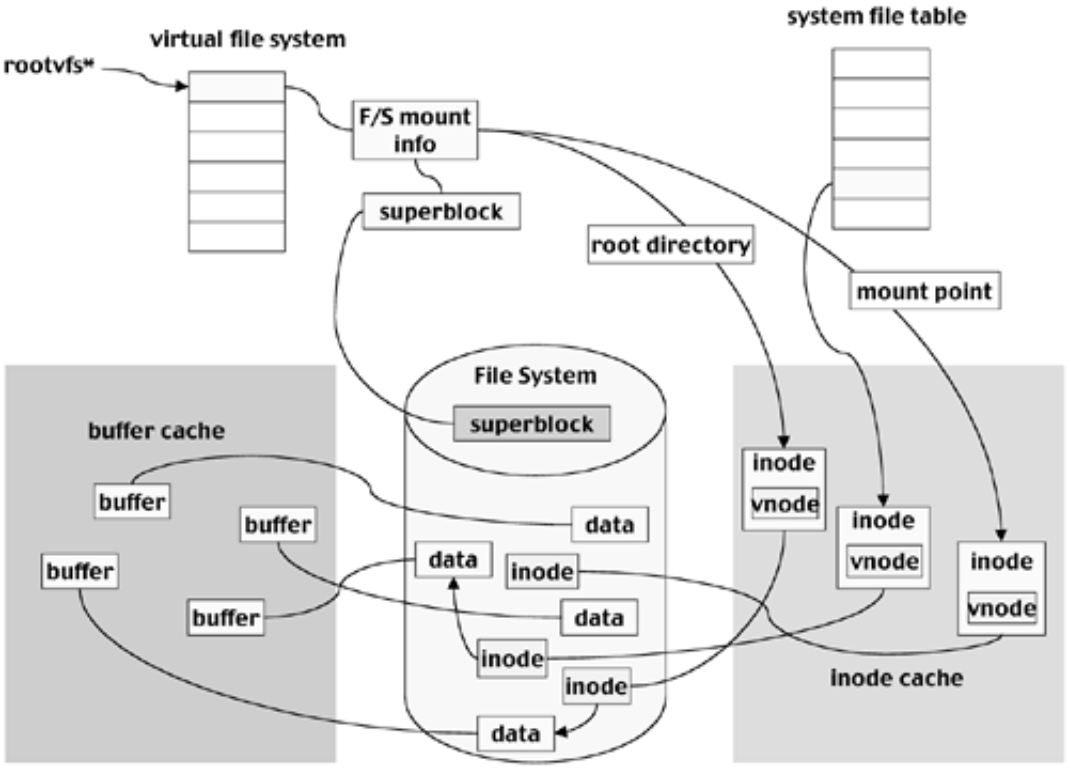
As the metadata utilized by various file system implementations may be quite different, the administrative commands used to manipulate, repair, and monitor them must be specific for the type they work with. When a developer creates a new file system type to be used with the HP-UX kernel, it is her or his responsibility to also create the utilities that work with the specific file system. So that the administrator doesn't have to learn a different set of command names for each type of file system, most of the common commands have been converted to command *wrappers*.

A wrapper command is merely a front end to the file system type-specific executable. Most wrapper commands allow you to pass the file system type as an option, or if the type is not passed, they simply examine the superblock for a magic number identifying the type. Once the type is known, the wrapper simply passes control to the type-specific executable. One way to see if a command is a wrapper is to find the "See also" section of its `main` page. For the `fsdb` command, you would find references to `fsdb_hfs` and `fsdb_vxfs`, among others. In most cases, you may also pass the type-specific executable name to the `man` command to learn about additional options and flags for the command.

The Kernel View of File Systems

To isolate the rest of the kernel from the type-specific structures used by various file system types, a Virtual File System (VFS) abstraction layer is employed. Figure 8-11 gives us an overview of the VFS components.

Figure 8-11. Kernel File System Tables



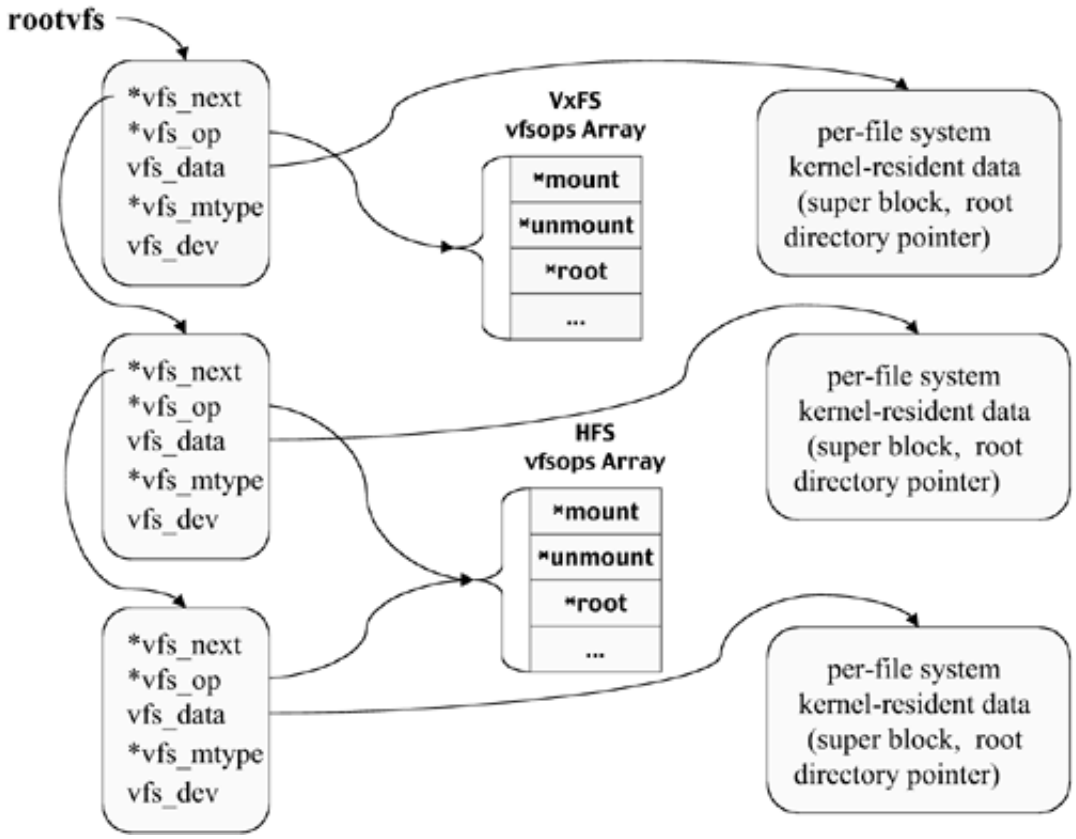
The VFS abstraction layer was first introduced to UNIX by Sun Microsystems engineers while they were developing the Network File System (NFS). They wanted local users and applications to be oblivious to the actual location of the files they were using. Current UNIX operating systems frequently build on this approach to masquerade the specifics of local and remote file system types mounted within their kernels.

The HP-UX kernel uses a number of VFS structures, virtual `inode`s (`vnode`), caches, and hash headers to facilitate file access. To get started, we examine two key components of the virtual file system: the `vfs` mount table and the `vnode`.

The `vfs` Mount Table

The starting point for the HP-UX file management subsystem is the pointer `rootvfs`. This kernel symbol directs us to the "/" (or root) file system's `vfs` data structure (see Figure 8-12).

Figure 8-12. Mounting a File System



Each new file system mount creates an additional entry in the linked list of `vfs` structures. Once a mount has been made, its `vfs` entry is maintained until the kernel is rebooted, even if the file system is unmounted (if it is later remounted, the original `vfs` structure is reused). Mount points configured by the `automounter` also add `vfs` structures to the list.

There are several notable components within the `vfs` structure. Among them are a pointer to the next `vfs`, a type value, and a pointer to an operational array (vector jump table) that maps type-specific kernel routines needed to manipulate the file system, and a pointer to its kernel-resident data structures. The kernel-resident data structures vary greatly in content between the various file system implementations. The type-specific routines mapped by the operations array know what to expect and how to work with the specifics of their design.

Listings 8.3 and 8.4 are annotated listings of the `vfs` structure and the `vfsops` vector array it points to.

Listing 8.3. `q4> fields struct vfs`

```
A forward-pointing link to the next mounted file system (in

chronological order)

   0 0    4 0 *          vfs_next

A hash chain for rapid lookup (used to see if a vfs structure

already exists before we create a new one)
```

```
   4 0    4 0 *          vfs_hash
A pointer to the file system type-specific operations array
   8 0    4 0 *          vfs_op
A pointer to the in-core copy of the vnode this file system is
mounted on
  12 0    4 0 *          vfs_vnodecovered
The vfs state flags
  16 0    4 0 int        vfs_flag
```

0x01VFS_RDONLY	read-only
0x02VFS_MLOCK	lock this vfs (keep subtree stable)
0x04VFS_MWAIT	there is a waiter for the lock
0x08VFS_NOSUID	disable suid for file system
0x10VFS_EXPORTED	this file system is exported (NFS)
0x20VFS_REMOTE	this is an imported file system (NFS)
0x40VFS_HASQUOTA	this file system has quotas enabled
0x200VFS_MOUNTING	used in conjunction with VFS_MLOCK
0x800VFS_LARGEFILES	enable use of large files

```
This file system's native block size
  20 0    4 0 int        vfs_bsize
For exported file systems (NFS) this is the UID remote "root"
users are remapped to and the export state flag
  24 0    2 0 u_short    vfs_exroot
  26 0    2 0 short      vfs_exflags
This points to the file system's type-specific in-core data
structures
  28 0    4 0 *          vfs_data
A reference count of processes currently sleeping on the file
system's mount point
  32 0    4 0 int        vfs_icount
The file system's specific type (HFS, VxFS, NFS, CDFS, ...)and
its unique id
  36 0    2 0 short      vfs_mtype
  40 0    4 0 int        vfs_fsid[0]
```

```
  44 0    4 0 int        vfs_fsid[1]
A log pointer, the most recent mount timestamp, file system
name, device number (Major+minor device number), and a chain
pointer to DNLC entries referencing this file system
  48 0    4 0 *          vfs_logp
  52 0    4 0 int        vfs_mnttime
  56 0 1024 0 char[1024] vfs_name
1080 0    4 0 int        vfs_dev
1084 0    4 0 *          vfs_ncachehd
```

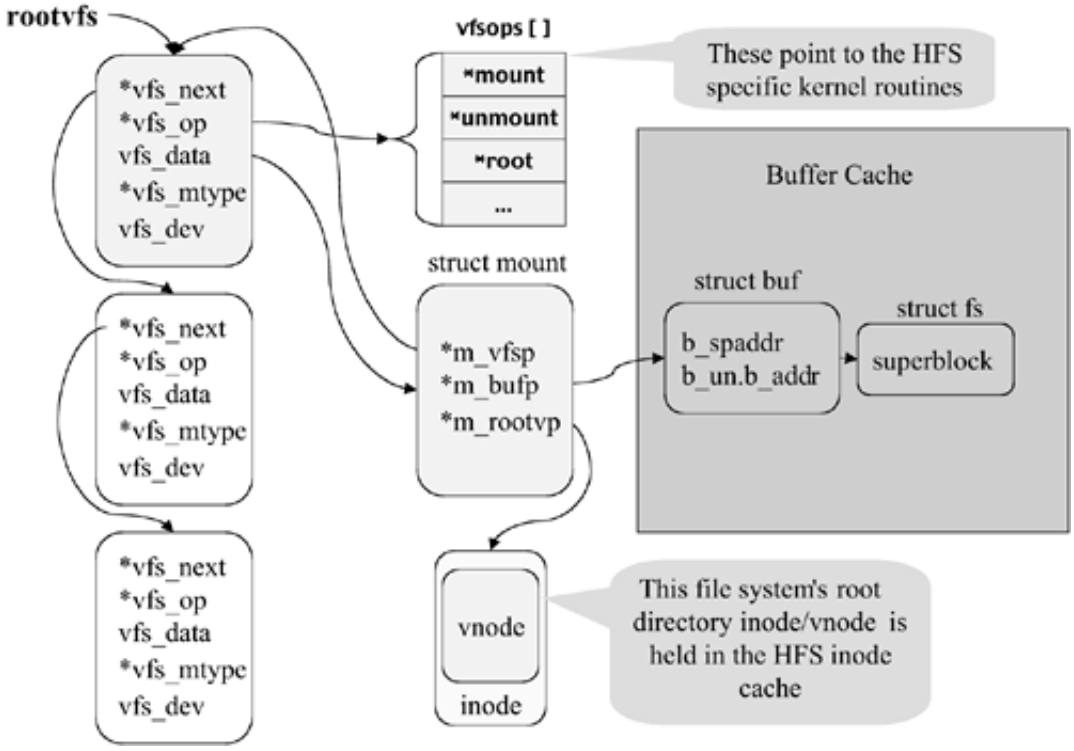
Listing 8.4. `q4> fields struct vfsops`

```
The following pointers form a vector jump table to type-
specific operational routines in the kernel
 0 0 4 0 * vfs_mount
 4 0 4 0 * vfs_unmount
 8 0 4 0 * vfs_root
12 0 4 0 * vfs_statvfs
16 0 4 0 * vfs_sync
20 0 4 0 * vfs_vget
24 0 4 0 * vfs_getmount
28 0 4 0 * vfs_freeze
32 0 4 0 * vfs_thaw
36 0 4 0 * vfs_quota
40 0 4 0 * vfs_mountroot
44 0 4 0 * vfs_size
```

The Type-Specific In-Core Data Structures

Each file system type requires its own specific in-core data. Figure 8-13 illustrates the point.

Figure 8-13. Type-Specific Data Structures (HFS Example)



In general, the in-core data has a copy of the file system's superblock, a pointer to the kernel's copy of the `inode`/`vnode` to which the file system is mounted, and copies of the file system's allocation tables.

In the case of HFS, the kernel-resident structures include a `mount` structure (with pointers to the mount point's `inode`/`vnode`), a pointer back to the file system's `vfs` structure, and a pointer to a copy of the superblock maintained in the buffer cache (more on the buffer cache later in this chapter). Let's examine the HFS `mount` and `fs` (superblock) structures in Listings 8.5 and 8.6.

Listing 8.5. `q4> fields struct mount`

```
First are forward and backward linkage pointers to a hash list
   0 0 4 0 *       m_hforw
   4 0 4 0 *       m_hback
Next is a pointer to the file system's vfs structure, the
block device number (major and \minor number) on which the
file system resides, and a pointer to the buffer header which
maps the in-core copy of the file system's superblock
   8 0 4 0 *       m_vfsp
  12 0 4 0 int     m_dev
  16 0 4 0 *       m_bufp
  20 0 4 0 int     m_maxbufs
The raw device number is also recorded here
```

```
  24 0 4 0 int     m_rdev
  28 0 4 0 *       m_nodes.m_inodp
  28 0 4 0 *       m_nodes.m_cdnodp
  32 0 4 0 *       m_qinod
Mount flags are stored here
  36 0 4 0 int     m_flag
  40 0 2 0 u_short m_qflags
  44 0 4 0 u_int   m_btimelimit
  48 0 4 0 u_int   m_ftimelimit
A pointer to the file system's root inode/vnode in-core, a
reference count, and a pointer to the mount point's in-core
inode/vnode
  52 0 4 0 *       m_rootvp
  56 0 4 0 u_int   m_ref
  60 0 4 0 *       m_iused
```

Listing 8.6. `q4> fields struct fs`

```
   0 0   4 0 int       fs_unused[0]
   4 0   4 0 int       fs_unused[1]
First we have the file system's super block address, the
offset of the cylinder group data block, the inode blocks, and
the first data block following the cylinder groups
   8 0   4 0 int       fs_sblkno
  12 0   4 0 int       fs_cblkno
  16 0   4 0 int       fs_iblkno
  20 0   4 0 int       fs_dblkno
  24 0   4 0 int       fs_cgoffset
  28 0   4 0 int       fs_cgmask
Next is the last modification timestamp, the number of blocks
(total space), and number of data blocks (minus the metadata
overhead) in the file system.
  32 0   4 0 int       fs_time
  36 0   4 0 int       fs_size
  40 0   4 0 int       fs_dsize
Additional static and tunable parameters include the number of
```

```
cylinders per group, block and fragment sizes, number of
fragments in a block (1,2,4,8), minimum reserved free space
(applies only to non-superuser allocation requests),
rotational delay for writing sequential blocks (also known as
interleave), and drive speed (in revolutions per second)
  44 0   4 0 int       fs_ncg
  48 0   4 0 int       fs_bsize
  52 0   4 0 int       fs_fsize
  56 0   4 0 int       fs_frag
  60 0   4 0 int       fs_minfree
  64 0   4 0 int       fs_rotdelay
  68 0   4 0 int       fs_rps
  72 0   4 0 int       fs_bmask
  76 0   4 0 int       fs_fmask
  80 0   4 0 int       fs_bshift
  84 0   4 0 int       fs_fshift
Maximum contiguous blocks and max number of blocks allocated
to a single file within a cylinder group
  88 0   4 0 int       fs_maxcontig
  92 0   4 0 int       fs_maxbpg
  96 0   4 0 int       fs_fragshift
 100 0   4 0 int       fs_fsbtodb
 104 0   4 0 int       fs_sbsize
 108 0   4 0 int       fs_csmask
 112 0   4 0 int       fs_csshift
 116 0   4 0 int       fs_nindir
 120 0   4 0 int       fs_inopb
 124 0   4 0 int       fs_nspf
This file system's unique identifier number
 128 0   4 0 int       fs_id[0]
 132 0   4 0 int       fs_id[1]
Mirror state information for primary swap and "/" partitions
 136 0   0 4 u_int     fs_mirror.state.root
 136 4   0 1 u_int     fs_mirror.state.rflag
 136 5   0 4 u_int     fs_mirror.state.swap
 137 1   0 1 u_int     fs_mirror.state.sflag
 137 2   2 6 u_int     fs_mirror.state.spare
 140 0   4 0 int       fs_mirror.mirtime
```

```
 144 0   4 0 int       fs_featurebits
 148 0   4 0 int       fs_optim
Cylinder group summary array address, summary array size,
cylinder group size
 152 0   4 0 int       fs_csaddr
 156 0   4 0 int       fs_cssize
 160 0   4 0 int       fs_cgsize
Number of tracks per cylinder group, sectors per track,
sectors per cylinder, total number of cylinders in the file
system, cylinders per group, and inodes per group
 164 0   4 0 int       fs_ntrak
 168 0   4 0 int       fs_nsect
 172 0   4 0 int       fs_spc
 176 0   4 0 int       fs_ncyl
 180 0   4 0 int       fs_cpg
 184 0   4 0 int       fs_ipg
 188 0   4 0 int       fs_fpg
Calculated cylinder group totals (this must be audited
following a power failure)
 192 0   4 0 int       fs_cstotal.cs_ndir
 196 0   4 0 int       fs_cstotal.cs_nbfree
 200 0   4 0 int       fs_cstotal.cs_nifree
 204 0   4 0 int       fs_cstotal.cs_nffree
Next are the file system modification flag, the clean/dirty
flag, the read-only flag, the spare flag (currently not used),
and a string holding the current mount point of the file
system
 208 0   1 0 char      fs_fmod
 209 0   1 0 char      fs_clean
 210 0   1 0 char      fs_ronly
 211 0   1 0 char      fs_flags
 212 0 512 0 char[512] fs_fsmnt
 724 0   4 0 int       fs_cgrotor
 728 0   4 0 *         fs_csp
 732 0   4 0 int       fs_csp_pad
 736 0   4 0 int       fs_unused2[30]
 856 0   4 0 int       fs_cpc
```

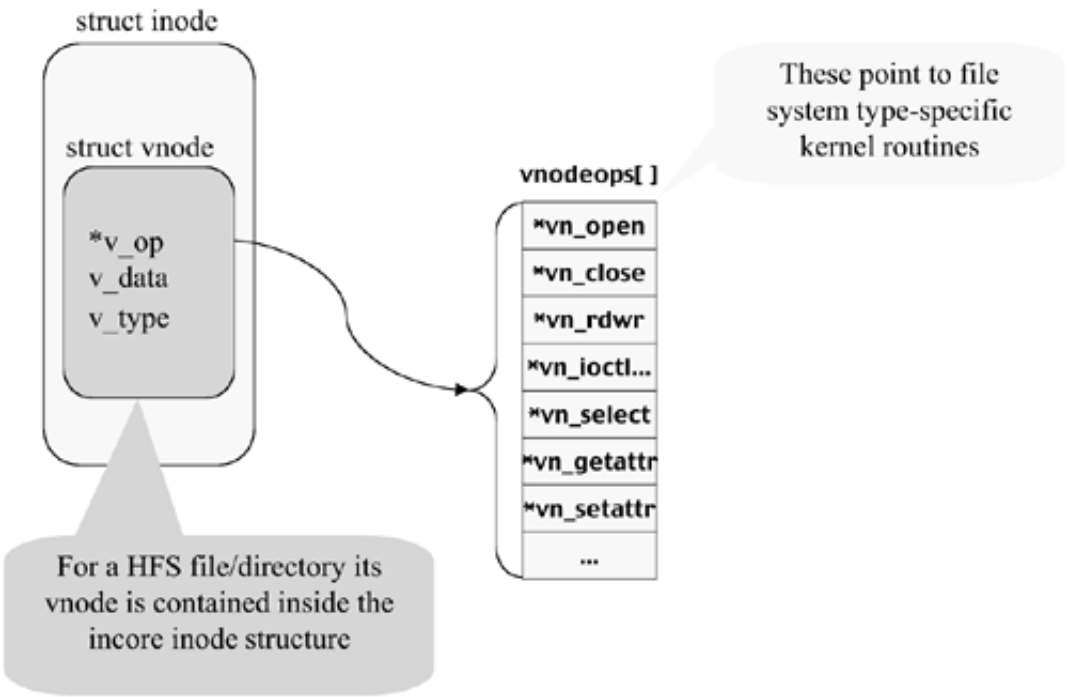
```
 860 0   2 0 short     fs_postbl[32][8]

This file system's "magic number" (used to verify its type),

its type name and pack name

1372 0   4 0 int       fs_magic

1376 0   6 0 char[6]   fs_fname

1382 0   6 0 char[6]   fs_fpack

1388 0   1 0 u_char    fs_rotbl[0]
```

The `vnode`

Next, we address the specifics of the `vnode` structure. We have learned that an `inode` contains all of a file's credentials and attributes, and is therefore the key to managing a file within a file system. The `inode` number is unique within the context of a single file system, and it is all that is needed to define a file. Within a kernel, there may be several file systems mounted together to create what appears to be a seamless, larger file system. Within this larger context, `inode` numbers are no longer guaranteed to be unique. Each mounted file system may have an `inode` 1, 2, and so on. To enable the kernel to refer to opened files with a unique handle, the virtual `inode` or `vnode` was created.

In a similar manner to the `vfs` structure, a `vnode` contains an operational array pointer to the kernel's type-specific routines for file manipulation tasks (see Figure 8-14).

Figure 8-14. The `vnode`



The key fields of the `vnode` include a type number, a pointer to a type-specific operations array, and a reference to the in-core copy of the file's `inode` data. Listing 8.7 is an annotated listing of a virtual file system vnode:

Listing 8.7. `q4> fields struct vnode`

```
 0 0 2 0 u_short v_flag
```

0x01	VROOT	root of a file system
0x02	VTEXT	vnode is a text prototype
0x04	VSWAP	vnode maps a swap space
0x08	VXOFTHOLD	vnode allows soft holds
0x010	VEXLOCK	exclusive lock
0x020	VSHLOCK	shared lock
0x040	VLWAIT	there is a waiter on a lock
0x080	VXDEVOK	cross device rename and link is allowed
0x0100	VMMF	vnode is memory mapped
0x04000	VNOBCACHE	data caching disabled
0x08000	VN_MPSAFE	semaphore not required

```
Shared lock count, exclusive lock count, private data count,
and reference count
 2 0 2 0 u_short v_shlockc
 4 0 2 0 u_short v_exlockc
 6 0 2 0 u_short v_tcount
 8 0 4 0 int     v_count
Pointer to vfs structure if this is a mount point
12 0 4 0 *       v_vfsmountedhere
Pointers to the type specific vnode operations array, unix ipc
socket, stream head, and the vfs the file is in
16 0 4 0 *       v_op
20 0 4 0 *       v_socket
24 0 4 0 *       v_stream
28 0 4 0 *       v_vfsp
vnode type, device number, type-specific data (hfs inode, vxfs
inode, rnode, ...), and parent file system type
```

```
32 0 4 0 enum4   v_type

36 0 4 0 int     v_rdev

40 0 4 0 *       v_data

44 0 4 0 enum4   v_fstype

Pointer to parent pseudo vas (if this maps to an executable

image file)

48 0 4 0 *       v_vas

Beta semaphore locking structure

52 0 1 0 u_char  v_lock.b_lock

54 0 2 0 u_short v_lock.order

56 0 4 0 *       v_lock.owner

Mapped memory cache buffers for this file

60 0 4 0 *       v_cleanblkhd

64 0 4 0 *       v_dirtyblkhd

vnode write count, FS independent locks, soft hold count, and

a copy of the va_nodeid

68 0 4 0 int     v_writecount

72 0 4 0 *       v_locklist

76 0 4 0 int     v_scount

80 0 4 0 int     v_nodeid

DNLC entry pointers if this vnode is currently described in

the directory name lookup cache

84 0 4 0 *       v_ncachedhd

88 0 4 0 *       v_ncachevhd

92 0 4 0 *       v_pfdathd

96 0 4 0 u_int   v_last_fsync
```

The HFS In-Core `inode` Contains Its `vnode`

For an HFS file, the in-core `inode` structure includes the `vnode` structure and a complete copy of its disk-resident `inode` (the `icommon` structure we examined earlier) plus additional fields of information (Listing 8.8):

Listing 8.8. `q4> fields struct inode`

```
  0 0  4 0 *         i_chain[0]

  4 0  4 0 *         i_chain[1]

The device number this inode came from
```

```
  8 0  4 0 int       i_dev
The inode number
 12 0  4 0 u_int     i_number
The state flags, lock word, and tid of last thread to lock this inode
 16 0  4 0 u_int     i_flag
 20 0  2 0 u_short   i_lockword
 24 0  4 0 int       i_tid
```

The virtual file system's vnode for the file is stored between offset 28 and 127

```
inode device pointer
128 0  4 0 *         i_devvp
132 0  4 0 int       i_diroff
Continuation inode pointer
136 0  4 0 *         i_contip
File system pointer
140 0  4 0 *         i_fs
Quota pointer
144 0  4 0 *         i_dquot
148 0  4 0 int       i_rdev
152 0  4 0 int       i_un.if_lastr
152 0  4 0 *         i_un.is_socket
156 0  4 0 *         i_fr.if_freef
160 0  4 0 *         i_fr.if_freeb
164 0  4 0 *         i_fselr.i_selp
168 0  2 0 short     i_fselr.i_selflag
172 0  4 0 *         i_fselw.i_selp
176 0  2 0 short     i_fselw.i_selflag
180 0  4 0 *         i_mount
```

Acopy of the disk-based inode (structure icommon) is stored between offset 184 and 311

```
Number of reader calls
312 0  2 0 u_short   i_rcount
Beta semaphore locking structures
316 0  1 0 u_char    i_b_sema.b_lock
318 0  2 0 u_short   i_b_sema.order
```

```
320 0  4 0 *          i_b_sema.owner

324 0  4 0 int        filler[0]

328 0  4 0 int        filler[1]
```

The relationship between the in-core `inode` data and the `vnode` varies between file system types, but the `vnode` structure itself is identical for all supported file systems. As the `vnode` uses operation's array pointers, it is up to the programmers of the referenced routines to have intimate knowledge of `inode` type-specific content.

When a file is opened by the kernel, it is identified by the device number of the volume on which the file system is located and its `inode` number. By using these two values, each file is guaranteed to have a unique identity within the scope of the operating system.

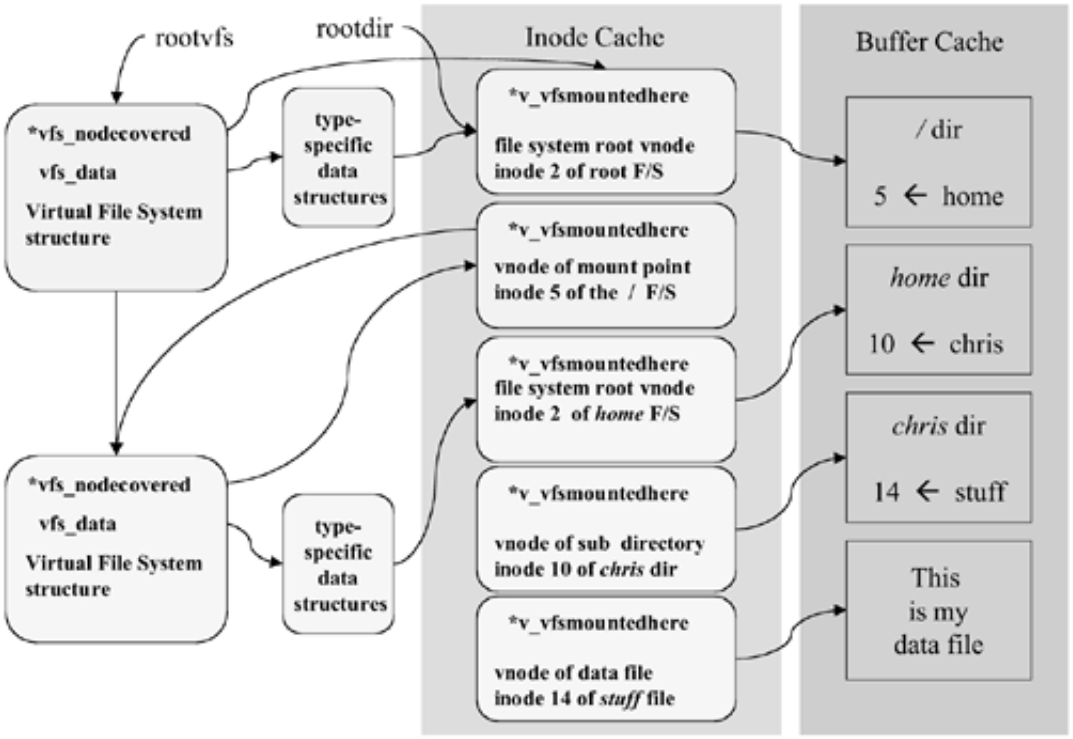
Caching `inode`/`vnode` Data in the Kernel

To improve system performance and reduce the amount of time spent accessing metadata on the disk, `inode`/`vnode` data is maintained in kernel-resident caches. Each type of file system is responsible for the allocation and maintenance of its `inode`/`vnode` cache. In most implementations caches are sized by kernel-tunable parameters or by an imbedded formula. A hash is formed using the device number and the `inode` number. Before a copy of an `inode` is read in from the physical disk, the `inode` cache is always checked to see if a copy of the data is present.

Smoke and Mirrors: Tying Together the File Subsystem Abstractions

Next, we need to tie the abstractions together into a seamless VFS implementation (see Figure 8-15).

Figure 8-15. Building a Seamless File System



Let's revisit our earlier scenario except that now `/home` is a separate file system mounted to the system's root file system. Again, our challenge is to follow the pathname */home/chris/stuff* through the kernel's virtual structures.

We start our search by locating the kernel's copy of the `/ inode` (2 on the first volume). The `inode`/`vnode` and root directory data of all mount points and file system roots are held in the caches and the buffer cache as long as a file system is mounted. To speed up this initial search, the kernel maintains a permanent shortcut pointer to the cached copy of the `/ inode`/`vnode` in the symbol `rootdir`.

From this `vnode`, we determine the file system device number, and from the associated `inode` we get the data block pointer. These two numbers are used to hash our way into the buffer cache, where we find a copy of the root file system's root directory data. We take the `inode` number associated with the *home* directory and the device number to hash our way back into the `inode` cache for the next step of our search.

When we find the cached `vnode` for the */home* directory (5 on the first volume), the `v_vfsmountedhere` pointer is valid and directs us to the kernel mount table and the `vfs` structure for the second mounted volume (did you ever play the children's game Chutes and Ladders?). We then follow the `v_data` pointer to find the root `inode`/`vnode` (2 on the second volume) and cached data block for the mounted */home* file system.

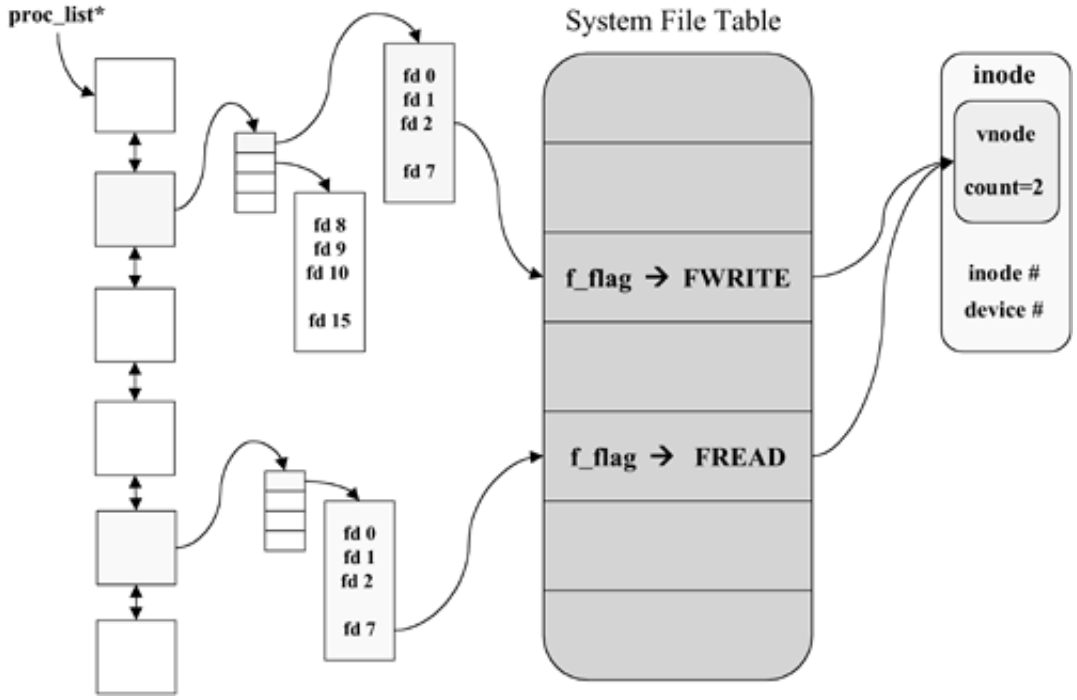
We continue our search within the */home* file system, locating the `inode`/`vnode` (10 on the second volume) and cached data for the */home/chris* directory and finally the `inode`/`vnode` (14 on the second volume) and cached data for the */home/chris/stuff* file. Accessing these last two stops on our search may or may not find the requested data in the appropriate caches. If they aren't loaded, then the system uses the routines referenced by the operational arrays and kernel functions to load copies from the disk into the caches.

This brings us to the end of our quest. Next we consider the attachment of an open file to a process.

Connecting an Open File to a Process

We have seen that the `proc` structure is the base of all process-related kernel structures. The definition of a process's resources includes mapping opened files to per-process file descriptors (see Figure 8-16).

Figure 8-16. System File Table



From the kernel's point of view, each distinct file `open()` call must be tracked, which is accomplished by making entries in the system file table. From the process's point of view, it is only concerned with its own file descriptors, but the kernel must manage all `open()` calls.

The number of file descriptors a single process may have is controlled by the kernel-tunable parameters `max_files` (for a regular user process) and `max_files_lim` (for a superuser process), and may be set up to 60,000. The default is 60 and is usually sufficient, considering that most processes only need to map `stdin`, `stdout`, and `stderr`.

To avoid wasting kernel memory space, process file descriptors are stored in blocks of eight for HP-UX 11.11 (HP-UX 11.0 used 32 descriptors per block), and the blocks are referenced through a partition table pointed to by the `proc` structure. Additional blocks are allocated as needed by the kernel.

On HP-UX 11.11 the `file_descriptor_t` structure contained:

```
fd_lock       a spinlock structure

*fd_filep     a pointer to a system file table file entry

fd_locker_tid a reference to a locking thread

fd_state      the current descriptor state

fd_pofile     reference to the ofile partition entry
```

The kernel file system table consists of `nfile` entries (kernel-tunable) of type `file`. See Listings 8.9 and 8.10.

Listing 8.9. `q4> fields struct file`

```
The file descriptor table flags are:
  0 0 4 0 int       f_flag
```

0x01FREAD	descriptor is readable
0x02FWRITE	descriptor is writable
0x04FNDELAY	no delay
0x08FAPPEND	append on each write

```
The kernel object represented by this entry
  4 0 2 0 short     f_type
```

0x01DTYPE_VNODE	regular file
0x02DTYPE_SOCKET	Berkeley IPC Socket
0x03DTYPE_STREAMS	Streams file type
0x05DTYPE_UNSP	user nsp control
0x06DTYPE_LLA	link-level LAN access

```
link count from the message queue
   6 0 2 0 short     f_msgcount
reference count
   8 0 4 0 int       f_count
type-specific file system operations array pointer
  12 0 4 0 *         f_ops
pointer to this file's vnode/socket kernel data
  16 0 4 0 *         f_data
  20 0 4 0 int       f_freeindex
  24 0 8 0 long long f_offset
  32 0 4 0 *         f_buf
credentials of user who opened the file
  36 0 4 0 *         f_cred
kernel beta-semaphore
  40 0 1 0 u_char    f_b_sema.b_lock
```

```
  42 0 2 0 u_short   f_b_sema.order
  44 0 4 0 *         f_b_sema.owner
data intercept audit flag
  48 0 2 0 short     f_tap
```

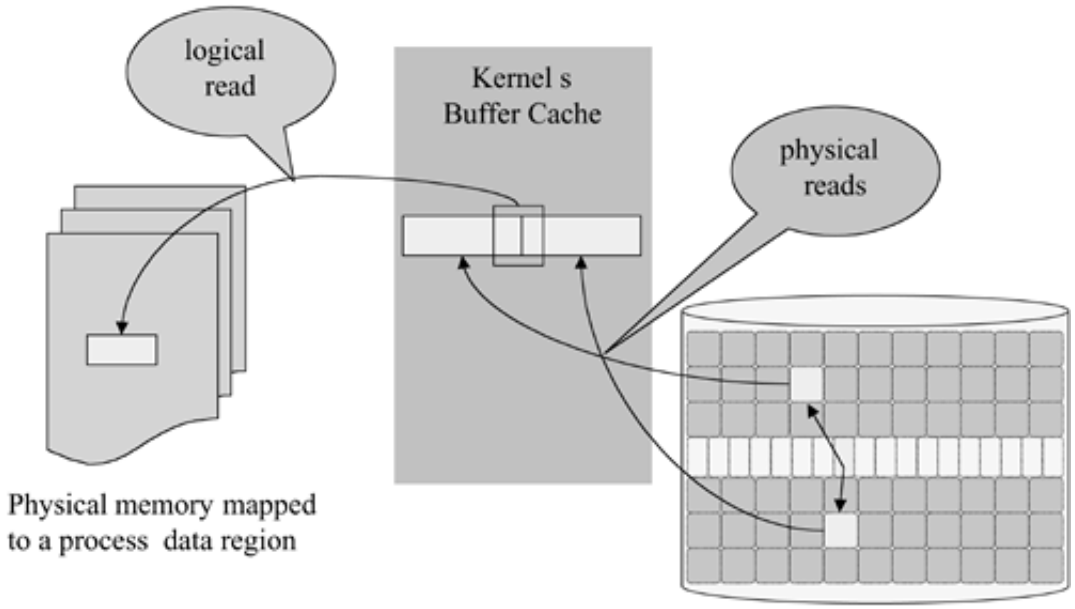
Listing 8.10. `q4> fields struct fileops`

```
 0 0 4 0 * fo_rw
 4 0 4 0 * fo_ioctl
 8 0 4 0 * fo_select
12 0 4 0 * fo_close
```

The Buffer Cache

Just as the kernel keeps copies of open `inode` data in a cache, it also maintains a much larger cache of recently accessed file data blocks (or extents, depending on the type of file system). This buffer cache is a key component of the HP-UX operating system. Figure 8-17 provides an overview of the buffer cache design.

Figure 8-17. The Buffer Cache



A basic premise of the UNIX operating system is that every thing is a file. This model serves many purposes and greatly reduces the overall complexity of the operating system. With this in mind, it is understandable that everything that may be done to improve file performance provides a large payback.

Being Flexible: The Dynamic Buffer Cache

The HP-UX implementation of a buffer cache follows a dynamic model. The buffer cache is mapped into the kernel's address space, but the physical pages allocated to it come from the same available memory pools as those used for user processes. In this manner, the buffer cache competes with user code for system memory resources.

The kernel allows the administrator to set a maximum and minimum percentage of the available memory that may be allocated for buffer cache use. Remember that *available memory* means the amount of physical memory remaining after the kernel allocates pages for its exclusive use during system initialization. The kernel-tunable parameters to enable a dynamic buffer cache are:

- `dbc_min_pct`: Minimum percentage of available memory that may be allocated for buffer cache use
- `dbc_max_pct`: Maximum percentage of available memory pages that may be allocated for buffer cache use

If the `dbc_min_pct` is set equal to or larger than the `dbc_max_pct`, then the dynamic buffer cache is capped at the size dictated by `dbc_min_pct`. This presents what at first seems to be a conundrum. How can it be a dynamic buffer cache if the size is fixed? While the size is fixed, it is simply a targeted value and not an absolute. As long as the buffer cache is considered to be dynamic, its pages may be aged and stolen by the `vhand` daemon during times of heavy memory pressure. This means that while the overall size appears to be static, the specific pages allocated to the cache may change from time to time, so they are considered dynamic.

If these two dynamic tunables are not set, then the following older kernel parameters take effect and the cache is flagged as static; that is, pages once allocated to the cache remain there and `vhand` cannot scan them.

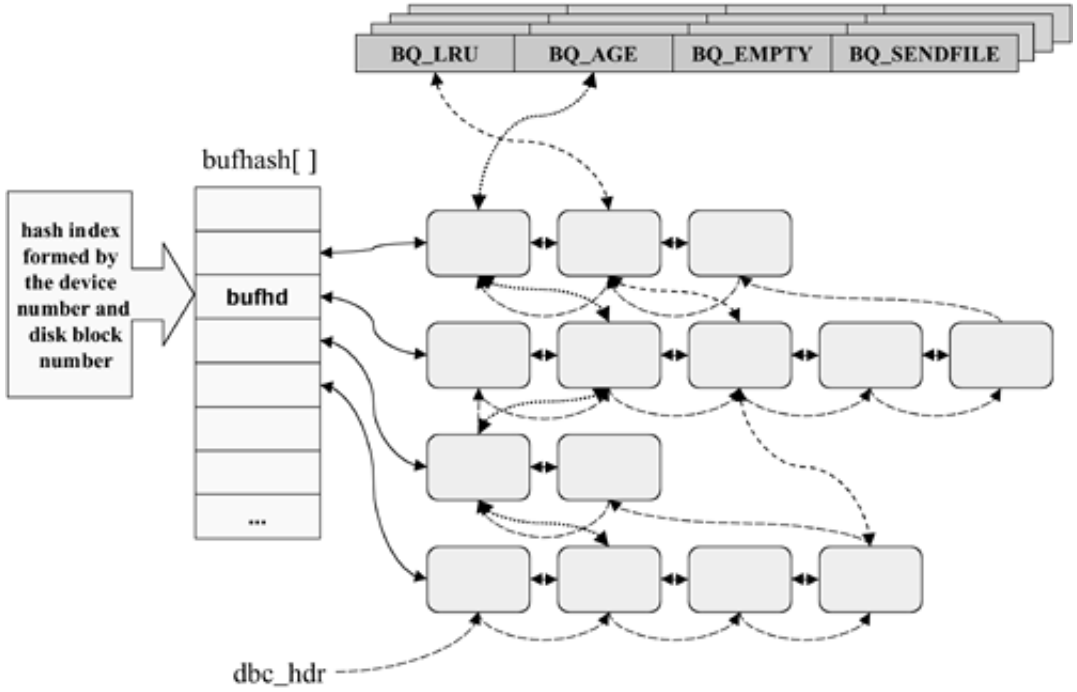
- `nbuf`: Number of buffer headers
- `bufpages`: Number of buffer pages

Once a buffer has been populated with a copy of file data, the kernel needs an efficient method to search for it in the cache. This is accomplished through the use of a hash (by now you expected this, didn't you!).

The Buffer Cache Hash

Figure 8-18 illustrates the basic design of the buffer cache. Individual buffers are pointed to by buffer headers' which are arranged in hash chains. In the case of a `read()`, the hash offset is calculated using the device number and block offset (obtained from the `vnode`) for the requested disk data.

Figure 8-18. The Buffer Cache Hash



Once the hash has been calculated, the appropriate hash chain is searched to see if the requested data is present in the cache. In the case of a miss, the kernel works with the appropriate hardware driver, requests a physical data transfer from the disk, and allocates a free cache buffer to store the data in. Requests made and satisfied by the buffer cache are said to be *logical* reads and writes. Reads and writes between the buffer cache and a disk are called *physical*.

A `write()` request is handled in a similar manner. If a buffer has already been started, a simple logical write is performed. When a buffer is filled or when a dirty buffer (one whose content differs from its counterpart on the disk) is found by the `sync` daemon during one of its scheduled scans, a physical write is scheduled. Physical writes may also be specifically requested by a programmer or when a file is closed.

Listings 8.11 and 8.12 show selected fields from q4 listings of the buffer header and buffer structure.

Listing 8.11. `q4> fields struct bufhd`

```
The buffer header state flag

   0 0 4 0 int b_flags
```

0x00800000	B_PAGEOUT	this is a buffer header, not a buffer

```
Forward/backward linkage pointers for the hash chain

   4 0 4 0 *   b_forw

   8 0 4 0 *   b_back
```

```
Modification timestamp
  12 0 4 0 int b_checkdup
Lock structure for this buffer hash chain
  16 0 4 0 *   bh_lock
```

Listing 8.12. `q4> fields struct buf (a partial listing)`

```
The buffer state flags
  0 0 4 0 int               b_flags
```

0x00000000	B_WRITE	
0x00000001	B_READ	
0x00000002	B_DONE	transaction completed
0x00000004	B_ERROR	transaction failed
0x00000008	B_BUSY	not currently on a free list
0x00000010	B_PHYS	physical I/O in progress
0x00000040	B_WANTED	send a wakeup when no longer BUSY
0x00000100	B_ASYNC	don't wait for I/O completion
0x00001000	B_PRIVATE	used by LVM
0x00004000	B_PFTIMEOUT	LVM power fail timeout
0x00008000	B_CACHE	
0x00010000	B_INVAL	buffer not valid
0x00020000	B_FSYSIO	buffer resulted from bread or bwrite
0x00040000	B_CALL	call b_iodone() from iodone()
0x00080000	B_NOCACHE	don't hold block in buffer cache
0x00100000	B_RAW	raw access I/O
0x00200000	B_BCACHE	buffer is from the buffer cache
0x00400000	B_SYNC	buffer write is synchronous

```
Linkage pointers connecting a buffer to a hash chain
    4 0 4 0 *                 b_forw
    8 0 4 0 *                 b_back
Linkage pointer connecting a buffer to a free list
```

```
   12 0 4 0 *                  av_forw
   16 0 4 0 *                  av_back
Linkage pointers connecting a buffer to a vnodes clean/dirty list
   20 0 4 0 *                  b_blockf
   24 0 4 0 *                  b_blockb
   28 0 4 0 *                  b_strategy
   32 0 4 0 *                  b_un.b_addr
   32 0 4 0 *                  b_un.b_words
   32 0 4 0 *                  b_un.b_filsys
   32 0 4 0 *                  b_un.b_cs
   32 0 4 0 *                  b_un.b_cg
   32 0 4 0 *                  b_un.b_dino
   32 0 4 0 *                  b_un.b_daddr
   36 0 4 0 int                b_un2.b_sectno
   36 0 4 0 int                b_un2.b_byteno
   40 0 4 0 int                b_dev
   44 0 4 0 int                b_blkno
   48 0 4 0 *                  b_sc
The count of how many times this buffer has been returned to a
free list since it was allocated
   52 0 4 0 int                b_bcount
   56 0 2 0 u_short            b2_flags
Error number (if B_ERROR is set in b_flags), buffer size, and offset
   58 0 2 0 short              b_error
   60 0 4 0 int                b_bufsize
   64 0 4 0 int                b_offset
   68 0 4 0 *                  b_merge
The pointer contains the address of the routine to call when a
pending I/O is finished
   72 0 4 0 *                  b_iodone
---------------------------------------
```

Free Lists: Allocating a New Buffer

When a buffer is in use, its `b_flags` is set to `B_BUSY`. As soon as any pending data transfer is complete, it is returned to a free list. There are several free lists.

When a page is first allocated to a dynamic buffer cache, its buffers are placed on the BQ_EMPTY free list. Requests to grow the buffer cache depend on its level of use.

Once a buffer is allocated, used, and freed, it is placed on the BQ_AGE list, and its b_count is incremented. Once the b_count exceeds 4, the buffer is placed on the BQ_LRQ list when it is freed. In this manner, a buffer that has been loaded to satisfy a truly random access read request is more likely to be reallocated than one for which sequential requests are being processed.

Each processor has its own set of free list headers. The rules for finding an available buffer with a fixed buffer cache are as follows:

- Search the processor's BQ_AGE list, then its BQ_LRU.
- Next, search the BQ_AGE lists of all the other processors.
- Finally, search their BQ_LRQ lists.

In the case of a dynamic buffer cache:

- Check the current processor's BQ_AGE and BQ_LRU.
- Next we check the BQ_EMPTY (we wait to check it, as the allocation of new pages to the cache is performed in background).
- Assuming that we can't find a buffer on one of our processor's free lists, we next check the neighboring processors' BQ_AGE and then their BQ_LRU lists.

If we are still unsuccessful, the thread blocks until a buffer is available.

An interesting observation is that items cached in the buffer cache don't behave like items in most other caches. When a buffer is involved in an I/O operation, it is B_BUSY. The rest of the time, it is placed on a free list. In this manner the more a buffer is referenced, the less its chances are of being reallocated. If the cache is not busy, a buffer may persist for quite some time following its use, if another process comes along requesting the data, it may be pleasantly surprised to find that it is already in the cache and may be accessed logically.

A fourth queue, BQ_SENDFILE, is used to list transitive buffers in use by kernel routine sendfile().

Bigger Isn't Always Better

The time required to search the hash chain is offset by the fact that a logical read or write is several orders of magnitude faster than a physical transaction. The greater the "hit rate" for data in the cache, the greater the payback is for this scheme. For files accessed in a sequential manner, the hit rate may exceed 90 percent! Adjusting the size of the buffer cache may help to increase the hit rate, but caution should be used to make sure that the system has plenty of available memory and that increasing the buffer cache doesn't cause an increase in the page-out rate, or the gains in the hit rate could be lost in the added overhead of vhand.

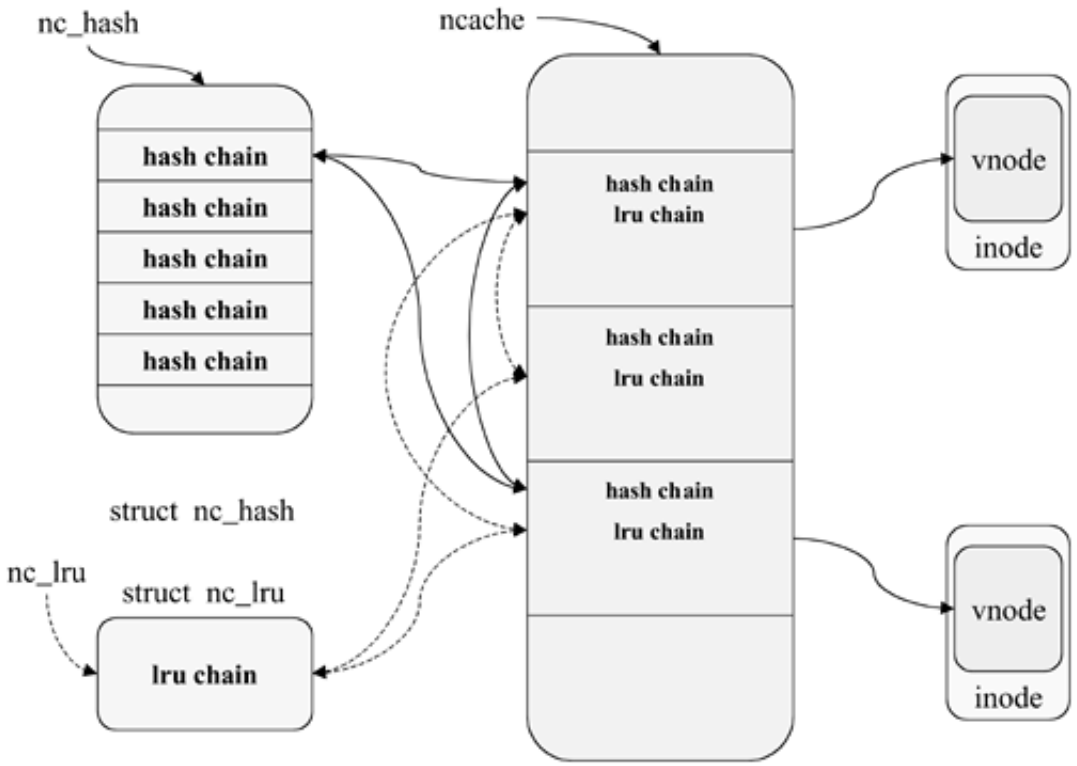
Another concern is that if the buffer cache gets too large, then the added length of the hash chain requires more search time. At some point the payback of caching may be diminished.

Prior to HP-UX 11.11, the conventional wisdom was that the buffer cache should not exceed 0.5 GB. Recent modifications (HP-UX 11.11) to the hashtable mechanics suggest that sizes in the 1-GB to 2-GB range may be considered

The Directory Name Lookup Cache, DNLC

The final component of the file system we examine in this chapter is the directory name lookup cache, or DNLC (Figure 8-19). Just as the kernel maintains caches for data blocks and inodes for improved performance, it also maintains a cache of recently referenced pathnames.

Figure 8-19. Directory Name Lookup Cache



We have seen how a user-supplied pathname is processed by the kernel through the VFS. If we could skip past a majority of the smoke-and-mirror translation tasks, we could improve the time required to open a file descriptor.

Consider a thread that has changed to a working directory and is accessing a number of the files in that directory. Each time an `open()` call is made, the kernel must convert the pathname to an appropriate `vnode` and data block. The DNLC caches individual directory names and accesses them via a hash based on their name strings.

Entries in the cache remain on a free list in a least recently used fashion. Another limit is that individual names must not be longer than 39 characters, the maximum space allocated for the directory name. Longer names may be used but they will not be cached (prior to HP-UX 11.0 the maximum length for cached names was 15 characters).

Hash chain headers are stored in the kernel's `nc_hash[]` array. The hash function uses the `vnode` address and the file/directory name to calculate the chain header to which it should be linked.

Consider the Listings 8.13 and 8.14of the DNLC structures.

Listing 8.13. `q4> fields struct nc_hash`

```
Each nc_hash[] array element contains two linkage pointers

   0 0 4 0 * hash_next

   4 0 4 0 * hash_prev
```

Listing 8.14. `q4> fields struct mcache`

```
First the entry is linked to its hash chain
   0 0  4 0 *        hash_next
   4 0  4 0 *        hash_prev
All entries are linked into a least recently used list
   8 0  4 0 *        lru_next
  12 0  4 0 *        lru_prev
This points to the directory's vnode
  16 0  4 0 *        vp
And a pointer to the parent directory's vnode
  20 0  4 0 *        dp
The entry's length, name, and credentials are stored next
  24 0  1 0 char     namlen
  25 0 39 0 char[39] name
  64 0  4 0 *        cred
Mount point directories are linked to the virtual file system table
  68 0  4 0 *        vfs_next
  72 0  4 0 *        vfs_prev
All entries sharing a parent directory are dual linked
  76 0  4 0 *        dp_next
  80 0  4 0 *        dp_prev
All entries sharing a vnode are dual linked
  84 0  4 0 *        vp_next
  88 0  4 0 *        vp_prev
```

Summary

In this chapter, we studied the critical components of the file system. Foremost are a file's `inode` on the disk and, once opened by the kernel, its `vnode`. These two incarnations of what is basically the same data define the credentials and attributes of all files in the HP-UX environment. Specialized files known as directories coupled with the kernel's `VFS` structures allow the creation of a seamless hierarchical directory structure that may incorporate files from a variety of local and remote file systems. A user or application needs no knowledge of the mount path specifics or file system types.

We also discovered that there are several caches and hashing schemes implemented by the kernel to improve system performance levels. In the next chapter we build on the topics we have covered so far and look at the process/thread life cycle.

Chapter 9. The Process Life Cycle, Cradle to Grave

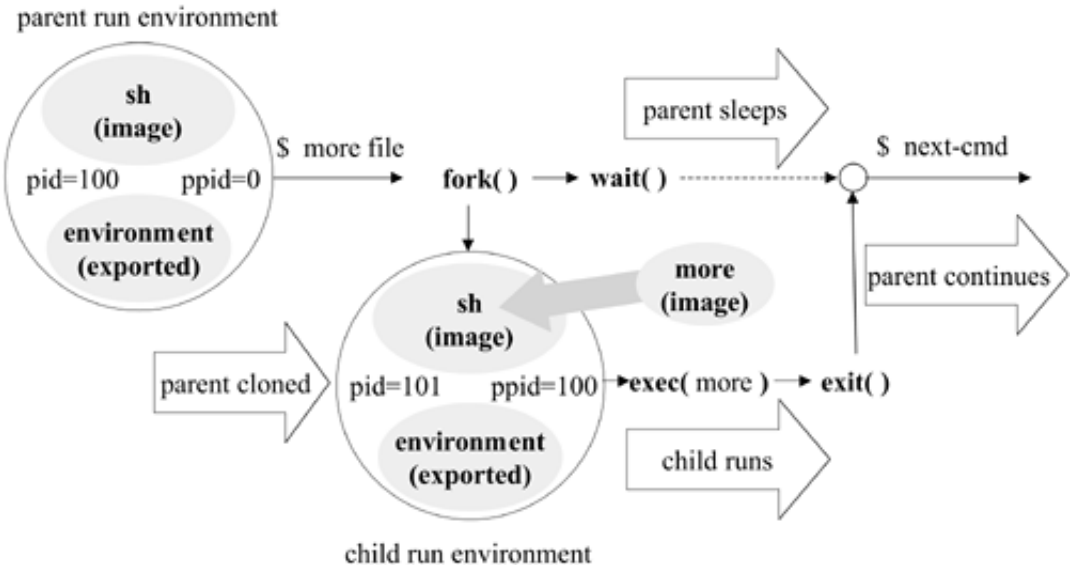
So far we have discussed the underlying PA-RISC hardware and many of the various kernel data structures that make up the HP-UX operating system. The point of view of our discussions has switched between that of the kernel and that of a process. We now have a broad enough perspective to examine the life cycle of a system process from cradle to grave.

We start with the birth of a process, which is the result of a system call to either `fork()` or `vfork()` and requires the establishment of a unique process table entry and the creation of the process's first thread. Then we examine the tasks performed by the kernel in the creation of the new process environment, including the creation of new memory regions and/or the attachment to existing shareable ones. Our study concludes with a discussion of the termination of a process, including breaking reference links, deconstruction of system resources, and the important role played by the parent process in the acknowledgment of the *death-of-child* signal. We also examine the mechanics of shared memory objects in greater depth.

The Birth of a Process

The UNIX operating environment presents a special challenge when it comes to starting a new process. The only way for a new process to be created is for an existing process to ask the system to make a copy, a complete "clone" of the process environment (see Figure 9-1). A parent process is said to *fork* a "child" process: this is the computer equivalent of cellular mitosis in living organisms.

Figure 9-1. Parent Creates a Child



Once the fork is completed, there are two essentially identical process environments under kernel control. They have the same process logical memory view, identical file descriptors (to facilitate features such as pipes and I/O redirection), and they map the same shared memory objects. The only notable differences are that the new process has a unique process ID (`p_pid` in its `proc` structure) and its own virtual memory view, and that all of its scheduling-related parameters, performance counters, and pending signal counts have been initialized.

Both the parent and child share the same execution trace and return from the same system call, but they have different return values passed to them. In this manner, each process can determine its role.

A parent may create any number of identical copies of itself through fork calls, which may be desirable in the case of a system daemon wishing to create a number of clones to handle individual client requests. Since there is an endless variety of programs we may wish to run, a process needs a way to morph itself into a different process—changing its stripes, so to speak. To this end, another call is key in process creation: once a child process has been created and has determined that its function is to usher in a new process image, it places the `exec()` system call.

When the `exec()` call is made, the kernel is asked to change out one process logical view for another while maintaining its kernel-resident run environment and basic identity. To put it another way, the kernel structures that make a process unique, are required for its management, and contain its environmental credentials are preserved, but all those directly associated with the current process's text and data are removed. The kernel then rebuilds the logical view in accordance with the new process's requirements (as defined in its executable program file). In effect, the kernel waves a magic wand and changes one process into another!

As the work of a process is finished, one of its threads must make one final system call to `exit()`. The kernel then starts the termination process. It is at this time that an interesting communication takes place between the kernel and the parent process of the process that is being deconstructed. When a process calls `exit()`, the kernel notifies the parent of the process of the impending death of its child. Before the kernel may complete the deconstruction and cleanup of the child's system resources, the parent process must acknowledge receipt of the kernel's signal and respond appropriately. Failure to do so results in the child process remaining in limbo and becoming what is commonly called a *zombie* process. Most versions of UNIX label a process in this state as *defunct* (political correctness rears its ugly head again—we wouldn't want to offend the undead!).

Before we explore each of these system calls in detail, let's consider a simple parent–child scenario.

A Simple Scenario

Consider the simple case of a user running a POSIX shell and issuing the command:

```
$ more /etc/hosts
```

In the beginning, we have a process environment for the initial POSIX shell program (`pid` = 100). The resulting child (`pid` = 101) will become a run environment for the `more` program. Let's break it down into specific actions:

- The shell must parse the line and isolate the first argument.
- The argument must be checked against the shell's list of intrinsic shell keywords, aliases, and function names.
- Assuming no match, the shell must search for an executable file that matches the command name; the contents of the shell's `PATH` variable tells it where and in what order to conduct the search.
- Once the executable has been located, its access and permissions are checked. If suitable, then the parent shell creates a clone of itself using either a `fork()` or `vfork()` system call.
- In the case of a typical interactive shell, the parent process requests that it be put to sleep and awoken when the child process requests an `exit()`. This is commonly called running the command in the foreground and is accomplished by the parent process making a `wait()` system call with the appropriate arguments immediately upon the return from the fork call.
- The child process calls `exec()` immediately upon return from the fork call. It requests that its process environment be replaced by that of the `more` program and that execution continue with the first executable line of the `more` program code.
- Once the child completes its task, it calls `exit()`, resulting in the parent being notified of the death of the child, and since the parent is currently waiting on just such a signal, it will wake and proceed.
- The kernel is now free to complete the deconstruction of the child and free its resources.
- In most cases, the shell then issues another prompt to the user, and the process starts all over again.

As a side note, consider that a parent process may choose not to wait for the death of its child (called running the child in background mode), but it still must respond to the receipt of the death-of-child signal with one of several variations of the `wait()` system call. The kernel cannot complete the deconstruction of a child process until the parent's wait handshake is received. When a child calls `exit()`, it is placed in a zombie state until the parent handshake is received (or until the system is rebooted).

Another point to consider is that if a process is multithreaded, the child resulting from a fork will have only one initial thread regardless of how many active threads the parent had at the time the call was made. As a

related point, if any active thread of a multithreaded process calls `exit()`, then the process and all sibling threads are halted and deconstructed. The process and the thread that made the `exit()` call remain in the zombie state until the parent handshake is received.

Which Came First?

So now, the age old question: which came first, the chicken or the egg? Since all processes start out as the cloned child of a parent, how does the first process get its start? In the world of HP-UX (and UNIX in general), that auspicious honor belongs to the first user-level process named `init` and assigned process identifier `p_pid=1`.

The `init` process is handcrafted by the kernel during system initialization and linked to a process and thread table structure (more on this later). The `init` process takes its marching orders from the `/etc/inittab` file, starting many processes, including the infamous `/sbin/rc` program, which in turn initiates most of the system daemons and services. Next time you are asked the chicken and egg question, simply smile a knowing smile and answer "init!" (If there was ever any doubt, your UNIX-savvy friends will now know for sure you are a fellow geek!)

Now that we have taken a first look at the process life cycle, we examine each of the related system calls in greater detail.

A Historic Look at the `fork()` Call

The original UNIX `fork()` call required that the full scope of the calling process's kernel run environment be duplicated. This included the allocation of new process and thread table structures, the duplication of the entire `vas`/`pregion` structures, and creation of new region structures in the case of private memory regions and copies of their data pages. If the `fork()` was called in prelude to an `exec()` call, then much of this work was almost immediately deconstructed, as the kernel replaced the original process view with the new one.

Another Approach: The Berkeley Virtual Fork

With the introduction of Berkeley's virtual memory UNIX kernel, a new "virtual" fork call was added to the growing system call list. The Berkeley kernel hacks decided that in the case of a "`fork()` immediately followed by `exec()`" scenario, it would make sense to simply allow the new child process to run in the parent's virtual memory space until it was time to start building its own view. This methodology was implemented in the Berkeley `vfork()` call.

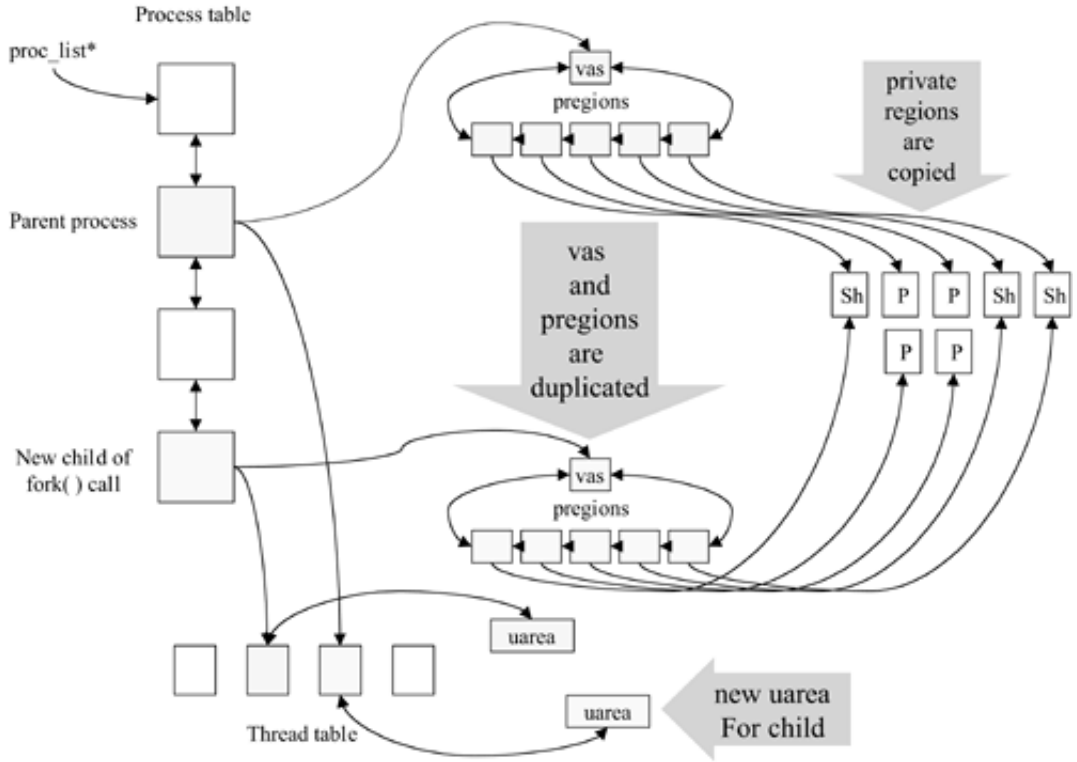
The idea behind the `vfork()` may seem very logical at first, but it flies in the face of many basic UNIX rules. The basic kernel memory management subsystem is designed specifically to make sure that one process doesn't corrupt the memory space of another. With the `vfork()`, there is no such protection between the parent and the newly created child. A child process of a `vfork()` should not modify any of the parent's data, but this is only a programming convention, and in actual practice there are no guarantees. To some, this aspect of Berkeley's `vfork()` was considered a bug, while other creative programmers considered it a feature and used it to provide a very primitive form of threading. We don't recommend this approach by any means! To address these concerns, Hewlett-Packard made several modifications to the basic `fork()` and `vfork()` calls over the years.

HP's New and Improved `fork()`

To improve system performance and avoid the potential corruption of the parent data by the child during a `vfork()`, HP reworked the mechanics of the basic `fork()` call and redirected the `vfork()` call to the same entry point in the kernel.

Hewlett-Packard's `fork()` was built around the concept of copy-on-write for the parent and copy-on-access for the child (referred to after this as copy-on-write/access). As in the older fork model and as illustrated in Figure 9-2, a new process structure is allocated and the parent's `vas`, `pregion`, and private memory `region` structures are copied. What makes this version different is that while private memory `region` structures are copied, their current in-core pages are not. Active `vfd` entries in the newly copied child `region` point to the same physical pages as those of the parent. This means that the parent and child have access to the same physical data space. To assure that no data corruption occurs, all the core resident data pages have their `pdir` entries, and `tlb` entries invalidated.

Figure 9-2. The `fork()`



When a page fault occurs for one of these pages, the kernel fault handler discovers a valid `vfd` entry. If the parent causes the fault, there is an entry in the `pdir`; if the child causes the fault, there is no `pdir` entry, as it is using a newly allocated space ID and thus has its own unique virtual page number. This seeming inconsistency in the synchronization of the various kernel and process memory-mapping structures is used to determine the disposition of the fault.

If either thread (parent or child) is attempting a first write, a copy of the page data is made and mapped to the faulting thread's `region`. A first read request by the parent is allowed, but if the request is from the child (indicated by the lack of a valid `pdir` entry), the kernel makes a copy of the data page for the child and maps it in the appropriate kernel tables. In this manner, each process ends up with its own private copy of the data, but the overhead of creating the page copies is postponed until they are actually accessed. If the `fork()` is followed by an `exec()`, we will not have wasted our time copying pages unnecessarily.

Later, Hewlett-Packard upgraded its `fork()` call to work with full copy-on-write access rules. As the parent and the child each has its own private data quadrants with unique space IDs, the kernel needed the ability to alias a single physical page to multiple virtual pages. This feature was introduced with the HP-UX 10.0 kernel release. The advantage to true copy-on-write is that a read access by either the parent or the child is allowed to function in a shared mode, avoiding the overhead of a page copy. In the case of a write request, a copy is made if the page shows more than one virtual address currently mapped (this means that the last to access the page ends up with the original copy).

The only complication to the HP approach is that since we create new private memory regions, swap reservations must be made. When this feature was first introduced in a workstation release of HP-UX several years back, it created a bit of a predicament. If a large parent program, such as a computer-aided design program with a relatively large private data region, needed to fork a simple child, say, to perform some menial task such as checking the user's email, the initial `fork()` resulted in a large swap reservation, which was immediately deconstructed by the following `exec()` call. This required that the system be configured with a large amount of swap space that wasn't actually used. As disk space was still quite expensive at the time, the original Berkeley `vfork()` was patched back into the kernel to avoid having to over state the system's swap space.

Hewlett-Packard also revisited the `vfork()` call and modified it to allow the child to simply map the parent's `vas`, `pregion`, and `region` structures to its `proc` table. Because this gives the child unrestricted access to the parent's data space, the only supported use of this call is when it is immediately followed by an `exec()`

or `exit()` in the child's logic.

As you may be thinking, this gets to be a bit confusing, so let's clear the deck a bit and create a timeline of the various flavors of `fork()` and `vfork()` that have been and are being used. Then, we examine the semantics of the current implementation of each call.

- HP-UX 7.x: `fork()` uses the original UNIX model and copies the parent's data area, and `vfork()` uses the original Berkeley model, sharing the data area between the parent and the child.
- HP-UX 8.x: HP-UX servers and workstations used the new copy-on-write/access `fork()` model implemented to handle both `fork()` and `vfork()` calls. The previously mentioned issue with swap reservation caused a patch for the workstation release of the O/S, which reimplements the original `vfork()` code.
- HP-UX 9.x: Servers use the combined copy-on-write/access `fork()`/`vfork()` calls, while workstations used the copy-on-write/access `fork()` and the original Berkeley `vfork()`.
- HP-UX 10.x and beyond (at least through 11i, the current release at time of writing): Servers and workstations alike use the new, true copy-on-write `fork()` call and a newer improved version of the `vfork()` call.

The fork1() Kernel Routine

Today the two system calls fork() and vfork() enter the kernel at the same point: fork1(). This kernel routine looks for the passed forktype argument (set to either FORK_PROCESS or FORK_VFORK) to determine which set of rules to play by. In either case, a number of kernel procedures are used. Let's first examine the actions for forktype=FORK_PROCESS, the full copy-on-write version.

The fork() System Call Mechanics

After entering fork1(), calls are made to obtain unique process and thread identification numbers. The current release of HP-UX reserves pid 0 through 7 for system use (0 for the swapper, 1 for init, 2 for page-out, 3 for statdaemon, 4 for the unhash daemon, 5 for netisr, 6 for the socket registration daemon, and 7 for commit). Thread identification numbers are unique within the scope of the kernel, and in a similar manner tids 0 to 7 are reserved for the system threads associated with the first eight system processes.

> **NOTE**
>
> in the future, HP-UX may be modified to allow each process to start numbering its threads with kt_tid = 0, but for now all threads have a unique tid.

Once we have a unique process and thread ID, we allocate the process table and thread table structures. Kernel routines check for available structures by following the freeproc_list and freekthread pointers. Although both of these structures are currently maintained as dynamic lists, the current implementation does not return unused process and thread structures to the kernel memory arenas; instead the structures are returned to the appropriate free list when their process or thread is deconstructed.

These free lists maintain a kind of high-water mark for structure utilization and save time when the kernel is asked to find new process or thread structures. If a free list is empty and we haven't exceeded the respective kernel parameter nproc or nkthread (the tunable maximums), the kernel memory allocator is asked to provide space for the new structure.

As our new structures are allocated, they are initialized and set to the idle state. The newly allocated proc and kthread structures are linked together and attached to a variety of hash chains.

- tidhash[] links kthread structures to chains based on the tid.
- pidhash[] links proc structures to chains based on the pid.
- uidhash[] links proc structures to chains based on the UID.
- sidhash[] links proc structures to chains based on the session ID.

These hashtables provide quick access to the process and thread structures based on common attributes. Once this work is completed, the process and thread table entries are linked to their appropriate active lists.

Now switch() is called to pass control to the kernel routine newproc(). Before we continue with the creation of our new process, a sanity check is made checking the thread state of the new child's kthread structure. If it is anything other than kt_stat=TSIDL, then the kernel panics, as we shouldn't be here under any other circumstance. Assuming we pass this test, the child's proc and kthread structures inherit basic data from the parent's environment, such as the UID and GID. We reload the child's scheduling and usage counters, and temporarily disable swapping for the parent (after all, we will be copying the parent's memory view to the child and don't want vhand attempting to swap it out while we are making our copy).

Now, as we are rounding the far turn and entering the home stretch, control is passed to procdup(), where we start duplicating the parent vas, pregion, and private region structures to create a new memory view

for the child.

Copying a Process Memory View

The kernel manages regions of pages to be used by a process for any of a number of purposes. Each process maintains its own virtual memory view, facilitated by assigning virtual space values to work in conjunction with an offset address for each region. Since the virtual view belongs primarily to the process, it is in the process `pregion` that `p_space` and `p_vaddr` (the space and offset used to create a virtual address) are recorded. The entire `vas`/`pregion` structure of the parent process must be duplicated for the child and attached to its `proc` structure (using `p_vas`).

The kernel checks each `pregion` in the parent's list to see if it points to a `region` of type `RT_SHARED` or `RT_PRIVATE`. In the case of a shared region, the kernel needs only copy the `pregion` to the child's list and increment the reference count (`r_refcnt`) in the `region`. In the case of a private region the kernel must build a duplicate `region` and `pregion` and map it to the child's list. This involves duplicating the region's page list (containing the b-tree's `broot`, `bnode`s, and chunks) and reserving swap space for it and for the potential number of pages it could map. A unique "space" value must also be obtained so that the child will have its own virtual view.

This is where copy-on-write comes into play. As a duplicate private region is created, all of its `vfd`s are set to copy-on-write, and any in-core page frames have their usage count incremented (`pf_use++`) in their `pfdat`. The page's entry in the `pdir` is flagged for copy-on-write behavior (this includes updating multiple `pde`s in the case of an aliased page), and any existing `tlb` entries are purged. This assures a page fault on first access and allows the fault handler to update access information and physical page copying at the time if necessary. In the case of a child making an immediate `exec()`, we have minimized the amount of page copying done at the time of the `fork()`.

Let's examine what happens when a copy-on-write page is accessed.

Copy-on-Write, First Read Access from the Parent

If the first access to the page is by the parent, the system experiences a `tlb` fault (since we purged the `tlb`s during the `region` duplication process). The resulting fault loads the translation from the `pdir` and sets the copy-on-write access bit.

We try one more time and, as there is now a valid translation in the `tlb` and this is a read request, access is granted (effectively creating a *shared read* access mode).

Copy-on-Write, First Read Access from a Child

If the first access to the page is by a child, a `tlb` fault occurs, no `pde` entry will exist (the child was placed in its own space, and its virtual page numbers have not been mapped in the `pdir`), and the kernel page fault handler is called next.

This handler determines the faulting thread's identity and follows pointers from its `kthread` structure to the `proc` structure to the `vas` to the `pregion` list. The `pregion`s are searched (using their `skiplist` links) to find the one containing the faulting virtual page. Finally, we calculate the page offset within the `region` and search the b-tree to locate the `vfd`/`dbd` data for the faulted page. This search will reveal a valid `vfd` entry with `pg_cw=1`.

Since this is a first access, the kernel also needs to add an alias entry to the `pfn_to_virt` table entry for this physical page before we continue. In this manner, we postpone the creation of the alias links until it is absolutely necessary. We also make an entry in the `pdir` for the new virtual page number. If the `pde` pointed to by the initial hash for the new virtual page is not available and we have to link to a sparse `pde`, we allocate it from the `aa_pdirfreelist`.

At this point the instruction is retried, but since the `tlb` has not been updated, it will fault. The handler finds the new valid `pde` in the `pdir` and uses it to update the `tlb`, setting the copy-on-write access bit in the process. We try one more time and finally our read request is allowed.

Copy-on-Write, First Write Access

As with a first read request, the virtual page translation will have been purged from the tlbs. For a parent the pdir will exist, and for a child there will not be a pdir entry. We mirror the actions taken during the first read mechanics we just discussed.

Once we get to the point where we have valid pdir and tlb entries, the instruction is retried but fails this time with an access protection fault (we tried to write to a page with the pg_cw bit set).

Once we enter the fault handler, the current page use count is checked. If pf_use=1 (in pfdat), then we know we are the last to reference the page and we simply change the access rights by clearing the copy-on-write bits in the tlb, pde, and vfd for the page. The instruction is retried, and this time around it should succeed. If, however, pf_use>1, then the kernel must create a copy of the page for use by this process.

To copy the page, we first need to get a write lock so no one may change its contents while our copy is in progress. We remove the current virtual-to-physical translations for the page (purge the tlb entry, free the pdir, and invalidate the vfd), as we will be creating new ones. Next, we find a free pfdat entry (or block until one is available) and update the appropriate virtual-to-physical and physical-to-virtual tables.

We set the access type in the pde to PDE_AR_KRW for our new page to keep anyone from accessing it as user data until we finish the copy. Once the copy is completed, the original page's pf_use is decremented and the new page's tlb is purged. It is necessary to purge the tlb because it was set during the copy process and we need the next access to fault and reload the access rights from the newly created pdir entry. Now we are ready to continue using our new private copy of the data.

Copy-on-Write for the Parent, Copy-on-Access for the Child

Before we move on, let's talk a bit about the older copy-on-write/access mechanics. In most cases, it behaves similarly to copy-on-write except that virtual page alias entries are not utilized and all first access by a child results in a private copy being made.

If the kernel parameter COW=1, copy-on-write is used for duplication of all r_type=RT_SHARED regions. Currently, the HP-UX kernel defaults to COW=0 and uses copy-on-write/access. The exception is in the case of private text regions encountered in EXEC_MAGIC programs, where it always uses copy-on-write (first implemented with the HP-UX 10.0 release). This parameter is not user-tunable in HP-UX 11.x and may only be changed using adb (not for the faint of heart).

In HP-UX 10.x, the COW was a user-tunable parameter. Why this has been changed remains a bit of a mystery, since Hewlett-Packard has stated it intends to move completely to copy-on-write in the future.

It's time to return to our discussion of duplicating a process's memory view during a vofk(). A special case exists when the region to be copied holds the current thread's uarea.

Creating the New uarea

The pregion referencing the parent's uarea is somewhat of a special case and is replaced by one for a new uarea created for the child's initial thread. Swap reservation must be performed to reserve space for the uarea and its b-tree structure. When a new uarea is created, its pregion is flagged as PF_NOPAGE to keep it from being paged by vhand. A thread's uarea may be paged but only if the entire process has been marked for deallocation/deactivation during times of extreme memory pressure.

Ready, Set, Run Queue

As soon as all the new kernel structures are created, we are ready to allow the new thread to compete for runtime. We change its proc table state to p_stat=SINUSE, the thread state to kt_stat=TSRUN, and link the kthread to the end of the appropriate run queue in accordance with its kt_spu_wanted value. We also flag the parent to allow swapping again, as we are through making copies.

In general, a new process and its first thread start on the same spu as the parent thread that created it. This makes sense because both initially share the same execution environment and need access to the same text page for their next instruction. See Listing 9.1.

Listing 9.1. q4> fields struct kthread (an edited listing)

```
These fields store the spu number of the run queue that the thread is currently on, the
➥ spu it would like to be on, and the spu group number (processor set) that it wants to be
➥ part of
 92 0  4 0 int      kt_spu
 96 0  4 0 int      kt_spu_wanted
100 0  4 0 int      kt_spu_group
The next field is used to express a mandatory processor
locking strategy (sometimes called a processor affinity lock)
104 0  1 0 u_char   kt_spu_mandatory
```

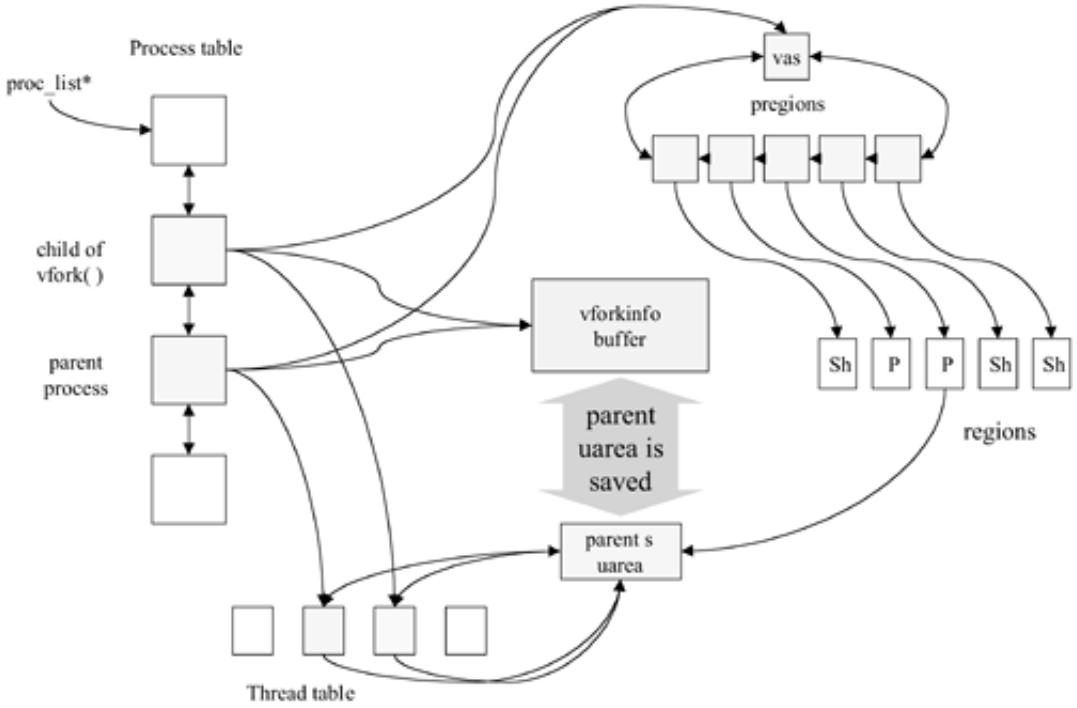
By sharing the same processor, we minimize cache-coherency issues that might arise if the parent and child were scheduled on separate processors. Various load-balancing mechanisms are employed by the HP-UX kernel to disperse threads to all the processors and locality domains on multiple processor systems. We cover processor load-balancing in greater detail when we talk about multiprocessor systems in Chapter 12, "Multiprocessing and HP-UX."

As we unwind from this litany of kernel procedural calls, an interesting situation occurs. We started down this path when a thread made a `fork()` system call. In most cases, every call results in a single return at some future point in time. With the `fork()` call, two separate returns result from a successful call, one in the parent context and one in the newly created child context. Upon returning from the `fork()`, the `pid` of the newly created child is passed to the parent thread, while the new child thread receives a 0 to let it know that it is the new kid on the block.

The `vfork()` System Call Mechanics

When you call `vfork()`, you enter the same kernel procedure, `fork1()`, but there are several deviations from the `fork()` sequence we just examined. The first point of departure occurs when control is passed to `newproc()`. In the case of `forktype=FORK_VFORK`, the newly created child shares the parent's memory view, which is accomplished by having the child's `proc` table simply point to the parent's `vas` structure, as shown in Figure 9-3.

Figure 9-3. The `vfork()`



To preserve the sanctity of the parent's data, the kernel allocates a `vforkinfo` buffer. This structure is used to hold state information about the `vfork()` and a copy of the parent's `uarea` and stack. See Listing 9.2.

Listing 9.2. `q4> fields struct vforkinfo (an edited listing)`

```
The first field is used to store the state of the vfork. Since
this is visible to both the parent and the child, the current
state of the call only needs to be stored in this one location
 0 0 4 0 enum4  vfork_state
Next we store a pointer to the parent and child kthread structure
 4 0 4 0 *      pthreadp
 8 0 4 0 *      cthreadp
We also store the number of buffer pages and the size of the
parent thread's uarea (which will be stored in the buffer)
12 0 4 0 int    buffer_pages
16 0 4 0 u_long u_and_stack_len
Space is provided to save the rp and a pointer to the last
vforkbuffer (to handle nesting of vforks)
20 0 4 0 *      saved_rp_ptr
24 0 4 0 long   saved_rp
This pointer directs us to the beginning of the uarea copy in
```

```
the buffer

28 0 4 0 *      u_and_stack_buf

Next is the pointer to the previous p_vforkbuf

32 0 4 0 *      prev

And finally a pointer to the parent thread's uarea

36 0 4 0 *      p_upreg
```

When the buffer is initialized, it is linked by the pointer `p_vforkbuf` in the `proc` tables of both the parent and child processes, and its state is set to `vfork_state=VFORK_INIT`. The structure remains in use until the child calls either `exit()` or `exec()`. As the waiting parent resumes operation, the parent's `uarea` and stack are restored from the buffer and its space is then freed.

The next step in the `vfork()` sequence is to check if the parent is a multithreaded process. If so, the thread obtains exclusive write access to the process memory view and suspends all the sibling threads. This is necessary because the child will soon begin running in the parent's memory view, sharing its `uarea` and data regions—we can't have sibling threads also working in the same view!

This is an important issue for programmers to understand. When deciding to use `vfork()` with a multithreaded process, until the child calls `exec()` or `exit()`, the parent and all of its siblings are suspended. This suspension should last a very short time, but in a symmetrical multiprocessing (SMP) environment with siblings distributed to various processors, this could result in a number of forced context switches taking place.

The next deviation for `vfork()` occurs when control is passed to `procdup()` where we set `vfork_state=VFORK_PARENT` in the `vforkinfo` buffer. The child is prepared for running, and the call returns in both the parent and the child. Note that in the `vfork` there is no time spent copying the `vas` and `pregion` structures, as the child's `proc` pointer, `p_vas`, simply points to the parent process's `vas`.

Following the return in the parent thread, a call is made to `sleep()`. As this call is entered, a check is made to see if the requesting thread is currently in a `vfork` and if `vfork_state=VFORK_PARENT`. If this is the case, several additional steps are taken. First, a calculation is made of the size of the parent's `uarea` and stack, and they are copied to the address pointed to by `vforkinfo _u_and_stack_buf` in the `vforkinfo` buffer so they may be restored when the parent wakes.

We now set `vfork_state=VFORK_CHILDRUN`, and the parent thread sleeps, waiting for a wakeup call. The correct use of the `vfork()` is for the child to immediately call either `exec()` or `exit()` upon its return. While a programmer may choose an alternate logic, any deviation from the stated use is not supported by Hewlett-Packard!

Before we take a look at `exec()`, let's spend a little time defining the process and `kthread` states.

Process and Thread States: Idle Hands

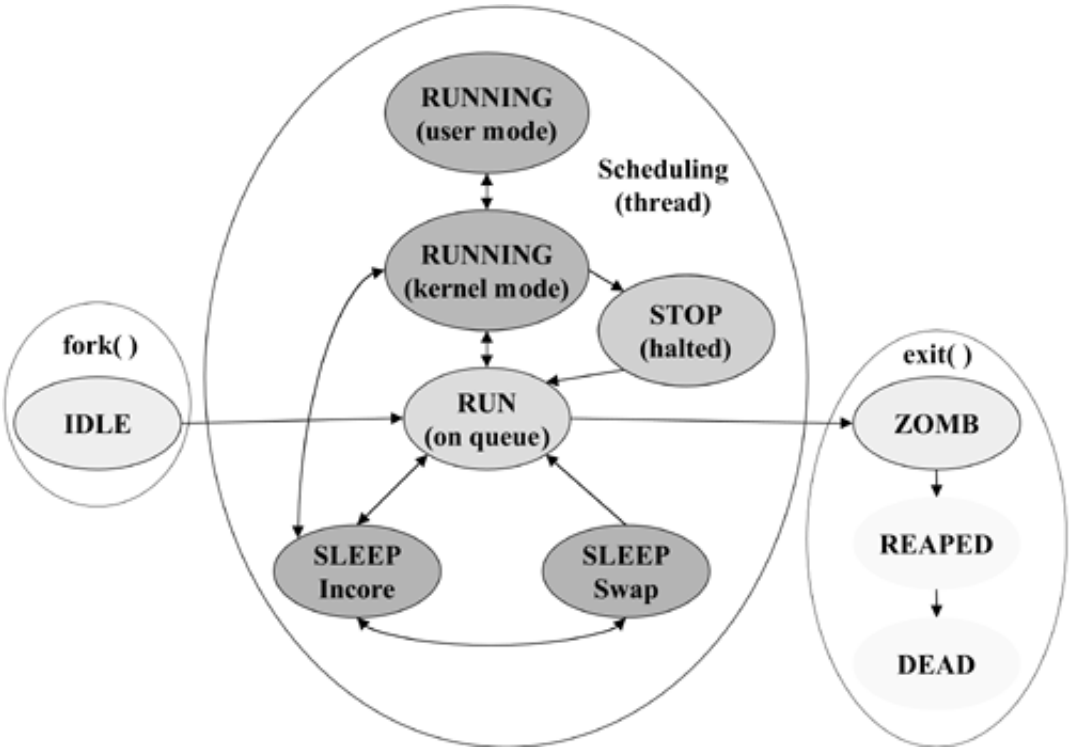
The *idle* state indicates that the kernel is in the middle of initializing a process/thread run environment. The thread can't be placed on a run queue, yet but soon it will be ready to compete with other threads for scheduling consideration. The amount of time a process/thread spends in the idle state is very short, and it is rare to see this state reported by a `ps` command. The common process table states, `p_stat`, are:

`SUNUSED` = 0	table entry unused (on the free list)
`SWAIT` = 1	entry abandoned
`SIDL` = 2	intermediate state during creation
`SZOMB` = 3	intermediate state during deconstruction
`SINUSE` = 5	entry in use

Prior to the advent of kernel threads, a process could also be in the state `SRUN`, waiting on an active run queue; `SSLEEP`, blocked or sleeping; or `SSTOP`, halted. With the introduction of kernel threads, these three scheduling-oriented process states were replaced with the inclusive `SINUSE` state designator for the process.

As the current scheduler works directly with kernel threads, the `TSRUN`, `TSSLEEP`, and `TSSTOP` states are set at the thread level and recorded in the `kt_stat` field. Figure 9-4 abstracts the overall process life cycle and the relationship between the various states.

Figure 9-4. Process Life Cycle



The common thread table states are:

`TSUNUSED`	= 0	table entry unused (on the free list)
`TSSLEEP`	= 1	thread is on a sleep queue
`TSRUN`	= 2	thread is on a run queue
`TSIDL`	= 3	intermediate state during construction
`TSZOMB`	= 4	intermediate state during deconstruction
`TSSTOP`	= 5	thread is halted

The state `TSSUSP` is also defined but is not currently used. Threads that have been halted by either a debugger or by job control are reporting the `TSSTOP` state at this time.

PREV < Day Day Up > NEXT

Process Identity Crisis: The exec() System Call

When an established thread desires to replace its logical memory view with that of another program, "changing its stripes," so to speak, it places a system call to exec(). The exec() call is passed the name of an executable file. The file may be either a compiled binary image or a text file to be processed as a script by an interpreter program such as the POSIX shell or PERL.

In the case of a compiled program, the file starts with a header written in a format recognizable to the operating system and containing a "magic number" indicating the hardware platform, O/S type, and version for which the code was compiled. If the magic number does not match those acceptable to the kernel, an error message stating "bad magic" is generated and the exec() call fails. HP-UX 32-bit executables use the Spectrum Object Module (SOM) header format. The name echoes the history of the PA-RISC processor design, which was originally called the spectrum computing family. When the 64-bit PA-RISC hardware was introduced (PA-RISC 2.0), HP-UX adopted the POSIX ELF-64 header format for wide executables (there is an ELF-32 definition, but HP-UX does not currently use it).

The Kernel's Draft Horse, getxfile()

The exec() call relies heavily on the kernel routine getxfile() to do the majority of its work. The sequence of actions taken depends on the type of fork used to create the calling thread. The first challenge is to determine if the filename passed is a suitable object for running. A program file must not be open to any process for writing; the file must be executable on the system (magic number check), and the requesting thread's UID or GID must have the proper credentials for execution. Assuming these tests are passed in the case of fork()/exec(), the vas and pregion lists must be deconstructed, while for the vfork()/exec() case, we must create a uarea, vas, and pregion/region list from scratch. In both cases, getxfile() must then build the new logical memory view in the system's virtual space.

Calling exec() Immediately After vfork()

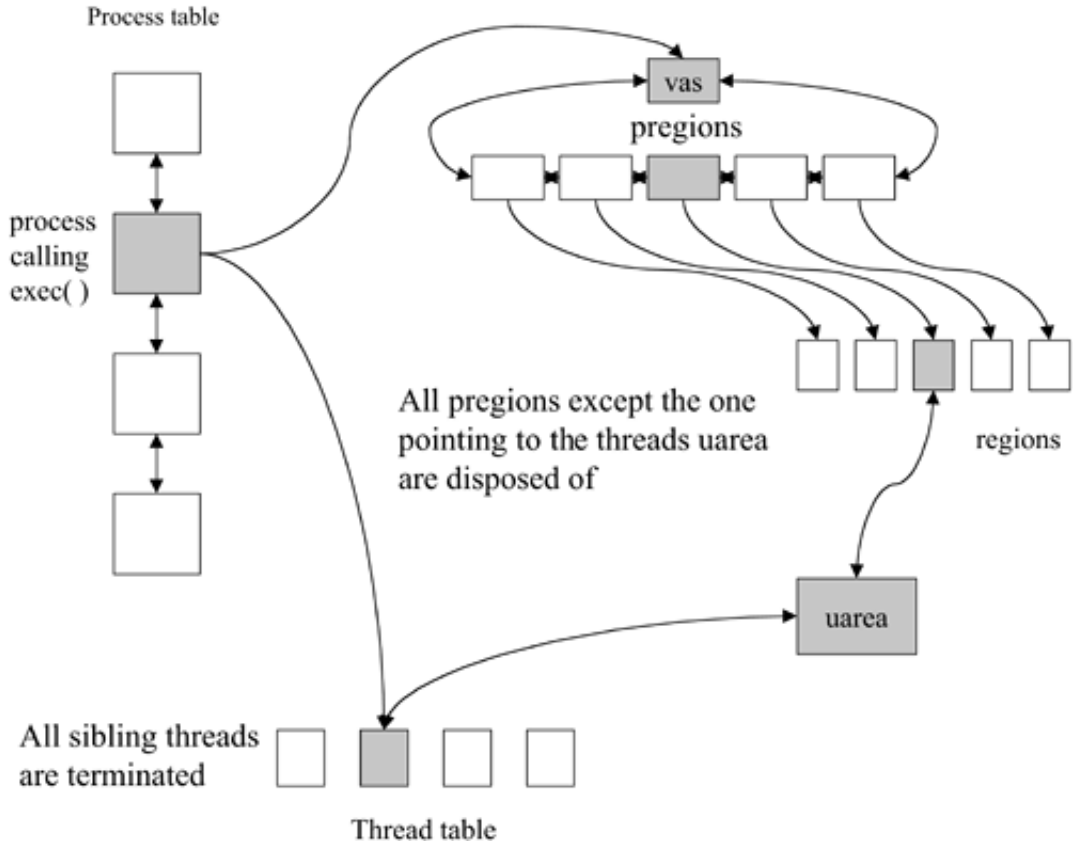
In the vfork()/exec() case (where vfork_state=VFORK_CHILDRUN), we first need to request the creation of a new uarea for the thread along with minimal vas, pregion, and region structures. Next, the content of the parent's uarea (the one we are currently using) is copied to the new one, and another routine is called to switch the thread's execution trace to the new uarea and kernel stack.

The vfork_state is changed to VFORK_CHILDEXIT, and we initiate the cleanup of the parent's context. Once the parent uarea and stack have been restored, the parent wakes and releases the vforkinfo buffer as it resumes.

Calling exec() after a fork()

Following a call to fork(), the newly created child has a vas, pregions, private and shared regions, and an initial thread and uarea. In this case getxfile() must dispose of most of the existing pregions and regions. This function is performed by the kernel procedure dispreg(). Figure 9-5 shows the state of a process's memory view after the disposition of its old image.

Figure 9-5. The exec() Call: Disposing of Old Regions



Removing `pregions`, `regions`, and b-trees

The first step in removing `pregion`s, `region`s, and b-trees is to acquire a write lock for the process and terminate any existing sibling threads. Next, we walk the `pregion` list and dispose of all except our thread's `p_type=PT_UAREA`. If we are attempting to perform an *exec self* (defined as a process attempting to execute itself with another copy of the same program image), we also save the `p_type=PT_TEXT` and `p_type=PT_NULLDREF pregion/regions` to save time when we reconstruct the new logical view.

Disposing of a `pregion` involves several steps. First, it must be removed from the active list. If `vhand`'s `agehand` or `stealhand` is currently pointing to it, they are moved to `p_next` so `vhand` won't waste time. Next, we follow the `p_reg` pointer to its associated `region` structure.

If we are the last `pregion` to reference the `region` (`r_refcnt=1`), we schedule its cleanup (if others are still referencing an `RT_SHARED` region, we simply decrement its `r_refcnt`). After dealing with the `region`, each `pregion` structure is also freed and returned to the kernel memory arenas.

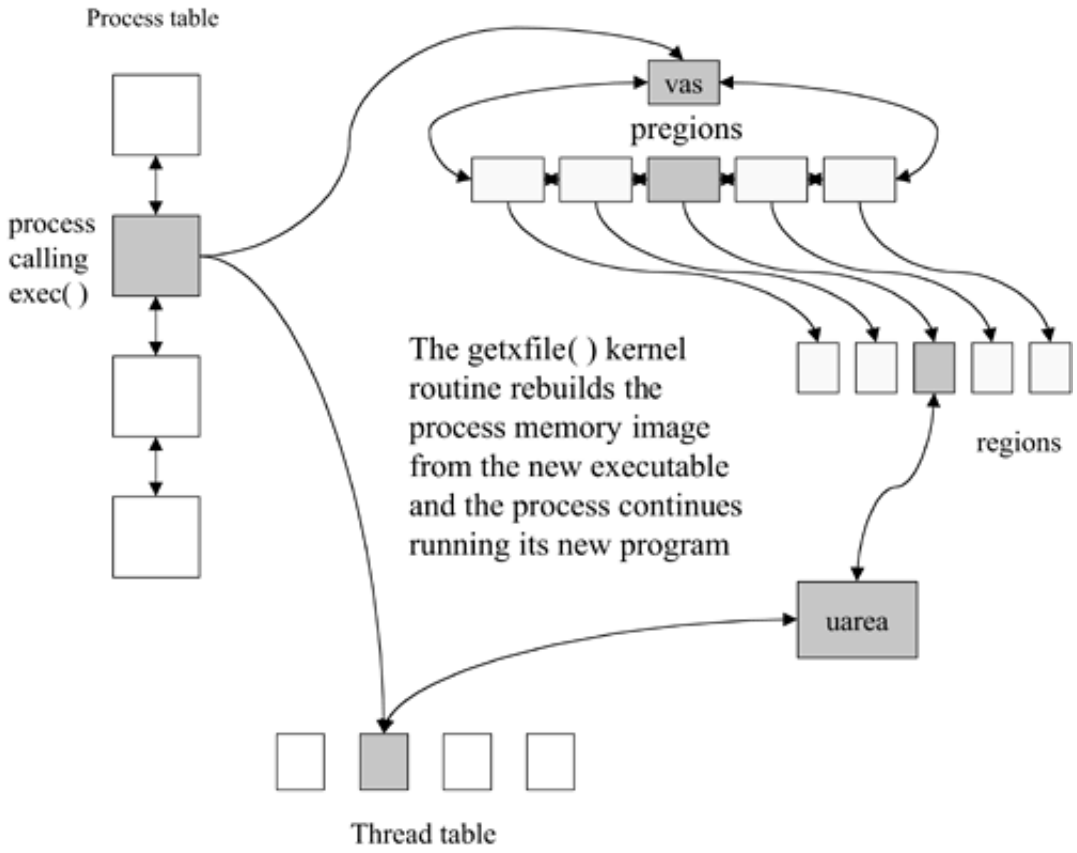
The actual cleanup of a region requires that we wait for any pending I/O operations to its pages to complete. We then walk the b-tree and delete all virtual address translations, purging them from the processor TLBs and freeing the associated `pdir` entries. If the virtual-to-physical `pde` was allocated from an alias or sparse `pdir` entry, it is returned to the appropriate free list. If it was located in the machine-visible hashtable, it is simply marked as invalid (`pde_valid` is cleared). If this is the only virtual reference to a physical page (`pf_use=1` in the page's `pfdat`), we must free the physical page and adjust the system's available page count, `freemem` (if any process threads are currently blocked, waiting on a memory page, we wake them). The page entry in `pfdat_ptr` and `virt_to_virt_prt` is updated, and appropriate adjustments to the swap reservation and allocation structures are also made.

Finally, we release any reserved or allocated swap and physical pages used to hold the `region`'s page list (the pages needed to hold the b-tree nodes and chunks). The `region` is unlinked from the forward and backward pointers to the systemwide region list, and we return its structure to the kernel memory arena.

Building the New Logical View

At this point, regardless of which path we followed, `getxfile()` sets up the new logical view and maps it within the system's virtual address space (see Figure 9-6). To take stock, we have a new `proc` and `kthread` structure and a minimal `vas` with a `uarea`.

Figure 9-6. `getxfile()` Rebuilds the Memory View



The `exec` routine opens the program file and passes its `vnode` to `getxfile()` along with its attributes and load-specific information from the header. If the program file is a text file, the `vnode` passes points to the executable image of the interpreter program, which is to be run to process the script.

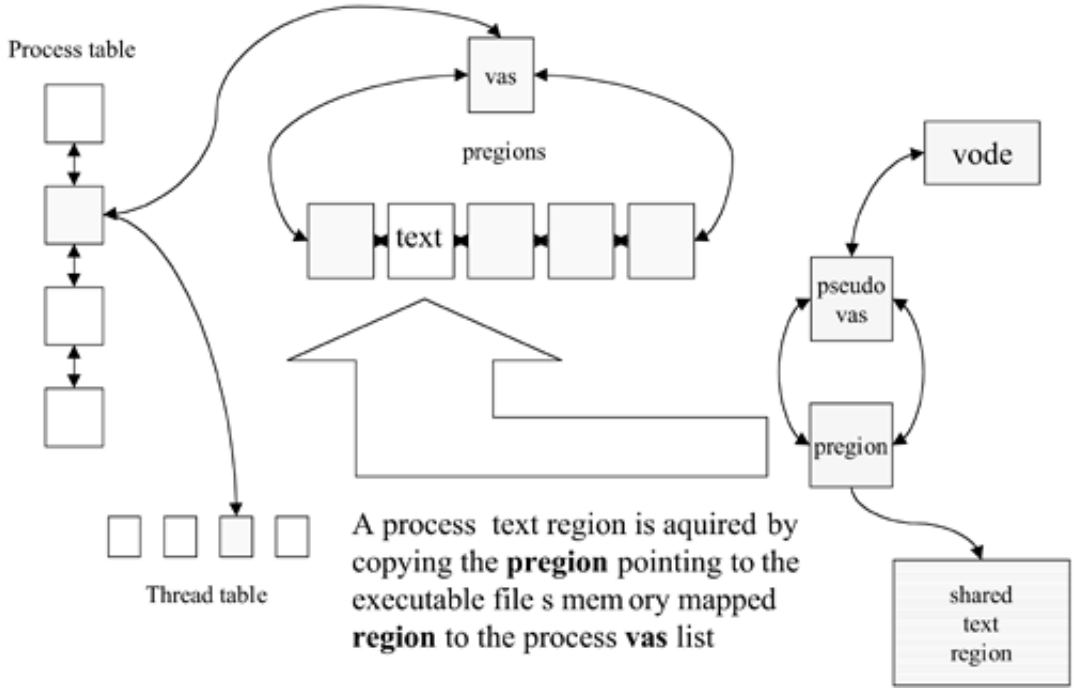
Now it's finally time to build the new image map! The compiler magic type tells us what type of mapping (`DEMAND_MAGIC`, `EXEC_MAGIC`, `SHMEM_MAGIC`, `SHARE_MAGIC`) to use as we locate the regions in quadrants. First, the null dereference and text regions are configured and mapped (if we are doing an exec self, then these regions will have been retained). Private regions for data (initialized, BSS, and heap) and user stack are added next. Additional regions are added as required for shared memory objects and memory-mapped files (both private and shared varieties).

The transformation is now complete, and we return from the `exec()` call wearing our new image and ready to run its code. Before we examine the final call in a the process's life cycle, `exit()`, let's take a closer look at the way shared memory objects are mapped into the system's virtual space and attached to a process's logical view.

Shared Memory Objects Revisited

When a program filename is passed to `exec()`, it must find the file (also called the front store) and open it. If the file is already open, there will be an existing `vnode` for the file in the kernel memory. If not, a new `vnode`, pseudo `vas`, a single `pregion`, and a `region` are created, as illustrated in Figure 9-7. In most cases, a `vas` structure is pointed to by a `proc` structure (`p_vas`), but in the case of an executable or memory-mapped file, it is a `vnode` (`v_vas`) that references it. In turn, the `vas` (`va_fp`) points back to the `vnode` for which it was created.

Figure 9-7. Executable and Memory Mapped Files

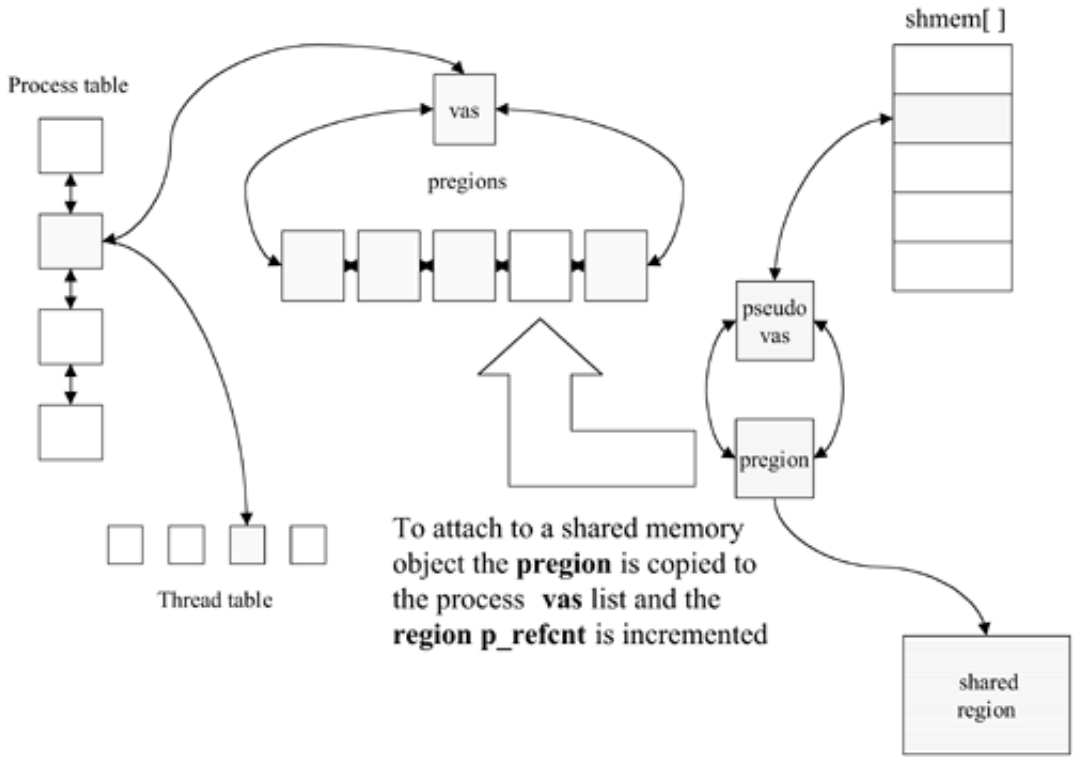


When a process needs to attach a text region or a memory-mapped file to its virtual view, it simply copies the file's `pregion` structure and increments the associated `region` reference count.

Shared Memory, System-V, and POSIX

In a very similar manner system-V, shared memory regions are also maintained by first allocating an entry from the system's shared memory descriptor table, `shmem[]`. An active table entry points to a pseudo `vas`, `pregion`, and `region` created to manage the shared memory space (see Figure 9-8). The System-V shared memory allocation, attach, detach, and deallocation requests are mapped through this table and its related structures.

Figure 9-8. Shared Memory Objects



The following kernel-tunable parameters are used to size and configure System-V shared memory capabilities.

- `shmmni` sets the maximum number of shared memory regions the kernel may manage.
- `shmmax` sets the maximum size in bytes for a shared memory region.
- `shmseg` sets the maximum number of shared memory regions a single process may attach to its virtual view at one time.

System-V shared memory regions are managed as independent kernel resources; that is, they may continue to exist even if there are no current processes attached to them. If a programmer decides to use shared memory regions, he or she is responsible for their allocation and eventual cleanup. The administrative commands `ipcs` and `ipcrm` may be used to check the status of shared memory regions and, if necessary, to remove them from the system.

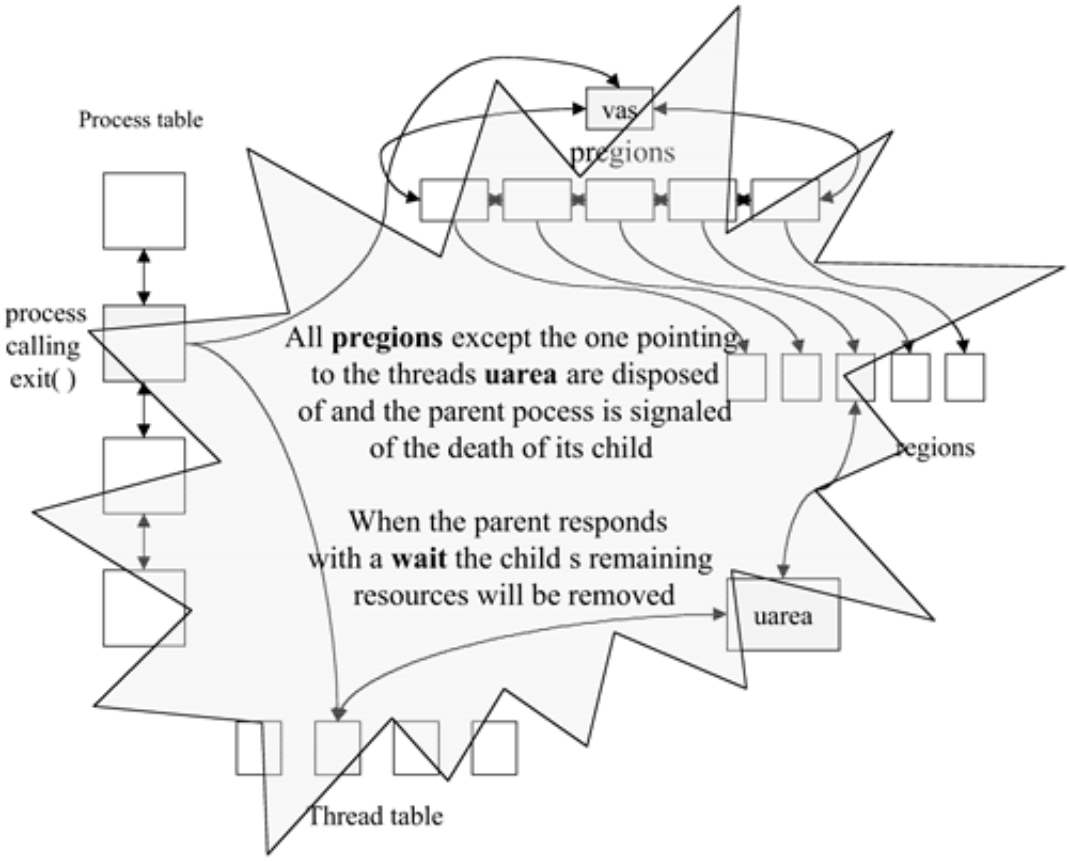
Note that some programmatic schemes involve creating a shared memory region and having it persist as various daemons or agents are launched and momentarily attach to and detach from it. In a production environment, just because a shared memory region has no processes currently attached does not necessarily mean that you should remove it!

The POSIX system definition also has a set of shared memory calls. The calls are implemented on HP-UX as shared memory mapped files. Programmers are encouraged to use the System-V shared memory calls over the POSIX calls but both will function.

The `exit()` System Call Mechanics

When a thread makes an `exit()` system call, it is asking to be removed from its run queue and to have its cumulative resources returned to the kernel for dispersal to their appropriate free lists and arenas (Figure 9-9). As we mentioned earlier, when any thread of a multithreaded process calls `exit()`, all of its threads are halted and removed from kernel tables and structures.

Figure 9-9. The `exit()`



Actually, there are two variations of the exit call, the original `exit()` and `__exit()`. When a programmer codes a call to `exit()`, an additional call is made to `atexit()`. The idea is that the programmer may wish to have specific functions performed as part of the exit procedure (i.e., some standard I/O library routines require a final "flush" of buffers to avoid data loss when a program terminates). Since an `exit()` may be called on behalf of a thread by the kernel (as the result of a fault or signal) or at the request of another process (by the `kill` command or system call), a programmer may need to schedule cleanup actions as part of the program's exit logic. This feature is embedded into the basic `exit()` call.

There are some conditions in which these cleanup actions may not be desired or in fact may cause problems. When is an executing thread not running in its own run environment? During a `vfork()`. If a child of a `vfork` calls `exit()` while it is still running in the parent context, then any scheduled cleanup actions are actually modifying the parent environment. Once the child completes its exit, the parent may wake to find that its context has been corrupted. For this reason and others, the `__exit()` call was created. This call

bypasses the call to `atexit()`. It proceeds directly to exit, does not pass atexit, and does not collect $200! In the kernel both calls are handled by `exit1()`.

Walk, Don't Run, to the `exit1()` Nearest You

As part of the kernel mechanics, when a thread calls `exit()` or `__exit()`, or when a fault results in the kernel scheduling `exit()` on behalf of a thread, a sequence of events takes place involving the kernel, the exiting process, and the parent process. In the case of a multithreaded process, all sibling threads are immediately halted, and in all cases deconstruction of the process's resources is begun. A write lock is obtained for the process (we don't want two siblings attempting to exit at the same time). We set `p_flag=SWEXIT`, ignore all signals, release semaphores, cancel pending callouts, and release our virtual memory regions (provided we are the last active process to reference them; for shared regions, it is a case of "last one out turns off the lights"). We close all open file descriptors, perform any requested Sys-V semaphore undos, and destroy any adopted processes. If the exiting process has children, we turn them over to `init` and send a `SIGCHLD` to `init` if there are any newly adopted zombies waiting for the reaper. Finally, the `kthread` and `proc` structures are removed from their active lists.

All kernel structures are released with the exception of the `proc`, calling `kthread`, `vas`, and one remaining `uarea`, `region`, and `pregion`. The state in the `proc` and `kthread` structures are set to `p_stat=SZOMB` and `kt_stat=TSZOMB` respectively.

A special case exists when the thread calling exit is a first-generation result of a `vfork()` call. In this case (indicated by the `proc` structure `p_flag = SVFORK`), we set `vfork_state=VFORK_CHILDEXIT` in the `vforkinfo` buffer. The kernel deconstructs the child and wakes the parent. When the kernel next schedules the parent thread to run, the context switch, `swtch()`, calls `resume()` to restore the thread's save state. In the case of `vfork_state=VFORK_CHILDEXIT`, the save state comes from the `vforkinfo` buffer. The buffer is released, any sibling threads that were suspended during the `vfork` are placed back on their run queues, and the parent thread continues.

Night of the Zombies: Responsible Parenting

At this point, the kernel notifies the parent process (as indicated in the exiting process's `p_ppid` field) of the death of its child. This is accomplished by sending a signal to the parent process. The actual signal is called `SIGCHLD`. It is the responsibility of the parent process to respond to this signal using one of several system calls.

The calls fall into the generic label of `wait` and consist of four variations, although once we get past the system call interface, they are all processed by the same kernel procedure, `wait1()`. The four user-callable versions of `wait` are as follows:

- `wait(*stat_loc)`: This is the classic wait call and suspends the calling thread until status information about one of its terminated or stopped children may be returned. If there is a waiting zombie when the call is made, it returns immediately.
- `wait3(*stat_loc, options, *resource_usage)`: This call allows the requesting thread to be specific about which child process it wishes to receive status information about. In addition, there are three optional flags that may be passed:

    `WUNTRACED` to receive status information about children that have stopped due to a signal receipt

    `WNOHANG`, which keeps the calling thread from suspending if no child status is available

    `WNOWAIT`, which causes the request to not be registered and allows a later wait to be requested against the same child if used in conjunction with `WNOHANG` when the child is not ready to exit

- `waitid(idtype, id,*infop, options)`: This call is used if the parent wants to wait for a child to simply change state. Three focus options are allowed by passing the appropriate `idtype` argument:

`P_PID` to specify a specific process ID

`P_PGID` to specify any process within a specific process group

`P_ALL` to wait for any of its children

In addition, the following options may be specified:

`WEXITED` to wait only for children that have exited

`WSTOPPED` to wait for a stopped child

`WCONTINUED` to wait for a child to continue

`WNOHANG` and `WNOWAIT` which function the same as in the `wait3` call

- `waitpid(pid, *stat_loc, options)`: This version functions the same as the wait call if the passed `pid=-1`. If `pid>0`, then the call is for a specific child. If `pid<-1`, then the call is for any child whose process group is equal to `pid` x `-1` (absolute value). The options available are similar to those in the `wait3` call.

In all four `wait` variations, version control is passed by the system call interface to the kernel routine `wait1()`, where the real work is done. First, a search is made for zombie children. If they are found, their status is passed back to the caller and the zombie is laid to rest. This is called *reaping a thread*. Stopped, traced, and continued children may also be looked for depending on the actual call specifics.

Before `wait1()` may modify any of the child resources, a lock must be acquired to make sure that the thread is being reaped only in response to a single request. The kernel also reaps any adopted zombies that may be queued for its attention.

The Grim Reaper Pays a Visit

When the kernel reaps a `kthread` with `kt_stat=SZOMB`, the thread's final resource usage statistics are added to its process, and it is laid to rest with a call to `kissofdeath()`. The reaped `kthread` structure is placed on the `freekthread_list`. Next, the process itself is reaped in a similar manner. Its resource usage statistics are added to the process that called the `wait`, and the `proc` structure is placed on the `freeproc_list`. A call to `abandonchild()` removes it from the parent's list of children. (Don't you just love this stuff? I mean, what a violent piece of code. We have daemons, zombies, kill processes, death-of-child signals, abandoned children, and calls for the kiss of death. UNIX internals is not for the fainthearted!)

It should be noted that a parent process may choose to ignore the `SIGCHLD` signal. In this case, its children remain in the `SZOMB` state. While a zombie has freed most of its system resources, it does occupy a place in the process and thread tables. If the parent aborts without issuing the necessary wait, the children are frozen in the zombie state indefinitely—basically until the system is rebooted, as only the parent of record may issue a wait for a zombie child.

Another possibility is that the parent process may exit or abort before its children. In this case, HP-UX tries to solve the potential problem by reassigning the parent process ID of the orphaned child to the patron of all processes, `init`. The `init` process periodically issues a `waitid("P_ALL"...)` to clean up any adopted zombies. This approach has reduced the number of inadvertent zombies on a system to minimum. The limiting factor is that the "adoption" is only performed if a child calling exit passes the kernel an invalid parent ID number. If a parent process is blocking an attempted death-of-child signal and aborts without handling it, the child will remain a zombie. Responsible parents always clean up after their children—or at least pass the responsibility on to another!

Summary

We have now covered the creation, execution, and deconstruction of a process/thread's existence on an HP-UX system. Hopefully, you are developing an appreciation and a basic understanding of the behind-the-scenes work done by the HP-UX kernel on behalf of an application program. As there are many constituent parts to a program's run environment, it is the allocation, cleanup, and balancing of these resources that occupies much of the kernel's time.

Savvy programmers aware of these issues may make subtle changes in their approach to the use of system resources and improve performance. There are no magic bullets or secret formulas contained here, but knowledge can help you achieve effective utilization of your system's valuable resources.

Next, we examine the way HP-UX interacts with device files ("special files") in the I/O subsystem.

Chapter 10. I/O and Device Management

The I/O subsystem is one of the most complicated parts of the HP-UX operating system. As each new generation of hardware is developed, it brings along with it new types of busses, new device adapters, and new devices. Over time the kernel has had to change to accommodate these new advances while maintaining compatibility with old hardware. As a result, the I/O subsystem has grown into a large and complex framework that involves many smaller subsystems and a lot of "glue" to hold it all together.

PA-RISC I/O Architecture

The PA-RISC I/O architecture has specific requirements for all PA-RISC systems. These are designed to provide a standard set of rules for how the I/O hardware works and communicates with other components.

Every PA-RISC system must have at a minimum at central bus to which must be attached at least one processor module and at least one memory module. If the system has more than one processor or more than one memory module, they all reside on the central bus.

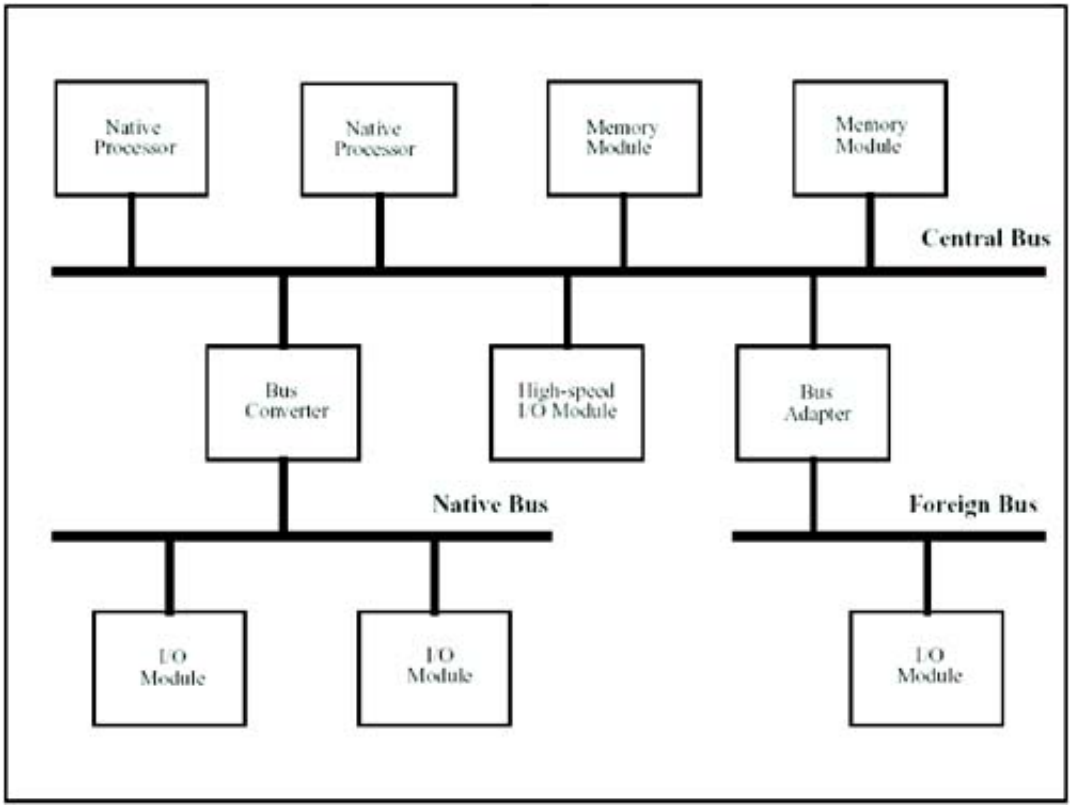
There are also a number of possible native busses. A native bus is one that adheres to the PA-RISC I/O architecture. As hardware has evolved since the introduction of PA-RISC, a number of new native busses have been introduced. One of the earliest of these was the HP Precision Bus (HP-PB), also called the native I/O (NIO) bus. Some others include the high-speed connect (HSC) bus, Gecko System Connect (GSC) bus, summit bus, and runway bus. Each provides different capacities and performance points, but all are native busses.

In addition to the native busses, PA-RISC systems must be able to connect to a variety of busses that were not designed with PA-RISC in mind. These are known as foreign busses. Some examples are Extended Industry Standard Architecture (EISA), Peripheral Component Interconnect (PCI), and Versa Module Eurocard (VME). These busses are designed by a variety of non-HP organizations and are used on many types of systems beyond PA-RISC. Because they don't conform to the PA-RISC architecture, we have to have some hardware that adapts their interface to that of the PA-RISC system. These adapters, which connect native busses to foreign busses, are called bus adapters.

It may also be desirable to connect between two different native busses, perhaps to expand capacity, for example. The PA-RISC I/O architecture specifies that a native bus can have no more than 64 modules on it. To get beyond this limitation, we can attach one native bus to another, much the way some people chain extension cords and power strips together to get more electrical outlets. The hardware that connects one native bus to another is a bus converter.

Figure 10-1 shows a block diagram of a typical PA-RISC system. In this example, we have two processors and two memory modules connected to the central bus. There is also an I/O module connected directly to the central bus, along with a bus converter that connects to a native bus and a bus adapter that connects to a foreign bus.

Figure 10-1. PA-RISC I/O Block Diagram



All of these components must be able to communicate with each other. This is part of what makes the I/O subsystem such a large and complex part of the kernel.

IODC and PDC

All native I/O modules provide an area of memory on the card known as I/O-dependent code (IODC). IODC normally resides in ROM on the interface card, but is frequently copied into main memory and accessed there. There are two components to IODC. The first is an area that contains information about the I/O module. By examining this information, the system can uniquely identify a particular type of module and determine how to interact with it. The second is a set of standard entry points that can be used to interact with the module. Not all modules provide all of these entry points. Some examples of these are `ENTRY_INIT`, which is used by the system at boot time to tell the card to initialize itself, and `ENTRY_IO`, which can be used at boot time to read or write through the module. This last is particularly important because this is how the kernel gets loaded into memory at boot time. In order for a device to be bootable, the I/O card it is connected to must have IODC that supports `ENTRY_IO`.

Processor-Dependent Code (PDC)

Similar in concept to IODC, each native processor also has an area of processor-dependent code (PDC). It contains a set of standard entry points to control and query the processor. Among these are entry points for resetting the processor (`PDC_RESET`), handling hardware errors or machine checks (`PDC_CHECK`), and handling transfer of control requests (`PDC_TOC`). PDC also provides entries for retrieving model information, controlling the system's chassis display, and managing the real-time clock.

I/O Address Space

PA-RISC uses memory-mapped I/O space. This means that there are no specific instructions for performing I/O in the system. Instead, all I/O modules take up a portion of the memory space, and the system uses `LOAD` and `STORE` operations to interact with the memory modules. The 32-bit and 64-bit address layouts are slightly different, but the concepts are the same. Figure 10-2 shows the 32-bit layout, and Figure 10-3 shows the 64-bit layout.

Figure 10-2. 32-Bit I/O Address Space

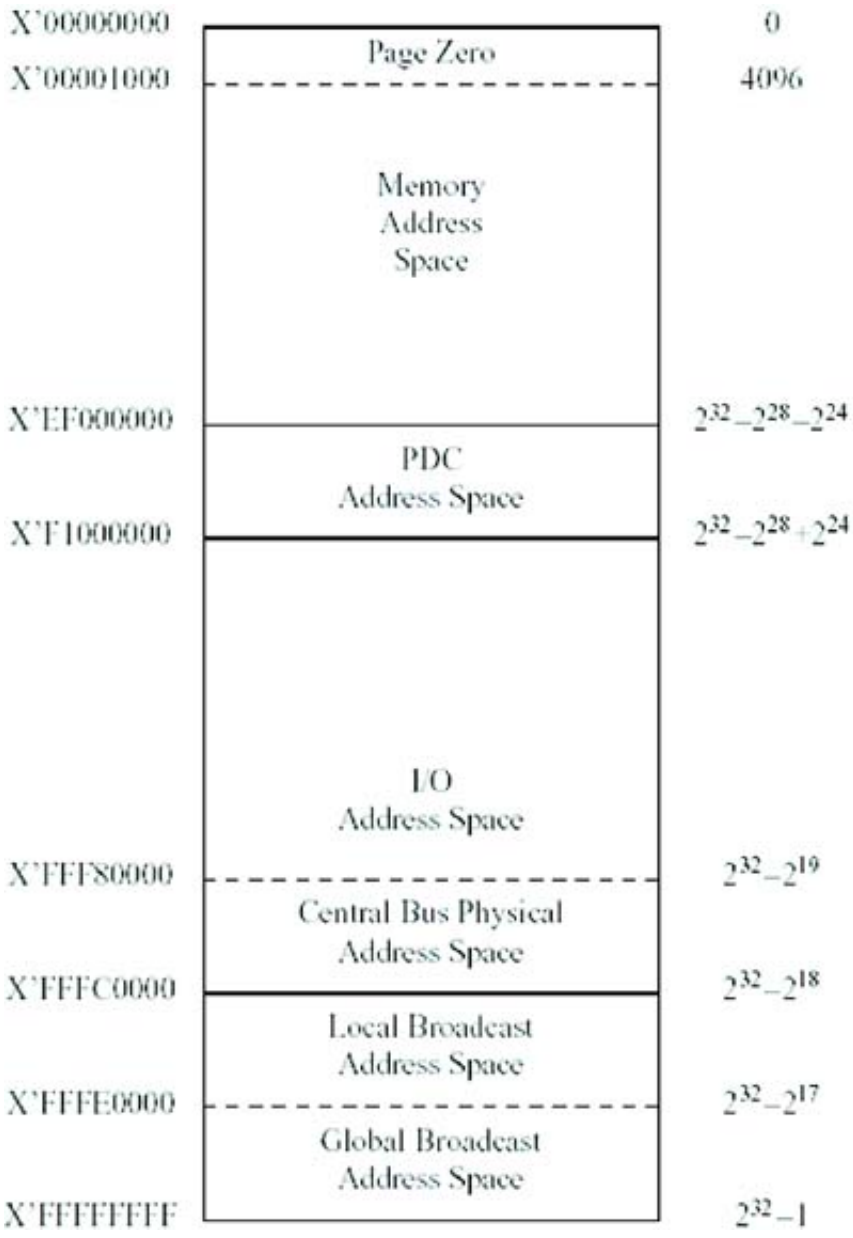
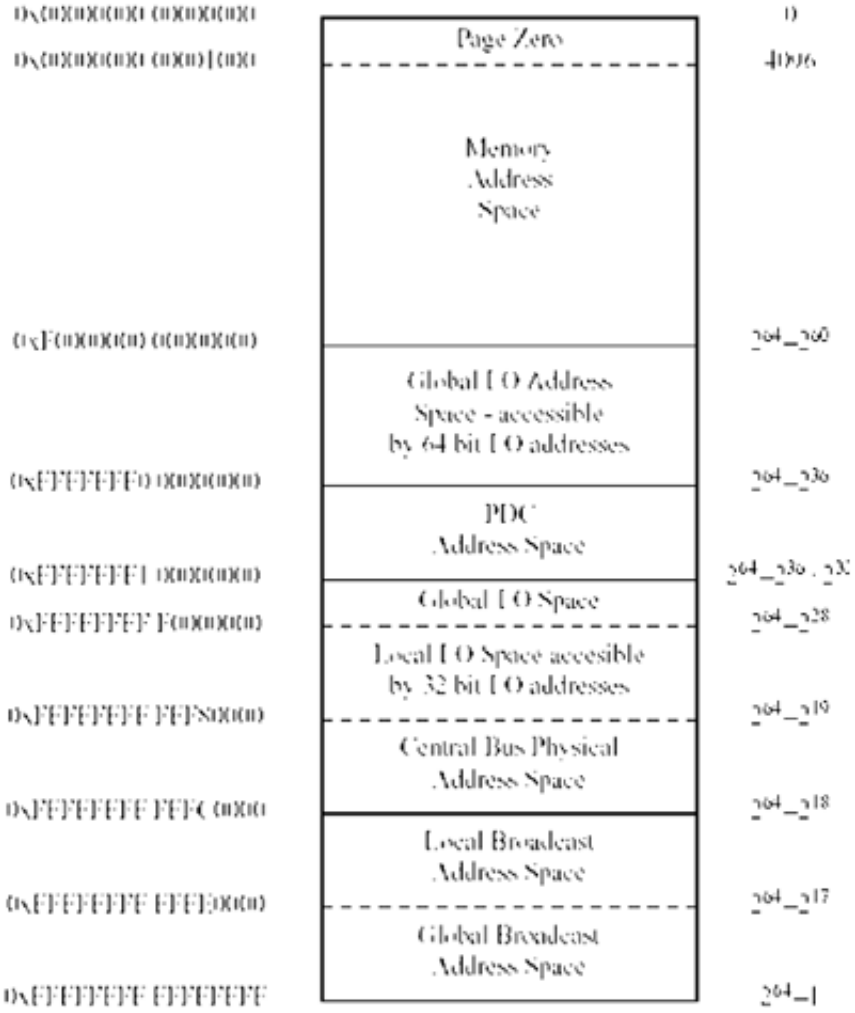


Figure 10-3. 64-Bit I/O Address Space



Both address-space models reserve a portion of memory for PDC address space. In the 32-bit model this is from 0xEF000000 to 0xF0FFFFFF. In the 64-bit model, it is in the range 0xFFFFFFF0 00000000 to 0xFFFFFFF0 FFFFFFFF. This area of address space is reserved for use by PDC. Memory in this range can be used both for PDC code itself and for data storage.

Both models also reserve a range of addresses for interfacing with I/O modules. In the 32-bit model this range is from 0xF1000000 to 0xFFFFFFFF. The 64-bit model has a range of addresses that correspond with the 32-bit ranges but with the upper bytes "F extended." This gives a range from 0xFFFFFFFF F1000000 to 0xFFFFFFFF FFFFFFFF. This range of 64-bit addresses is called the local I/O address space. In addition, the 64-bit model has a pair of ranges the make up the global I/O address space, which is accessible only by 64-bit processes. These ranges are from 0xF0000000 00000000 to 0xFFFFFFEF FFFFFFFF and from 0xFFFFFFF1 FFFFFFFF to 0xFFFFFFFF EFFFFFFF.

As you can see from the figures, the 32-bit I/O addresses and the corresponding local I/O addresses in the 64-bit model are further divided into four regions:

- A general I/O space for addressing modules on the bus
- An area reserved for the system's central bus
- A local broadcast area for communicating with all modules on the bus

- A global broadcast area for communicating with all modules in the system.

The general I/O space is further divided according to the topology of the system. Each bus in the system is given a 256-KB chunk of the address space. This 256-KB chunk is then divided into 64 chunks of 4 KB each, one for each possible module on the bus. The module's space is divided into 64 register sets, each with 16 registers. Of course, not every module will use all 1,024 registers, but that is the space that is available to the module. These addresses associated with a module are known as the module's hard physical address.

Now that we have all these addresses partitioned into busses and modules and register sets and registers, how do we know what to do to talk to an I/O module? That's the part that is unique to every type of module, and that's why every module type has to have its own driver. The system can query the IODC in the module, find out what kind of module it is, and assign it to a driver, and the driver then knows what registers are available and how to use them to communicate with the card.

Modes of Operation

There are two general modes of operation that can be used in communicating with an I/O module. The first is direct I/O. A direct I/O module cannot initiate a transfer of data on its own. In order for data to be transferred, the processor (or some other bus module) must explicitly do a `LOAD` or `STORE` operation. The only way a direct I/O module can initiate communication is by sending an interrupt to a processor. This puts some severe limitations on what can be done with a direct I/O card. The throughput on this type of card is low, and the demands on the processor are high because it has to be involved in every data transfer. However, there are some advantages too. Direct I/O is much easier to implement in hardware, particularly when interfacing to existing devices that are not designed with the PA-RISC architecture in mind. For this reason, they're also less expensive. For a low-speed operation to industry-standard devices, such as parallel and serial ports, direct I/O can be a good solution.

When the need for performance is higher, then we turn to the other type of I/O card: direct memory access (DMA). A DMA I/O card can initiate its own data transfers across the bus, normally to and from a memory module. The processor can give a DMA I/O card a command such as "read" or "write" and a range of memory, and the card will transfer all the data in the range. When the operation is complete, it signals the processor with an interrupt. During the time the I/O is taking place, the processor is freed to do other work—it doesn't need to get involved in the transfer until it is signaled that the transfer has completed. While this adds some complexity to the hardware on the module, it is an absolute necessity if we're to keep up with devices such as disks and tapes.

PA-RISC further defines two different types of DMA modules. Type A DMA modules support *data chaining*. This allows the system to supply a list of data ranges rather than a single range. When the module finishes transferring to one range, it begins transferring to the next. The system is signaled when the entire DMA chain is complete. Type B DMA modules take this one step further. They allow chaining of the commands as well as the data lists. Thus, the processor can pass the module a list of commands to be executed, a list of the memory ranges with which to transfer the data, and let the module do all the work. The module then returns a completion list to the system indicating the status of the various commands.

External I/O Interrupts

Now we have I/O modules connected to system busses. Each module has a range of addresses through which the system can communicate with the module. How does the module go about getting the attention of the processor when it needs to? This is done through the External Interrupt process.

Each processor has one control register called the external interrupt request register (EIRR) and another called the external interrupt enable mask (EIEM). These are 32-bit registers on a 32-bit machine and 64-bit registers on a 64-bit machine. Each bit in these registers corresponds to an external interrupt number. At boot time when the system is initializing the I/O modules, each module is instructed which interrupt number to use. All direct I/O cards use the same interrupt number: interrupt 3. They also send broadcast interrupts, meaning that when a direct I/O module sends an interrupt, all the processors get it. DMA cards, on the other hand, can each be given their own interrupt bit, and each can be assigned to a processor. When these cards send an interrupt, it's to a specific processor and to a specific bit on that processor. The programming of the modules and the interrupts themselves both take place in the same way: by writing to registers in the I/O space. Each module has a register called the `IO_EIM` register, which tells it where to send its interrupts. The processor writes a value to the `IO_EIM` register for each module as part of the initialization process. To generate an interrupt, the card writes to the `IO_EIR` register of the processor. Remember that a processor is just another module with an address on the bus just like any other. An I/O module writes to the processor's `IO_EIR` register to raise an interrupt.

After each instruction, the processor performs a logical AND between its EIEM and its EIRR registers. If a bit is turned on in the EIRR and that interrupt is enabled by the corresponding bit being on in the EIEM, then an interrupt is generated. (The I bit in the Processor Status Word, or PSW, must be on too, enabling the overall interrupt system.)

When the processor is interrupted, the first thing it does is turn off the I bit in the PSW, preventing any further interrupts until it is ready for them. It then branches to the first-level interrupt handler (FLIH) for the interrupt it has received. The address of this handler is found in an interrupt vector table (IVT), which is pointed to by the interrupt vector address (IVA) register (control register 14). Recall from Chapter 1, "PA-RISC 2.0 Architecture," that there are many types of traps and interruptions that the processor has to deal with. Each type is assigned a number; an external interrupt is a Trap Type 4. The address of the FLIH is computed as IVA + (32 * trap type). So for an external interrupt, the FLIH is found as the fourth entry in the table pointed to by the IVA. In HP-UX this is the routine `ihandler()`.

Each processor in the system has its own interrupt control stack (ICS). When an interrupt occurs, the system switches to using the ICS rather than the user process's stack or the kernel's stack. It then saves the current system state onto that stack. Next, it calls one of three interrupt handlers: `mp_ext_interrupt()`, `up_ext_interrupt()`, or `external_interrupt()`. Which it calls depends on whether the system is multiprocessor or uniprocessor and whether or not the kernel debugger is running. We perform this little bit of magic by actually patching the code at boot time. That is, part of the boot-up process is to determine if we're a uniprocessor or multiprocessor system and go poke the right branch address into `ihandler()`.

Regardless of which of the three error handlers is used, a certain set of steps is performed. The first is that we determine which external interrupt bit we're dealing with. The interrupt handler locates the highest priority unmasked interrupt to determine which to handle. It then sets that bit in the EIRR to off, indicating the interrupt has been received. Next, we mask off lower priority interrupts by modifying the EIEM. This holds off any lower priority interrupts until we're done with the current one. Now we can turn the I bit back on in the PSW, enabling interrupts overall. Finally, we're ready to call the correct interrupt handler for the external interrupt we received.

The information on what to mask and where to find the specific handler is kept in a table called the `eirr_switch`. It is an array of `struct eirrswitch`:

```
struct eirrswitch {
        void          (*int_action)();
        unsigned long   eiem;
        void           *int_arg;
        unsigned long   io_int;
};
```

Each entry in this array contains the pointer to the interrupt handler, the value that should be put into the EIEM, the argument to be passed to the handler, and a flag indicating whether or not the interrupt is an I/O interrupt. The flag is required because for an I/O interrupt, only the argument from `int_arg` is passed. For other interrupts, we also pass the bit number and a pointer to the save state.

The values that get loaded into the EIEM are sometimes called the system priority level (SPL). The values for SPL are defined in the file `/usr/include/machine/spl.h` and are summarized in Table 10-1. Higher SPL levels are more restrictive. That is, they have fewer possible interrupts enabled.

Table 10-1. SPL Levels

SPL Level	Alternate Name	Value
none	SPLPREEMPTOK	0xffffffff
SPL0	SPLNOPREEMPT	0xfffffff0

SPL1		0xffffffc0
SPL2	SPLSOFT	0xffffff00
SPL3		0xfffffe00
SPL4		0xfffffc00
SPL5	SPLIO	0xe1000800
SPL6	SPLSYS	0x40000000
SPL7	SPLLWI	0x00000007

We saw how hardware modules in the system communicate with each other. We looked at the address space of I/O modules, communication between modules using that address space, and the mechanics of how interrupts are generated and handled. In the next section, we take another step back and look at the framework that exists within the kernel for managing the various drivers.

I/O Framework

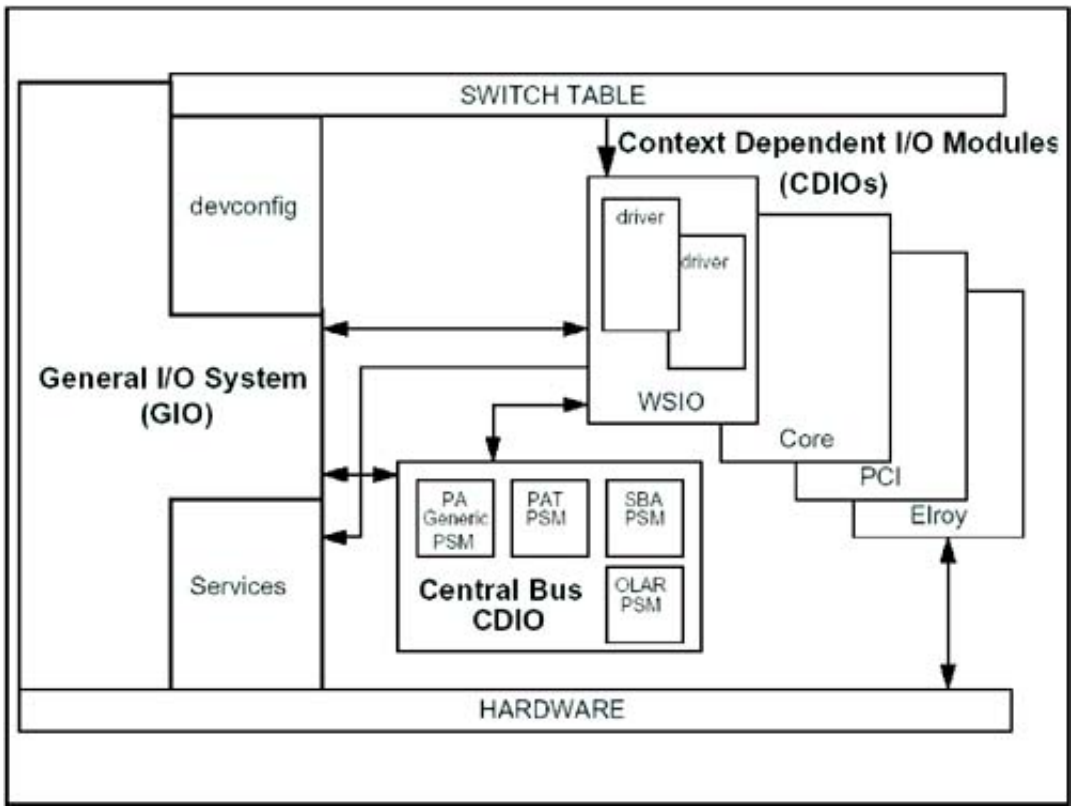
Prior to the 10.0 release of HP-UX, there were two distinct families of HP products that used the HP-UX operating system. One was the workstation family, or S700 systems. These systems evolved from the Motorola 68000-based single-user systems. The other family was the server family, or S800 systems. These were the original PA-RISC machines, using the hardware developed for both the HP-UX servers and the for systems running Hewlett-Packard's proprietary operating system MPE.

Over time, the S700 systems were moved to the PA-RISC platform. It became clear that a single operating system that ran on both workstations and servers made good sense. With release 7.0, HP-UX had merged most of the kernel functionality between the two platforms. Hewlett-Packard no longer had different releases for the two platforms. However, the I/O systems still remained incompatible—a workstation had to run the workstation version of 7.0, and a server ran the server version of 7.0.

Release 10.0 finally converged the two I/O systems. This coincided with the release of the K-class system, the first system to have I/O hardware from both the server and workstation platforms.

This convergence was made possible by creating a framework in which the various drivers could coexist despite having been developed in very different environments. Figure 10-4 shows the block diagram of this I/O framework.

Figure 10-4. Converged Workstation and Server I/O Systems



The key components of the framework are the general I/O (GIO) system and the context-dependent I/O

(CDIO) systems. GIO handles all the functions that are done on a systemwide level. The CDIOs provide environments for the drivers to work in. The SIO CDIO provides the environment for the server I/O subsystem, and the WSIO CDIO provides the environment for the workstation I/O subsystem. In addition, there are CDIOs for other driver environments and hardware bus types.

One special case of the CDIO is the central bus CDIO (CB-CDIO). This CDIO is responsible for system platform hardware. Within this CB-CDIO environment are a set of platform support modules (PSMs). As new functionality is added to the system—hardware for things like online addition and replacement (OLAR), for example—new PSMs are created to support that functionality. A system that supports OLAR has the OLAR PSM included in the kernel to provide the software interface.

General I/O

As mentioned, the GIO subsystem provides global functionality to the I/O system. Every system has GIO in the kernel. Some of the functions of the GIO include the following:

- **Management of I/O Configuration:** The overall configuration of the I/O system is a global concept and is managed in the GIO. The GIO maintains a structure called the `iotree`, which describes the relationship between the modules in the system.
- **Algorithms for Device Discovery:** At system boot time (and when requested by the `ioscan` command) the system must probe the I/O modules to determine what is present and bind the hardware to the correct drivers. The GIO oversees this process, although the individual CDIOs do the actual interrogation of the modules.
- **System Administration Interface:** Users and system administrators must see a consistent view of all I/O modules. The GIO provides this through a pseudo-driver called `devconfig`.
- **Services:** The GIO provides services that can be used by all of the CDIOs. These services include such things as inter-CDIO communication and dynamic module loading and unloading.

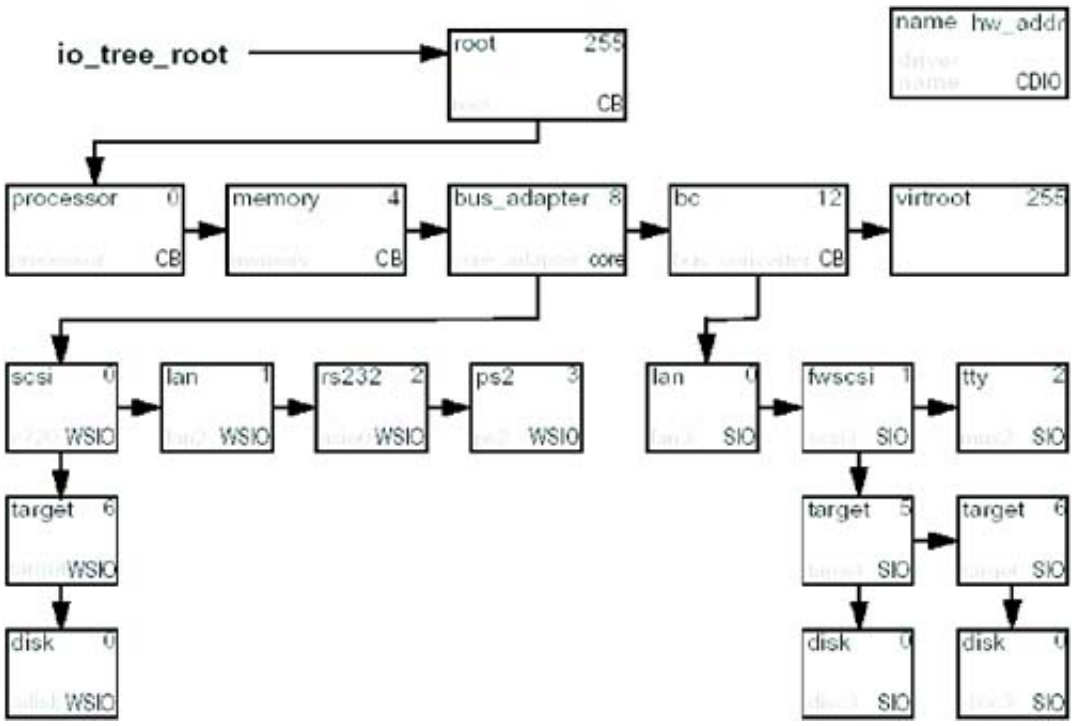
The GIO is not directly involved in individual I/O transactions or with individual devices. It provides a global environment in which the CDIOs operate and a uniform interface to the user.

I/O Global Data Structures

One of the most important data structures provided by the GIO is the `iotree`. The data structure of the `iotree` is an "object" in that the GIO provides procedures for manipulating the tree structure, and the structure may only be manipulated by those procedures.

The basic building block of the `iotree` is an *`iotree` node.* Each node in the tree has a pointer to its parent, a pointer to its first child (the one with the lowest address), and a pointer its first sibling. Each node also has information about the driver that manages the node. Figure 10-5 is a diagram of a small `iotree`, showing the pointer structure.

Figure 10-5. `Iotree` Example



The figure illustrates that each node in the tree represents a module and that for each of these modules we record the driver name, the class, the hardware address, and the CDIO that corresponds to that module.

The process of building the `iotree` starts with the routine `io_virt_mode_config()`, which is called by `init_main()` at boot time. This routine first initializes the GIO, then calls `io_scan()` to begin scanning the I/O busses. `io_scan()` puts all the nodes in the tree into the SCAN state, then calls `gio_scan_subtree()`, which starts configuring the central bus. Each CDIO in the system is given a chance to claim each device on the bus. Information about each device is passed to a `Scan` routine in the CDIO. If the CDIO recognizes the device, it claims it.

This scanning of the bus continues recursively. As each CDIO is told to scan a module and given a chance to claim it, it also calls the claim routines for the interface card drivers within that CDIO. Those drivers in turn get a chance to claim any interface cards they support on the bus.

As each module gets claimed by a driver, its `iotree` node gets put into the CLAIMED state. Once the entire scan is complete, any nodes that did not get claimed by a driver are still in the SCAN state. These are then moved to the UNCLAIMED state, indicating that there is no driver in the kernel that supports them.

Another important data structure maintained by the GIO is the device switch tables. This pair of tables is used to associate device files in the system with device drivers. Each device file (also known as a *special file*) has three characteristics: the type (block or character), the major number, and the minor number. When a special file is accessed, the system uses either the block device switch (`bdevsw`) or the character device switch (`cdevsw`) depending on the type of the file. It then uses the major number as an index into that table to get a structure containing the information about the driver. The structures for block and for character devices are slightly different, but they contain similar information. Here's the declaration for the character device switch:

```
struct cdevsw

{

        d_open_t        d_open;

        d_close_t       d_close;

        d_read_t        d_read;
```

```
        d_write_t       d_write;
        d_ioctl_t       d_ioctl;
        d_select_t      d_select;
        d_option1_t     d_option1;
        int             d_flags
        void            *d_drv_info;
        pfilter_t       *d_pfilter_s;
        aio_ops_t       *d_aio_ops;
};
```

The declaration for the `bdevsw` looks like this:

```
struct bdevsw
{
        d_open_t        d_open;
        d_close_t       d_close;
        d_strategy_t    d_strategy;
        d_dump_t        d_dump;
        d_psize_t       d_psize;
        int             d_flags;
        int             (*reserved) __(());
        void            *d_drv_info;
        pfilter_t       *d_pfilter_s;
        aio_ops_t       *d_aio_ops;
};
```

With the exception of `d_flags`, all of these fields are pointers either to the functions that handle certain operations or to information about the driver. As an example, the major number for the `stape` driver is 205. The 205th entry in the `cdevsw` table will have an entry that looks something like this:

```
{stape_open,
stape_close,
stape_read,
stape_write,
stape_ioctl,
NULL,
NULL,
```

```
stape_info,

NULL,

C_ALLCLOSES | C_MGR_IS_MP,

}
```

The first five of these entries are pointers to the routines that handle opens, closes, and so on for SCSI tapes. The last entry, the `d_flags` field, indicates that the system should call the `stape_close` routine for all closes and that the driver is aware of multi-processor systems and can be used in that environment. The `stape_info` entry is a pointer to the `drv_info_t` for this driver.

Exploring the system:

Load up all of the `cdevsw` entries:

```
q4> load struct cdevsw from &cdevsw max 256
```

Keep the entry for major number 205:

```
q4> keep indexof == 205
```

Print the `cdevsw` data:

```
q4> print -tx
```

Get the name of the open routine:

```
q4> ex d_open using a
```

The block and character switch tables are populated during the I/O discovery process described above. As each device is claimed by a driver, the GIO puts that driver's information into the appropriate device switch.

Context-Dependent I/O

CDIOs encapsulate functionality that is either specific to a particular type of bus or provides a driver environment. Not all CDIOs need be present in a kernel. The kernel contains only those CDIOs that are required by the particular hardware for which it was built. Thus, an S700 system would not have the SIO CDIO because that hardware doesn't exist in S700 systems, and an older S800 system such as the G-class wouldn't have the WSIO CDIO for the same reason.

Each CDIO must provide a number of interface functions that allow the GIO to communicate with it. These functions are called by the GIO to initialize the CDIO and also as part of the device discovery process. The following are the functions used for CDIO initialization:

- `install()`: The first call made to a CDIO. This function is called while the system is still in real mode, before virtual memory translations have been turned on. Generally it just registers the CDIO with the GIO.
- `module_init()`: This function too is called in real mode. This is a second place for CDIOs to perform tasks that must be done before virtual memory translations are started.
- `init_begin()`: This is the first function in the CDIO that is called in virtual mode. This allows the CDIO to perform any configuration that needs to be done with virtual translations on.
- `install_drv()`: This function is used only by the WSIO CDIO. It allows the CDIO to set up structures that are needed before drivers are installed.
- `init_middle()`: This function is called after all drivers have been installed. The CDIO can expect that no more driver installation will take place until after the system is up and running.
- `init_end()`: This is the final configuration call made to the CDIO. CDIOs can do any post-configuration cleanup in this routine.

Once the CDIO is initialized, there are three functions used as part of the device discovery process. These functions allow for the configuration of drivers within the CDIO:

- `scan()`: This function is called by the GIO to allow the CDIO to claim I/O modules. The GIO passes an `ionode`, which represents a particular hardware path, to the CDIO using this call. The CDIO probes the hardware modules connected to that hardware path, looking for any modules that it wishes to claim. If it finds hardware to claim, it creates a node in the `iotree` for the module.
- `config()`: This function is called by the GIO to tell the CDIO to configure the driver associated with a node.
- `unconfig()`: This function is called by the GIO to tell the CDIO to unconfigure a driver. This may be because a particular hardware module is no longer available, or it may be in preparation for unloading a dynamically loadable driver.

The names of these routines are typically preceded with the name of the CDIO, so the config routine for WSIO is `wsio_config()`, for SIO it's `sio_config()`, for CORE it's `core_config()`, and so on.

Of course, the remaining functionality in the CDIO is unique to each CDIO. For example, the SIO CDIO provides a routine called `io_send()`, which is used by all SIO drivers to send messages to other parts of the SIO subsystem. The WSIO CDIO provides DMA services that are used by all WSIO drivers to simplify doing DMA transfers. Each CDIO provides a set of functionality suited to the particular drivers that run under that environment.

Central Bus CDIO

The CB_CDIO is a special case of a CDIO. It is installed like any other CDIO and interfaces with GIO in the same way. But its job is to manage the central bus of the system rather than a particular I/O environment or set of drivers. Every system has the CB-CDIO compiled into it.

Up through release 11.0 of HP-UX, much of this functionality was provided by a CDIO called the Precision Architecture CDIO (PA-CDIO). This CDIO supported the PA-RISC specific portions of the kernel. When Hewlett-Packard began looking at porting the HP-UX system to other platforms, it became apparent that a more generic interface was needed to support the platform hardware. So, with HP-UX 11.0 release 99OP, Hewlett-Packard removed the PA-CDIO and replaced it with the CB-CDIO.

One of the key features of the CB-CDIO is the support for PSMs. A PSM is a piece of code that installs into the CB-CDIO just as a device driver installs into other CDIOs. But rather than supporting a particular type of

device, a PSM supports a particular platform's specific functionality. These can support a particular functionality, such as the `olar_psm`, which supports OLAR, or they can support particular system hardware such as the `sapic_psm`, which supports a particular programmable interrupt controller (PIC). Providing PSMs as modules that can be loaded as if they were device drivers allows the kernel to include only those PSMs needed on a particular platform. In fact, it is possible for PSMs to be dynamically loaded after boot time, just as a driver is, to enable new platform functionality.

The CB-CDIO provides a wide range of functionality to the kernel:

- **GIO Interface:** The CB-CDIO has to follow all the rules of any CDIO, so it must implement the standard interface to the GIO.
- **PIK Interface:** The platform-independent kernel (PIK) interface contains routines that provide services to kernel functions that are the same across all platforms.
- **PDK Interface:** The platform-dependent kernel (PDK) interface provides services to those parts of the kernel that are unique to a particular platform.
- **PSI Switch Table:** The platform support interface (PSI) switch table provides the access to the underlying PSM.

Another important feature provided by the CB-CDIO is I/O forwarding. I/O forwarding is the process of having I/O requests issued by the processor that will receive the interrupt when the I/O completes. Remember that each I/O module is told at boot time what interrupt it should generate and which processor it should interrupt. In this sense, each module is "owned" by a particular processor. Ideally, we want the I/O to complete on the same processor on which it started. This improves performance because data is more likely to be associated with the I/O in the cache. The CB-CDIO manages this by maintaining an I/O queue for each processor, called `mp_io_queue`. When a driver is ready to queue an I/O request to be started, it calls `ioforw_sched()` to have the I/O placed on the queue of the correct processor. CB-CDIO maintains a map, called `io_forw_map`, which maps devices to the processors that handle the interrupts for those devices. `ioforw_sched()` places the I/O onto the `mp_io_queue` for the correct processor, and that processor then starts the I/O. This way, when the interrupt comes back from the device at I/O completion, that processor is likely to have the relevant data in its cache.

I/O Odds and Ends

There are a few I/O topics that don't really fit into the categories we've talked about so far, but they are worth knowing about.

Device Files and the Device Switch Table

Device files in HP-UX are represented by two numbers: a major number and a minor number. Within the kernel, these two numbers are concatenated to form a single 32-bit value known as a `dev_t`. The first 8 bits of the `dev_t` are the major number, and the remaining 24 bits are the minor number. The major number is associated with a driver through the device switch tables, which we talked about earlier. The minor number is passed as an argument to that device driver's routines.

As an example, let's say we want to do an `open()` system call to a tape device called `/dev/rmt/c0t3d0BESTnb`. The major number for this device is 205 (decimal), and the minor number is `0x0030c0`. Within the kernel, this is a `dev_t` value of `0xcd0030c0`. The kernel handles this open request by looking in the device switch table and finding entry 205, then calling the routine pointed to by the `d_open` field in that entry: the function `stape_open()`. The first argument to `stape_open()` is the `dev_t` that we're opening. In this example the SCSI tape driver is part of the WSIO CDIO, and as such it gets access to common SCSI services. One of the first things that `stape_open()` does is call `scsi_lun_open()`, again passing the `dev_t`. `scsi_lun_open()` decodes the `dev_t` into card, target, and logical unit numbers (LUNs), and does what it has to do to open the correct target and LUN. `stape_open()` now interprets the remaining bits in the `dev_t` in its own way and determines that this is a no-rewind Berkeley tape device file.

The `ioconfig` File

The `ioconfig` file is a persistent version of the `iotree`. That is, it is a place on disk where I/O configuration can be stored so that it will exist from one boot of the machine to the next. There are actually two copies of the `ioconfig` file on a system—one in `/etc/ioconfig` and another in `/stand/ioconfig`. The copy in `/etc/ioconfig` is the authoritative copy, and the one in `/stand` is a backup in case the `/etc` directory is unavailable at boot time.

During system startup, the `ioinit` program is run from `/sbin/ioinitrc` (which in turn gets started by `init`). `ioinit` compares the two files to see if they match. If they don't, the one in `/etc` is written over the top of the one in `/stand`. Then `ioinit` reads the configuration information from the file and uses that to rebuild the `iotree` entries from the previous boot of the system. This ensures that instance numbers and major numbers won't change between one boot and the next.

Instance Numbers

So, what are these instance numbers that the `ioconfig` is keeping for us? An instance number is just a way of uniquely identifying a particular card. Card types are assigned a class by the driver that claims them. This class might be `tty` for a serial interface, `lan` for a LAN interface, or `ext_bus` for a device like a SCSI card that supports multiple remote devices. Within these categories, each card is given an instance number, so for example, we might have `ext_bus` numbers 0, 1, and 2 in a particular system. Each number corresponds to a particular card. The numbers are assigned by the GIO when they are bound to a driver, and they need not be sequential. They must be unique within a class but not across classes, so you can have only one `ext_bus0`, but you can also have a `lan0` and a `tty0`.

These numbers are generally represented as a part of the minor number of the device file. In our SCSI tape example the `ext_bus` instance was 0 (for SCSI card 0) and is represented by the `c0` part of the device filename and by the `0x00` in the upper byte of the minor number.

I/O Merging It is important that these instance numbers not change once they've been discovered, which is why we have the `ioconfig` file. If we reassigned the instance numbers at each boot, then removing a card or adding a new card could cause the instance numbers to change, meaning that the device files would now be pointing to different devices.

I/O merging is a technique to improve disk throughput by gathering together contiguous disk accesses. When a new disk I/O is started, rather than just tacking it on the end of the queue, the driver looks through the queue of I/Os that are waiting to be started to see if any of them are scheduled to access adjacent data on the disk. If so, the new I/O is combined with the old so that the system will do one larger I/O rather than two smaller ones. This takes advantage of the disk device's ability to read ahead and cache data, and also of the physical nature of disks. A disk access is much quicker if the heads don't have to be moved.

When the I/O is completed, there is a corresponding process that unmerges the I/O back to the original requests.

Summary

In this chapter, we looked at the I/O subsystem from a variety of levels. We covered some of the physical and architectural aspects of the I/O system, looking at how modules are accessed, addressed, and claimed by drivers.

We looked at how the I/O portion of the kernel is made up of a complex framework consisting of the GIO, which acts as interface and overseer; a variety of CDIOs, which encapsulate the drivers; and the CB CDIO, which manages the platform-specific parts of the I/O system.

Finally, we looked at how I/O requests work their way through the system from a user request to a device special file to the device switch tables and into the correct driver.

All this gives us a more detailed look at some of the I/O interfaces we touched on in earlier chapters. Chapter 15, "System Initialization," looks at how this fits in with the other things that happen at boot time.

Chapter 11. The Logical Volume Manager

When is a device not a device? When it is a pseudo-device!

In the logical volume manager (LVM) we see yet another type of abstraction layer. HP-UX builds file systems, allocates swap space, and allows raw data storage on disk volumes. The LVM subsystem allows the administrator to create logical volumes from the pooled space of physical volumes. Once a logical volume has been configured, it may be used in the same manner as a physical volume. Logical volume management gives the administrator a great deal of control over the size of individual file systems, swap, and raw space. In this chapter, we examine the underlying disk- and kernel-resident metadata structures used by LVM.

LVM Design Concept

Why have multiple file systems? Most UNIX file systems allow a file to grow within the boundary of the volume on which it was created. As such, a single runaway file may completely fill a file system. A file system that is 100 percent full presents challenges to the kernel when it needs to manage the space. If file systems holding operating system directories are filled, the kernel may have difficulties performing many of its tasks and in an extreme case even crash.

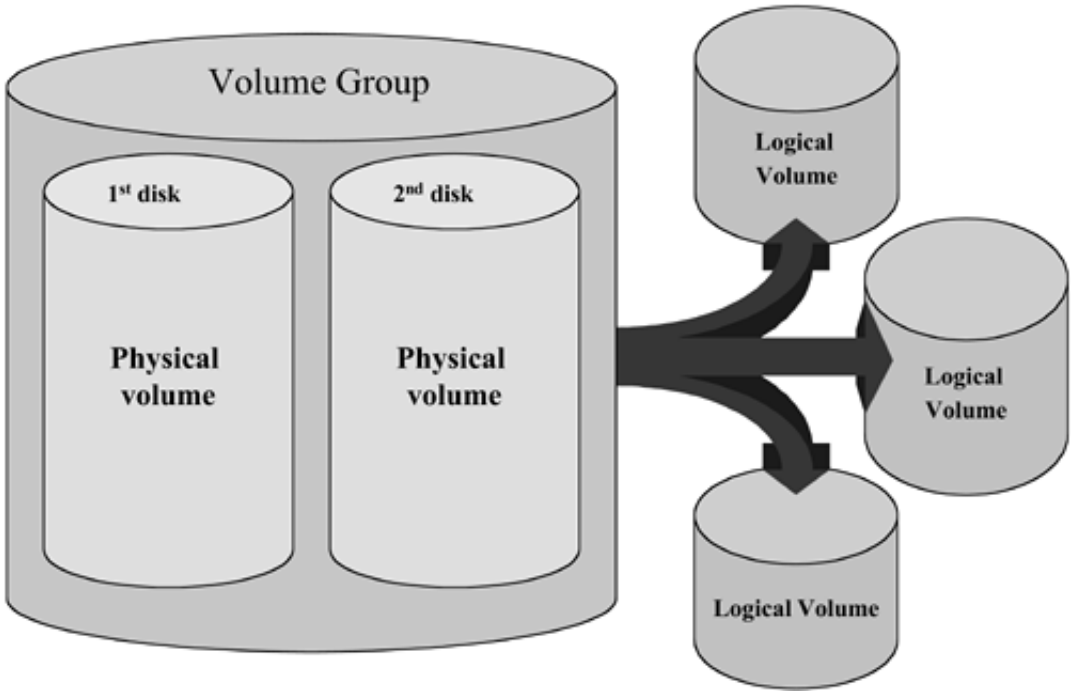
To avoid this scenario, most administrators prefer to create a number of individual file systems, sizing each to match its use. This allows the isolation of system directories (e.g., `/usr`, `/lib`) from user (e.g., `/home`) and application (e.g., `/opt`, `/var/opt`) directories. The kernel's virtual file system layer is then used to build what appears to applications and users as a single, seamless virtual file system. The kernel handles all transversal of mount points and directory structures during file open calls.

Traditionally, a physical volume could hold a single file system. This meant that in order to have multiple file systems, you needed multiple physical volumes. Another limit was that the size of a file system was dependent on the size of the underlying physical volume.

Early versions of HP-UX implemented a very limited form of hard disk sectioning that allowed the division of a physical disk in up to seven distinct *sections*. Each section could then be used to hold a file system, swap space, or raw storage. The problem with this approach was that it lacked flexibility. The sizing of the various disk sections was hardcoded in the driver and could not be adjusted by the administrator. As new disk models were created, they could not be used until the driver code was updated. Hewlett-Packard introduced LVM to address these issues (HP-UX 8.x).

The basic function of the LVM subsystem (see Figure 11-1) is to combine the space of one or more *physical volumes* into a *volume group*. Once the volume group has been created, its space may be assigned to one or more *logical volumes*.

Figure 11-1. The Logical Volume Manager



The administrator may size the individual logical volumes to match their needs. In addition, a logical volume may be extended or reduced following its initial creation. The volume group may also be extended by adding additional physical volumes or reduced by removing physical volumes. Keep in mind that logical volume management is an entirely separate issue from file system maintenance. The LVM abstraction isolates the mechanics of the volume manager from those of the various file systems employed on HP-UX. While logical volumes and volume groups may be extended and reduced, the effects of these actions on resident file systems must be handled by file system type-specific utilities. In some cases, this may require the archiving of the file system's data and its complete recreation in order to match it to a new volume size. Care and planning must be practiced before any reconfiguration or reallocation of the physical space within a volume group is attempted.

The LVM subsystem is a very robust design and greatly enhances the administrator's ability to manage the system's physical disk space. In addition to the basic sizing of logical volumes, LVM allows the optional creation of mirrored disk volumes (one-way mirroring or two-way mirroring) and volume data stripping. As you might guess, there are many issues and options the administrator needs to be aware of before attempting to manage disk space with LVM.

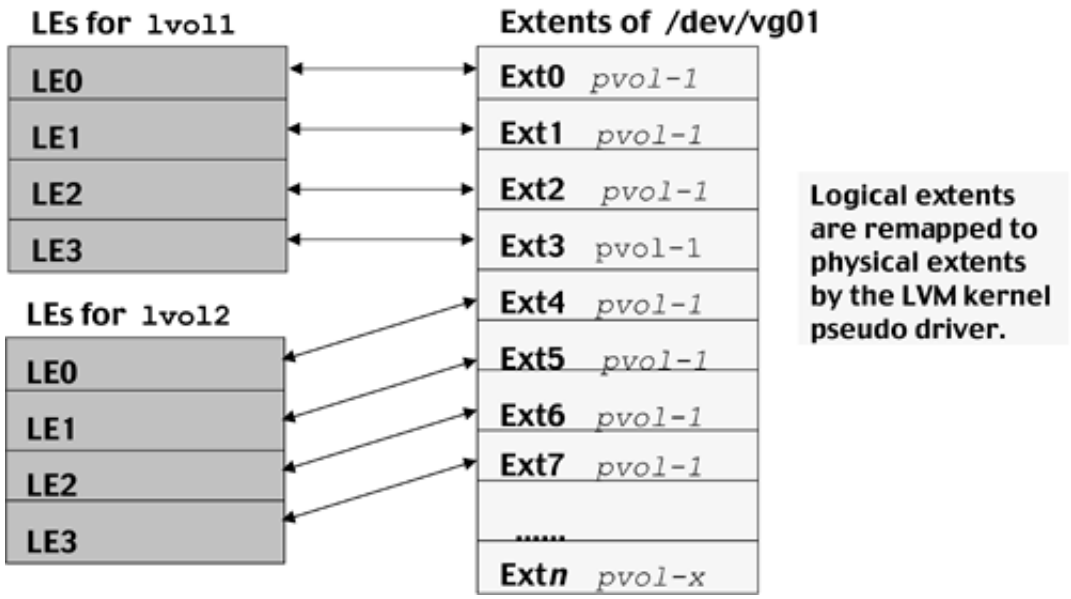
The LVM Extent

As with many kernel subsystems, LVM first divides its managed space into a fixed-size *extent*. While the extent size is tunable (1, 2, 4, 8, 16, 32, 64, 128, or 255 MB on a per-volume-group basis), the default 4 MB is frequently used. All of a volume group's physical volumes pool their *physical extents* in the group's *virtual extent* pool. Individual extents from this pool are mapped to a logical volume's *logical extent* map.

LVM's Smoke and Mirrors

Through the use of pseudo-device files, requests to logical volumes are directed to the LVM pseudo-driver (major number 64) in the kernel. A file system created on a logical volume is mapped to a collection of logical extents (see Figure 11-2), which in turn contain the blocks of the file system. The job of LVM is to translate a file system block number to a logical extent number and an offset within the extent. Next, it has to figure out which physical extent the logical extent maps to. Once these translations are made, the LVM pseudodriver passes the read/write request to the hardware driver responsible for the physical disk on which the mapped physical extent exists. In this manner, the offset within a logical extent is converted to an offset within a physical extent. As LVM performs these activities behind the smokescreen provided by the device switch tables, the requesting thread is blind to the redirection and translations that take place.

Figure 11-2. Mapping Extents



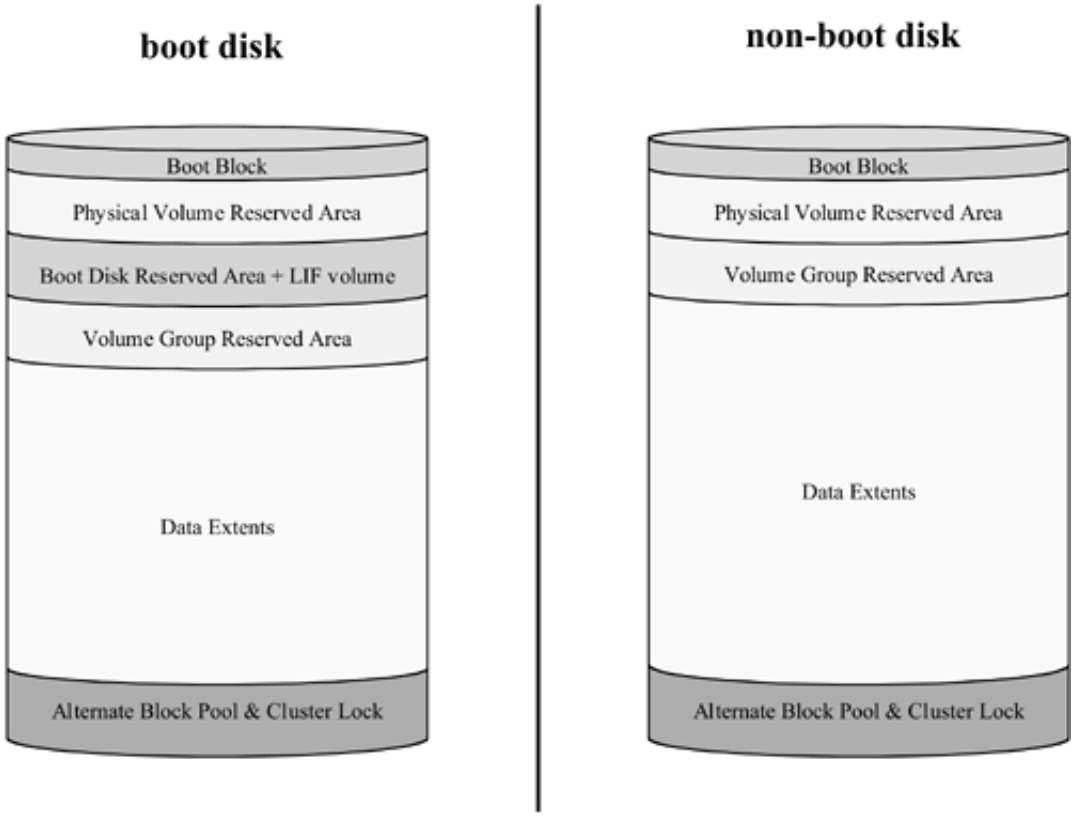
As a quick review, let's examine the commands required to initialize a logical volume group and create logical volumes.

Building a Volume Group

To create a new volume group on your HP-UX system, there are several basic steps to follow (see Figure 11-3).

- First, any physical volume intended for use with LVM must be physical volume–created (using the `pvcreate` command). This basic initialization consists of building template data structures at the beginning of the disk (see Figure 11-4). The templates used depends on whether the physical volume will be used as a bootable disk. Once a physical volume has been designated for use with LVM (pvcreated) the `mkfs` and `newfs` commands will refuse to work with it as a physical volume. Both commands have a –f option to allow a forceful override of this protection mechanism.

Figure 11-4. Metadata Disk Layout

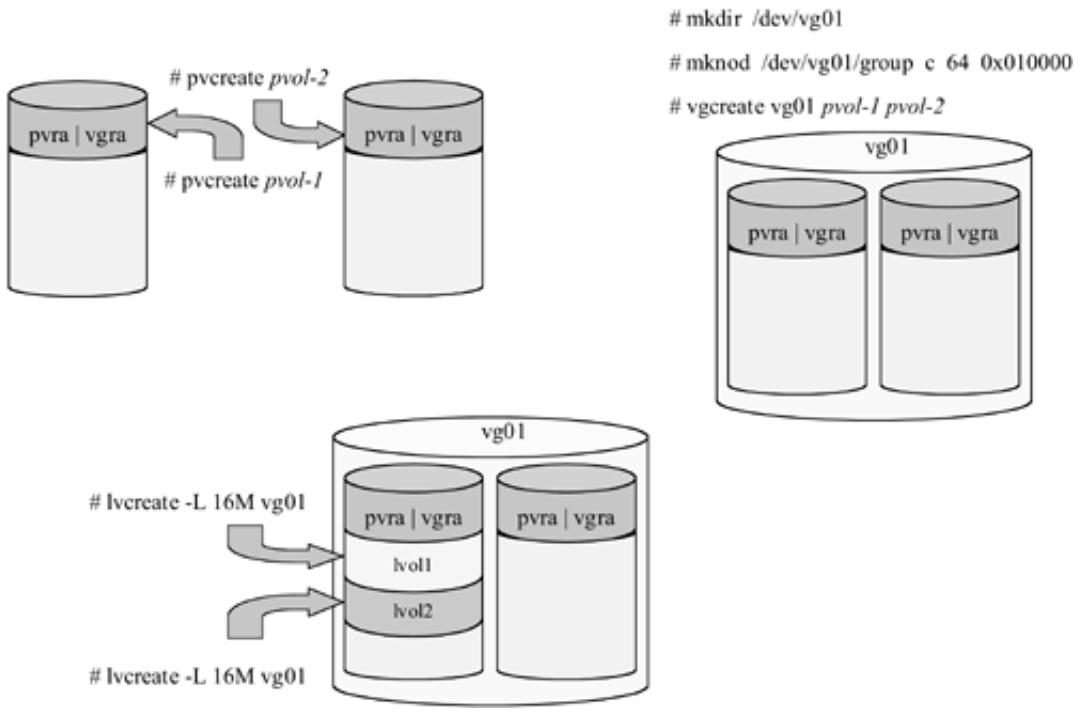


- Second, device files must be created to allow access to the volume group and its logical volumes. This is accomplished by creating a new directory under */dev* (i.e., */dev/vg01*). We must also create a pseudo-device file under our new directory */dev/vg01/group*. The name of our new directory is arbitrary (although the general wisdom is to start the name of the new directory with *vg* so that it may be easily spotted when listing the contents of */dev*) but the *group* filename is required.

- Third, we simply create the volume group with the `vgcreate` command. The physical volume names passed to this command fix the order of each physical volume within the volume group. Physical disk order greatly influences the allocation order of extent resources within a volume group. Physical volumes may also be a assigned to a labeled subgroup within the overall volume group. These "physical volume groups" (PVG) come into play during the creation of mirrored extents when the *strict* allocation policy option is enabled. This assures that multiple mirrors of the same extent are not located on the same physical disk.

- Finally, we are ready to `lvcreate` our logical volumes. A volume group is kind of like a checking account: you may write a check for any amount as long as you don't overdraw the account. In general, LVM simply allocates virtual extents from its free pool. They are taken from the first available physical extent, meaning that extents from the first physical volume are used before those of the second. It should be noted that command line options allow the administrator a great deal of control over the allocation process.

Figure 11-3. LVM Administration Review



You may notice that in Figure 11-3 the `mknod` command (used to create */dev/vg01/group*) has a specific major and minor number. For LVM, the major number (8 bits) is always 64. The minor number contains three 8-bit sections and is usually represented as a six-digit hexadecimal number. The first two digits (8 bits) represent the volume group number. The next two digits are always set to 00, and the last two digits reflect the logical volume number.

The volume group's `group` file is, in effect, logical volume 0 and is utilized by the LVM administrative commands to provide a path of access to the LVM metadata resident on the volume group's physical volumes.

Disk-Resident Data Structures

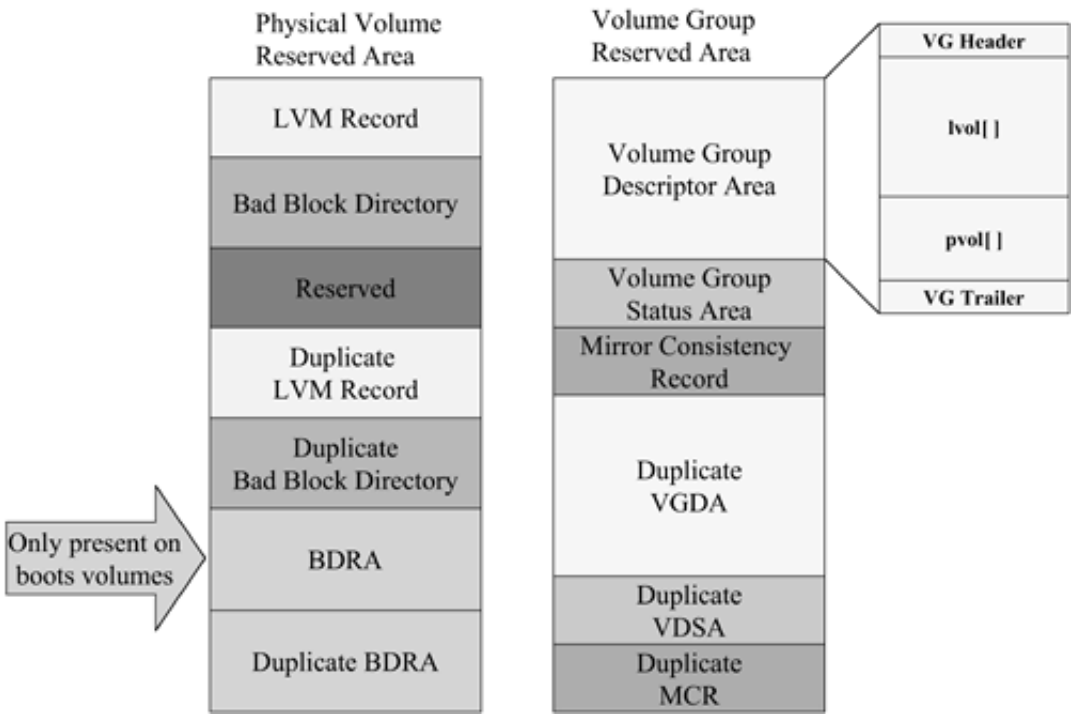
We start our examination of LVM data structures with the layout of the physical volumes. Figure 11-4 shows us an overview of the LVM metadata structures on a physical volume. Bootable LVM disks are created with the `pvcreate -B` option and have a logical interchange format (LIF) file system header located in the first 8 KB of the disk.

The LIF header is actually an ancient file system type used by the HP-UX boot loader. In the case of a LVM disk, it is a simple directory structure containing pointers to boot files stored in the boot disk reserved area (`BDRA`) on bootable disks.

Following the boot block, we see the physical disk reserved area (PDRA). This structure contains the LVM record, which stores information about this specific physical volume and offset pointers for each of the LVM structures on the disk. In effect, you may think of the LVM record as a type of LVM superblock for the disk. In addition, there is a bad block directory with relocation information for blocks identified by LVM as needing to be replaced. As we mentioned there is an area for the `BDRA` if this is a bootable disk. Also note the duplicates of each structure.

Next is the volume group reserved area (`VGRA`), which in turn contains the volume group descriptor area (`VGDA`), the volume group status area (`VGSA`), and the mirror consistency record (`MCR`). The `VGDA` structure is critical to LVM's ability to map logical to physical extents. The majority of the extent-mapping information is held in the `VGDA` along with information about the volume group the disk belongs to. The `VGSA` contains stale extent and missing physical volume information. The `MCR` provides space for the mirror write cache (`MWC`) consistency data stored on the drive. As with the `PVRA`, there are duplicates of each structure. Figure 11-5 illustrates the `PVRA` and `VGRA` in greater detail.

Figure 11-5. PVRA and VGRA Components



After the VGRA is the bulk of the disk's space, the area mapped as physical extents. Only whole extents may be allocated within LVM, so a partial extent will remain unused. The worst case for this would be a wasted space slightly smaller than the basic extent size. This is one consideration when deciding on the extent size: the larger the extent size, the larger the possible waste may be. Considering that the size of modern disk drives is in gigabytes, the potential to waste less than 4, or even 8 or 16 mega-bytes, doesn't seem to be much of a concern.

Bringing up the end is an area for bad block relocation, known as the bad block pool, and a small optional structure for cluster-locking information if the volume group is to be used in conjunction with an HP Service Guard cluster environment. Bad block relocation under LVM control may be disabled during configuration. Considering that most modern disk drive controllers handle the relocation of bad blocks, many administrators choose to disable bad block checking within LVM.

The PVRA and VGRA

The kernel defines the location of several key structures on an LVM physical volume (see Table 11-1). Disk drives are block-oriented devices. Access to their data must be performed a *sector* at a time. The basic size is either 1 KB per sector or 2 KB per sector on newer drives. The primary LVM record is located at sector 8. The primary bad block directory is at sector 9 for 1 KB/sector drives and at sector 10 for 2 KB/sector drives. The secondary LVM record starts at sector 72. The secondary bad block directory is at sector 73 for 1 KB/sector drives and at sector 74 for 2 KB/sector drives. Overall size of the PVRA is set to 128 sectors, and the bad block directory is set to 55 sectors.

Table 11-1. Kernel Parameters for LVM Disk-Based Structures

Kernel	# define	Sector #
Primary LVM record	PVRA_LVM_REC_SN1	8
Primary bad block directory	PVRA_BBDIR_SN1	9 (or 10)
Secondary LVM record	PVRA_LVM_REC_SN2	72
Secondary bad block directory	PVRA_BBDIR_SN2	73 (or 74)
Primary boot data record	BDRA_BDR_SN1	128
Secondary boot data record	BDRA_BDR_SN2	136
Overall size of the BDRA	BDRA_SIZE	16
Length of the boot disk record	BDRA_BDR_LENGTH	2
Length of the physical volume list	BDRA_PVL_LENGTH	6

Now let's examine the disk-resident structures in greater detail (Listings 11.1 and 11.2). We use our friend q4 to examine the fields of the various structures. These listings have been annotated, and in some cases redundant fields have been truncated. The PVRA begins with the lv_lvmrec structure.

Listing 11.1. q4> fields struct lv_lvmrec

```
The first element of this structure is the structures magic ID
and is set to LVMREC01. It is followed by the double-word
physical volume and volume group unique ID numbers
  0 0 8 0 char[8] lvm_id
  8 0 4 0 u_int   pv_id.id1
```

```
 12 0 4 0 u_int   pv_id.id2
 16 0 4 0 u_int   vg_id.id1
 20 0 4 0 u_int   vg_id.id2
Next are the pointers and lengths of the other structures on
this disk
 24 0 4 0 u_int   last_psn
 28 0 4 0 u_int   pv_num
 32 0 4 0 u_int   vgra_len
 36 0 4 0 u_int   vgra_psn
 40 0 4 0 u_int   vgda_len
 44 0 4 0 u_int   vgsa_len
 48 0 4 0 u_int   vgda_psn1
 52 0 4 0 u_int   vgda_psn2
 56 0 4 0 u_int   mcr_len
 60 0 4 0 u_int   mcr_psn1
 64 0 4 0 u_int   mcr_psn2
 68 0 4 0 u_int   data_len
 72 0 4 0 u_int   data_psn
We also see the physical extent size configured for this
volume group and additional structure pointers
 76 0 4 0 u_int   pxsize
 80 0 4 0 u_int   pxspace
 84 0 4 0 u_int   altpool_len
 88 0 4 0 u_int   altpool_psn
 92 0 4 0 u_int   maxdefects
 96 0 4 0 u_int   io_timeout
100 0 4 0 u_int   bdra_len
104 0 4 0 u_int   bdra_psn
108 0 4 0 u_int   bdr_len
112 0 4 0 u_int   bdr_psn1
116 0 4 0 u_int   bdr_psn2
120 0 4 0 u_int   pvl_len
124 0 4 0 u_int   pvl_psn1
128 0 4 0 u_int   pvl_psn2
132 0 4 0 u_int   cl_lock_flags
136 0 4 0 u_int   cl_lock_psn
140 0 4 0 u_int   cluster_id
```

```
144 0 4 0 int     conf_act_mode
148 0 4 0 u_int   orig_pv.pv_id.id1
152 0 4 0 u_int   orig_pv.pv_id.id2
156 0 2 0 u_short orig_pv.pv_pxcount
158 0 2 0 u_short orig_pv.pv_pxalloc
160 0 1 0 u_char  orig_pv.pv_num

Following the bad block directory information is the BDRA for
bootable disks.
```

Listing 11.2. `q4> fields struct lv_bootdata`

```
Again we begin with a magic ID, HPLVMBDR, a timestamp, and a
version number
   0 0 8 0 char[8] bd_magic
   8 0 4 0 u_int   bd_timestamp
  12 0 2 0 short   bd_version
Next the root volume group's boot, dump, and swap volumes are
identified (note that some of these fields are not currently
being used but are in place for future enhancements)
  14 0 2 0 short   bd_numrootPVs
  16 0 2 0 short   bd_numswapPVs
  18 0 2 0 short   bd_numdumpPVs
  20 0 4 0 u_int   bd_rootVGID.id1
  24 0 4 0 u_int   bd_rootVGID.id2
  28 0 4 0 u_int   bd_swapVGID.id1
  32 0 4 0 u_int   bd_swapVGID.id2
  36 0 4 0 u_int   bd_dumpVGID.id1
  40 0 4 0 u_int   bd_dumpVGID.id2
  44 0 4 0 int     bd_rootvg
  48 0 4 0 int     bd_swapvg
  52 0 4 0 int     bd_dumpvg
  56 0 4 0 int     bd_rootlv[0]
--------------------------------
 180 0 4 0 int     bd_rootlv[31]
 184 0 4 0 int     bd_swaplv[0]
```

```
--------------------------------
 308 0 4 0 int     bd_swaplv[31]
 312 0 4 0 int     bd_dumplv[0]
--------------------------------]
 436 0 4 0 int     bd_dumplv[31]
 440 0 4 0 int     bd_rootPVs
 444 0 4 0 int     bd_swapPVs
 448 0 4 0 int     bd_dumpPVs
 452 0 4 0 u_int   bd_rootPVsize
 456 0 4 0 u_int   bd_swapPVsize
 460 0 4 0 u_int   bd_dumpPVsize
 464 0 4 0 int     bd_rootPVcksum
 468 0 4 0 int     bd_swapPVcksum
 472 0 4 0 int     bd_dumpPVcksum
 476 0 2 0 short   bd_boot[0]
-----------------------------------
 482 0 2 0 short   bd_boot[3]
 484 0 2 0 short   bd_rootdisks[0]
-----------------------------------
 546 0 2 0 short   bd_rootdisks[31]
 548 0 2 0 short   bd_swapdisks[0]
-----------------------------------
 610 0 2 0 short   bd_swapdisks[31]
 612 0 2 0 short   bd_dumpdisks[0]
-----------------------------------
 674 0 2 0 short   bd_dumpdisks[31]
 676 0 2 0 short   bd_rootmaint[0]
-----------------------------------
 738 0 2 0 short   bd_rootmaint[31]
 740 0 2 0 short   bd_swapmaint[0]
-----------------------------------
 802 0 2 0 short   bd_swapmaint[31]
 804 0 2 0 short   bd_dumpmaint[0]
-----------------------------------
 866 0 2 0 short   bd_dumpmaint[31]
 868 0 4 0 int     bd_flags
 872 0 4 0 int     bd_reserved[0]
-----------------------------------
```

```
2040 0 4 0 int     bd_reserved[292]

2044 0 4 0 int     bd_checksum
```

Let's switch our focus to the `VGRA` and its components. The first part of the `VGRA` is the `VGDA`, which includes four main structures: `VG_header`, `lvol[]`, `pvol[]`, and `VG_trailer`. The configurable maximum number of logical volumes and physical volumes per volume group are used to size the `lvol[]` and `pvol[]` arrays respectively. The volume group tunables `max_lv` and `max_pv` may be set during the `vgcreate` command. Let's take a look at Listings 11.3 and 11.4:

Listing 11.3. `q4> fields struct VG_header`

```
First are the timestamps and identifier

     0 0 4 0 int     vg_timestamp.tv_sec

     4 0 4 0 int     vg_timestamp.tv_usec

     8 0 4 0 u_int   vg_id.id1

    12 0 4 0 u_int   vg_id.id2

Next is the maximum number of logical volumes and physical

volumes for the volume group

    16 0 2 0 u_short maxlvs

    18 0 2 0 u_short numpvs

The maximum number of physical extents for the volume group

and its status flag

    20 0 2 0 u_short maxpxs

    22 0 2 0 u_short flags

    24 0 4 0 u_int   reserved2

    28 0 4 0 u_int   reserved3
```

Listing 11.4. `q4> fields struct LV_entry`

```
We start with the maximum size for the logical volume and its

state flags

   0 0 2 0 u_short maxlxs

   2 0 2 0 u_short lv_flags
```

LVM_LVDEFINED	Logical volume entry defined
LVM_DISABLED	lvol unavailable

LVM_RDONLY	lvol read only
LVM_NORELOC	bad blocks not relocated
LVM_VERIFY	all writes to be verified
LVM_STRICT	allocate mirror on distinct pvols
LVM_NOMWC	no mirror consistency checks for this lvol
LVM_PVG_STRICT	allocate mirrors from distinct PVG's
LVM_CONSISTENCY	mirror consistency recovery required
LVM_CLEAN	lvol has no pending writes
LVM_CONTIGUOUS	allocate contiguous physical extents for this lvol

```
Next is the configured scheduling strategy (sequential or
parallel)
   4 0 1 0 u_char  sched_strat
The number of mirrors, number of stripes and the stripe size
are recorded in the next three parameters
   5 0 1 0 u_char  maxmirrors
   6 0 2 0 u_short stripes
   8 0 2 0 u_short stripe_size
  10 0 2 0 u_short reserved2
The timeout is the number of seconds allowed before a
scheduled LV I/O fails
  12 0 4 0 u_int   lv_io_timeout
```

Each `pvol[]` entry consists of a `PV_header` and a `PX_entry[]`. The `PX_entry[]` array is sized in accordance with the maximum number of extents allowed per physical volume (`max_pe` is settable during the `vgcreate` command). This array contains the final word when it comes to which logical extent is mapped to which physical extent. The index into the `PX_entry[]` array represents the physical extent number; the array data contains the logical volume and logical extent IDs to which it is mapped. See Listings 11.5, 11.6, and 11.7.

Listing 11.5. `q4> fields struct PV_header`

```
The physical volume identifier, an extent count, and the pvol
flags
   0 0 4 0 u_int   pv_id.id1
   4 0 4 0 u_int   pv_id.id2
   8 0 2 0 u_short px_count
  10 0 2 0 u_short pv_flags
```

LVM_PVDEFINED	this entry is used
LVM_PVNOALLOC	no extent allocation is allowed
LVM_NOVGDA	pvol contains a VGDA
LVM_PVRORELOC	no new defects relocated
LVM_PVMISSING	pvol is missing
LVM_NOATTACHED	pvol not attached
LVM_PVPOWERFAIL	pvol is power-failing
LVM_PVNEEDSYNC	pvol needs re-sync
LVM_PVALTLINK	pvol not the primary link
LVM_PVINUSE	pvol is being configured
LVM_PVCFGRSTORD	pvol had config data restored
LVM_PVSWITCHLINK	pvol path requires switch
LVM_PVMOSWBACK	don't switch links back
LVM_PVSPARD	pvol is a spare
LVM_PVDATA_SPARED	pvol failed, data has been spared

```
The number of entries in the pvol extent map
  12 0 2 0 u_short pv_msize
The maximum number of defects that may be relocated
  14 0 2 0 u_short pv_defectlim
```

Listing 11.6. `q4> fields struct PX_entry`

```
The physical extent table entries map to a logical volume and
a logical extent number
   0 0 2 0 u_short lv_index
   2 0 2 0 u_short lx_num
```

Listing 11.7. `q4> fields struct VG_trailer`

```
The trailer structure is a finishing thought to the VGDA
```

```
 0 0 4 0 int     vg_timestamp.tv_sec

 4 0 4 0 int     vg_timestamp.tv_usec

 8 0 4 0 u_int   reserved1

12 0 4 0 u_int   reserved2

16 0 4 0 u_int   reserved3

20 0 4 0 u_int   vgda_cksum

24 0 8 0 char[8] vgda_magic
```

You may be wondering why there are duplicate copies of so many of the disk-resident data structures. When a disk-based structure is updated by the LVM pseudo-driver, only one of the copies is written on each physical volume (except in the case of a volume group with a single physical volume, where both copies are updated). When the data needs to be read by the LVM driver, it chooses the one with the newest copy. As an additional sanity check, the timestamps in the structure header and trailer are compared. If they match, we can assume that the last write to the disk was successful; if they don't match, we try another copy. Remember that all the disks in the volume group contain the same extent-mapping information for redundancy.

Following the `VGDA` is the `VGSA` consisting of the `SA_header` and a trailer (Listing 11.8).

Listing 11.8. `q4> fields struct SA_header`

```
The first two structures hold timestamps

   0 0 4 0 int     sa_h_timestamp.tv_sec

   4 0 4 0 int     sa_h_timestamp.tv_usec

Next is the maximum number of physical extents per physical

volume and the maximum number of physical volumes for the

volume group

   8 0 2 0 u_short sa_maxpxs

  10 0 2 0 u_short sa_maxpvs

  12 0 4 0 u_int   reserved1

The final component in the structure is a checksum

  16 0 4 0 u_int   sa_checksum


The corresponding trailer is 16 bytes in length and consists

of the VGSA magic number ("VGSA0001"} and a timestamp
```

The MCR completes the primary structures in the VGRA and contains the disk copies of data from the MWC. The `mwc_entry` structure (Listing 11.9) contains the disk copies of mirror consistency cache information. We discuss the way these records are used later in this chapter.

Listing 11.9. `q4> fields struct mwc_entry`

```
Timestamps surround 126 sets of logical volume number, the
track group shift, and the logical track group number
   0 0    4 0 int        b_tmstamp.tv_sec
   4 0    4 0 int        b_tmstamp.tv_usec
   8 0    2 0 u_short    ca_p1[0].lv_number
  10 0    2 0 u_short    ca_p1[0].ltgshift
  12 0    4 0 u_int      ca_p1[0].lv_ltg
-------------------------------------------
1008 0    2 0 u_short    ca_p1[125].lv_number
1010 0    2 0 u_short    ca_p1[125].ltgshift
1012 0    4 0 u_int      ca_p1[125].lv_ltg
1016 0    4 0 int        e_tmstamp.tv_sec
1020 0    4 0 int        e_tmstamp.tv_usec
Finally we have a 1024-character pad
1024 0 1024 0 char[1024] pad
```

LVM: The Kernel View

In Figure 11-6 we see an overview of the LVM pseudodriver organization within the kernel. As mentioned earlier, an application thread requests a file open by referencing a file system pathname. The virtual file system resolves this to a specific `vnode` within the kernel file system table. The `vnode` contains the device number of the file system on which the file is located. The device number for LVM-based file systems directs all I/O requests to the LVM pseudodriver via the kernel's device switch table. Once an I/O request is received by the driver, the driver must pass it through several layers.

Strategy Layer: This layer receives the initial request for a block I/O transaction to a specific file system block. The kernel facilitates this request by passing a `buf` structure containing the logical volume number (`b_dev`), request flags defining the transaction (`v_flags`), the block number within the logical volume (`b_blkno`), the byte count of the request (`b_bcount`), and various options (`b_options`). The strategy layer must validate the request, checking the availability of the requested volume and its size against the block size of the volume.

Mirror Consistency Layer: If a logical volume has mirroring configured, then this layer must coordinate mirror writes. A volume may be configured to cache mirrored write request in an MWC. A volume group is divided into logical track groups (LTG) and cached write requests are first registered in one of the MWC's cache records (one per LTG) in kernel memory and also written to an `mwc_entry` on one of the physical volumes of the volume group. Originally, each volume group had 32 LTGs, but with the 11.i release this was increased to 126.

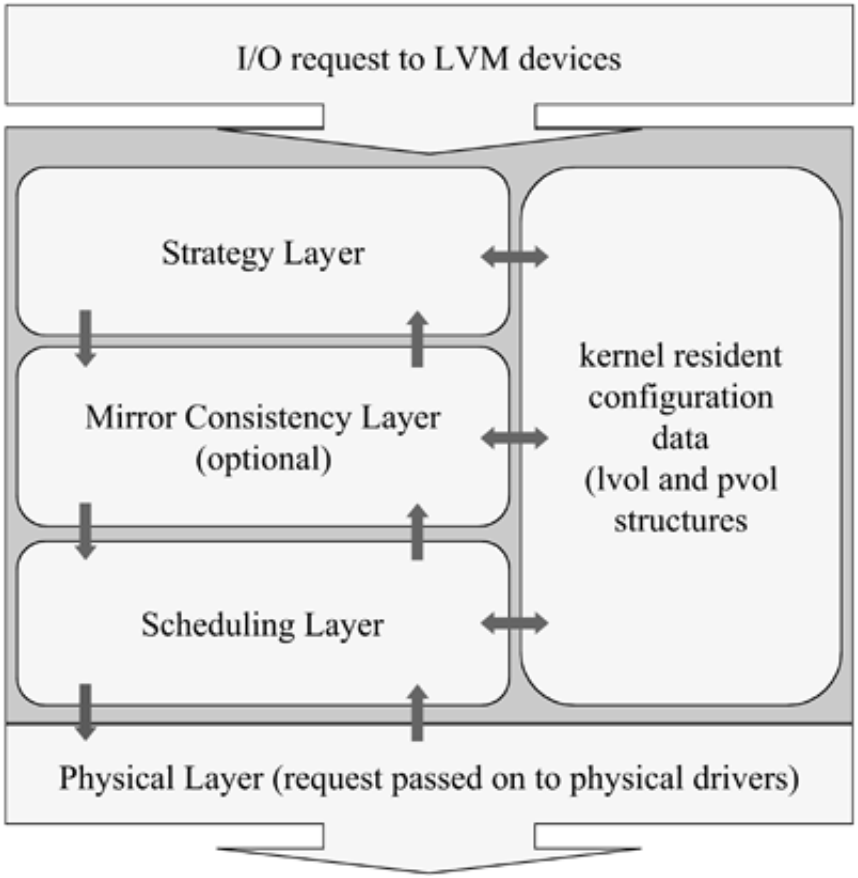
The cached data does not contain the user data, but it does register the intent to update data on the disk. When the write has been completed to all mirror copies, the cache record is cleared. In the case of a system crash and reboot, the cache records on the physical volumes may be used to identify which LTG's may be out of sync. If the MWC is not used, then all logical extents on the mirrors would have to be resynchronized.

Scheduling Layer: This layer makes full use of the kernel-based copies of the volume group's kernel-based configuration information. The actual location and number of mirrors are converted into one or more physical requests. The strategy layer accepts the `buf` pointer from the previous layers and directs it on its way through LVM.

A logical volume may be configured to follow one of several scheduling strategies. It may be `LVM_RESERVED` if the request is to the reserved area on the disk (the actual metadata structures used to configure and manage the volume). Normal read and write requests may be either `LVM_SEQUENTIAL` or `LVM_PARALLEL`. This effects the methodology used for mirrored read and write requests. Finally, the request could be flagged as `LVM_STRIPE` for parallel striped operations.

Physical Layer: This last layer is where the rubber meets the road. The LVM driver passes requests and their associated `buf` structures to the actual physical device drivers responsible for the mapped physical volumes.

Figure 11-6. The LVM Pseudodriver Architecture

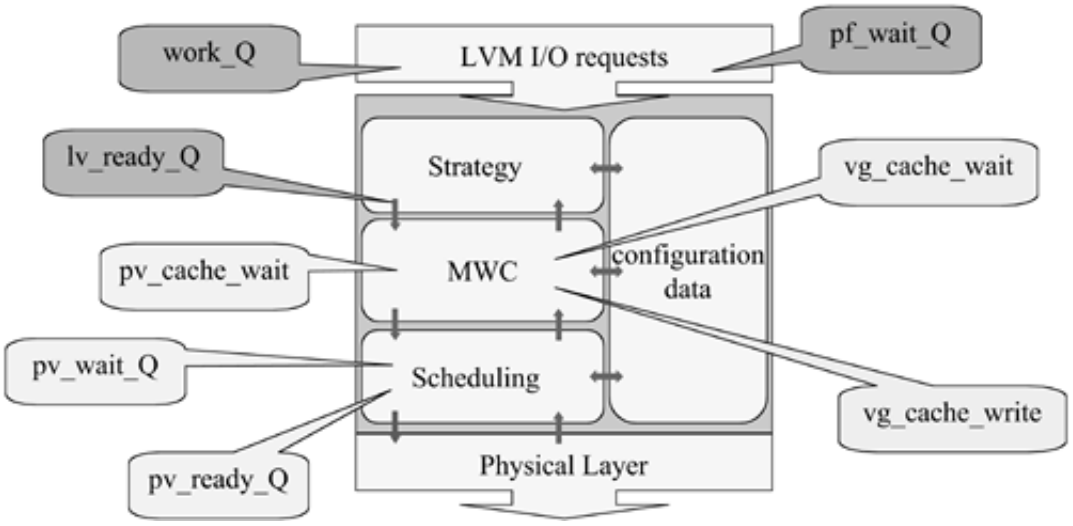


Work Queues: Keeping a Request on Track

Because the LVM pseudodriver is a kernel resource, it is common for it to be processing multiple requests in the various layers at any one point in time. In addition to servicing multiple threads, the buffers may be queued while awaiting access to a physical device or to be transferred to the next layer within the LVM driver itself.

In Figure 11-7, we see that there are a number of queues that a request may find itself on.

Figure 11-7. Work Queues



They may be divided into four different categories.

> **Global queues:** The `pf_wait_Q` contains all requests that may not be completed due to a power failure. A request on this queue is waiting for termination and cleanup.
>
> **Per-volume-group queues:** The `vg_cache_wait` queue holds requests waiting on a free entry in their volume group's MWC. The `vg_cache_write` queue provides a linked list of physical volumes available for an MWC update to disk. When the LVM driver needs to store MWC data to a physical disk in its volume group, it selects the one at the head of this queue.
>
> **Per-physical-volume queues:** All requests scheduled for a specific physical volume are linked to the `pv_read_Q`. The `pv_cache_wait` holds requests waiting for their MWC data to be written to a physical disk.
>
> The LVM system supports a feature known as physical volume links (`pv links`). This feature allows for the automatic switchover from one bus to an alternate bus; that is, when a failure occurs on a bus controller, if the system has an alternate path available, I/O is switched to it. The `pv_wait_Q` holds requests waiting for the `pv links` switch to take effect.
>
> Currently, `pv links` support an active/passive mode of operation. This references the fact that only one interface may be active at a time. If it fails they standby passive interface is activated. In the future, this may be enhanced to allow active/active configurations where both interfaces may be used to share the load and increase overall throughput while providing true hot standby functionality
>
> **Per-logical-volume queues:** The `work_Q` is actually a per-logical-volume array of all outstanding requests for the volume. The array entries are the other queues on which the individual requests are currently linked. Since this master queue has knowledge of all current outstanding requests for a volume, the kernel strategy layer makes use of this information to serialize I/O request whenever possible.
>
> The `lv_read_Q` is a holding place for requests waiting to be passed to the MWC layer in the pseudo-driver.

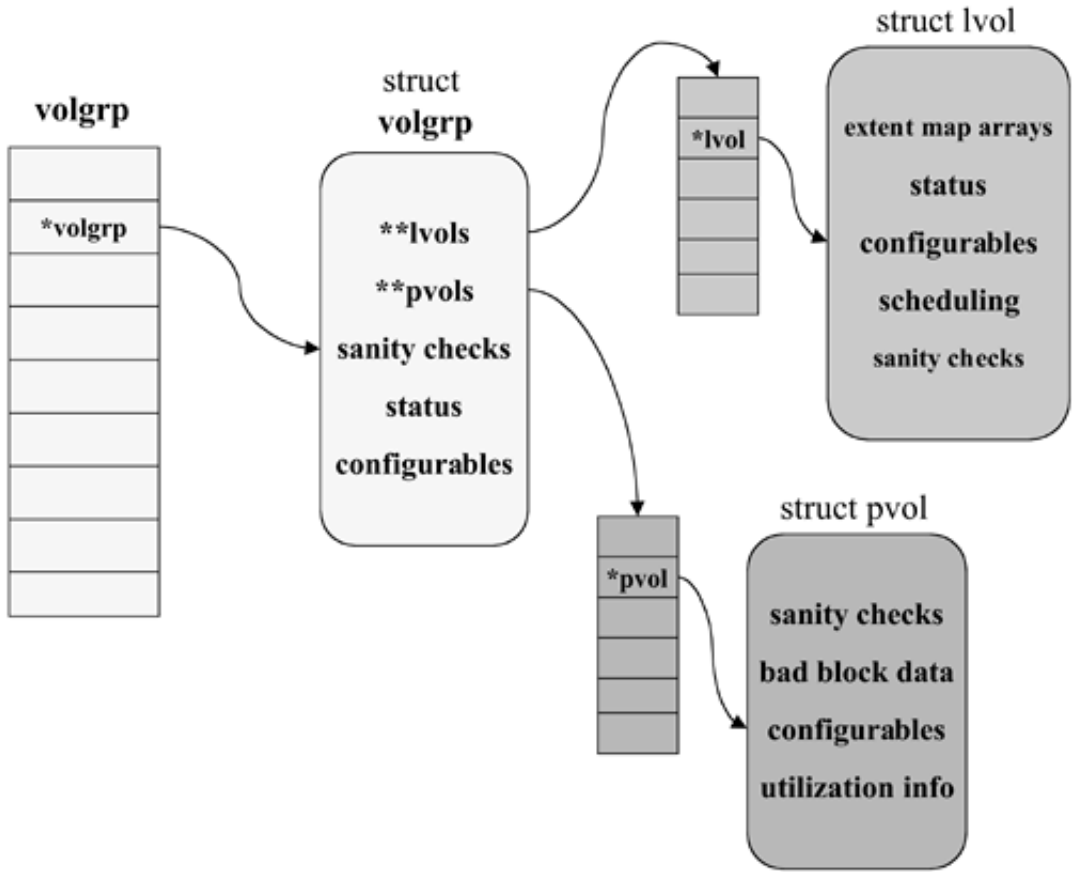
Next, let's consider the data structures stored in the kernel to support the operations of the LVM subsystem.

Kernel Resident Data Structures

When a volume group is activated (at boot or via the **`vgchange -a y`** command), its metadata is copied

to kernel-resident structures. Figure 11-8 presents an overview of these structures. The starting point for these structures is the kernel `volgrp[]` array. As individual `volgrp` structures are among the largest in the kernel, their number is limited by the kernel-tunable `maxvgs` and defaults to 10. Note: if you are creating a new volume, group directory and *group* file the volume group number passed to the `mknod` command (the first two digits of the minor number argument) should not exceed this tunable value. The volume group's number is used as the index into the kernel resident `volgrp[]` array.

Figure 11-8. Kernel-Resident Configuration Structures



Let's begin by examining the `volgrp` structure (Listing 11.10).

Listing 11.10. `q4> fields struct volgrp`

```
We start with lock pointers and counters
   0 0 4 0 *                  vg_lock.interlock
   4 0 4 0 u_int              vg_lock.delay
   8 0 4 0 int                vg_lock.read_count
  12 0 1 0 char               vg_lock.want_write
  13 0 1 0 char               vg_lock.want_upgrade
```

```
  14 0 1 0 char               vg_lock.waiting
  15 0 1 0 char               vg_lock.no_swap
Next a pointer to the lvol array (sized to 256), the number of
logical volumes, a lock, a pointer to the pvol array and its
size, the major number for the volume group pseudo-device file
(0x40, acts as a sanity check), the volume group identifier, a
count of the open volumes
  16 0 4 0 *                  lvols
  20 0 4 0 u_int              num_lvols
  24 0 4 0 *                  vg_pvolsListLock.lvc_slock
  28 0 4 0 *                  pvols
  32 0 4 0 u_int              size_pvols
  36 0 4 0 u_int              num_pvols
  40 0 4 0 int                major_num
  44 0 4 0 u_int              vg_id.id1
  48 0 4 0 u_int              vg_id.id2
  52 0 2 0 short              vg_extshift
  54 0 2 0 short              vg_opencount
  56 0 4 0 u_int              vg_flags
```

VG_LOST_QUORUM	run quorum is lost
VG_ACTIVATED	volume group is activated
VG_NOLVOPENS	disallow lvol opens
VG_READONLY	volume group activated read-only

```
  60 0 4 0 *                  vg_intlock.lvc_slock
Total number of requests processed in the strategy layer and
the current pending requests
  64 0 4 0 int                vg_totalcount
  68 0 4 0 int                vg_requestcount
  72 0 4 0 *                  vg_ca_intlock.lvc_slock
```

Byte offset 80 through 539 holds a variety of MWC information structures.

This section is examined later in this chapter.

```
Pointers and offsets to various related structures
 540 0 4 0 *                 vg_vgda
 544 0 4 0 u_int             vg_LVentry_off
 548 0 4 0 u_int             vg_PVentry_off
 552 0 4 0 u_int             vg_PVentry_len
 556 0 4 0 u_int             vg_VGtrail_off
```

byte offset 560 through 1087 contains volume group status area data.

```
Configured limits for the volume group, logical volumes,
physical volumes, physical extents, extent size, data area
length, status area length, mirror cache size, the volume
group number (a quick sanity check), available physical
volumes, data area and status area block sizes, cluster
locking ID, and configuration mode (used in conjunction
with service guard configuration)
1088 0 2 0 u_short      vg_maxlvs
1090 0 2 0 u_short      vg_maxpvs
1092 0 2 0 u_short      vg_maxpxs
1100 0 4 0 u_int        vgda_len
1096 0 4 0 u_int        vg_pxsize
1104 0 4 0 u_int        vgsa_len
1108 0 4 0 u_int        mcr_len
1112 0 4 0 int          vg_num
1116 0 2 0 u_short      vg_npv_avail
1118 0 2 0 u_short      vg_npv_newavail
1120 0 2 0 u_short      vgda_blkfactor
1122 0 2 0 u_short      vgsa_blkfactor
1124 0 4 0 u_int        vg_cluster_id
1128 0 4 0 int          vg_config_mode
```

CLV_VG_CONF_STD	non-special mode
CLV_VG_CONF_EXCL	exclusive activation mode
CLV_VG_CONF_SHAR	shared activation mode

```
The remainder of the structure holds shared mode data if

applicable, volume group switching, and spare information
```

The `lvol` and `pvol` data is populated from that found in the `PVRA` and `VGRA` structures on the volume group's physical disks (Listings 11.11 and 11.12).

Listing 11.11. `q4> fields struct lvol`

```
Various queue pointers and addresses
    0 0 4 0 *                  work_Q
    4 0 4 0 *                  lv_ready_Q.lv_head
    8 0 4 0 *                  lv_ready_Q.lv_tail
   12 0 4 0 int                lv_ready_Q.lv_count
The logical ext array pointer (used during re-sync operations)
   16 0 4 0 *                  lv_lext
Three physical extent pointer maps for mapping mirrored

extents
   20 0 4 0 *                  lv_exts[0]
   24 0 4 0 *                  lv_exts[1]
   28 0 4 0 *                  lv_exts[2]
Pointer to the schedule queue, the number of stripes and the

strip size
   32 0 4 0 *                  lv_schedule
   36 0 2 0 u_short            lv_stripes
   38 0 2 0 u_short            lv_stripesize
An assortment of lock pointers and primitives
   40 0 4 0 *                  lv_lock.interlock
   44 0 4 0 u_int              lv_lock.delay
   48 0 4 0 int                lv_lock.read_count
   52 0 1 0 char               lv_lock.want_write
   53 0 1 0 char               lv_lock.want_upgrade
   54 0 1 0 char               lv_lock.waiting
```

```
   55 0 1 0 char               lv_lock.no_swap
   56 0 4 0 *                  lv_intlock.lvc_slock
   60 0 4 0 int                lv_complcnt
Next are the cumulative request count, the pending request
count, and the current status flag
   64 0 4 0 int                lv_totalcount
   68 0 4 0 int                lv_requestcount
   72 0 2 0 short              lv_status
   74 0 2 0 short              lv_allow_cfgcmd_rslvr
   76 0 2 0 u_short            lv_ref
   78 0 2 0 u_short            lv_rawavoid
   80 0 2 0 u_short            lv_rawoptions
The lvol's physical extent count, maximum number of logical
extents, and the number of in-use logical extents
   84 0 4 0 u_int              lv_curpxs
   88 0 2 0 u_short            lv_maxlxs
   90 0 2 0 u_short            lv_curlxs
   92 0 2 0 u_short            lv_flags
```

LVM_RESERVED	group file lvol0 strategy
LVM_SWAUENTIAL	sequential scheduling flag
LVM_PARALLEL	parallel scheduling flag
LVM_STRIPE	striping enabled
LVM_DYNAMIC	dynamic scheduling (not in current use)
LVM_STRIPE_NEW	new stripe (not in current use)

```
Current scheduling strategy, mirror count, and a pointer to
the bit allocation map for the logical volume
   94 0 1 0 u_char             lv_sched_strat
   95 0 1 0 u_char             lv_maxmirrors
   96 0 4 0 *                  lv_bitmap
  100 0 2 0 u_short            lv_partner
  102 0 2 0 u_short            lv_mimwchit
  104 0 2 0 u_short            lv_mimwcmiss
```

```
  108 0 4 0 u_int              lv_mirxfers
  112 0 4 0 u_int              lv_mircount
  116 0 4 0 u_int              lv_miwxfers
  120 0 4 0 u_int              lv_miwcount
  124 0 4 0 *                  lv_vg
```

Byte offset 128 through 511 contains raw buffer data.

Byte offset 512 through 895 contains logical volume disk sort information.

```
Number of seconds for a request timeout
  896 0 4 0 u_int              lv_io_timeout
```

Listing 11.12. `q4> fields struct pvol`

```
Pointers to the volume group structure, the lvmrec structure,
and the bad block directory for this pvol
     0 0  4 0 *                  pv_vg
     4 0  4 0 *                  pv_lvmrec
     8 0  4 0 *                  pv_bbdir
The maximum and current number of entries in the bad block
directory
    12 0  4 0 u_int              pv_maxdefects
    16 0  4 0 u_int              pv_curdefects
    20 0  4 0 u_int              pv_vgdats[0].tv_sec
    24 0  4 0 long               pv_vgdats[0].tv_usec
    28 0  4 0 u_int              pv_vgdats[1].tv_sec
    32 0  4 0 long               pv_vgdats[1].tv_usec
    36 0  4 0 int                pv_vgra_psn
    40 0  4 0 int                pv_data_psn
    44 0  4 0 u_int              pv_pxspace
The total number of physical extents and the number of free
extents for the pvol
    48 0  2 0 u_short            pv_pxcount
```

```
    50 0  2 0 u_short            pv_freepxs
    52 0  4 0 *                  pv_intlock.lvc_slock
    56 0  4 0 int                pv_armpos
Work queue pointers
    60 0  4 0 *                  pv_ready_Q.lv_head
    64 0  4 0 *                  pv_ready_Q.lv_tail
    68 0  4 0 int                pv_ready_Q.lv_count
Cumulative number of transfers to this pvol, number of pending
requests, status flags, and the pvol's index number within its
volume group
    72 0  4 0 int                pv_totxf
    76 0  2 0 short              pv_curxfs
    78 0  2 0 u_short            pv_flags
    80 0  1 0 u_char             pv_flags2
    81 0  1 0 u_char             pv_num
    84 0  4 0 int                pv_sa_psn[0]
  88 0  4 0 int                pv_sa_psn[1]
  92 0  4 0 u_int              pv_vgsats[0].tv_sec
  96 0  4 0 long               pv_vgsats[0].tv_usec
 100 0  4 0 u_int              pv_vgsats[1].tv_sec
 104 0  4 0 long               pv_vgsats[1].tv_usec
 108 0  4 0 *                  pv_cache_wait.lv_head
 112 0  4 0 *                  pv_cache_wait.lv_tail
 116 0  4 0 int                pv_cache_wait.lv_count
 120 0  4 0 *                  pv_cache_next
 124 0  4 0 *                  pv_mwc_rec
 128 0  4 0 u_int              pv_mwc_latest.tv_sec
 132 0  4 0 long               pv_mwc_latest.tv_usec
 136 0  4 0 int                pv_mwc_flags
 140 0  4 0 int                pv_mwc_loc[0]
 144 0  4 0 int                pv_mwc_loc[1]
 148 0  4 0 int                altpool_psn
 152 0  4 0 int                altpool_next
 156 0  4 0 int                altpool_end
Physical volume defects array
 160 0  4 0 *                  pv_defects[0]
-----------------------------------------------
 412 0  4 0 *                  pv_defects[63]
```

```
 416 0  4 0 *                  freelist
 420 0  4 0 *                  freelist_ptr
 424 0  4 0 u_int              freelistsize
 428 0  4 0 u_int              bbdirsize
```

Byte offset 432 through 523 contains the physical volume attribute data.

```
vnode and pv-links information
 524 0  4 0 *                  currentPhysicalLink
 528 0  4 0 *                  pv_wait_Q.lv_head
 532 0  4 0 *                  pv_wait_Q.lv_tail
 536 0  4 0 int                pv_wait_Q.lv_count
```

Byte offset 540 through 927 contains physical volume buffer data.

```
the size of a read/write request (multiple of 1 KB)
 928 0  2 0 u_short            pv_blkfactor
 930 0  2 0 u_short            sgio_flags
```

Byte offset 934 through 1011 contains physical volume spare information.

Now we have in the kernel all the configuration data necessary to allow translation from a logical volume offset to a physical volume offset.

One-Way and Two-Way Mirroring Mechanics and Options

Mirroring allows for either two (one-way) or three (two-way) mirroring of individual logical volumes under LVM control. The basic premise of mirroring is very straightforward, but the behind-the-scenes mechanics require additional considerations.

A major concern with mirrored operations is assuring that each mirror copy has the same data. This really comes to bear when a write is requested. To help make sure that the data is consistent across all copies,

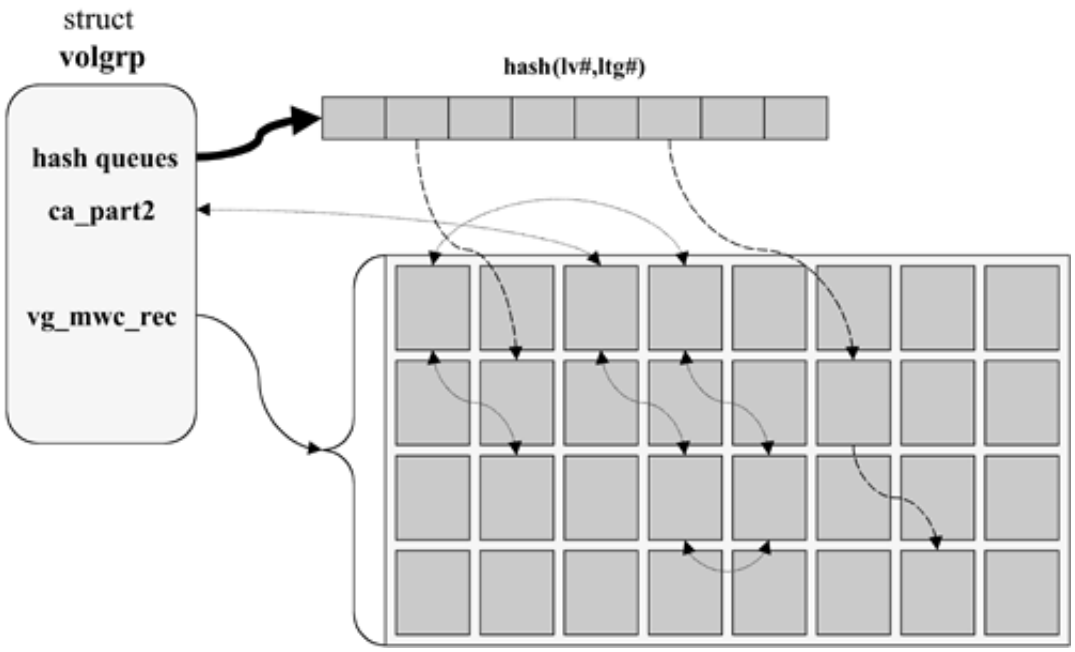
LVM incorporates an MWC strategy. When a logical volume is configured to be mirrored, it may be configured to use one of three mirror-caching policies.

> **NONE:** Choosing this policy disables all internal consistency checks. Extents are not marked as stale at activation, and mirrors are not synchronized. This may be suitable for a swap volume.
>
> **NOMWC:** No MWC records are keep and there is no performance cost during normal operation. At volume group activation time all but one volume will be marked as stale. The activation process may take some time as all extents of the stale volume(s) will have to be copied from the non-stale volume.
>
> **MWC:** If this policy is selected, then for any write request to proceed on a mirrored volume, a request is passed to the MWC layer of the driver. An entry must be made to an MWC structure and copied to one of the volume group's physical volumes before it may continue through to the next LVM layer. If a resynchronization is required, all track groups represented by an active entry in the disk-based copies of the cache data must be synchronized. Activation here proceeds more quickly than with NOMWC since only LGT's with incomplete MWC entries will need to be synchronized. The performance hit here lies with the requirement that the MWC be written to disk before LVM advances the request through its various work queues.

Note that while all mirrors should be identical following resynchronization, there is no way to know which mirror had the most current data. The mirror chosen to be copied to the others is selected by random choice.

The MWC data on a physical disk is stored in the `mwc_entry` structure we examined earlier in this chapter and is sized to hold 126 individual entries (as of HP-UX 11i). There is also a kernel-based copy of this information (see Figure 11-9).

Figure 11-9. Mirror Write Consistency Records



Listing 11.13 is an extracted portion of a **listing (with annotation) created using `q4> fields struct volgrp`**.

Listing 11.13. `q4> fields struct volgrp`

```
This contains the spinlock structure for controlling mp-access
to this data
  72 0 4 0 *                  vg_ca_intlock.lvc_slock
```

Byte offset 80 through 463 contains the vg_cache_lbuf structure used by the MWC to store MWC entries to a physical volume

---

```
Next we have the linkage pointers to the vg_cache_wait or
vg_cache_write wait queues
 464 0 4 0 *                  vg_cache_wait.lv_head
 468 0 4 0 *                  vg_cache_wait.lv_tail
 472 0 4 0 int                vg_cache_wait.lv_count
 476 0 4 0 *                  vg_cache_write.lv_pvhead
 480 0 4 0 *                  vg_cache_write.lv_pvtail
 484 0 4 0 int                vg_cache_write.lv_pvcount
The vg_mwc_rec points to the memory-resident copy of the lmv
record data
 488 0 4 0 *                  vg_mwc_rec
This points to the beginning of the memory-resident cache
array followed by a pointer to the least recently used element
in the list
 492 0 4 0 *                  ca_part2
 496 0 4 0 *                  ca_lst
This is the hash list used to speed searches for a cached
entry
 500 0 4 0 *                  ca_hash[0]
 ---------------------------------------
 528 0 4 0 *                  ca_hash[7]
The number of current free entries, the total number of
entries, and the number of changed entries in memory (dirty
entries)
 532 0 1 0 u_char             ca_free
 533 0 1 0 u_char             ca_size
```

```
 534 0 1 0 u_char              ca_chgcount
The cache flags
 535 0 1 0 u_char              ca_flags
```

CACHE_ACTIVATED	cache has been initialized
CACHE_INFLIGHT	cache being written to disk
CACHE_CHANGED	memory cache is currently dirty
CACHE_CLEAN	something is waiting for disk write to complete

```
 536 0 2 0 u_short             ca_clean_lvnum
```

Summary

As we wrap up our look at LVM, a few observations are warranted. LVM was originally introduced in HP-UX 8.x for use by servers. It wasn't until the release of HP-UX 10.0 that it was supported for use by all HP-UX system implementations. It has become a mainstay of current configurations and a workhorse of the kernel. Alternate volume managers (Veritas VxVM) offering extended features are also available for use with the current HP-UX kernel. LVM may not have all the features or administrative tools offered by other volume mangers, but it is built around a basically straightforward model and has relatively low overhead. LVM is undergoing continuing scrutiny and enhancements.

As a testament to the basically solid design of LVM, the Linux operating system volume manager is modeled on the HP LVM implementation.

We have referenced locking structures in several of our listing. In Chapter 12, we will further our understanding of this critical kernel mechanism as we discuss the implementation of multiprocessing within the HP-UX kernel.

Chapter 12. Multiprocessing and HP-UX

There are many challenges in the implementation of a multiprocessor computer system. One of the first decisions to be made is whether the system will be symmetric or asymmetric. In a symmetric system, all processors have equal access to the system and play equal roles. In an asymmetric system, one processor is "in charge," and it delegates work to the others. With the exception of some boot time tasks, HP-UX is symmetric. During boot, one processor does the initial bring-up, but once the other processors are started, all have equal access. (See Chapter 15, "System Initialization," for details of the system boot process). The HP-UX implementation has the following characteristics:

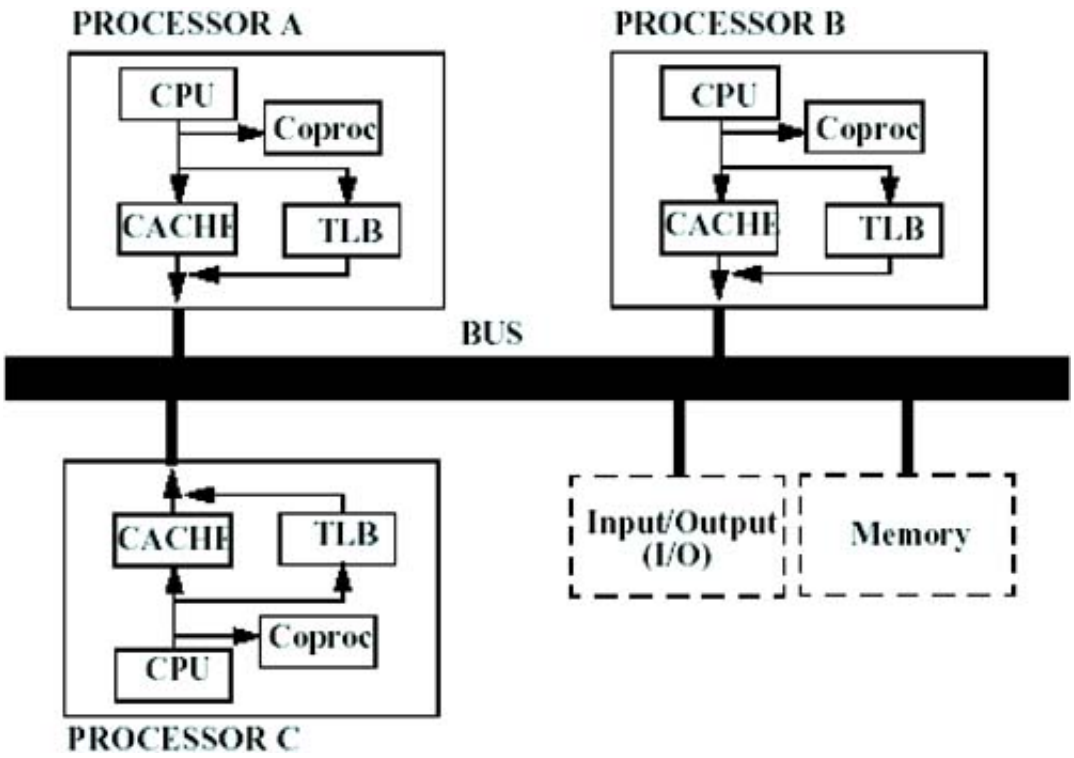
- Each processor has equal access to all system resources.
- System memory is shared by all processors.
- All I/O devices are shared by all processors.
- All processors execute a single copy of the kernel.

As with any design consideration, there are benefits to this design and there are design challenges. In this chapter, we look at how we take advantage of the benefits of multiprocessing and how we address some of the problems caused by having multiple processors.

Hardware Overview

Figure 12-1 shows an example of a typical multiprocessor system.

Figure 12-1. A Typical Multiprocessor System



Note that while the processors share the system memory and I/O, each has its own cache and TLB. In order to get the necessary speed benefits from the cache, the cache has to be physically close to the CPU. But this brings up another challenge: how do you maintain consistency between the caches on the various processors? For example, say processor A and processor B both have the same memory address cached in their respective data caches. If processor A modifies the data at that address, it will change the copy in its cache and mark the cache as dirty, but it will not necessarily write that data to main memory immediately. The data that processor B has in its cache is now no longer valid.

We get around this problem with a *cache coherency check transaction.* Every time a processor modifies data that is in its cache, it sends out a cache coherency check transaction. All other processors must check their caches to see if they have copies of the data in their own caches. If a processor has the data in cache and it has been modified, then the data is immediately flushed to main memory. Then that cache line is marked invalid so that the next time the data is needed, it must be read from memory again.

To use our example from above, when processor A wants to modify the data, it first sends out the cache coherency check transaction. Processor B sees this transaction and finds that it has that address in its own cache. If the cache line in processor B's cache is dirty, meaning that it needs to be written to memory, processor B immediately flushes that cache line out to memory. It then marks the cache line as invalid so that if it needs the data again later, it will have to load it from memory. If processor B has the data in cache but has not modified it (it is "clean"), then it just has to invalidate the cache line.

Multiprocessing Data Structures

Each processor has a data structure that is used to keep track of data specific to that processor. This structure is of type struct `mpinfo`, which is `typdef`ed to `mpinfo_t`. The global variable `mpproc_info` is declared as a pointer to a `struct mpinfo`. At boot time, the system allocates enough memory to hold one struct `mpinfo` for each processor. It then points `mpproc_info` to the beginning of this allocated memory. In this way, `mpproc_info` can be used as an array of `struct mpinfo` by using the processor index. For example, `mpproc_info[0]` is the `mpinfo` structure for the first processor in the system. The `mpinfo` structure is also sometimes called the *per-processor data pointer* (PPDP).

The `mpinfo` structure is a huge structure—over 13 KB in an 11.11 system. A large portion of this is the `ki_proc_info_data`. This structure is used for kernel profiling and tracing, and has counters to keep track of `syscall` and procedure calls. Table 12-1 lists some of the more useful and interesting members of the `mpinfo` structure. Note that the code uses the terms "processor" and "SPU" interchangeably. An SPU is a System Processor Unit.

Table 12-1. The `mpinfo` Structure

Type and Name	Description
`prochpa`	Pointer to the processor's hard physical address (HPA)
`iva`	Interrupt vector address
`procindex`	Processor number
`spinlock_depth`	Number of spinlocks held
`entry_int_level`	Interrupt level the processor was at when the first spinlock was taken
`spu_state`	State of the SPU: one of `SPU_ENABLED`, `SPU_DISABLED`, `SPU_PENDING`, `SPU_INTR_ENABLED`, `SPU_DISABLED_HW_FAILURE`
`threadp`	Pointer to the current thread structure
`uareasid`	Space ID of the `uarea` structure
`prevthreadp`	Pointer to the previous thread structure
`mpcntrs`	Per-processor counters
`mp_rq`	Run queues

The `mpcntrs` member has some interesting information in it and is worth a further look (Listing 12.1).

Listing 12.1. Declaration for the `mpcntrs` structure

```
typedef struct mpcntrs {
         u_long fsreads;          /* # of actual reads to FS blocks */
         u_long fswrites;         /* # of actual writes to FS blocks */
         u_long nfsreads;          /* # NFS reads issued */
         u_long nfswrites;         /* # NFS writes issued */
```

```
        u_long bnfsread;        /* # bytes read via NFS */
        u_long bnfswrite;       /* # bytes written via NFS */
        u_long phread;          /* # physical reads issued */
        u_long phwrite;         /* # physical writes issued */
        u_long runocc;          /* # of times rq occupied since boot */
        u_long runque;          /* cumulative len of rq since boot */
        u_long sysexec;         /* # of exec()'s since boot */
        u_long sysread;         /* # of read/readv()'s since boot */
        u_long syswrite;        /* # of write/writev()'s since boot */
        u_long sysnami;         /* # of filename lookups since boot */
        u_long sysiget;         /* # of inode fetches since boot */
        u_long sysselect;       /* # of select calls since boot */
        u_long dirblk;          /* # of directory disk blocks read */
        u_long semacnt;         /* # of SysV sema ops since boot */
        u_long msgcnt;          /* # of SysV msg ops since boot */
        u_long muxincnt;        /* # of mux input xfers since boot */
        u_long muxoutcnt;       /* # of mux output xfers since boot */
        u_long ttyrawcnt;       /* # of raw chars read since boot */
        u_long ttycanoncnt;     /* # of canonical chars processed */
        u_long ttyoutcnt;       /* # of chars output since boot */
        long activeprocs;       /* # of proc  table entries allocated */
        long activethreads;     /* # of thread table entries allocated */
        long activeinodes;      /* # of inode table entries allocated */
        long activefiles;       /* # of file  table entries allocated */
} mpcntrs_t;
```

These statistics can be useful in comparing various types of loads across the processors. Note that these numbers are per-processor, so each processor has its own set of counters. The numbers are also cumulative since boot time. The exception is the four counters that start with "active": `activeprocs`, `activethreads`, `activeinodes`, and `activefiles`. These counters represent the current number of items used by this processor. For example, `activeprocs` is incremented when a process is created and decremented when the process is destroyed. Examining these values for the different processors can tell us if a particular type of activity is getting tied up on a particular processor.

Exploring the System:

To load all the `mpinfo` structures and display some data from each:

```
q4> load struct mpinfo from mpproc_info max
➥  nmpinfo
```

```
q4> print -x procindex%d prochpa spinlock_depth
```

To display the `mpcntrs` data structures from a particular processor:

```
q4> load struct mpinfo from mpproc_info max
➥ nmpinfo

q4> keep procindex == 1

q4> print -tx | grep mpcntrs
```

Another part of the `mpinfo` structure that is worth looking into is the `mp_rq` member. This is the run queue structure for this processor. Each processor has its own set of run queues. As we discussed in Chapter 5, "Process and Thread Management from the Process's Viewpoint," threads are placed on a run queue when they become ready to run. The details of how threads move on and off the run queues are covered in that chapter. Here we look at the actual structure of the queues themselves (Listing 12.2).

Listing 12.2. Declaration for the `struct mp_rq`

```
struct mp_rq {
     int bestq;
     int neavg_on_rq;                    /* 256X avg of neff_on_rq */
     int nready_free;                    /* active, not locked to any spu */
     int nready_free_alpha;              /* active, not locked, needs alpha */
     int nready_locked;                  /* active, locked to this spu */
     int nready_locked_alpha;            /* active, locked, needs alpha*/
     int asema_ignored;                  /* alpha related. see pm_policy.c */
        u_int ticks_last_migration;      /* ticks_since_boot at last migration */
     ulong_t cycles_last_migration;      /* IT at last migration */
     struct _horse *nexthorse;           /* next horse in the carousel (FSS) */
     lock_t *run_queue_lock;             /* lock for this processor's run Qs */
     struct mp_threadhd qs[NQS];
     int fsid_active;
     int spares[9];
};
```

The queues themselves are in the array of `struct mp_threadhd` members. The value NQS is the number of queues per processor. NQS is currently defined at 160, so there are 160 `struct mp_threadhds`. The `struct mp_threadhd` is simply a pair of pointers to threads. It is used as the head of a doubly linked list of threads:

```
typedef struct  mp_threadhd {

        struct kthread *th_link;   /* linked list of running threads */

       struct kthread *th_rlink;

} mp_threadhd_t;
```

The `th_link` member points to the first thread in the queue, and the `th_rlink` member points to the last thread in the queue. Within the threads, `kthread->th_link` is the forward pointer and `kthread->th_rlink` is the backward pointer.

Synchronization

One of the challenges of multiprocessor systems is synchronizing access between the processors. In the early days of UNIX, it could be assumed that there was only one processor and only one kernel, and once that kernel started running, it had absolute control of the system. If the kernel had a critical sequence of instructions, it could turn off interrupts and execute that code without the risk of anything interrupting the flow. With multiprocessor systems, even with interrupts off, it is possible that two processors could try to access the same data at the same time. For an example of why this is a problem, consider the following code, which stores a value into a circular buffer, increments the index into the buffer, then wraps the index back to the beginning if it is past the end:

```
buffer[index++] = new_data;

if (index == BUFFER_SIZE) {

  index = 0;

}
```

This looks pretty reasonable until you consider the case where two processors execute the same code at the same time. Let's say your buffer has 200 entries, so `BUFFER_SIZE` is 200. You've got 199 entries in the buffer, so index is at 199. Processor A stores the new data at `buffer[199]` and increments index to 200. Now processor B stores its new data at `buffer[200]` (off the end of the buffer) and increments the index to 201. Next, processor A gets to the test and checks to see if index is equal to 200. It's not, it's 201, so it keeps on going. Processor B does the same. Now you've got your index pointing past the end of the buffer and it's never going to come back. It'll just keep on writing over memory until something breaks.

In order to prevent this kind of data contention, we have two general classes of locks: *spinlocks* and *semaphores*. In general, spinlocks are used for short waits, and they protect against access by another processor. Semaphores can be used for longer waits, and they protect against access by another thread on any processor. Here are some of the other key differences between the two:

- Spinlocks may be held for only a short time. If any spinlock is held for more than 60 seconds, the system will panic. Semaphores can be held indefinitely.
- If the system tries to get a spinlock and finds it already locked, it will busy wait. It will do no other work until the spinlock is free. If it tries to get a semaphore and finds it locked, it can put the current process to sleep and switch to another process.
- A process can sleep while holding a semaphore. Processes that are holding spinlocks are not allowed to sleep.

Let's look into the details of spinlocks first.

Spinlocks

Listing 12.3 is the structure that represents a spinlock.

Listing 12.3. `Spinlock` structure

```
struct lock {

    volatile uintptr_t   sl_lock;
```

```
    volatile uintptr_t   sl_owner;
    char                *sl_name_ptr;
    volatile uint16_t    sl_flag;
    volatile uint16_t    sl_next_cpu;
    uint32_t             sl_arena_alloc_flag;
};
```

This structure is also `typdef`ed to `lock_t`; you'll see `lock_t` used more commonly than `struct lock`.

The first word in the structure, `sl_lock`, is the one that indicates whether or not the lock is currently locked. If `sl_lock` is zero, then the spinlock is locked. If it is nonzero, it is not locked. When it is not locked, this field should contain the per-processor data pointer (PPDP) of the processor that had it last. (Remember that the PPDP is also the address of the `mpinfo` structure for that processor.)

The second field, `sl_owner`, does the opposite of `sl_lock`. When the spinlock is locked, this field contains the PPDP of the owner. When it is unlocked, it contains zero.

`sl_name_ptr` is a pointer to a character string containing the name of the spinlock. Each spinlock is given a name when it is allocated. Prior to HP-UX release 11.11, this pointer had to be left-shifted two bits before it was used. Starting with 11.11, the pointer can be used as is.

You might wonder why we use zero to indicate locked. It seems more logical that a zero would be unlocked and nonzero would be locked. The answer comes from the PA-RISC architecture. In order to implement any kind of locking mechanism, the hardware must provide an atomic operation for testing a value and setting the value. By atomic, we mean that it is a single operation—the test and the set must both happen. The instruction can't be interrupted partway through. Many hardware architectures implement a Test and Set instruction. On PA-RISC this operation is done with the Load and Clear Word (LDCW) instruction.

The instruction is given a register and an address. The instruction first loads the contents of memory from the given address into the given register. It then clears the value in memory—sets it to zero. When the instruction completes, the memory location will always be zero. The register will be whatever value was in memory *before* the instruction was executed. We use this instruction to set the value in `sl_lock` in the spinlock structure. After the instruction executes, the spinlock will be locked. We then examine the register we passed to the LDCW instruction. If that register is zero, it means that the spinlock was already locked by some other processor. If it is nonzero, then it was previously unlocked, and now is locked by us.

So what do we do if we try to lock a spinlock and find it is already locked? We spin, of course. More specifically, we call `wait_for_lock()`, which does some accounting, then sits in a tight loop waiting for the lock to free up. When it finds the lock free, it tries to grab it. If it succeeds, then `wait_for_lock()` returns and the processor continues. If, when we try to grab the lock, we find that we can't, that means some other processor beat us to it. We continue to spin.

During this spin we also keep a timer. If we find after 60 seconds that we're still waiting for the lock, then the system will panic with a Spinlock deadlock message. Sixty seconds is far too long for a spinlock to be held. If we're not able to get it within 60 seconds, something is broken.

Notice that once we find the lock free, there's a chance we still can't get it because another processor got there first. On a very busy system this can cause problems on a popular spinlock. To avoid this, the system uses a method of handing the spinlock from one processor to another. Rather than just freeing a spinlock, a processor can check to see if another processor is waiting for it. If so, it can assign it to that other processor. This is where the other two fields, `sl_flag` and `sl_next_cpu`, are used.

`sl_flag` is set to one if there is another processor waiting for the spinlock. So when we try to lock a spinlock and find it already locked, one of the things that `wait_for_lock()` does is `set sl_flag` to one. The `spinunlock()` code checks this flag when unlocking the spinlock. If the flag is one, indicating that someone is waiting, rather than just releasing the spinlock, the code donates it to the next processor.

Arbitration

Picking who to donate the spinlock to involves two more pieces of the puzzle: the `sl_arb array` and the `sl_next_cpu` member. `sl_arb` is a global array of pointers to spinlocks, one for each processor in the

system. When a processor starts to wait for a spinlock, in addition to setting `sl_flag` to one, it sets its own `sl_arb` array entry to point to the spinlock it is waiting for. Now, when the `spinunlock` code is ready to donate a lock to another processor, it can look through `sl_arb` for an entry that points to the spinlock that's being released. When it finds one, that processor is the one that gets the lock next. The `sl_next_cpu` member is used to decide where in the `sl_arb` array to start looking. Each time the lock is donated to a processor, that processor's index is put into `sl_next_cpu`. The next time we go through the process of looking for someone to donate to, we start at `sl_next_cpu+1`. This way, we proceed round-robin through the list of waiting processors. The one that got it last time is the last one to get it this time.

Prearbitration

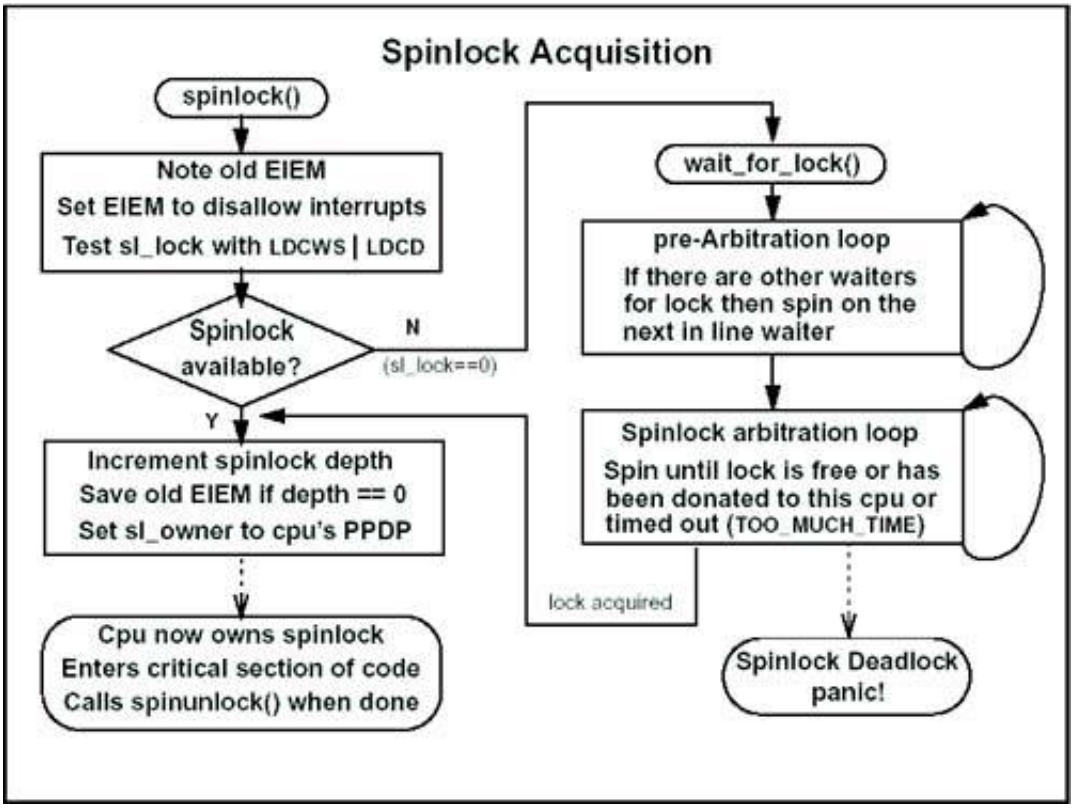
There is one more shortcoming to the spin and arbitrate method we've discussed so far. With a heavily used spinlock, many processors can be continuously looking at the same memory. This causes contention between the caches on those processors—the caches need to be continually flushed and updated to verify correctness. To avoid this contention, one more step was added to the process of acquiring a lock: *prearbitration*. Similar to the `sl_arg` array, there is a global array called `pre_arbitration`, with one entry per processor. Each entry in this array is of type `struct pre_arbitration`. In this structure is a pointer to a lock and a sequence number.

Now when we try to get a lock and find it locked, we first call the `preArbitration()` routine. This routine looks through the `pre_arbitration` array for another processor that is already pointing to the lock we want —that is, another processor that's already waiting for this lock. If we find none, then the routine returns and we fall through to the usual spinlock code, spinning on the lock and waiting for it to be released. However, if we do find one or more processors already waiting for the lock, we find the one that is last in line based on the sequence number. Then we set our own `pa_lock` field to indicate which lock we're waiting for, *and then we spin, watching the processor that is in line before us.* We're looking at the `pa_lock` field of that processor's `pre_arbitration` structure, waiting for it to change to zero. When we see the `pa_lock` field for the guy in front of us go to zero, we know that he has now acquired the lock and we're next in line. We exit `pre_arbitration()` and continue to spin on the spinlock as we would without arbitration.

Each of the `pre_arbitration` structures is aligned and sized such that it will be on its own cache line. That means that there is only one processor spinning on that cache line and it doesn't have to be updated on the other processors.

The whole process sounds complicated, but it's really not as bad as it sounds. Figure 12-2 shows a flowchart of the process for acquiring a spinlock.

Figure 12-2. Flowchart of Spinlock Acquisition



Compared to acquiring the lock, releasing it is a pretty simple operation. First, we check for the lock's `sl_flag` to see if anyone is waiting for it. If no one is waiting, we simply set `sl_owner` to zero and `sl_lock` to our PPDP. Then we restore the interrupt mask from before we had the lock, decrement the spinlock depth, and we're done.

If we find there is someone waiting for the lock, the process is still not much more complicated. We look through the `sl_arb` array to find out who is waiting for it. Then we set `sl_owner` to the PPDP of the processor that is waiting, leaving `sl_lock` at zero. The lock remains locked, but now the next processor in line is the owner of it. Recall that in `wait_for_lock()` the processor is looking for either `sl_lock` to become nonzero or for `sl_owner` to match its own PPDP. When it sees `sl_owner` change to its own PPDP, it knows that it now owns the lock and can continue.

Exploring the system:

To display the `io_tree_lock`:

```
q4> load struct lock from io_tree_lock
```

```
q4> print -tx
```

Semaphores

Now that we've covered spinlocks, let's have a look at the other synchronization primitive offered by the kernel: semaphores. Semaphores in the HP-UX kernel can be roughly divided into two types: mutual exclusion semaphores, or *mutexes*, and synchronization semaphores. Mutual exclusion semaphores are further divided into *alpha semaphores* and *beta semaphores*.

The key difference between an alpha semaphore and a beta semaphore is that alpha semaphores are used to protect data structures, which must be consistent when a context switch occurs. It is assumed that a structure is inconsistent while the lock is held. That's the idea of the lock—to prevent access while the structure is inconsistent. For that reason, a process can't sleep while an alpha semaphore is held. Over time, the alpha semaphores have been replaced with beta semaphores such that only one remains now: the file system semaphore, `filesys_sema`. Alpha semaphores are now officially deprecated, so there will be no new alpha semaphores created. The `filesys_sema` is the last.

There are several differences between mutual exclusion semaphores and synchronization semaphores. Table 12-2 lists the key differences.

Table 12.2. Comparison of Semaphore Types

Mutual Exclusion Semaphores	Synchronization Semaphores
Lock and unlock operations are always done by the same thread.	Lock and unlock are normally done by different threads. When the first thread locks the semaphore, it blocks. When another thread unlocks the semaphore, it allows the first to proceed.
Semaphores are initialized to one. The first lock will succeed, all following locks will block.	Semaphores are initialized to zero so that the first lock will block.
Semaphores have only two states: locked or unlocked. For this reason, these are also called binary semaphores.	Semaphores can take any value. For this reason, these are also called counting semaphores.

Alpha Semaphores

The actual implementation of alpha semaphores is fairly straightforward. The tricky part—dealing with contention between processors—is handled using a spinlock. Alpha semaphores are represented by a `struct sema` data type, also called a `sema_t`. The first field in a `struct sema` is a pointer to a spinlock, `sa_lock`. This protects the semaphore structure from concurrent access. Any time the semaphore structure is modified, the code first grabs the `sa_lock` spinlock, then modifies the semaphore, then releases the spinlock. In addition to the `sa_lock` field, there are two other fields—`sa_count` and `sa_owner`—which implement the core of the semaphore's functionality. `sa_count` is one if the semaphore is unlocked and zero if it is locked. `sa_owner` is a pointer to the thread that owns the semaphore if it is locked.

Because there is only one alpha semaphore in the system, there are a few alpha semaphore routines that are no longer used. For example, `pxsema()` and `vxsema()` are used to release one semaphore and acquire another in the same operation. Since there's only one semaphore, there's never a need to do this. The following are the more useful semaphore functions:

`initsema()` initialize a semaphore.

`psema()` acquires a semaphore.

`vsema()` releases a semaphore.

`owns_sema()` checks that a thread owns the semaphore.

`sema_add()` adds a semaphore to a thread's list of semaphores.

`sema_delete()` removes a semaphore from a thread's list.

Most of these routines are very simple. `initsema()` initializes the fields in the semaphore, setting the `sa_count` to one and allocating a spinlock structure for `sa_lock`. `sema_add()` and `sema_delete()` are actually more complicated than they need to be. They maintain a linked list of semaphores owned by a thread. The `kthread` structure has a field, `kt_sema`, which points to a list of owned semaphores. Within the semaphore, the `sa_next` and `sa_prev` pointers continue the list. However, since there is only one alpha semaphore, this list will either be empty or contain one entry, the `filesys_sema`. The `owns_sema()` routine simply checks to see if the `sa_owner` field is equal to the current thread.

Acquiring a semaphore with the `psema()` routine can be complicated. The first step is to lock the spinlock associated with the semaphore. Next, we check `sa_count`. If it is one, indicating it is free, then we set it to zero, set `sa_owner` to the current thread, and release the spinlock. The complicated part comes in if we find that the semaphore is already owned.

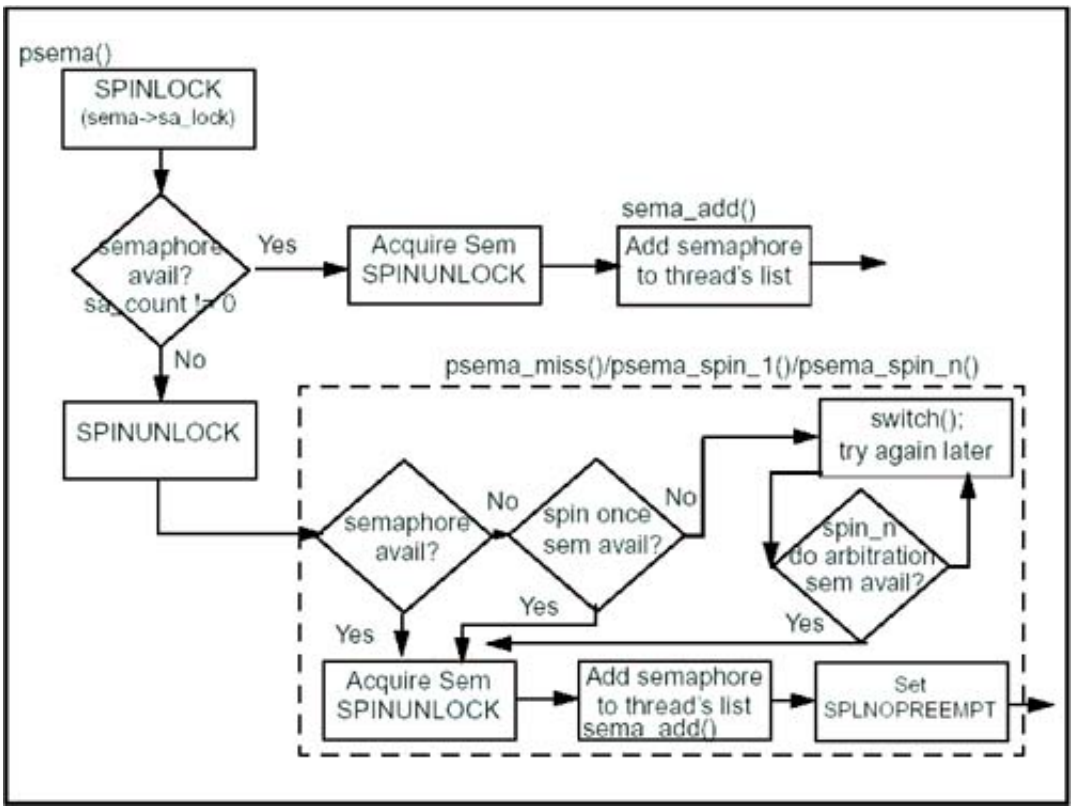
Because a semaphore can be held for a long time, we don't want to just spin waiting for it—we want to be able to context-switch and let the processor do other work. On the other hand, changing context is an expensive process. If the wait is going to be a short one, we'd rather not do the context switch. To balance out these requirements, when we try to get a semaphore and find it locked, the first thing we do is a short spin wait. The routine `psema_spin_1()` is called to spin for up to 50,000 clock cycles trying to get the lock. If we fail to get the lock after 50,000 cycles, we then call `psema_switch_1()` to give up the processor and let another process take over.

When we return from `psema_switch_1` (when it's our turn on the processor again), we go into another section of spinning, this time by calling `psema_spin_n()`. The difference between `psema_spin_1()` and `psema_spin_n()` is that `psema_spin_n()` uses an arbitration mechanism to determine how long to spin and who should get the semaphore next. The semaphore has a field, `sa_turn`, which says which SPU is next in line in the waiters list. Each time we get ready to spin on the semaphore, we first look at the `sa_missers[]` array in the semaphore. This array keeps track, in priority order, of the SPUs that are waiting for the semaphore. The earlier entries in `sa_missers` have been waiting longer and thus have a higher priority for getting the semaphore. If the current SPU isn't in the `sa_missers[]` array it gets added at the end of the list. Now we look at the `sa_turn` member in the semaphore and find out whose turn is next. We find that SPU in the `sa_missers` array, and if it is farther down the list than we are, we set `sa_turn` to our own SPU, indicating that we're next in line.

This `sa_turn` field has two effects. First, when we're spinning, waiting for the semaphore, we spin longer if it's our turn. If `sa_turn` equals our SPU, then we'll spin for 50,000 cycles waiting for the semaphore. If not, we spin only 5,000 cycles. Second, once we find the semaphore is free, we acquire it only if `sa_turn` points to our SPU or it is the value `SA_NONE`.

We go through this cycle of "find out whose turn it is, then spin" nine times before we context-switch. We continue doing this until we get our turn at the semaphore. Figure 12-3 shows the flowchart of the process of acquiring an alpha semaphore.

Figure 12-3. Flowchart of Alpha Semaphore Acquisition



As in the spinlock case, releasing the semaphore is trivial. We call `sema_delete()` to remove the semaphore from the thread's list, then we set `sa_count` to one, indicating the semaphore is free.

An interesting construct is seen in the semaphore code and in fact is used in a lot of code that uses spinlocks. In order to avoid contention for the spinlock, the semaphore code grabs the spinlock only when it has to—that is, when it is going to make a change to the semaphore. The spinlock isn't needed when reading the contents of the semaphore, only when modifying it. However, if the code reads the semaphore and finds that it needs to change it—as in the case when it finds it free and is now going to lock it—then the spinlock has to be acquired in order to make the change. But there's one more test that has to be done: now that you've got the spinlock, you have to make sure that the structure didn't change since you last looked at it. This leads to code that looks like this:

```
if (sema->sa_count == 1) {
  SPINLOCK(sema->sa_lock);
  if (sema->sa_count == 1) {
       sema->count = 0;
       sema->owner = u.u_kthreadp;
  }
  SPINUNLOCK(sema->sa_lock);
}
```

We check the semaphore; if it's free, we grab the spinlock. Then we check to see if it's still free, and only then can we lock the semaphore for ourselves.

Exploring the system:

To examine the file system semaphore:

```
q4> load struct sema from filesys_sema
```

```
q4> print -tx
```

Beta Semaphores

Most of the functions on beta semaphores are similar to those for alpha semaphores. One concept that is different is the idea of *disowning* a semaphore. There are a few cases when a semaphore must be acquired and held until an I/O operation completes. When the I/O completes later and an interrupt is generated, the "current thread" is whichever one is running when the interrupt occurs. Before the I/O starts, the system disowns the semaphore, leaving it locked but without an owner. Then, when the I/O completes later, the system can unlock the semaphore even though it is in a different thread context.

Internally, there are several differences between the alpha and beta semaphores. The beta semaphore structure itself has three fields: `b_lock`, `owner`, and `order`. `b_lock` is a bitmask value. The lowest order bit is on if the semaphore is currently locked, and the second bit is on if someone is waiting for the semaphore. The `owner` field will be zero if the semaphore is not locked, negative one if the semaphore is locked but disowned, or it will point to the thread which owns the semaphore.

The `order` field is used to ensure that if a thread acquires several semaphores, it does so in the correct order. To prevent a deadlock situation, all semaphores have an order. Every time a semaphore is acquired, the system checks all other semaphores already held by that thread. The order of the new semaphore must not be less than the order of any semaphore already held. This check of a semaphore order takes some extra time and really shouldn't be necessary. Semaphore should never be violated if there are no bugs in the system. Because of this, this feature is actually turned off in most kernels. When a new kernel is under development, semaphore order checking is turned on. When the kernel is ready for release, the kernel is rebuilt in "performance mode" with this check turned off to eliminate the overhead.

Another difference between the alpha and beta semaphore implementations is the lack of a spinlock and information about waiters in the `b_sema` structure. This information is kept in a separate structure, the beta semaphore hash. The system maintains a single table of information about beta semaphores, called `b_sema_htbl`. This table is an array of `struct bh_sema`. Each `bh_sema` structure has a pointer to a spinlock and forward and backward pointers to a doubly linked list of threads. Entries in this array are accessed by running the address of the semaphore through a hashing algorithm to get an index into the array.

The spinlock entry from this beta semaphore hashtable is used just as the spinlock is on an alpha semaphore. Before modifying the semaphore, the code must acquire the spinlock. The spinlock is released immediately after the semaphore is updated. The thread pointers make up a list of threads that are waiting for the semaphore. The `bh_sema->fp` points to the first waiting thread, and the remaining threads are linked using the `kthread->kt_wait_list` pointer. `bh_sema->rp` points to the tail of the list, and `kthread->kt_rwait_list` is the reverse pointer within the thread structure.

When a thread tries to acquire a beta semaphore and finds it locked, it sets the second bit in `b_lock` by ORing `b_lock` with `SEM_WAIT`. It then inserts its thread pointer onto the end of the list of waiters. When the semaphore becomes available, the `b_vsema()` routine checks to see if the `SEM_WANT` flag is set. If so, it goes to the list of waiters and gets the first entry off the list. It then checks the `kt_beta_misses` value from the thread structure of the first waiter. This value indicates the number of times the thread tried to get the semaphore, found it locked, and went to sleep. If this value is less than 10, then the semaphore is simply freed. If the value is 10, indicating that the thread has slept at least 10 times waiting for the semaphore, the semaphore is donated to the thread.

Exploring the system:

To display the `pid_sema` beta semaphore:

```
q4> load struct b_sema from &pid_sema

q4> print -x b_lock order owner
```

To display the beta semaphore hashtable entry for `pid_sema`:

```
q4> idx = (((&pid_sema) >> 6) & (64-1))

q4> load bh_smea_t from &bsema_htbl skip idx

q4> print -x beta_spinlock fp bp
```

Synchronization Semaphores

Where mutual exclusion semaphores are generally used to protect data structures from simultaneous access, synchronization semaphores are used to synchronize events. The implementation and usage are almost the same as for the mutual exclusion semaphores.

The data structure for a synchronization semaphore is `struct sync`, or `sync_t`. A `struct sync` has many of the same fields as the other semaphores. `s_lock` is a pointer to a spinlock to protect the semaphore during modification, `count` is the current value of the semaphore, `wait_list` is a pointer to a list of threads waiting for the semaphore, and `order` is the locking order.

The two main operations on a synchronization semaphore are `psync()` and `vsync()`. `psync()` corresponds to the lock operation on a mutual exclusion semaphore. The `psync` operation first decrements the `count` field in the semaphore. If the result is less than zero, the process blocks. If the result is greater than or equal to zero, the process continues. The `vsync()` operation is the inverse. It increments the value in the `count` field.

The `count` field can be thought of as the number of `psync()` calls that can be done before blocking. Because the `count` field can take on values other than just one or zero, you can also think of a synchronization semaphore as keeping track of how many of a particular resource are available. In this case, the `psync()` call corresponds to using a resource, and `vsync()` corresponds to freeing one up.

There is an additional call, `cvsync()`, which is a conditional `vsync()`. It performs the `vsync` operation only if there is a thread waiting for the semaphore.

An example of how this might be used is if you have a thread that uses a resource, such as memory, and another that frees up that resource. If the first thread finds it is low on memory, it can call `psync()` to wait until more is available. The thread that is freeing memory can call `cvsync()` once it has made more memory available. This technique is used by the scheduler to suspend itself when memory is critical and to wait for the `statdaemon` to reactivate it when memory becomes available.

Summary

We looked at a variety of techniques for managing multiple processors in a system and the interaction between them. While adding additional processors to a system makes more processing power available, it also introduces some very complex problems that must be overcome to keep the system running smoothly.

Chapter 13. Kernel Services

In addition to managing the system hardware and scheduling processes, the kernel provides a variety of services to both user applications and kernel subsystems. The services we most commonly associate with the kernel—the system calls—are covered in Chapter 4, "Programs, Processes, and Threads." The kernel also provides a variety of services to other subsystems within the kernel. In this chapter, we look at what some of those services are and how they are handled.

The Callout Table

The kernel sometimes needs to be able to schedule events to happen in the future. For example, a driver might want a timeout on an I/O operation. The driver starts an I/O and is interrupted by the I/O system when the I/O completes, but if for some reason the I/O doesn't complete, the driver needs to know about that too. The driver sets a *callout* for some number of ticks in the future, specifying a routine to be called at that time. If the I/O completes successfully, then the callout is cancelled. If the callout routine gets called, then the driver knows something went wrong.

Interface

The basic function for setting a timeout is called `timeout()`. It takes three arguments: a pointer to a function to call, a `void*` argument to pass to the function, and an integer representing the number of ticks in the future at which the function should be called. There are also a few other variations on this routine, including `timeout_on_spu()`, which allows the caller to specify on which system processor unit (SPU) the timeout function should run.

There are also functions for canceling a timeout or for finding timeouts. The `untimeout()` function is passed a function pointer and an argument, just like the first two arguments to `timeout()`. If a timeout is registered for that function and that argument, then the timeout is cancelled and the return value will be the number of ticks remaining until the timeout expires. The `find_timeout()` function does the same, but removal of the timeout is optional. In fact, `untimeout` just calls `find_timeout` with the `remove` value set to one.

Data Structures

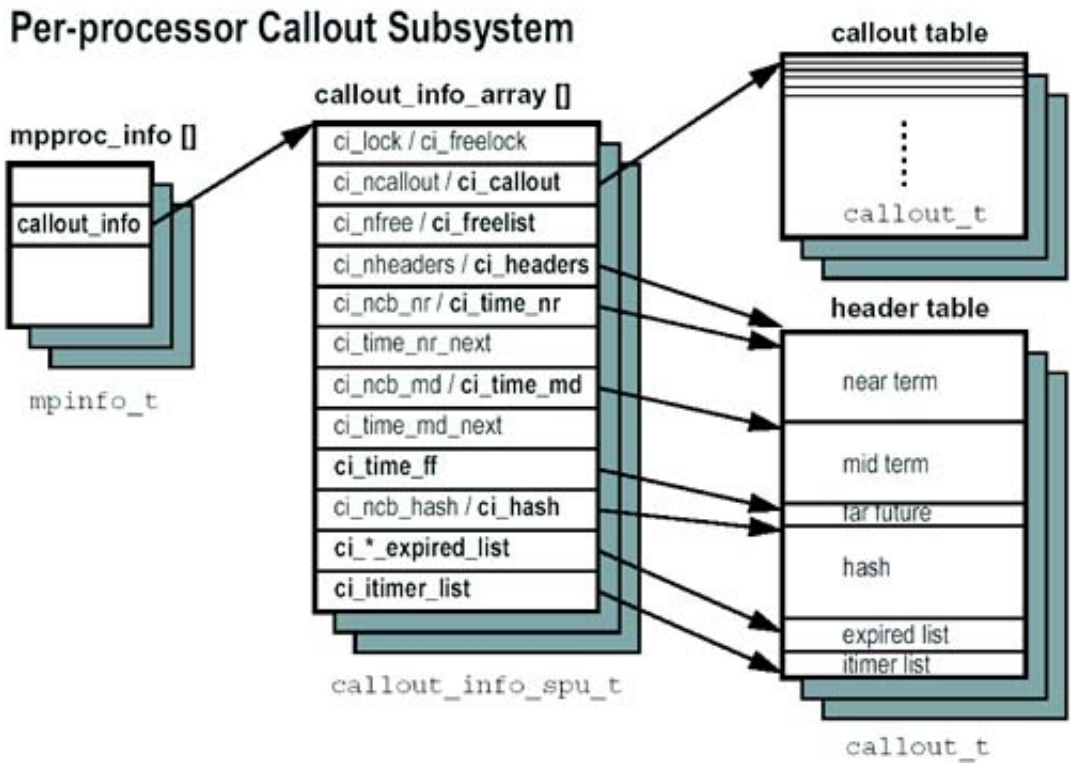
The primary data structure used for callouts is the `struct callout`, or `callout_t`. The structure is used for two different things: either a "real callout"—that is, to represent a callout event—or a "callout header." At boot time a fixed number of callout structures are allocated. The number is determined by the `ncallout` system tunable. These callout structures are divided between the SPUs so that each SPU has its own pool of callout structures. The reason we divide them into per-SPU pools is to reduce lock contention. If we had a single pool, then we'd have to have a single lock controlling access to the pool. Because a callout scheduled on one SPU is triggered by that SPU, it makes sense for each SPU to have its own pool of callouts. An SPU can access its pool without blocking operations on another SPU. In the `mpinfo` structure for the SPU there is a field called `callout_info`, which points to a `struct callout_info_spu` (typedef `callout_info_spu_t`). This structure maintains information about the callouts that pertain to that SPU. This structure has (among other things) a pointer to the first `callout_t` in its pool (`ci_callout`), one past the last `callout_t` in the pool (`ci_calloutEND`), and a pointer to a linked list of free callouts (`ci_freelist`).

When a new callout is scheduled, the system gets an available callout structure from the `ci_freelist`, fills in the appropriate data, and then attaches it to a list of callouts that are scheduled. If an SPU runs out of free callouts, it can go to the pools belonging to the other SPUs and "borrow" callout structures from those free lists.

The `callout_info` structure also maintains three sets of linked lists of active events based on when they are scheduled to occur. These are the near-term list pointed to by `ci_time_nr`, the midterm list pointed to by `ci_time_md`, and the far-future list pointed to by `ci_time_ff`. The division of events into groups like this avoids the overhead of having to search the entire list of callouts at every interrupt. We only have to search the near-term list to see if anything has expired. When the near-term list is empty, we migrate events from the other lists.

To enable us to find a particular callout, there is also a hashtable and hash chains. A hashing function determines which of several hash chains to search for a callout, then that chain is searched to see if the callout is on the list. Figure 13-1 shows the relationship of the callout information in the `mpinfo` structure and the various lists of callouts.

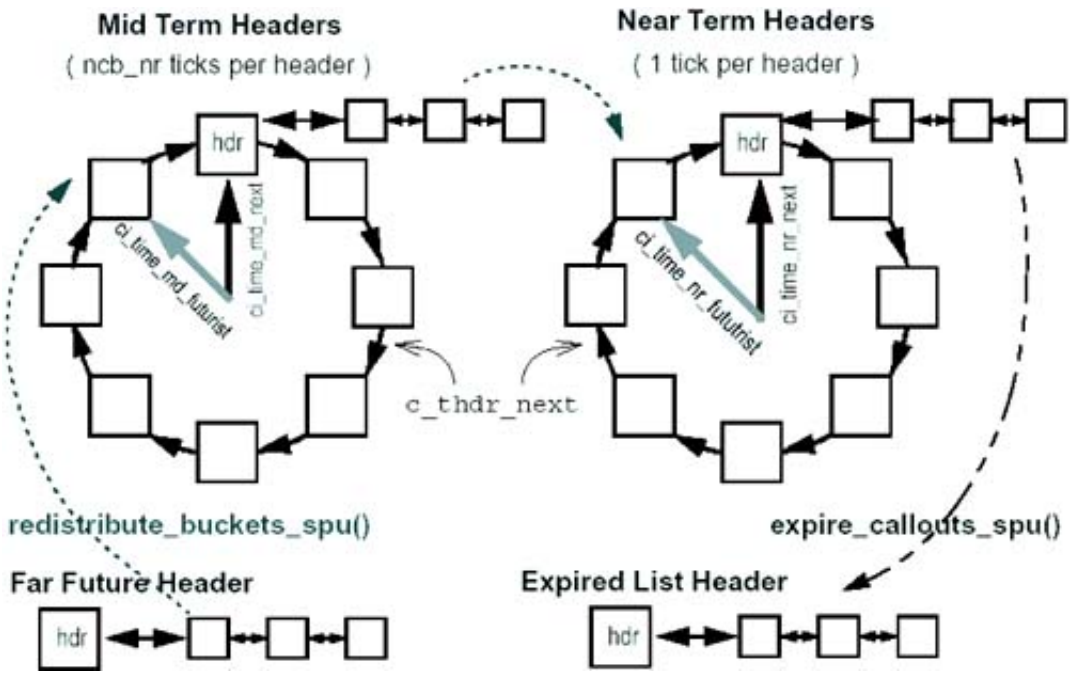
Figure 13-1. Per-Processor Structures for Callouts



Callout Processing and Expiration

In the previous section we mentioned the three sets of callouts—the near term, the midterm, and the far future. While the far future is a single linked list, the midterm and near term are both collections of lists. The code is designed such that the near and the midterm can have a different number of lists, but currently both have 128 lists. These sets of lists are arranged as a circular array, as shown in Figure 13-2.

Figure 13-2. Callout Table Headers



You can think of these as analogous to a clock. On a clock each time the minute hand makes a complete revolution, the hour hand moves forward one tick. Similarly, each list in the midterm wheel represents the same amount of time as the entire near-term wheel. On the near-term wheel all callouts that expire in a particular tick are on the same list. With 128 lists in the wheel, that means anything expiring in the next 128 ticks is near term. The `ci_time_nr_next` pointer keeps track of which list is the next one due to expire. The `ci_time_nr_futurist` pointer is the end of the queue—the callouts that expire the latest of those considered near term. Each time we get a clock tick interrupt, we check the expiry time for those callouts pointed to by `ci_time_nr_next`. If the callout is due (or past due), we call the callout functions and then move `ci_time_nr_next` to the next list.

In the midterm wheel, each list represents 128 ticks. When the near-term wheel makes one full revolution—that is, when `ci_time_nr_next` catches up to `ci_time_nr_futurist`–all of the callouts pointed to by `ci_time_md_next` get moved onto the near-term wheel. We've essentially moved the hand on the midterm wheel up a tick. When the midterm wheel has made a full revolution we go to the far-future list.

The far-future list is just a single list, kept in sorted order. As the midterm wheel needs replenishing, callouts are moved from the far-future list.

All of this magic happens every clock interrupt. The clock interrupt calls `per_spu_hardclock()`, which in turn calls `expire_callouts_spu()`. `Expire_callouts_spu()` calls `redistribute_buckets()` to handle the moving of callouts from one wheel to the next.

While all this sounds like a lot of work, it's actually much less overhead than the alternative. It used to be that all callouts were kept in one big linked list, sorted by expiry time. With that method, you only had to check the head of the list to see if the next callout was due to expire. The problem was the process of keeping the list sorted. With the possibility of thousands of callouts on the list, it becomes very expensive in terms of processing power to walk down the list and find the right place to insert a new callout.

Kernel Memory Allocation

In Chapter 3, "The Kernel: Basic Organization," we talked about how the kernel manages virtual and physical memory. Memory is divided into pages of 4 KB each. The kernel's page allocator handles allocating pages of memory to processes that need them. However, the kernel itself needs dynamic memory too, and it uses a different process. The memory allocations for the kernel are frequently for small areas of memory —the size of a `proc` structure or the size of a network buffer, for example. It would be impractical to just call the page allocator and request a page each time we need a small piece of memory. That's where the kernel memory allocator comes in. It handles requests for memory from the kernel, getting pages from the page allocator as needed. It then divides these pages into smaller pieces so that it can be used efficiently.

Beginning with HP-UX version 11.11, the method of tracking and allocating kernel memory has changed. Prior to 11.11 HP-UX, a scheme known as the McKusick & Karel memory allocator, also called the "bucket allocator," was used. In release 11.11, this was replaced by the new arena allocator.

Although our focus in this book is on HP-UX version 11.11, the bucket allocator is pervasive enough that it's worth a brief look. Without going into too much detail, we look at the bucket allocator before examining the newer arena allocator.

The Bucket Allocator

The bucket allocator divides memory requests into buckets by size, where each size is a power of two. Thus, we have a 32-byte bucket, a 64-byte bucket, a 128-byte bucket, and so on. When a kernel subsystem needs to allocate memory, the request is rounded up to the next power of two. So, for example, if a request is for 100 bytes, it will come from the 128-byte bucket.

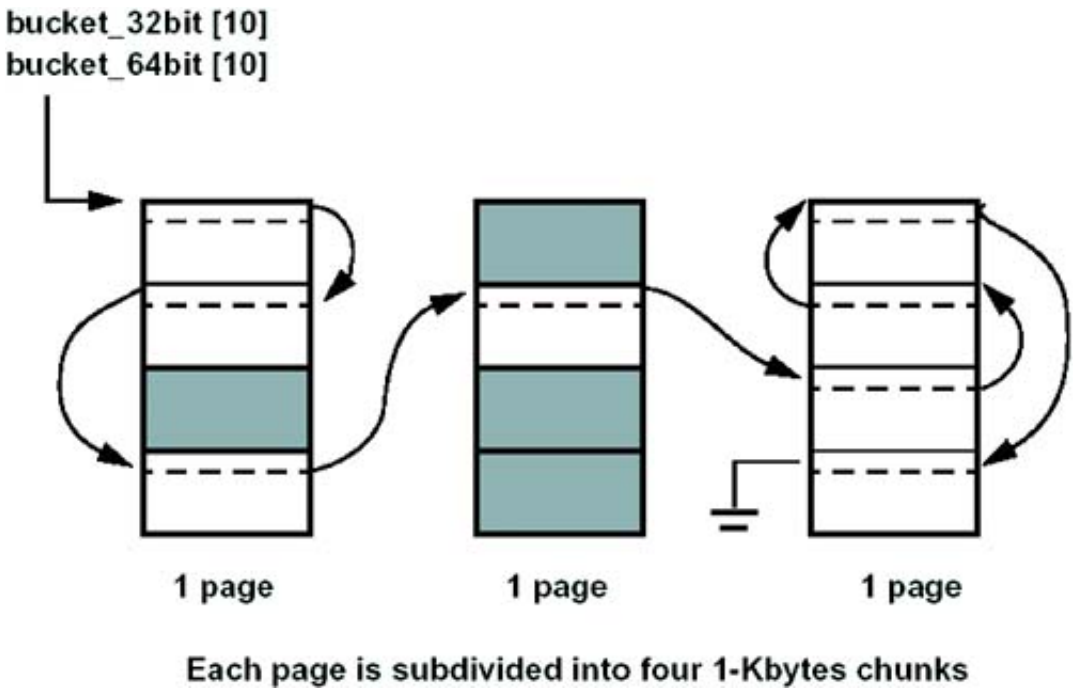
The arrays `bucket_32bit[]` (for 32-bit kernels) and `bucket_64bit[]` (for 64-bit kernels) contain the head of a list of free areas of memory. The index into the array is the power of two of the size of the buckets: 2^5 is 32, so the head of the 32-byte bucket list is at `bucket_64bit[5]`, and so on.

Memory is allocated using the `MALLOC` macro. This macro checks the `freelist` pointer to see if an area of memory of the requested size is on the list. If so, then the first address off the list is returned. If not, then a new page is requested from the page allocator by calling `kmalloc()`. This page then is broken into pieces, the pieces are put on the free list, and the first of them is returned to the requestor.

When the memory is no longer needed, it is returned to the bucket lists by use of the `FREE` macro. If a lot of memory allocations are made and then freed, this can leave many pages of memory on the bucket free lists. The `vhand` process periodically looks for blocks of memory on the free lists that can be coalesced back into pages and returned to the page pool.

When the memory is on the free list, the first word in the memory area is used as the link to the next free area. Figure 13-3 shows an example of three pages, each of which has been broken into four 1-KB areas. In the figure, four are in use and eight are on the free list. The third page in this example would be a candidate for being returned to the page pool.

Figure 13-3. Buckets on the Free List



One of the shortcomings of the bucket allocator is the risk of corruption if a freed pointer is reused. When a subsystem frees memory, the first word in that area of memory is now used as a link in the linked list. If the pointer to that memory gets reused after it has been freed, the result is that the entire free list gets corrupted. This corruption causes a panic the next time the system tries to get a piece of memory from that bucket. These types of corruption can be very hard to track down, because by the time the panic occurs, the code that caused the corruption may already be done and gone.

The Arena Allocator

The arena allocator has several advantages over the bucket allocator. With this allocation method, each subsystem creates its own "arena" of memory. Memory is then requested from and returned to that arena. This allows the allocator to tailor its performance to a particular use and also helps to contain any possible memory corruption. A problem in one arena can't affect another subsystem. The arena allocator refers to each piece of memory allocated or released as a *memory object*.

One feature of the arena allocator is that it distinguishes between fixed-size memory objects and variable-sized memory objects. If an arena is created as a fixed-size arena, then all allocations made from that arena are of the same size. This is a very common operation in the kernel—for example, one arena, the `CALLOUT` arena, is used for allocating callout structures. Fixed-size arenas can manage memory more efficiently than can variable-sized arenas because each page of memory can be divided up optimally for that particular size of data object.

Each object in the arena has a header associated with it. When the object is on a free list, the list link is kept in this header, not in the first word of the object itself as with the bucket allocator. By moving the link outside of the memory object, we reduce the chance of someone corrupting the free list by reusing a freed pointer.

The arena allocator divides objects into three size categories. A *small object* is one that is less than 4 KB. A *large object* is from 4 KB to 32 KB. An *xlarge object* is an object larger than 32 KB.

Arena Allocator Data Structures

The top-level data structure for arenas is the `struct kmem_arena_globals`. There is exactly one of these,

called karena, allocated at system initialization time. This holds system global arena information such as the pointer to the linked list of arenas and the number of arenas.

Each arena is represented by a struct kmem_arena. The kmem_arena_create() function creates a new kmem_arena, initializes it, and links it onto the list of arenas in the system. kmem_arena_create() takes four arguments: the size of the objects in the arena, then name of the arena, a pointer to a kmem_arena_attr structure, and a flags field. For an arena that stores variable-sized objects, the size will be zero.

The attribute structure, kmem_arena_attr, allows for a great deal of tuning of the arena's attributes. Table 13-1 lists the fields in the kmem_arena_attr structure.

Table 13-1. The kmem_arena_attr Structure

Type	Name	Description
size_t	kat_struct_size	The size of the attribute structure itself.
int (*)	kat_ctor	Pointer to a constructor function, called when new objects are created.
void (*)	kat_dtor	Pointer to a destructor function, called when objects are destroyed.
long	kat_maxcnt	Maximum number of objects.
long	kat_minfcnt	Minimum number of objects to be kept on the free lists.
long	kat_maxpgcnt	Maximum number of pages in the arena for variable-sized objects.
long	kat_refillcnt	Minimum number of objects to be created at each refill.
kat_flags_t	kat_flags	Attribute flags.
size_t	kat_align	Alignment requirements for objects.

Objects and Object Headers

As mentioned, each object has an object header associated with it. When an object is on the free list, this header has a pointer to the next object on the list. When objects are in use, the header points back to the free list header in the arena it came from. Headers are stored differently depending on the size of the object.

For small objects, the header is stored immediately before the object in memory. The header in this case needs only a single pointer—the one that points to the free list if free or to the free list head if in use. We don't need a pointer to the object itself because we know it immediately follows the header. The small object header looks like this:

```
typedef union kmem_obj_hdr {

  union  kmem_obj_hdr    *ko_fnext; /* Next element in free list */

  struct kmem_flist_hdr  *ko_fhdr;  /* Ptr to Free list header */

} kmem_obj_hdr_t;
```

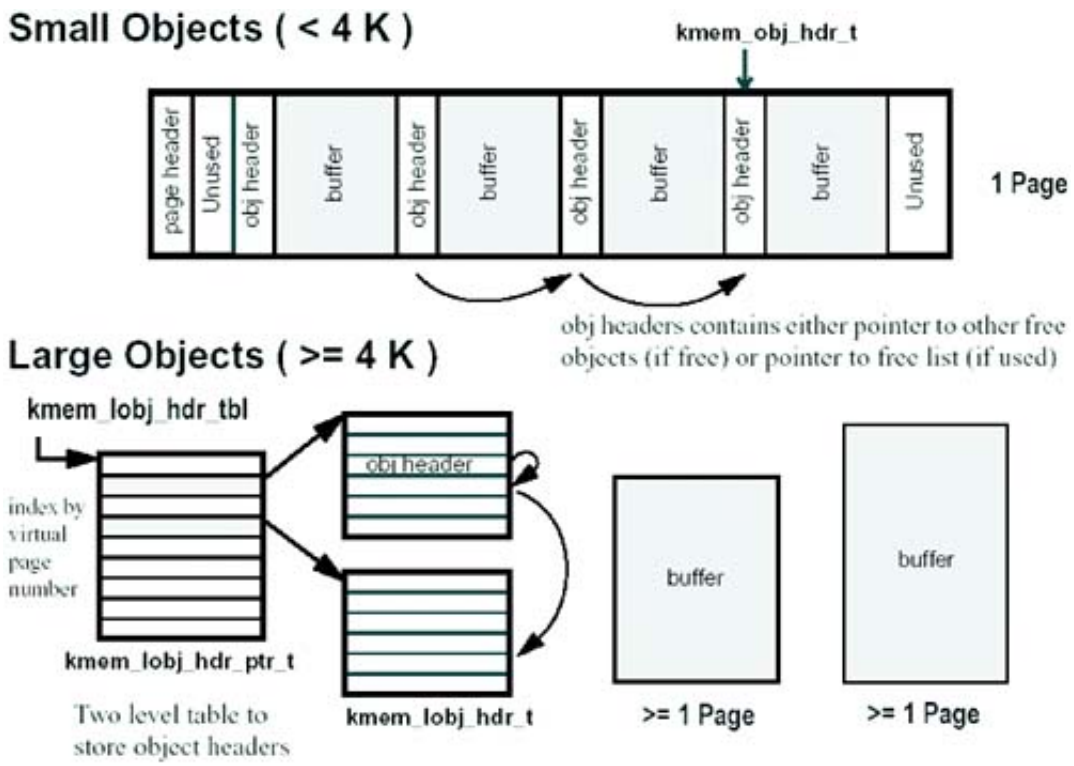
For large objects, we have a global two-level structure for pointers to the object headers. The top level, *kmem_lobj_hdr_tbl*, is allocated at system initialization time. It is an array of pointers to second-level tables, which in turn are arrays of pointers to objects. To find a particular object header, the system uses the upper bits of the virtual page number to find the right entry in the first-level table, then follows that pointer and uses the lower bits of the virtual page number to find the second-level table entry. This is the object header.

Because the object headers for large objects are not next to the object itself, we need another field in the header. In addition to the pointer to the free list, we need a pointer to the object itself. The large object header builds on the small object header but adds this extra pointer:

```
typedef struct kmem_lobj_hdr {
     union  kmem_obj_hdr  kl_ohdr;         /* Object header for large objects */
     union  {
      void            *kl_vaddr;   /* Virtual address of object. */
      size_t          kl_objsize; /* Object size */
   } kl_un;
} kmem_lobj_hdr_t;
```

Note the additional `object size` field. This is used for xlarge objects. Xlarge objects are not cached—that is, when freed they don't go back on a free list, they just get released. For small and large objects, the size of the object is in the free list header. But for xlarge objects we have no free list, and therefore no free-list header, so we have to store the size here. Figure 13-4 shows the layout of objects and object headers in memory.

Figure 13-4. Memory Arena Objects and Object Headers

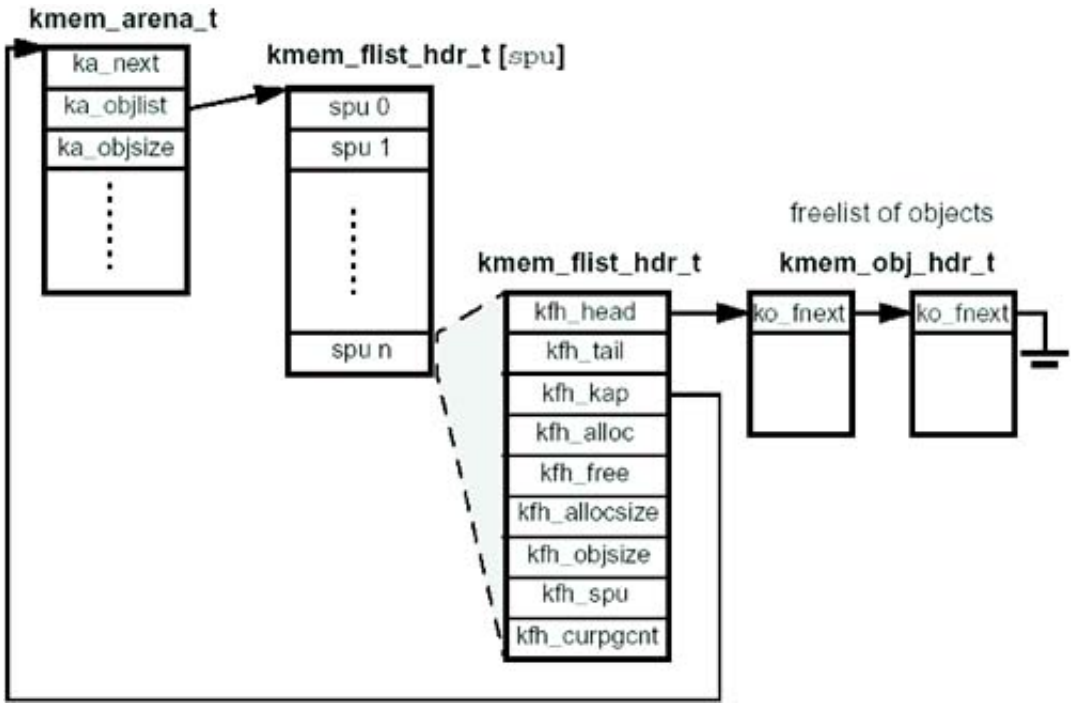


This combination of objects and object headers gives us better performance and better control over memory than does the bucket allocator.

Object Free Lists

For fixed-size objects' the arena maintains one free list per SPU. The `ka_objlist` entry in the `kmem_arena` structure is a pointer to an array of `kmem_flist_hdr` structures, one for each SPU. This header has a pointer to the first free object, `kfh_head`. In addition, it has fields used for managing the list, such as the object size, current page count, and a pointer back to the arena it belongs in. Figure 13-5 illustrates the free list for fixed-size objects.

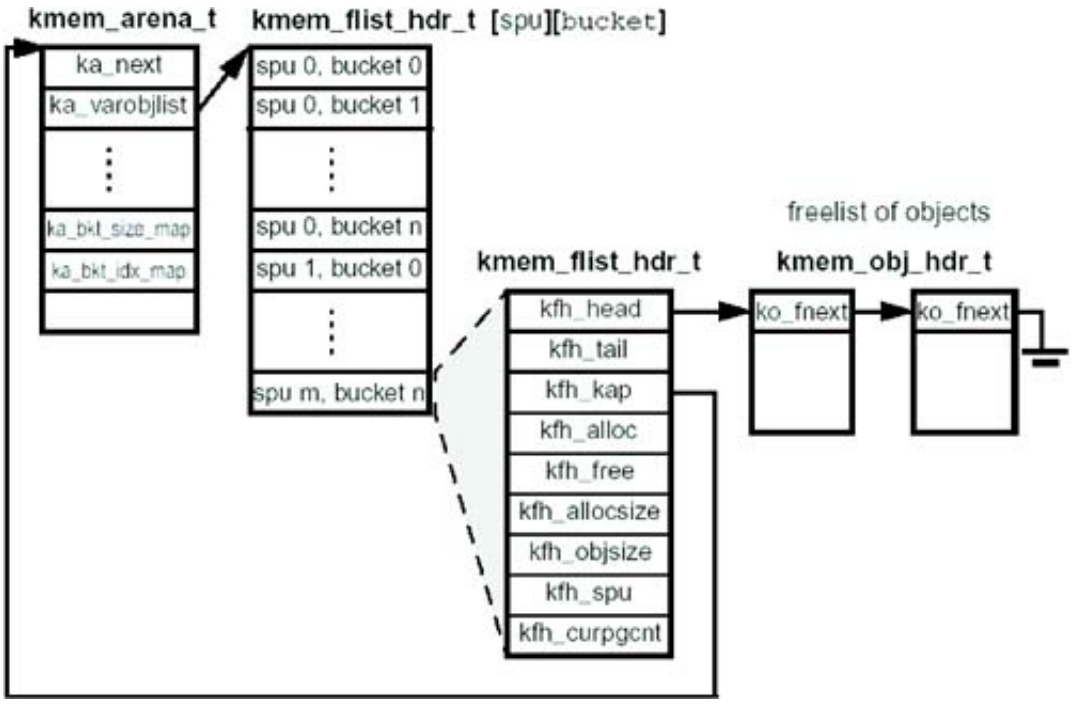
Figure 13-5. Free List for Fixed-Size Objects



The free lists for variable-sized objects are different. Free memory objects are handled similarly to the way they were with the bucket allocator, with a set of buckets pointing to lists of different sized objects. In this way, when a new object is needed, the allocator can go to the free list and get an object that is already the correct size. One difference between this scheme and the older bucket allocator is that the bucket sizes are different. Because of the extra overhead of the object headers, it doesn't make sense to make the buckets exact powers of two. For example, if you have a 1024-byte bucket, then you could only get three of these on a page. If instead we use a 960-byte bucket, then we can get four objects of up to 960 bytes on a page efficiently. The next smaller bucket would be a 768-byte bucket, which gives you five objects on a page, and so on. The size of the various buckets is kept in an array, `kmem_bkt_default_size_map`. When the arena is created, the creator can opt to use the standard power-of-two buckets if that layout is required for compatibility with older code, but it is a less efficient use of memory.

For these bucket lists for variable-sized objects, we use the `ka_varobjlist` pointer in the arena structure. This points to a two-dimensional array, indexed by CPU number and bucket number. Then, entries in this array are the `kmem_flist_hdr` structures, just like for the fixed objects. Figure 13-6 illustrates the free lists for variable-sized objects.

Figure 13-6. Free List for Variable-Sized Objects



The arena allocator is an improvement over the older bucket allocator in that it gives us more control over the size and alignment of objects, makes more efficient use of memory, and helps isolate one area of the kernel from another in terms of memory usage.

Managing Kernel Virtual Addresses: The `Sysmap`

In addition to allocating and freeing memory, the kernel has to keep track of its own virtual address space. Recall that for user processes, this is handled by the `VAS`, `pregion`, and `region` structures associated with each thread. The kernel doesn't have these structures for its own address space. All kernel addresses are in the same space—space 0, or `KERNELSPACE`–and it has to manage the available virtual addresses within that space. When physical pages are assigned for kernel use, the kernel has to map new virtual addresses to those physical pages, and when the pages are freed, the virtual addresses must be freed too.

The kernel keeps track of available addresses using a *resource map*. A resource map is a collection of structures, each of which describes a range of address space. Resource maps are allocated at system initialization time and have a fixed size. The first entry in the map is a header that points to the end of the map and has a pointer to the name of the map:

```
struct map {
  struct mapent *m_limit; /* address of last slot in map */
  char *m_name; /* name of resource - use in overflow warning message */
};
```

All the remaining entries contain a size and an address:

```
struct mapent {
  unsigned long m_size; /* size of this segment of the map */
```

```
  unsigned long m_addr; /* resource-space addr of start of segment */
};
```

The `m_addr` field is the address of the beginning of a range of free addresses, and the `m_size` field is the size of the range.

The kernel stores its dynamic and static data in the first quadrant of space zero. On a 32-bit kernel this is a 1-GB range, on a 64-bit kernel it's 4 TB. A resource map called `sysmap_32bit` manages the first 1 GB of this address space. On the 64-bit kernel there is also a resource map called `sysmap_64bit` that manages the remaining part of the first quadrant, beyond the first gigabyte.

The resource map is arranged in sorted order by address. Entries in the map are moved to make room to insert a new entry and shifted to fill in a hole created by a deleted entry. There are no blank entries—the map starts with the second entry (the first is the header), and the end of the map is indicated by an `m_addr` value of zero. Because zero is used as an end-of-list marker, all the actual page numbers are incremented by one so that we can manage page zero as well. So, page zero is indicated by an `m_addr` of one, and page `0x33a7` would be indicated by an `m_addr` of `0x33a8`.

When a new range of virtual addresses is needed, the kernel uses a "first fit" algorithm to find a range. It traverses the `sysmap` starting at the beginning and looks for the first range that is big enough for the range it needs. When it finds one, it uses that address range and adjusts the size field to indicate that the available range is now smaller. So, for example, if we need four pages of addresses and the first entry we find in the map that is big enough is `[9, 0x1b37,]` then we've found a range at `0x1b37` that is nine pages long. The kernel uses pages `0x1b37` to `0x1b3a` for the four pages and changes the entry in the `sysmap` to read `[5, 0x1b3b]`.

If we ever run into a case where a range of virtual addresses is needed and there is no entry in the `sysmap` big enough, we will panic with the message `kalloc: out of virtual space`. This can be caused if the `sysmap` has become fragmented, containing many small segments and no large ones. Of course, it can also be caused if we really are out of virtual addresses, but that is rare unless a buggy subsystem has run away and grabbed up all the address ranges.

As memory is freed and the address ranges associated with it are no longer needed, those address ranges get put back into the table. If possible, the kernel coalesces adjacent ranges into one larger range; otherwise a new entry is inserted into the table.

The size of the `sysmap` is fixed at system boot time. The `nsysmap` and `nsysmap64` tunables set the initial size of the `sysmap`s. In addition, the kernel adjusts those numbers upward at boot time based on the amount of memory in the system. If the `sysmap` ever becomes full, it will be unable to accept new entries. Again, this can happen if the address space becomes fragmented such that the table is used up by a lot of small, disjoint addresses. When the kernel needs to free a range of addresses and can't record it in the `sysmap`, it prints a message to syslog similar to this: `sysmap: rmap ovflo, lost [3f600, 3f6a3]`. This tells us that this address range has been lost and can't be used again.

Summary

In addition to all of its other jobs, the kernel provides a range of services both to user processes and to kernel subsystems. These include a large number of system calls provided for applications as well as managing timeouts and kernel memory for kernel subsystems.

Chapter 14. Signals

Signals are the software equivalent of a hardware trap or interrupt. They are a way of informing a process or thread that an event has occurred. Just like their hardware counterparts, signals can be caused either by the instruction being executed or by an external event.

A signal that is caused by the current instruction is a *synchronous signal*. Examples are floating-point exceptions and memory faults. Another way to think of a synchronous signal is as a signal that a thread causes to be sent to itself.

A signal that is caused by an eternal event is an *asynchronous signal*. These are signals that originate outside the current thread—for example, a signal due to data being ready from a terminal, or another thread using the `kill()` system call to send a signal.

Another aspect of synchronous versus asynchronous signals has to do with multithreaded processes. Since a synchronous signal is caused by a thread, it also delivered to that thread to be handled. An asynchronous signal can be delivered to any thread within the process.

Being synchronous or asynchronous is a property of how the signal is delivered—not the type of signal. The `SIGSEGV` signal is normally caused by a bad memory access and so is normally sent synchronously—but it could also be sent asynchronously. A process could use the `kill()` system call to send a `SIGSEGV` signal.

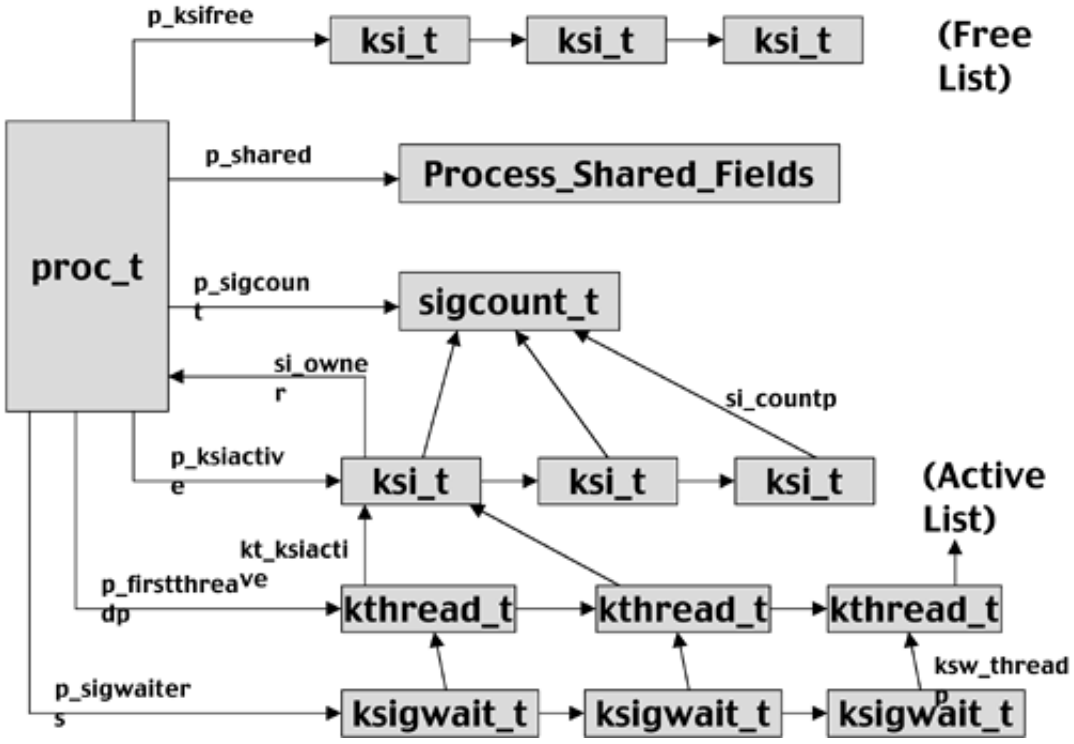
In general, a process can choose to ignore a signal, hold the signal pending, or handle it by calling a signal-handling routine. For each signal type, there is also a default action, and a process can request the default treatment. This default is different for different signals. The scenario gets a bit more complicated when a process is multithreaded. In addition to the above, the application can specify which thread will handle a signal. In this way one or more threads can be used specifically as signal handlers while other threads are allowed to run without interruption.

Let's take a look at how signals are processed—from setting up a signal handler to signal generation, delivery, and handling.

Signal Data Structures

Before we get into the process of handling signals, let's look at the associated data structures. Figure 14-1 shows the various signal-related structures and how they relate to the `proc` and `kthread` structures.

Figure 14-1. Signal-Related Data Structures



With the advent of multithreaded processes in HP-UX version 11.0, the concept of signals became much more complicated, and several new data structures were created. At the same time, much of the data that used to be kept in the `proc` structure was moved into a new structure called a `struct Process_Shared_Fields`. The `proc` structure has a field, `p_shared`, that points to the `Process_Shared_Fields` structure. The `Process_Shared_Fields` is mostly made up of a group of fields of type `ksigset_t`, which is merely an array of integers used as a bitfield—one bit per signal type. A variety of routines exist in the kernel for manipulating these bitfields—setting or clearing individual bits, setting or clearing all bits, testing bits, and so on. The `Process_Shared_fields` structure is shown in Listing 14.1.

Listing 14.1. `Process_Shared_fields structure`

```
struct Process_Shared_Fields {

  ksigset_t pshared_sigignore;     /* signals being ignored */

  ksigset_t pshared_sigcatch;      /* signals being caught by
```

```
                                    user */
  int       pshared_nsig;         /* # of signals recognizedby
                                    process */
  void    (*pshared_signal[_NSIG-1])();    /* disposition of
                                    signals */
  ksigset_t pshared_sighandmask[_NSIG-1];  /* signals to be
                                    blocked */
  ksigset_t pshared_sigreset;     /* reset handler after
                                    catching */
  ksigset_t pshared_siginfowanted;/* sigs to be delivered witv
                                    siginfo*/
  ksigset_t pshared_sigrestart;   /* restart interrupted sys
                                    call */
  ksigset_t pshared_signodefer;   /* Do not block sig in
                                    handler */
  ksigset_t pshared_sigonstack;   /* Signals to take on
                                    sigstack */
  void    (*pshared_sigreturn)(); /* handler return
                                    address
  proc_sigcontexttype_t pshared_sigcontexttype;
};
```

In addition to the above structure, there are a few signal-related fields that are still in the `proc` structure (Listing 14.2).

Listing 14.2. Signal-related fields in `proc` structure

```
struct proc {
         ...
   ksigset_t p_sig;            /* sigs pending on the
                                          proc */
   ksigset_t p_ksi_avail;      /* signals with siginfo
                                          available */
   ksigset_t p_ksifl_alloced;  /* signals with freelist
                                          entries */
   ksi_t *p_ksiactive;         /* active list of pending
                                          signals */
   ksi_t *p_ksifree;           /* free list of ksi_t's */
```

```
   struct sigcount *p_sigcountp; /* # signals pending at
                                                receivers */
   ksigwait_t  *p_sigwaiters;   /* list of sigwaiting
                                                threads */
   int       p_cursig;          /* sig which caused proc
                                                stop/exit */
              ...
};
```

Each process may also have a `struct sigcount` associated with it, pointed to by the field `p_sigcountp` in the `proc` structure. The `sigcount` structure keeps track of how many signals the process has outstanding—that is, sent but not yet received. This structure is only used for signals sent with the `sigqueue()` system call, which allows the sender to specify data along with the signal. The `kill()` system call does not use the `sigcount` structure. There is a pool of these structures available to the system, kept on a free list. When a process sends a signal, a `struct sigcount` is taken from the pool and attached to the process. When the last of the signals is received, the structure is returned to the pool. The reason this is a separate structure rather than just a counter in the `proc` structure is that a process could send a signal, then exit before the signal is received. This gives us a persistent record of the signals sent. Listing 14.3 is the `sigcount` structure.

Listing 14.3. `sigcount` structure

```
typedef struct sigcount {
   unsigned int     sc_count:16;   /* sigqueues sent by this
                                      process and
                                      pending at a receiver */
   sc_owner_state_t sc_owner_state:1;
   unsigned int     sc_unused:7;
   unsigned int     sc_cookie:8;  /* DS ID token */
   struct sigcount *sc_next;      /* for freelist management */
} sigcount_t;
```

Along with the `sigcount` structure, signals sent with `sigqueue()` also need a place to store the data that is sent along with the signal. This information must be outside of the `proc` structure for the same reason that the `sigcount` does—it may have to persist after the process terminates. The structure that holds this data is a `struct __ksi`, or `ksi_t`. The more interesting fields include a pointer to the `sigcount` structure that's associated with the data, the number of the signal, the cause of the signal (user-generated, timer-generated, I/O completion, etc.), an `errno` that can be associated with the signal, and two data fields. The first of these fields, `si_value`, is used by the `sigqueue()` system call. When a user sends a signal using `sigqueue()`, he or she can specify a value to be sent along with the signal. The second data field, a large union called `__data`, contains various information depending on the signal number. Some examples are the faulting address for `SIGSEGV` or `SIGBUS`, or the status returned by a child for `SIGCLD`. Here's a partial list of what's in the `ksi_t`:

```
typedef struct __ksi {
```

```
   struct sigcount *si_countp;     /* sender's pending
                                              signal count */
   int               si_signo;   /* signal number */
   sigval_t          si_value;
   int               si_code;    /* cause of signal */
   int               si_errno;   /* associated errno */
   union {
     ...
   } __data;
} ksi_t;
```

Each `kthread` structure also has some data related to signals. These are as follows:

```
struct kthread {
     ...
     char    kt_cursig;        /* number of current pending signal,
                                     if any */
     ksigset_t kt_sig;        /* signals pending to this thread */
     ksigset_t kt_ksi_avail;  /* signals with siginfo available */
     ksi_t *kt_ksiactive;     /* list of pending queued signals*/
     ksigset_t kt_sigmask;    /* current signal mask */
     int         kt_proc_sig; /* signo of sig directed at proc */
     ...
       }
```

Finally, we have the `ksigwait` structure. Each thread that is explicitly waiting for a signal—with the `pause()`, `sigpause()`, `sigwait()`, or `sigsuspend()` system calls—will have one of these structures. The `ksigwait` structures are pointed to by the `proc` structure, and they in turn point to the thread that is waiting for the signal.

Signal Anticipation

Signal anticipation is what an application can do to plan for signals. From the application's point of view, the preferred method of planning for signals is with the `sigaction()` system call. Other system calls are available too, but they have essentially the same functionality. `Sigaction()` is the POSIX standard and should be used in all new development.

`Sigaction()` expects three arguments: the signal number, a `struct sigaction` containing new values to be used for that signal, and a `struct sigaction` into which the old values can be returned. The file `/usr/include/sys/signal.h` includes symbolic definitions of the signal numbers, the definition of the `sigaction` structure, and a variety of other signal-related declarations. The `sigaction` structure contains a pointer to a handler routine, a mask that indicates what signals should be blocked while handling this signal, and a set of flags for modifying the signal-handling behavior. We'll see how these are used shortly.

If the user has specified a non-`NULL` value for the second argument, indicating that he or she wants to change the existing values for the signal, `sigaction()` first does a few sanity checks to make sure that the values passed in the `sigaction` structure are valid. We first check to see if the user is trying to change the action for `SIGKILL`, `SIGSTOP`, or `SIGCANCEL`. These signals cannot be caught. Any attempt to change the handler to anything other than `SIG_DFL` (the default handler) will cause `sigaction()` to return `EINVAL`. Next, we validate the flags to make sure that there are no unsupported bits set in the flags field. If any bits other than those defined in `signal.h` are set, `sigaction()` will again return `EINVAL`. Finally, if the user is trying to change the behavior of the `SIGGFAULT` signal, he or she must also have the `_SA_SIGGFAULT` flag set in the flags field. `SIGGFAULT` is used by the graphics libraries and should normally not be changed by a user application. This flag is a way of saying, "Yes, I really do want to change `SIGGFAULT`."

Once we've passed the sanity checks, the next thing `sigaction()` does is call `setsigvec_setup()`. This routine is responsible for allocating any needed memory and for acquiring any locks needed to ensure that the operation will be atomic and can't be interrupted. Specifically, if the `SA_SIGINFO` flag is set, then the kernel must allocate memory in which to store signal information to pass to the signal handler. This is done first, before any locks are acquired, because it has to `MALLOC` memory, and `MALLOC` might sleep if memory is not available. Next, if the signal being modified is `SIGCLD`, we obtain three locks: the `MTPROC_REAP_LOCK`, the systemwide `sched_lock` spinlock, and the `exit_lock` spinlock for the current process. This is because `SIGCLD` changes modify the behavior taken when a child exits. The `MTPROC_REAP_LOCK` makes sure that the process isn't currently being reaped; the `sched_lock` makes sure the scheduler can't modify it; the `exit_lock` makes sure that it's not currently trying to exit. Finally, we obtain the spinlock that controls the process structure: `p->p_lock`. Once all this is done, we're back to `sigaction()`.

The actual setting and retrieving of the signal handler values is really pretty straightforward. If the user has passed a non-`NULL` pointer for the third argument to `sigaction`, indicating that he or she wants to get the current values, we just extract the handler and flag values from the `proc` structure (and its associated `Process_Shared_Fields` structure) and pass them back to the user. `Sigaction` then calls `sigsetvec` to set the new values, which again is pretty straightforward. The handler is copied to `p->p_shared->p_signal[sig-1]`. The mask is copied into `p->p_shared->p_sighandmask[sig-1]`. We then go through the flags value and copy each of the flags into its correct place—for most of the flags, this is a bit in `p->p_shared`.

There are a few special cases in `setsigvec()`. The `SA_NOCLDSTOP` and `SA_NOCLDWAIT` flags are looked at only if the signal number is `SIGCLD`. If the request is turning off the `SA_SIGINFO` flag, we go through each non-zombie child of the current process and delete the structure that is holding the `siginfo` data. If the action is being set to `SIG_IGN`, indicating that the user wants to ignore this signal, `setsigvec()` goes through all threads and removes any pending or blocked signals for this signal number.

There are also bitfields in the `proc` structure that get updated at this point. `p->p_sigignore` is a `ksigset_t` that says which signals are ignored, and `p->p_sigcatch` is a `ksigset_t` that says which signals are being caught. Note that these are actually in the `proc` structure itself, not in the `Process_Shared_Fields`.

Finally, after all this is done, `sigaction()` calls `setsigvec_cleanup()` to release any locks acquired in `setsigvec_setup()`.

We now have our signal handler set up—we saved off what signal the user is interested in, what he or she wants to do with that signal, and a set of flags and a mask that modify what happens when the signal arrives. We look in more detail at how these flags and masks are handled when we look at signal handling. But first, let's look at how signals get delivered.

Signal Delivery

Signal delivery became much more complicated with the advent of multithreaded processes. The kernel has to support the old paradigm of delivering a signal to a process as well as to deliver the signal to a particular thread within the process. Signal delivery is further complicated because the interface to the `kill()` system call is complicated—a pretty good feat for a system call with only two arguments.

The `kill()` system call is the way a process sends a signal. The first argument is the PID of the process to which the signal should be sent, and the second argument is the number of the signal to send. The complication comes from the PID argument, which can take on values other than an actual PID. If the specified PID is zero, the signal is sent to all processes in the same process group as the process sending the signal. If PID is –1 and the user is not root, the signal is sent to all processes belonging to the user. If the PID is –1 and the user *is* root, the signal is sent to all processes other than system processes. If the PID is the constant `KILL_ALL_OTHERS` (defined in `/usr/include/sys/signal.h`), the behavior is similar to the case where PID is –1 except that the signal is not sent to the sending process. And finally, if PID is negative but not –1 or `KILL_ALL_OTHERS`, the signal is sent to all processes whose process group is the absolute value of PID. Table 14-1 summarizes the various actions of `kill()`.

Table 14-1. Actions of the `kill()` System Call

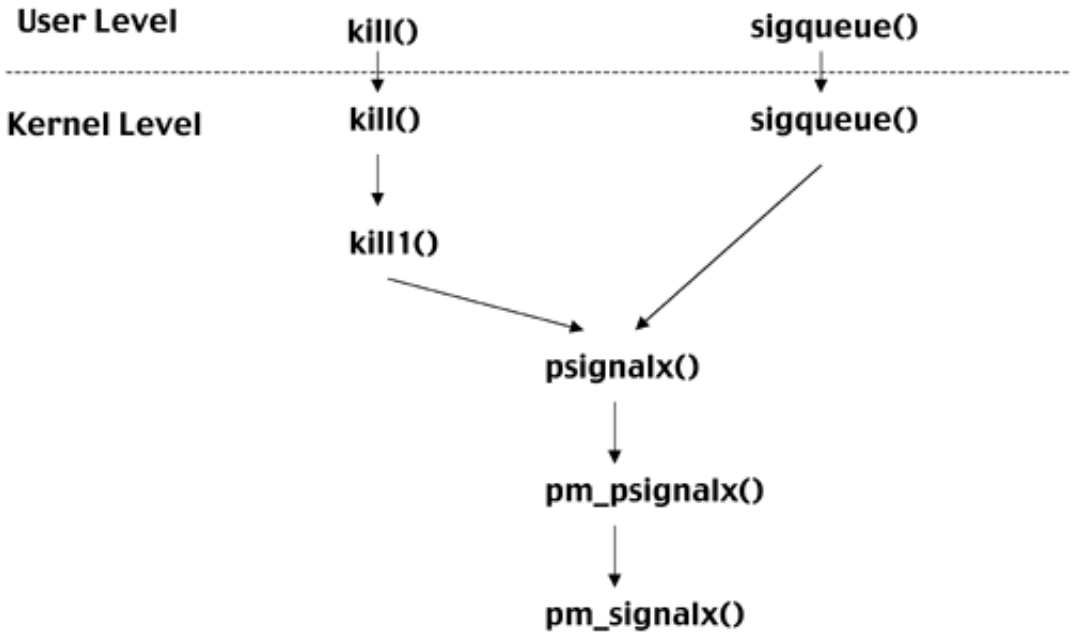
PID	Effective UID	Signal sent to
>0	Any	Specific process
0	Any	All processes in current process group
–1	Root	All processes
–1	Nonroot	All processes with same UID
KILL_ALL_OTHERS	Root	All processes except sender
KILL_ALL_OTHERS	Nonroot	All processes with same UID except for the sending process
<0	Any	Process group `|PID|`

The first thing the `kill()` kernel function has to do is validate the arguments and sort out where the signal should be delivered. `kill()` then calls `kill1()` and passes it the signal number, the process group or process to signal, a flag indicating whether the signal is going to a group or a single process, and a flag indicating whether or not this is a "kill all" signal.

The job of `kill1()` is to interpret the arguments from `kill()` and then call `psignalx()` for each process that should be sent the signal. Depending on the arguments, this could mean going through the process group hash to find all processes in a group, or it could mean looping through all processes looking for the correct ones.

There is also a second system call, which provides a means of initiating a signal: `sigqueue()`. `sigqueue()` does not have the broadcasting functionality that `kill()` has—the PID passed to `sigqueue()` must be a valid PID, and only that process receives the signal. However, `sigqueue()` does have the ability to send a value along with the signal. This value is of type `union sigval`. After validating arguments, `sigqueue()` creates a `sigcount` structure for the process if one doesn't already exist. It then goes to the list of free `ksi` structures to find an available one and copies the `sigval` into it. Finally, it calls `psignalx()` to deliver the signal. The `psignalx()` routine is where the logic for `kill()` and `sigqueue()` converge, as show in Figure 14-2.

Figure 14-2. The `kill()` and `sigqueue()` System Calls



`psignalx`() is a relatively simple front end to `pm_psignalx()`. The first thing `psignalx()` does is check to see if the pointer to the `ksi_t` that was passed to it is on the stack or in `malloc`ed memory. Recall that we maintain for each process a list of free `ksi_t` structures that we use to hold the `siginfo` data. It is also possible for a function to just declare a type `ksi_t` as a local variable and use that. If `psignalx()` finds that the `ksi_t` pointer it has received points to the stack, it copies that data into a `ksi_t` from the free list. `psignalx()` then grabs the `sched_lock` and calls `pm_psignalx()`, passing a pointer to the `proc` structure, the signal number, and the (possibly new) pointer to a `ksi_t`.

`pm_psignalx()` is still not the one that does the real work—it's another front end. It locks the process lock for the process, checks to make sure the process isn't a zombie, then calls `pm_signalx()`.

At last, in `pm_signalx()` we're ready to actually post the signal. Signals can be posted either to a process or to a particular thread within the process. Recall that the `proc` structure has `p_sig` for pending signals, and each thread structure has `kt_sig` for pending signals. One of the arguments to `pm_signalx()` is a pointer to a `kthread` structure. If this pointer is `NULL`, then the signal gets posted to the process…sort of. We'll see some exceptions to that rule soon.

The first thing `pm_signalx()` does is look at the process to see if there are any threads that have called `sigwait()` to wait for the signal. If it finds one, then it just does a wakeup for that thread and it is done. For a thread in `sigwait()`, there is no other action taken on the signal other than waking up the sleeping thread.

If no thread is in `sigwait()`, then we look for threads in `sigpause()`, `pause()`, or `sigsuspend()` that are waiting for this signal. If a thread is found here, then that thread becomes the target of the signal. In other words, even if a signal is sent to a process, if there is a thread waiting for it, that thread is the one that gets it.

Next, we check to see if the process is multithreaded. If there is only a single thread in the process, that thread becomes the target of the signal. If there are multiple threads, we check each to see if it has registered a handler for the signal. If so, that thread gets the signal. If we have multiple threads, no thread is waiting for the signal, and no thread has a handler for the signal, *then* we deliver the signal to the process itself. Finally, we're ready to actually raise the signal by setting the appropriate bit either in `p_sig` or `kt_sig`.

The next thing we have to do is decide whether or not to wake up the thread we just set the bit for. This depends on a combination of what state the thread is in and what signal we're sending it. It's just a straightforward case statement to get it to behave as described in the man pages.

Finally, there's just a little cleanup to do. If the thread was a `sigwaiter` and we woke it up, then we remove it from the `sigwaiters` list. If we had `siginfo` along with the signal, here is where it gets enqueued onto

the receiving thread's list. And if the signal we're delivering is `SIGKILL`, we mark the process as terminating so it won't get back into user space to run anymore.

So now the signal has been delivered. The receiving thread or process has its bit set to let it know the signal has arrived, and it has been woken up if appropriate. Let's now have a look at what the receiving process does with the signal.

PREV < Day Day Up > NEXT

Signal Recognition

There are three places a thread checks to see if it has been signaled: at the end of every system call, when it wakes up from a sleep, and after handling a trap of some kind. The routine that checks for signals is `issig()`. The kernel routines `real_sleep()`, `syscall()`, and `trap()` each call `issig()` to see if any signals are pending for the thread.

The job of `issig()` is to determine what signal, if any, should be processed. `issig()` loops through the bits in `p_sig` and `kt_sig` looking for pending signals. If it finds no signals pending, it returns zero. If it finds a signal that is pending, it returns the number of the signal. The signal number is also placed in the thread structure, in `kt_cursig`.

Although this sounds like a relatively simple process, `issig()` does a few other things that add to its complexity. Most of the default actions for signals are handled by `issig()`. If the action for the signal is set to `SIG_DFL`, `issig()` looks at the signal number. Default actions are handled by `issig()` for `SIGTSTP`, `SIGTTIN`, `SIGTTOUT`, `SIGSTOP`, `SIGCONT`, `SIGIO`, `SIGCANCEL`, `SIGPWR`, `SIGWINCH`, `SIGCLD`, `SIGURG`, and `SIGDIL`.

Signals with an action of `SIG_HOLD` or `SIG_IGN` are handled by `issig()` too. If the action is `SIG_IGN`, the bit is reset and any `siginfo` that goes with it is thrown away. For `SIG_HOLD`, no action is taken—the bit stays set, the `siginfo` stays, but the signal is not acted on.

Once `issig()` has found a signal that needs to be handled, it has one more step to do before returning. If the signal was directed at the process, then any `siginfo` data that goes with it is also attached to the process. `issig()` is working at the thread level, so when it returns a signal number, it is indicating that the current thread is to receive the signal. Just before it returns, `issig()`moves the `siginfo` data from the process to the thread so that the routines that actually handle the signal will be able to find it.

The second step in signal recognition is handled by the routine `psig()`. `psig()` looks at the value it `kt_cursig` to find the current signal number, then uses that to look up the signal action in the `p_signal` array in the `proc` structure. If the action is `SIG_DFL`, `psig()` takes the default action—for those signals whose default actions weren't handled by `issig()`. These include `SIGILL`, `SIGIOT`, `SIGBUS`, `SIGQUIT`, `SIGTRAP`, `SIGEMT`, `SIGFPE`, `SIGSEGV`, `SIGSYS`, `SIGXCPU`, `SIGXFSZ`, and `SIGGFAULT`. In all these cases, a core file is generated for the process and the process is terminated.

Ignored and held signals don't make it to `psig()`, and those with default actions have now been handled either by `issig()` or `psig()`. What we're left with is a signal that has a user handler installed. `psig()` then calls the function `sendsig()` to prepare for calling that function.

`psig()` is also the route into the signal handler for synchronous signals such as `SIGSEGV` or `SIGILL`. These signals are caused by a thread trying to do an invalid operation. The invalid operation gets caught by the trap handler, which then calls `psig()` directly. In this way, these signals get handled immediately, not queued up to be found later. This detection happens in the `trap()` function in the kernel—but this isn't the same as the detection of asynchronous signals in `trap()` that we mentioned above. In other words, `trap()` can generate a signal in two ways: first, for a trap of any kind, `trap()` checks to see if an asynchronous signal has been queued and needs to be handled; second, `trap()` checks to see if the exception that caused it to be called should immediately generate a synchronous signal.

Signal Handler Launch

The function in the kernel that arranges for the signal handler to be called is `sendsig()`. Its job is to set up the stack frames for the call to the user function. It also has to make sure that when the routine finishes, it returns to the right place to clean up the signal data structures. It calls a variety of utility routines to do the work it has to do.

The first is `sendsig_alloc_sigcontext()`, which allocates space on the stack to hold the `sigcontext` structure. This space be can either on the user's stack or on a separate signal stack depending on the flags for this signal. Data is copied into this sigcontext structure by `sendsig_fillin_slm()`, `sendsig_copyoutsavestate()`, and `sendsig_fillin_xsi()`.

The stack frame used for calling the user's function is set up such that the return address points to the `sigcleanup()` routine. More specifically, since the signal handler is going to be in user space and `sigcleanup()` is a kernel routine, the return address points to a stub in `libc` that handles the callback into the kernel for cleanup.

We also have to manipulate things to actually get into the signal handler. Remember that we're in the kernel code, and we got here by means of a trap, a sleep, a wakeup, or a system call. Those are the things that cause signal detection. We have a save state, which points back to the user code that sent us into the kernel. (For details on this, see Chapter 4.) We have to modify that save state so that when we leave kernel mode, we go instead to the user's handler. `sendsig_setuphandler()` does this by modifying the value of GR 31—the return pointer—in the save state.

`sendsig()` has sanity checks throughout this process to make sure the user's stack hasn't been corrupted. If any utility routine fails because it's unable to manipulate the user's stack, the process gets immediately terminated. This is done by one of my favorite routines—`sendsig_the_barbarian()`. Even kernel engineers have a sense of humor.

Signal Handler Return

When the user's signal handler returns, it branches back to the return pointer it was given by `sendsig()`. This vectors into a routine in `libc`, which then goes to a system call to the `sigcleanup()` kernel routine. `sigcleanup()` has a pretty easy job—it essentially restores the save state that was saved by `sendsig()` so that when it returns, it will go back to the place the signal occurred. There are some sanity checks to make sure the user hasn't modified the stack in any way he or she shouldn't have, and once again `sendsig_the_barbarian()` is there to cut down any process with an invalid stack.

Summary

The generation and handling of signals is a complicated process, involving several transitions between user and kernel space. The kernel has to deal with signals that can be caught, ignored, or held. It also has to determine which thread should receive a signal and manage information passed from the sender to the handler.

Chapter 15. System Initialization

The first step in the system boot process is controlled by processor-dependent code (PDC). On a multiprocessor system, PDC must first identify the controlling native processor, or *monarch*. The monarch is the first processor initialized, and it handles all of the early boot process. Each processor on the bus has a value called `BOOT_ID`, which is set at the factory to 2. Monarch selection begins by looking for the processor with the highest `BOOT_ID`. If there are multiple processors with the same `BOOT_ID`, the one with the lowest slot number becomes the monarch. `BOOT_ID`s are rarely changed, so the monarch is almost always the processor in the lowest slot. The monarch processor then continues with the boot process while all other processors idle, waiting for an interrupt.

PDC next configures the initial memory module (IMM). This is the memory module that is mapped to physical address zero. The IMM selection is similar to the monarch selection process, and the IMM is the largest memory module. If there are multiple modules with the same size, the IMM is the one with the lower slot number. On most newer systems, the entire memory space appears to the system as a single module, so this selection process usually selects that one memory module. The module's soft physical address (SPA) is set to 0, mapping it to physical address zero.

Now that the system has a monarch processor identified and memory mapped, the next step is to identify the boot and console devices. This is done by the boot console handler (BCH). BCH looks in the system's stable store to find the hardware path to the system console. Once the console path is initialized, it displays some system status information and then proceeds based on the settings of two flags in the stable store: `Autoboot` and `Autosearch`. If the `Autoboot` flag is set, BCH attempts to boot the system without interaction with the user. Without the `Autoboot` flag, BCH asks the user to verify the boot path. `Autosearch` allows BCH to search all I/O modules for bootable devices. Without `Autosearch`, BCH attempts to boot only from the primary path stored in stable storage. Table 15-1 summarizes the effects of `Autoboot` and `Autosearch`.

Table 15-1. Autoboot and Autosearch

Autoboot	Autosearch	Nickname	
OFF	OFF	Manual boot	BCH interacts with the user to get the boot path.
OFF	ON	Boot search	BCH finds all bootable paths and presents the user with a list. The user must choose a boot path from the list.
ON	OFF	Auto boot	BCH attempts to boot from the primary path. If the boot fails, BCH prompts the user.
ON	ON	Auto search	BCH first tries the primary path. If that fails, BCH searches for bootable devices and tries to boot each one in turn.

PDC expects a bootable device to begin with a logical interchange format (LIF) volume. More information about the format of a LIF volume can be found on the lif(4) man page. PDC looks for a LIF header at the beginning of the bootable device. Within this header is a value called `IPL_ADDR`, which is the offset of the initial program loader (IPL). Also in the header are `IPL_SIZE` and `IPL_ENTRY`, which specify the size of the IPL program and the address of the entry point to the program. PDC loads the IPL program into memory, then branches to the start address.

The PA-RISC platform can run a variety of operating systems, including HP-UX, MPE, and Linux. The IPL program for each operating system is different. For HP-UX the IPL program is called ISL, for initial system loader.

ISL: The Initial System Loader

ISL also uses the `Autoboot` flag and the LIF volume. In an interactive boot, ISL displays a prompt and waits for a command. From ISL, the user can perform a variety of low-level maintenance functions, including updating the stable storage flags, changing boot and console paths, and examining the LIF volume contents. In an automatic boot, ISL looks in the LIF volume for a file called `AUTO`, reads a line from that file, and uses that as a boot command.

Normally, the `AUTO` file contains a single line that just reads `hpux`. `hpux` is the name in the LIF directory of the program that handles the next step in the boot process: the `HPUXBOOT` secondary loader. The header of `HPUXBOOT` contains its load address and entry point address. ISL loads `HPUXBOOT` into memory at its load address—`0xd00000`—then branches to its entry point address.

HPUXBOOT: The Secondary Loader

The secondary boot loader is called HPUXBOOT, but is also sometimes called Mongoose, the name it used while under development. More information on HPUXBOOT can be found in the hpux(1m) man page. Its job is to locate the kernel image and copy it into memory, then branch to it.

The first thing it does is relocate itself from its fixed starting point of 0xd00000 to near the end of the IMM. The 0xd00000 address was picked to guarantee that it will fit into a 16-MB memory module. But when the kernel gets loaded into memory, it gets loaded before HPUXBOOT, so HPUXBOOT needs to move itself out of the way. It calculates its new address so that it will end at the end of IMM minus 2 MB.

HPUXBOOT must now locate the HP-UX kernel and load it into memory. The command-line arguments to HPUXBOOT—either from the AUTO file or from a manual boot—can specify the location of the kernel. In most cases, the AUTO file does not specify arguments to HPUXBOOT, in which case HPUXBOOT looks for the kernel on the device from which it was booted.

HPUXBOOT is able to read the standard HFS file system. It looks for the kernel in two different locations: /stand/vmunix and /vmunix. These names are relative to the mount point of the file system, so the /vmunix case is for configurations, where /stand is a separate file system from /. If the system does not have a separate /stand file system, then the kernel is found in /stand/vmunix in the root file system. If /stand is a separate file system, then the kernel is found in /vmunix under that file system.

Once HPUXBOOT has located the kernel file, it reads the headers of that file to find the load point in memory and the entry point. The load point for the kernel is always address 0x20000. HPUXBOOT then branches to the entry point for the kernel.

We've now made it to the point where we're actually running in the kernel code, but we have a long way to go. Here is our state when we start the kernel:

- Only the monarch processor is running. All the other processors are waiting for an interrupt.
- The processor is in narrow mode, and virtual address translation is turned off.
- The space registers and most of the control registers are zero or uninitialized.
- General registers arg0, arg1, and arg2 were set by HPUXBOOT. Other than that, all general registers are uninitialized.
- The cache is either clean or invalid.
- All TLB entries are invalid.
- The only memory that is usable is the IMM.

We do have information in memory from PDC that we can use—for example, the location of the IODC routines and the size of memory. Also, HPUXBOOT is still in memory and we have pointers to its I/O routines to help out with doing disk I/O.

Real-Mode Initialization

The entry point for the kernel is an assembly language routine called `rdb_bootstrap()`. RDB refers to the remote debugger used internally by Hewlett-Packard. This is the first point in the kernel at which the debugger can start managing the kernel. `rdb_bootstrap()` turns off interrupts and initializes the interrupt vector table so that we'll be ready to handle any interrupts that come in. It also creates the first `procinfo` structure, `mpproc_info[0]`, and fills it with information about the monarch. It sets CR24 to point to this `procinfo` structure. If the kernel is compiled for 64-bit operation and the hardware is 64-bit-capable, `rdb_bootstrap()` turns on the `W`-bit in the Processor Status Word to enable wide mode.

`rdb_bootstrap()` then has to get ready to make a call into C code. It sets up the stack pointer (GR30) and global data pointer (GR27), creates a calling stack from, then calls `realmain()`.

`realmain()` handles the portion of kernel initialization that must be done in real mode before we turn on virtual address translation. `Realmain()` does its work by using a `calllist`—an array of pointers to functions. The only line in `realmain()` is a call to `DoCalllist()`, passing `realmain_calllist` as the first argument. `DoCalllist()` iterates over the functions in the `realmain_calllist` array, calling each one in turn. The `calllist` itself is constructed when the kernel is compiled, using an English-like file that describes the order of the initialization routines. This file contains the name of each routine to be called along with constraints on the order of execution—which other routines it must run before or after. A program parses this file at compile time and creates the `calllist`.

The `realmain calllist` contains close to 100 routines. The following is a summary of the general functions performed by the `realmain calllist`.

1. Set the transfer of control (TOC) vector in page zero of memory.
2. Copy the arguments passed by `HPUXBOOT` from `HPUXBOOT`'s address space into the kernel's.
3. Create a temporary resource map for the IMM, indicating that memory currently used by the kernel image is busy and the rest is free. This will be used to allocate memory before the virtual memory allocator has been set up.
4. Allocate and initialize systemwide spinlocks.
5. Allocate buffers for IODC to use to dump memory in the event of a panic.
6. Initialize the monarch by setting up its HPMC handler, filling in information about the processor type and enabling its coprocessors.
7. Purge the TLB, then initialize TLB entries for kernel and graphics.
8. Call `pa_memory_init()`, which scans the central bus for memory modules not yet initialized. Any memory modules found are initialized and added to the native module tree.
9. Create another resource map `real_free_map` to be used to track available pages now that we have initialized the rest of memory. The resource map we had been using is copied into this new one.
10. Create the 32-bit and 64-bit resource maps `sysmap_32bit` and `sysmap_64bit`.
11. Using the routines provided by `HPUXBOOT`, read the LVM configuration from `/stand/rootconf` and the stored I/O configuration from `/stand/ioconfig`.
12. Perform first-level I/O configuration. This calls `pa_module_init()` to scan the system's busses looking for modules. Each module is added to the `native_mod_tree`. If the module is a processor, it gets a crash table entry; if it is an I/O module, the module is reset and interrupt handlers are set up for the module.
13. The `fixed_mod_table` (from `/stand/ioconfig`) gets merged with the `native_mod_table`, which contains devices we've discovered.
14. The space used by `HPUXBOOT` is now freed—it is no longer needed.
15. Allocate and initialize a number of kernel data structures. These include:
    - The `pgclass` table, used by `crashconfg(1m)` to mark which pages to dump in the event of a panic

- `mpproc_info` structures for non-monarch processors
- `pdir` locks, the page directory, and the `pfn_to_virt` table
- The I/O alias table
- The space map
- The `inode` table
- The file table
- The directory name lookup cache (DNLC)
- The `proc` and `kthread` tables
- Physio's private buffer pool
- Processor save state areas
- Callouts
- Resource allocation maps for quadrants 2, 3, and 4

16. Handcraft `proc[0]`, the swapper. Assign it to `kthread[0]`. Create a virtual address space for it, and manually map it to a real address.
17. Allocate and initialize processor idle stacks and non-monarch interrupt stacks.
18. Map any extra I/O register sets into virtual space
19. Each page we allocated so far while in real mode is inserted into the `pdir` and reserved in `sysmap_32bit`.

Let's have a look at where we are when `realmain()` completes:

- We have discovered all our I/O devices and reset them, but have not yet assigned drivers to them.
- The virtual memory system is initialized and ready to run.
- The monarch is still the only processor running, although we have initialized much of the data for the other processors.
- The file systems are unitialized.
- The `proc` and thread tables are initialized, and the first entry in each is occupied by the swapper process.

When `realmain()` completes, it returns to its caller, `rdb_bootstrap()`. The next thing `rdb_bootstrap()` does is put us into virtual mode by setting the `W`, `D`, `C`, `P`, `Q`, and `I` bits. This gives us wide mode, data translation, code translation, protection identifier validation, interrupt state collection, and external interrupts enabled. The system is now running in virtual mode.

Virtual Mode Initialization

Once `rdb_bootstrap()` has placed the system into virtual mode, it has a couple of more steps to do. It flushes all the caches because the `realmain` routine does some code optimization. This act of having the kernel modify its own code can cause problems with the caches, so we flush them. We also create the kernel stack and switch to using that instead of the interrupt stack. Finally, `rdb_bootstrap` branches to `main()`, the virtual mode main program for the kernel.

`main()` is similar to `realmain()` in that it works from a `calllist`. This one is called `mainstreet_calllist`. One interesting difference in `main` is that it never returns. The thread that calls the `mainstreet_calllist` routines eventually become `proc 0`, the swapper. If `DoCalllist` ever returns in `main()`, we'll panic with the message `main: Road Ends`.

Like `realmain_calllist`, `mainstreet_calllist` has a lot of work to do. There are over 100 entries on the `calllist`. Rather than list them all, we just summarize the more important functions.

The first step is to initialize more systemwide data structures: semaphores, the per-processor run queues, `ttisr` data structures, and `cblock`s. Then, we finish filling out `proc[0]`, `kthread[0]`, and the user area for that thread. All other `proc` table and `kthread` table entries are placed on the `freeproc_list` or `freekthread_list`. We set up the external clock interrupt handler and start the clock ticking.

At this point, we're finally ready to start up the non-monarch processors. They get started up one at a time. For each processor, the monarch does the following:

- Setup the `procinfo` structure and IVA for the particular processor.
- Set the global variable `boot_rendez_state` to `RS_RENDEZ_IN_PROGRESS`.
- Set the `MEM_RENDEZ` pointer in page zero of memory to point to `force_start()`.
- Send a rendezvous interrupt to the processor on EIRR bit 0.
- Monitor the value of `boot_rendez_state` for 10 seconds. If it changes to `RS_RENDEZ_ACK`, then the processor woke up. If it doesn't, then the processor is considered bad.

When a processor gets the rendezvous interrupt, it branches to the interrupt handler at `MEM_RENDEZ`—`force_start()`. This is an assembly language routine that does many of the things that `rdb_bootstrap()` does—the processor goes into wide mode, clears and disables interrupts, and sets up its IVA, stack, and data pointers. It then calls `non_monarch_init()` to do real mode initialization. `non_monarch_init()` sets up the TLB for the processor and fills in the rest of the `procinfo` structure. It then sets `boot_rendez_state` to `RS_RENDEZ_ACK` to indicate to the monarch that it has started.

Once it has done this, the processor begins to spin, watching the global value `mp_sync_after_rendez.` As long is this value is `WAIT_FOR_ALL_PROCESSORS`, it continues spinning. We'll see how that happens shortly.

Back on the monarch, once all the non-monarchs have been started, we continue with system initialization by initializing the scheduler. Next, we perform second-level I/O configuration, as described in detail in Chapter 10, "I/O and Device Management." When this process is complete, we'll have the I/O tree completely filled out with drivers mapped to each device. Now we're ready for more subsystem initialization! We set up buffer hash queues and allocate the buffer cache, initialize System V semaphores and shared memory, and initialize LVM.

Finally, we're ready to start some disk activity—starting with mounting the root file system. We go through every file system that was registered during the second level I/O configuration, looking for one with an entry point for `vfs_mountroot`. When we find one, that entry point is called, causing that file system to be mounted read-only on `/`.

Next, we fork some system daemon processes—`vhand`, `statdaemon`, `unhashdaemon`, and `ttisr`. The process is very similar to the use of the `fork()` system call in a user program. We first have to call `prepare_newproc()`, which initializes a `proc` structure and a `kthread` structure, and gives us back pointers to those new structures. Then we call `newproc()`, passing it the pointers to the new `proc` and `kthread`. Just like `fork()`, `newproc()` returns twice: once in the parent and once in the child. We check for a return value

of FORKRTN_CHILD to indicate that we are the new child process. If we are the child, then we call the appropriate routine—vhand, statdaemon, unhashdaemon, or ttisr_daemon—to perform the daemon function. These calls never return; they run as long as the system is up.

Up to this point, all of the processes we've launched have been kernel processes. They run in kernel address space, use the kernel stack, and have access to all the kernel functions. The next step is to launch the first user space process—the pre_init_rc process. pre_init_rc is a script found in /etc/pre_init_rc and is responsible for performing an fsck on the root file system. We start out as before, calling prepare_newproc() and newproc() to fork a new process and kernel thread. We then handcraft a small data space—three pages—and attach it to the thread. The first page gets a fixed preamble, coded in assembly, which contains some simple support routines. These include a version of strlen(), a version of bcopy(), and an interface to allow system calls. We need this preamble because we don't have direct access to the kernel's system call handlers or to routines like bcopy(), since the process is a user space process. At the same time, we don't have access to libc or any of the other usual C libraries because the kernel doesn't know where those are yet. We've only got the root file system mounted. Immediately following this, the code we want to run gets copied into the page. The second page is reserved for data for the process, and the third is used for stack.

Three pages isn't much space to operate in, but it doesn't have to do much. The code that gets placed into the first page opens stdin, stdout, and stderr, then does an execve() system call. For the pre_init_rc script, we tell execve() to run sh out of /sbin and pass pre_init_rc as argv[1]. The execve() syscall takes care of loading the shell and expanding memory regions as required, and then the shell runs the pre_init_rc script.

After the pre_init_rc script has run, we unmount the root file system, then mount it again, this time writable. Now we're ready to fork the init process.

Init gets forked essentially in the same way as pre_init_rc—by creating a process, creating a three-page user space for it, then using that space to make an execve() system call to get init running.

We're on the home stretch now. Once init is up and running, it starts forking off the user processes. We still need to get the rest of the processors going—we left them spinning, looking at the mp_sync_after_rendez global. The monarch processor now sets that global to CONTINUE_EXECUTION. All of the other processors now continue with their startup, much as the monarch did—they go into virtual mode, start their clocks ticking, and eventually call idle(), where they loop, looking for processes to run.

The last thing that the monarch does in the boot process is call sched(). This causes that thread of execution to become the sched process. Note that this isn't like the other daemons, where we forked first and the child became the daemon. We don't fork for sched, so the thread of execution that has done all the work for us so far becomes sched.

Summary

We've looked at the boot process from the time that the system is powered on to the time it's running and ready to launch user processes. Most of the process is fairly routine—the creation and initialization of systemwide data structures. Some of the steps are described in more detail in other chapters, particularly the I/O configuration process. We've seen too that there are some unusual things that have to happen during boot, such as handcrafting processes and making the leap into user space to launch user-level processes. There's a lot of work that has to get done to get from power off to running, particularly in a multiprocessor system.

Chapter 16. Tools Overview

There are several tools that can be helpful in examining the internals of the kernel and getting a better idea of what is going on. Some can also be used to modify parameters in the in-memory kernel image or in the kernel executable on disk.

When using any tool that can modify data, extreme caution must be exercised. It is possible to cause serious data corruption and render your system unbootable if the tools are misused. Hewlett-Packard support generally does not cover problems caused by the misuse of these tools.

adb

The absolute debugger, adb, is not specifically a kernel tool. It is a general-purpose debugger. It allows the user to specify an executable file and an optional core file, then examine and modify the contents of the core file and control execution of the executable. However, when the executable file is `vmunix` and the core file is an HP-UX core dump or `/dev/mem`, then adb can be used to examine kernel variables and structures.

For the purpose of examining the kernel, adb can be used either on a core dump or on the running kernel. If you wish to work with a core dump, change directory to the location of the dump and use the command `adb -m vmunix` (including the trailing period). The `-m` specifies a multifile dump image, `vmunix` is the name of the executable, and (the current directory) is the location of the dump. If you want to examine the running kernel, use `adb -k /stand/vmunix /dev/mem`. Optionally, you can include the `-w` parameter, indicating you wish to be able to modify the kernel or memory images. This should be used with extreme caution—modifying kernel memory can cause a system panic or worse.

Commands in adb are cryptic even by UNIX standards. Most versions of adb don't even have a prompt, so there's no indication that you're running adb. The newer versions of adb use `adb>` as a prompt so that you'll know where you are. They also remind you upon startup that `$q` is used to quit and `$h` is help.

The built-in help tells you that the command format is `[address expr][,count][command][modifier]`. The address expression is usually either a symbol (such as `main` or `proc`) or an address (such as `0x7f003843`). The most common commands are `/` for examining values from the memory image and `?` for examining values from the kernel executable.

The modifier for these two commands is the format in which the output should be displayed. Table 16-1 lists some of the more commonly used formats supported by adb.

Table 16-1. adb Formats

Format Specifier	Meaning
a	Print the symbolic name for the address.
b	Print one byte in hexadecimal.
B	Print one byte in octal.
d	Print two bytes in decimal.
D	Print four bytes in decimal.
i	Print the mnemonic for the instruction at the given address.
o	Print two bytes in octal.
O	Print four bytes in octal.
s	Print a zero-terminated string.
x	Print two bytes in hexadecimal.
X	Print four bytes in hexadecimal.
Y	Print a date/time value.

The `count` field after the address is the number of times the command should be repeated. You can also put a count on the modifier. Be aware that the count on the address is assumed to be hexadecimal, and the one on the modifier is assumed to be decimal. For example, `realmain,20?i` decodes the first 32 instructions from `realmain`, and `realmain?20i` decodes the first 20 instructions from `realmain`. Because of this ambiguity in the default base for numbers in adb, it's always a good idea to explicitly state the base you're

using by prefacing numbers with `0x` for hex and `0d` for decimal.

adb can be handy for examining kernel tunables. The command `nproc/D` gives you the current value of the `nproc` tunable, `maxdsiz/D` gives you the current `maxdsiz` value, and so on. In a few cases, adb can be used to modify tunables as well. This doesn't work for all tunable values; for example, if a tunable is used at boot time to determine how big to make a system table, then changing it later isn't going to help. In fact, changing it to a bigger value later might cause a system panic if the kernel runs off the end of the table. Also be aware that changing a tunable in the running kernel only changes the kernel image in memory, not the kernel file on disk. The next time you reboot and the kernel is reloaded, you'll be back to the old values.

To be able to modify values, you must have used the `-w` switch when you ran adb. This is an extra safeguard to keep you from accidentally modifying values. You should use `-w` on the command line only if you plan to be changing things. One example of a tunable you can change on the fly is `st_ats_enabled`, which enables the automatic tape-sharing mechanism in the tape driver. This flag is checked on every tape open, so changing it affects all future tape operations. To enable it, type `st_ats_enabled/W 1`, and to disable it, type `st_ats_enabled/W 0`. And, of course, you can check the current state of the flag with `st_ats_enabled/X`.

q4

q4 is a tool that was designed for analyzing kernel core dumps. However, like adb, it can also be used on a running system. You do need to be aware that most kernel data structures are very dynamic, changing constantly. Thus, if you're looking at a live kernel rather than a dump, you need to realize that the data you're looking at is a snapshot.

Like adb, q4 takes a kernel file and a memory image or core dump directory as parameters. If the parameters are omitted, it looks for `vmunix` and a core dump in the current directory. So, if you're looking at a core dump, change to the directory that has the dump in it and just type `q4`. If you want to run q4 on the running kernel, type `q4 /stand/vmunix /dev/mem`.

A very useful option to q4 is `-p`, which tells q4 to start Perl and set up an interface with the Perl process. This allows you, from within q4, to run Perl scripts that interact with q4. Many Perl scripts are packaged with q4 to perform common tasks. You may find that you need to tell q4 where Perl is by setting the `Q4_PERL_PATH`, and q4 also needs a Perl startup script that can be pointed to by the environment variable `Q4_STARTUP_SCRIPT`. If your files are in the usual places for release 11i, the following starts q4 with Perl mode on the running kernel:

```
export Q4_PERL_PATH=/usr/contrib/bin/perl

export Q4_STARTUP_SCRIPT=/usr/contrib./lib/Q4/sample.q4rc.pl

/usr/contrib./bin/q4 -p /stand/vmunix /dev/mem
```

Examining Data and Code

q4 has an `examine` command that is similar to the data display commands in adb. The format is

```
examine <address> [for <count>] using <format>
```

For example, to display the value of `nproc`, type

```
examine &nproc using D
```

Note the ampersand before the symbol `nproc`. q4 assumes that the symbol you are giving it is a pointer. This is true for most kernel values, since most kernel space is allocated dynamically. But for something like `nproc`, which is a static global value, you need to use the address-of operator, `&`.

If you want to look at assembly code for a function, you use the `code` command. This is simply `code <function>`. For example, to get the assembly code for `realmain`, you would type

```
code realmain
```

q4 doesn't do the disassembly itself—it relies on adb to do that. But it does figure out the length of the function for you, and the syntax is easier to remember than the adb command.

Getting Data Type Information

q4's strength lies in its awareness of data structures. Unlike adb, which requires working with raw data in memory, q4 can display complex kernel data structures in a format that is easy to read.

We can look at what is in a kernel data structure with the *`fields`* command. This command takes a struct, union, or type as an argument and displays the members of the structure. For example, to see what the fields are in a spinlock structure, we would type

```
q4> fields struct lock
OFFSET        SIZE       FLAVOR  NAME
 bytes +bits bytes +bits
     0     0     8     0 u_long  sl_lock
     8     0     8     0 u_long  sl_owner
    16     0     8     0 *       sl_name_ptr
    24     0     2     0 u_short sl_flag
    26     0     2     0 u_short sl_next_cpu
    28     0     4     0 char[4] sl_pad
```

q4 tells us the offset of each member in the structure, the member's size, its type, and its name. Notice that q4 knows about pointers such as `sl_name_ptr`, but it doesn't know what type they point to.

When you specify a type in q4, you must include the `struct` or `union` keywords if the type is a structure or a union. In the above example we used `struct lock` to indicate that it was a structure. If the type has a `typedef`, you can use that instead, so we could also have said `fields lock_t`.

Loading a Structure into a Pile

The basic unit that q4 works with is called a *pile*. A pile is simply an array of structures. You create a pile with q4's *`load`* command. The `load` command needs to know what kind of structure you want to examine, how many of them to load, and how to find them. The simplest form of the load command is *`load <type> from <address>`*. The type can be any publicly visible kernel structure or `typedef`. If you're using a structure, you must include the `struct` keyword. The address can either be a numeric value or a symbol. For example, to load the first `mpinfo` struct from the `mpproc_info` array, we would type

```
q4> load struct mpinfo from mpproc_info
```

Because `struct mpinfo` is also `typedef`ed to `mpinfo_t`, we could also say

```
q4> load mpinfo_t from mpproc_info
```

Displaying the Contents of a Pile

Once you've got a pile with a structure in it, you need to be able to look at the data. To print that pile, we use the *`print`* command. The `print` command with no arguments prints out the entire structure. If you're

going to print a lot of elements, you probably want to add the `-t` flag to the `print` command. This option tells the command to print one element per line with the names on the left rather than print the data in columns with the names across the top. The `mpinfo` structure is very large—it has over 2,000 elements in it —so even with the `-t` option, you'll get over 2,000 lines of output. Fortunately, you don't have to see it all.

You can pipe q4 commands through shell commands. So, after you've got your `mpinfo` pile loaded, you might want to do `print -tx | more` to see it a page at a time. You could find out how many items there are in an `mpinfo` structure by doing `print -tx | wc -l`. You can also tell q4 exactly which items to print. If you type `print -tx procindex prochpa`, you'll get something like the following:

```
procindex  0

  prochpa  0xfffffffffed21000
```

Using Skip to Load Structures

So far, we've loaded up the first `mpinfo` structure into a pile. But `mpinfo` is an array—what if you want to look at the `mpinfo` structures for other processors? The *`skip`* keyword tells q4 how many array elements to skip over. If you want to look at the second entry (the entry for processor 1), for example, type

```
q4> load struct mpinfo from mpproc_info skip 1
```

Now, if we look at `procindex` and `prochpa`, we get

```
procindex  0x1

  prochpa  0xfffffffffed61000
```

Loading Multiple Structures

Now we know how to load up a structure and examine its contents. But a pile can be more than one structure—you can load up a whole array of structures. We do this with the *`max`* keyword to the `load` command, which tells q4 the maximum number of structures to load; the default is one. Continuing with our `mpinfo` example, we can load the `mpinfo` structures for all of the processors by taking advantage of the global symbol `nmpinfo`, which tells us how many there are:

```
q4> load struct mpinfo from mpproc_info max nmpinfo

loaded 2 struct mpinfos as an array (stopped by max count)
```

q4 tells us how many were loaded and what made it stop.

Loading Linked Lists

The linked list is such a common structure in the kernel that q4 has knowledge of linked lists built in. To load a list of structures into a pile, you have to tell q4 which element in the structure points to the next structure in the list. You do this with the *`next`* keyword.

For an example, let's look at the process table. Prior to HP-UX release 11.11, the process table was a statically allocated table of entries. Beginning with HP-UX 11.11, the process table is a linked list. The head of the list is pointed to by the symbol `proc_list`, and each entry in the list has a member called `p_factp`, which points to the next entry. To load up all of the current process table entries, we type

```
q4> load struct proc from proc_list next p_factp max 10000

loaded 155 struct procs as a linked list (stopped by null pointer)
```

q4 again tells us how many structures were loaded and why it stopped—in this case because it encountered a null pointer. q4 detects the end of a linked list either when it finds a null pointer or when it finds a pointer that points back to the first entry in the list.

Notice the large value for `max`. In this case, we wanted all of the `proc` structures, so we picked a `max` value large enough that it wouldn't be exceeded. Without specifying max at all, we would get only one structure loaded.

Using Values from a Pile

In addition to using global symbols, we can also specify the name of a member in the current pile as an address. Since there are so many cases where one data structure points to another, it makes sense to be able to use the member names rather than have to type in addresses. For example, in the `proc` structure there is a member named `p_firstthreadp` that points to the first thread in that process. Once you've got a `proc` structure on the pile, you can load the first thread for that process by typing

```
q4> load struct kthread from p_firstthreadp
```

Manipulating Piles

Each time you execute a `load` command, a new pile is created. Previous piles are pushed onto a stack, so that at any time you will have access to all piles you've created. The *`history`* command lists the piles in the stack. For example,

```
q4> history

HIST NAME   LAYOUT COUNT TYPE          COMMENTS

   1 <none>  array     1 struct mpinfo stopped by max count

   2 <none>  array     2 struct mpinfo stopped by max count

   3 <none>  array     2 struct mpinfo stopped by max count

   4 <none>   list   155 struct proc   stopped by null pointer
```

The highest numbered pile is the current pile. You can recall a previous pile with the *`recall`* command. The argument to the `recall` command is a pile number from the `HIST` column in the history display. A copy of the specified pile is pushed onto the stack and becomes the new current pile. Given the history in the previous example, we can recall the first `mpinfo` pile:

```
q4> recall 1

copied 1 item
```

```
q4> history
HIST NAME   LAYOUT COUNT TYPE          COMMENTS
   1 <none>  array     1 struct mpinfo stopped by max count
   2 <none>  array     2 struct mpinfo stopped by max count
   3 <none>  array     2 struct mpinfo stopped by max count
   4 <none>   list   155 struct proc   stopped by null pointer
   5 <none>  array     1 struct mpinfo copy of 1
```

A pile can also be given a descriptive name that will be displayed in the history. The name is applied to the current pile with the *name* command. The syntax is just `name it <name>`. Continuing the above example, if we want to assign the name `proc1` to the pile we just recalled, we type

```
q4> name it proc1
so named
q4> history
HIST NAME   LAYOUT COUNT TYPE          COMMENTS
   1 <none>  array     1 struct mpinfo stopped by max count
   2 <none>  array     2 struct mpinfo stopped by max count
   3 <none>  array     2 struct mpinfo stopped by max count
   4 <none>   list   155 struct proc   stopped by null pointer
   5 proc1   array     1 struct mpinfo copy of 1
```

Now we can use `proc1` instead of the pile number if we want to recall the pile. This can be really handy if you have a lot of piles that are all the same data type—it can be hard to tell them apart.

Another handy pair of commands is *keep* and *discard*. For both of these commands, the argument is an expression. A new pile is created that contains either only those structures for which the expression is true (keep) or only those for which the expression is false (discard). As an example, let's say we want to look at the process table entry for PID 1. We can recall the list of all processes we have in our stack, then keep just the one with `p_pid` equal to one:

```
q4> recall 4
copied 155 items
q4> keep p_pid == 1
kept 1 of 155 struct proc's, discarded 154
q4> history
HIST NAME   LAYOUT COUNT TYPE          COMMENTS
   1 <none>  array     1 struct mpinfo stopped by max count
   2 <none>  array     2 struct mpinfo stopped by max count
   3 <none>  array     2 struct mpinfo stopped by max count
   4 <none>   list   155 struct proc   stopped by null pointer
```

```
  5 proc1   array     1 struct mpinfo copy of 1
  6 <none>   list   155 struct proc   copy of 4
  7 <none> mixed?     1 struct proc   subset of 6
```

We now have a pile that contains the process table for process one.

An Example: Getting the `uarea` of a Process

Let's look at an example that goes through several of these steps. In this example, we get the `uarea` of a process, PID 2003. Because of the way structures are interconnected, there are many ways to do this, and some of them are quicker and easier than this example, but the idea here is to show a variety of q4 commands. Following are the steps we go through:

- Load all processes.
- Keep process 2003.
- Get the `VAS` structure for the process from the `p_vas` pointer.
- Using the `VAS` region list, load all regions for the process.
- Keep the region that has a type of `PT_UAREA`.
- Print the space and offset of that region.
- Load a `struct user` from that space and offset.

```
q4> load struct proc from proc_list next p_factp max 10000
loaded 150 struct procs as a linked list (stopped by null
pointer)
q4> keep p_pid == 2003
kept 1 of 150 struct proc's, discarded 149
q4> load vas_t from p_vas
loaded 1 vas_t as an array (stopped by max count)
q4> load preg_t from va_ll.lle_prev next p_ll.lle_prev max 100
loaded 21 preg_ts as a linked list (stopped by loop)
q4> keep p_type == PT_UAREA
kept 1 of 21 preg_t's, discarded 20
q4> print -x p_space p_vaddr
  p_space            p_vaddr
0x6546c00 0x400003ffffff0000
q4> load struct user from 0x6546c00.0x400003ffffff0000
loaded 1 struct user as an array (stopped by max count)
```

Tracing Stacks

To get a stack trace, you use the *trace* command. q4 can trace four kinds of things: processes, kthreads, processors, and crash events. Note that trace will not show you function calls in user processes—this is the kernel stack.

If you are looking at a crash dump rather than the running kernel, you can get a stack trace of crash events using the *trace event* command. Given a crash event number, this shows you the stack trace from the time the process entered kernel space to the time the system panicked. Crash event 0 is generally the most useful, although there are times when you need to see other crash events as well. For example, here is the stack trace from a data page fault panic:

```
q4> trace event 0

stack trace for event 0

crash event was a panic

panic+0x14

report_trap_or_int_and_panic+0x84

trap+0xd9c

nokgdb+0x8

getnewbuf+0x1cc

ogetblk+0x110

getblk1+0x260

vx_getblk+0x50

vx_write_default+0x564

vx_write1+0x4d8

vx_rdwr+0x164

vno_rw+0x84

write+0x104

syscall+0x28c

$syscallrtn+0x0
```

Here we see that the process entered the kernel by doing a write() system call. The kernel progressed through vno_rw, vx_rdwr, and so on until it was in getnewbuf. At 0x1cc bytes past the beginning of getnewbuf, it executed an instruction that caused a trap—in this case a Data Page Fault. The trap handler, trap, decided it couldn't handle the trap and called report_trap_or_int_and_panic to cause the system to panic.

The *trace processor* command takes a single argument, the number of the processor you wish traced. q4 prints a stack trace for whatever was running on that processor.

You can use *trace process at* or *trace thread at* to get a kernel stack trace for a particular process or thread. The value following at should be the address of a struct proc or a struct kthread. Tracing a process will print a stack trace of all the threads for that process.

You can also use *trace pile* to get a trace of whatever is in the current pile. The structures in the pile must be of type struct proc, struct kthread, struct mpinfo, or struct crash_event_table_struct. Here's an example of using trace pile to get the stack trace for a particular process:

```
q4> load struct proc from proc_list next p_factp max 10000
```

```
loaded 150 struct procs as a linked list (stopped by null

pointer)

q4> keep p_pid == 2003

kept 1 of 150 struct proc's, discarded 149

q4> trace pile

stack trace for process at 0x0'42846040 (pid 2003), thread at

0x0'42847040 (tid 2104)

process was not running on any processor

_swtch+0xc4

_sleep+0x318

fifo_rdwr+0x2f8

vno_rw+0x1ac

read+0x10c

syscall+0x204

$syscallrtn+0x0
```

This shows us that this particular process has done a `read()` system call on a `fifo`. The `fifo` is empty, and the process has been put to sleep and will be woken up when data is available to read.

Using Perl Scripts in q4

q4 ships with a directory full of Perl scripts that can automate some common operations. Because these were designed for Hewlett-Packard internal use, they are not well documented and some have very limited, specific uses.

The Perl files are usually located in `/usr/contrib/lib/Q4`. Each script is a file with a `.pl` suffix. To use these files, you first have to use the *`include`* command to have them read into q4. For example, to include the `whathappend.pl` script, type

```
q4> include whathappened.pl
```

Once the file is included, you run it with the *`run`* command:

```
q4> run WhatHappened
```

The argument to the `run` command is the name of a subprogram within the Perl file. In most cases, this is the same as the file but in mixed case, as in the example for `WhatHappened`. In other cases, you may have to look at the Perl file to find out how it works. Table 16-2 lists some of the more useful Perl scripts.

Table 16-2. q4 Perl Scripts

File Name	Command	Purpose
bucketwalk.pl	BucketWalk	Checks the memory buckets in the bucket allocator for corruption.
callout.pl	callout	Reports information related to callouts.
ioscan.pl	Ioscan	Re-creates ioscan-like information from a dump.
lvm.pl	LVM	Prints summary of LVM configuration.
netinfo.pl	Netinfo	Simulates lanscan and netstat.
openfiles.pl	OpenFiles	Displays all files open by all processes.
sleep_queue.pl	(various)	Various commands to check sleep queues.
whathappened.pl	WhatHappened	General information used in analyzing dumps.
wsioscsi.pl	WsioScsi	Information about the state of the WSIO SCSI subsystem.

Summary

Both adb and q4 give you the ability to examine kernel data structures either on a running system or in a crash dump. adb is better for simple tasks and allows modification of kernel values. q4 allows you to get an in-depth view of the kernel but is a view-only tool.

PREV < Day Day Up >

[SYMBOL] [A] [**B**] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [R] [S] [T] [U] [V] [W]

[SYMBOL] [A] [B] [C] [**D**] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [R] [S] [T] [U] [V] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [R] [S] [T] [U] [V] [W]

General I/O System [See GIO]
GIO 2nd 3rd
Global Virtual Addresses
GSC

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [**M**] [N] [P] [Q] [R] [S] [T] [U] [V] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [**N**] [P] [Q] [R] [S] [T] [U] [V] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [**Q**] [R] [S] [T] [U] [V] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [**R**] [S] [T] [U] [V] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [R] [S] [T] [**U**] [V] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [R] [S] [T] [U] [**V**] [W]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [P] [Q] [R] [S] [T] [U] [V] [**W**]

WSIO CDIO