HP Computer/Instrument Systems
Training Course

OSF/1 Internals

Student Workbook

HEWLETT
PACKARD

H2596-90001
Printed in U.S.A. 05/91

Notice

Product Development
Application Support Division
100 Mayfield Avenue
Mountain View, CA 94043 U.S.A.

Contents

Overview

Course Description

OSF/1 Internals is a four-day course designed to introduce the fundamentals of the OSF/1 operating system. The course does not contain source-code level material. It offers students a deep technical introduction to the OSF/1 kernel.

Prerequisites

The OSF/1 Internals course is not a beginning operating systems course. It assumes familiarity with operational principals of the following:

- virtual memory
- one or more of the following file systems:
  - UFS
  - NFS
  - System V
- UNIX execution environment

Familiarity with the operational principals of the following is recommended:

- sockets
- streams
- system-level C programming

Audience

The intended course audience is made up of system programmers, system support personnel, application engineers, system administrators, and customer support staff. The course assumes that the student is familiar with UNIX and C programming at the system call level. No knowledge of Mach is assumed.

Overview

Course Goals

After completing this course the student should be able to demonstrate an understanding of OSF/1 Internals by describing:

- how OSF/1 enhances traditional UNIX
- how Mach is utilized in OSF/1
- how OSF/1 exploits parallel architectures
- the security features of OSF/1

Exercises

At the end of each module are two sets of exercises. The first set tests the student's performance with respect to each of the major objectives. The second set tests for a deeper understanding of the material: these exercises may require the student to synthesize the knowledge gained from the course, and, in some cases, require that the student delve into other materials. These questions should be considered optional. They may be used by the student as a means for studying the material at a level deeper than is presented in this course.

The answers to the exercises are given in the appendix at the end of the book. The answers to the first set of exercises consists merely of a reference to the topics in the book where the answer can be found. The answers to the second set of exercises is sketched out in the appendix; they are not fully developed.

Agenda

The following schedule will vary depending on the number of questions raised or the level of interest shown by the students.

Day 1:	Module 1 through Module 3
Day 2:	Module 4 through Module 5
Day 3:	Module 6 though Module 8
Day 4:	Module 9 through Module 11

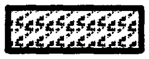
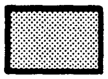
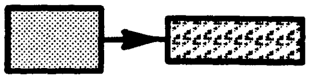
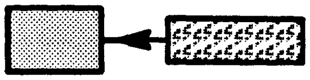
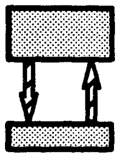
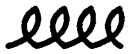
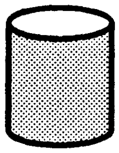
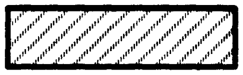
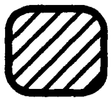
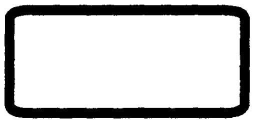
Overview

Recommended Readings

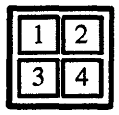
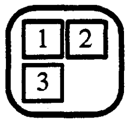
A discussion of most of the topics covered in this course can be found in Open Software Foundation, 1990a (this and other bibliographic citations appear in the bibiography). Two recommended books on UNIX are Bach, 1986 and Leffler, 1989 (the latter covers Berkeley UNIX and is thus the most relevant). A description of the programmer's interface to OSF/1 can be found in Open Software Foundation 1989.

Overview

Slide Conventions

	port
	task
	task has send rights
	task has receive rights
	a task with an unspecified number of additional *vm_map_entry*s.
	thread
	disk
	buffer
	pmap
	encloses an indirect reference to a routine
helvetica font	system call

Overview

⬆	indicates the flow of control
	boxes with square corners contain all of the incore pages
	boxes with rounded corners contain all of the pages assocated with the *vm_object*
🐞	bug

Overview

Module 1 — Introduction

Module Contents

Module Objectives

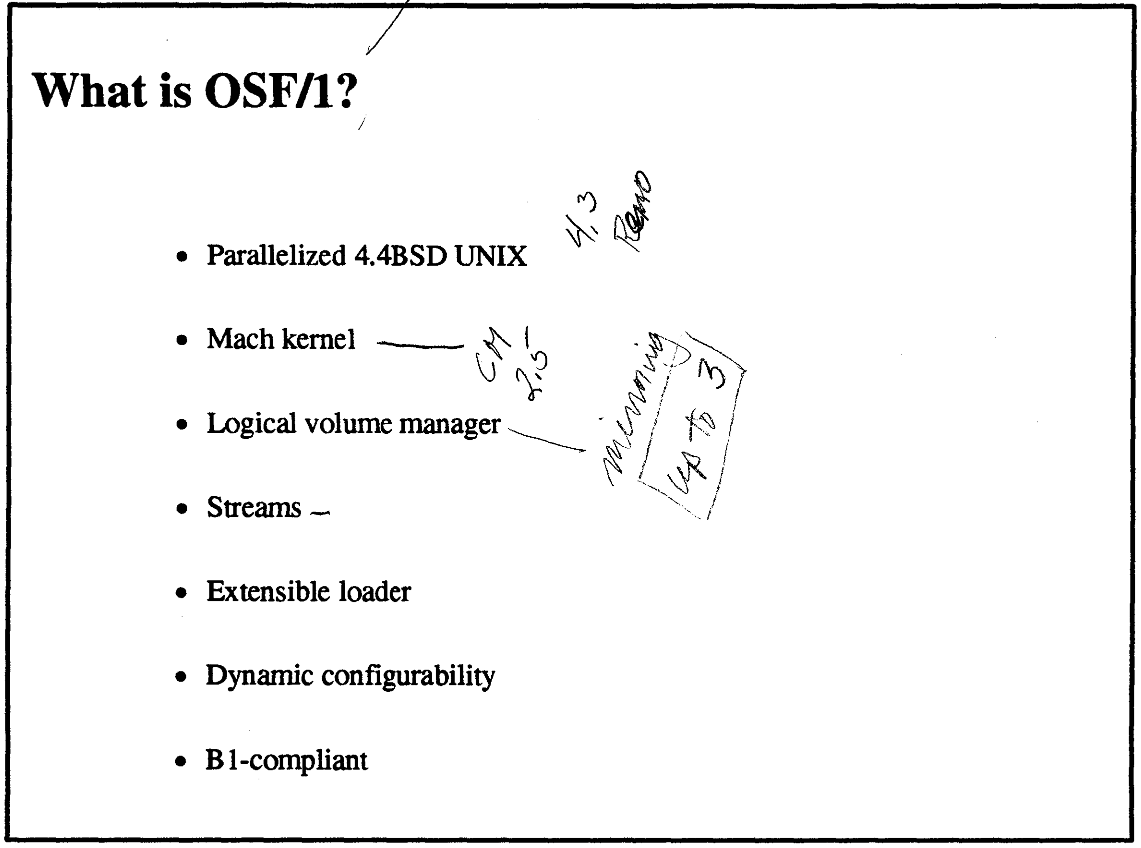
In order to demonstrate an awareness of the components of OSF/1, including its Mach functionality, the student should be able to:

- list the components of OSF/1
- describe the functionality that OSF/1 supplies that is supplied neither by traditional UNIX nor Mach
- explain how Mach and UNIX coexist within the OSF/1 kernel
- list the five fundamental abstractions of Mach and briefly describe each
- give an example of how the OSF/1 kernel can be easily extended to provide functionality not found in the traditional UNIX kernel

Module 1 — Background and Introduction

OSF/1 Ver 1.1

1-1. What is OSF/1?



What is OSF/1?

- Parallelized 4.4BSD UNIX 4.3 Reno
- Mach kernel CM 2.5
- Logical volume manager memory up to 3
- Streams
- Extensible loader
- Dynamic configurability
- B1-compliant

H2596 1-1. 1 © 1991 Hewlett-Packard

What is OSF/1?

OSF/1's UNIX technology is derived from the latest version of Berkeley UNIX—4.3renoBSD (the test version of 4.4BSD). This code has been modified by Encore so that it can efficiently exploit multiprocessors: all user and kernel processing (with minor exceptions) can take place in parallel on multiple processors. OSF/1 supports all the major UNIX file systems: the S5 file system (derived from that of AT&T), the UFS file system (derived from 4.3renoBSD), and the NFS file system (derived from a totally new implementation done at the University of Guelph). The latter two file systems have been parallelized.

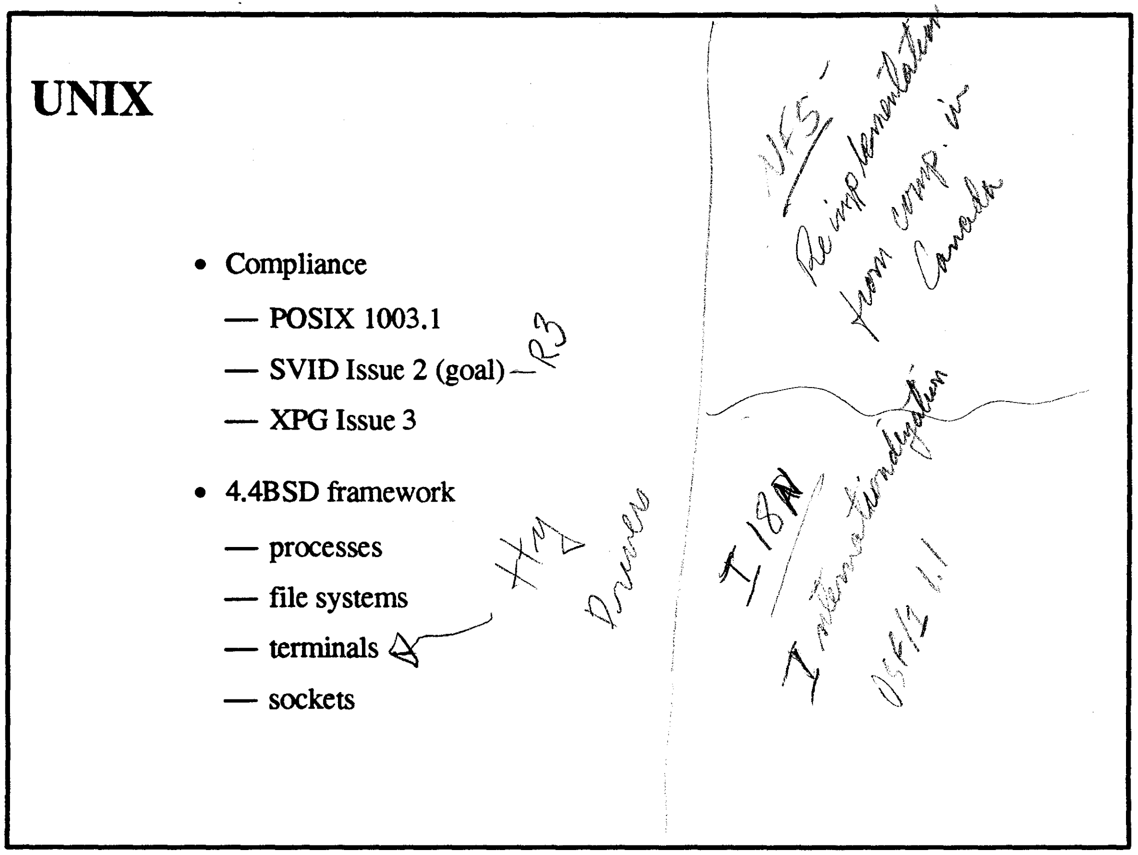
Mach is intended to be a foundation for further operating-system development. It is a simple, extensible kernel that can be used to construct the sort of functionality expected of normal operating systems, such as processes, file systems, etc. Unlike many operating systems, Mach was designed from the ground up for parallel and distributed environments.

Mach is relatively easy to port to many different architectures (there are Mach implementations on most of today's major architectures).

OSF/1's *logical volume manager* (derived from IBM's AIX operating system) allows file systems to span volumes, thus eliminating a major restriction on their use, as well as providing additional reliability through disk mirroring when desired. The *streams* implementation (derived from technology supplied by the Mentat Corporation) is compatible with that of SVR3, but is transparently parallelized: existing streams code can be made to run in parallel without modification. The *extensible loader* allows multiple load formats, shared libraries, and run-time loading, as well as other useful capabilities. The loader lets the user load modules into the kernel dynamically. Thus device drivers, streams modules, file systems, and protocols can be added to a running system.

OSF/1 can be compiled to be either C2- or B1-compliant, depending on the user's security requirements (this technology is derived from that supplied by SecureWare).

1–2. What is OSF/1?



UNIX

- Compliance
  - — POSIX 1003.1
  - — SVID Issue 2 (goal) — R3
  - — XPG Issue 3
- 4.4BSD framework
  - — processes
  - — file systems
  - — terminals
  - — sockets

X11 Drivers

NFS / Reimplementation from comp. in Canada

I18N / Internationalization OSF/1.1

H2596 1-2. 2 © 1991 Hewlett-Packard

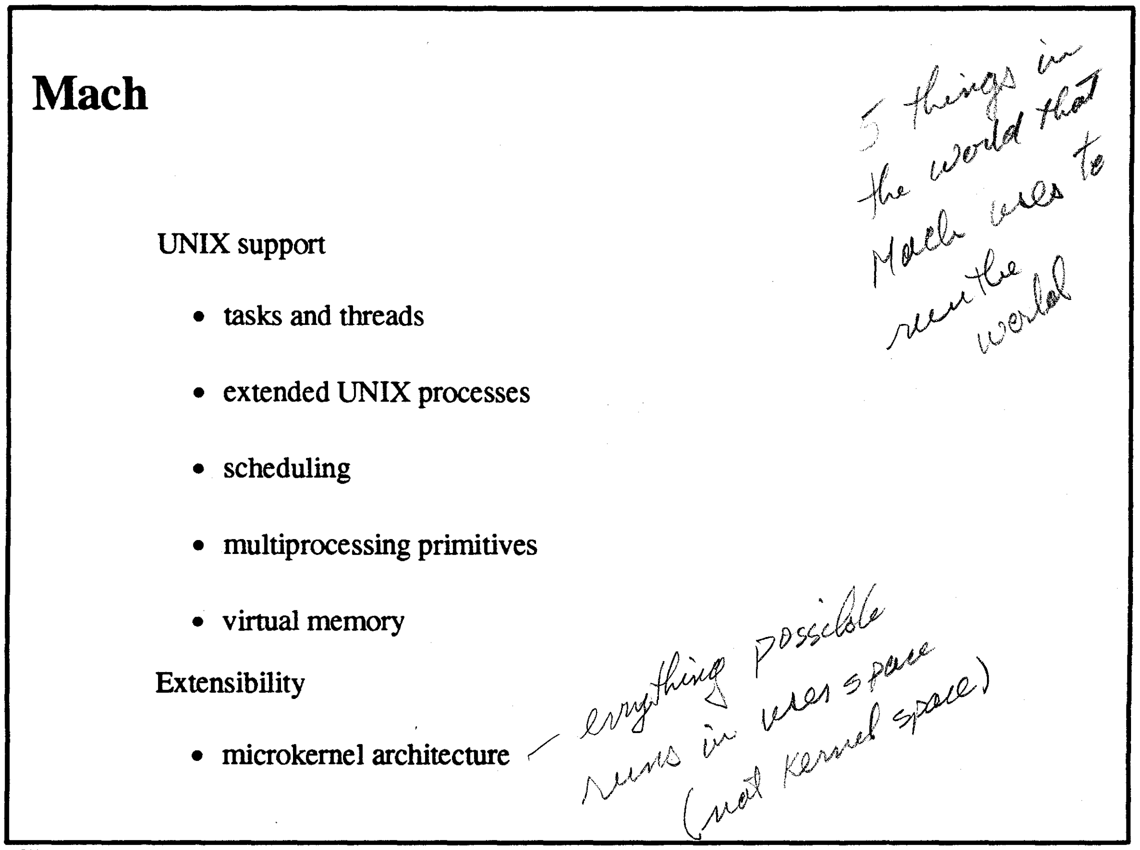
UNIX

The UNIX portion of OSF/1 includes both traditional UNIX functionality and new functionality implemented within the UNIX framework. OSF/1 is fully compliant with all of the standards given on the slide; the final arbiter in the face of conflicting specifications is the AES. Compliances are described in detail in the Open Software Foundation, 1989.

The base technology for UNIX is 4.4BSD. OSF changed the code in a number of places, primarily for integration with Mach and for parallelization. A very good description of Berkeley UNIX can be found in Leffler, 1989. Required System-5 functionality that is not in BSD has been added. In particular, OSF/1 includes an SVR3-compatible streams package, which allows transparent parallelization of streams modules.

1–3. What is OSF/1?

Job Control ?



Mach

5 things in the world that Mach uses to run the world

UNIX support

- tasks and threads
- extended UNIX processes
- scheduling
- multiprocessing primitives
- virtual memory

Extensibility

- microkernel architecture – everything possible runs in user space (not kernel space)

H2596 1-3. 3 © 1991 Hewlett-Packard

Mach

The primary function of Mach in OSF/1 is to support UNIX. UNIX processes are built from the Mach notions of tasks and threads. Unlike traditional single-threaded UNIX processes, OSF/1 processes can be *multithreaded.* Mach is responsible for scheduling the various threads. OSF/1 allows users to select from a number (currently two) of scheduling policies and, on multiprocessors, it allows user control of processor allocation.

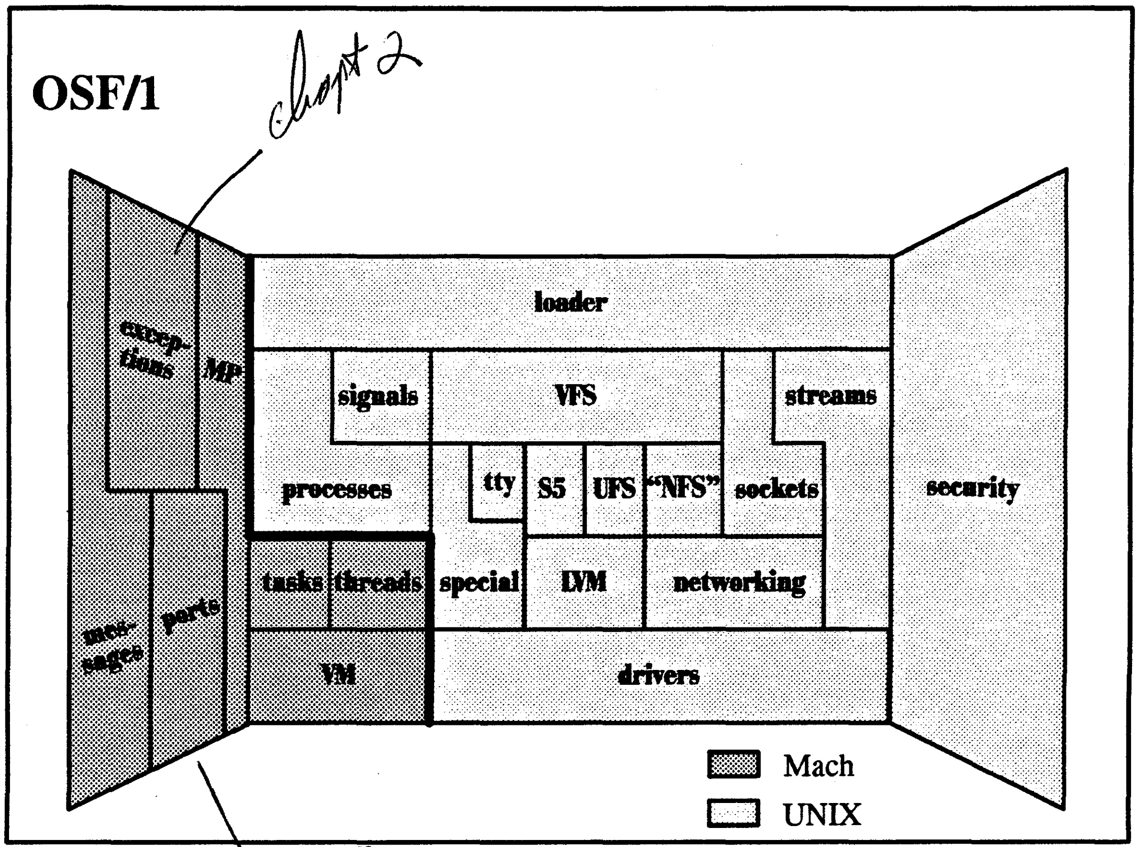
Traditional UNIX event-oriented synchronization has been extended and made safer in Mach; Mach supplies varieties of interprocessor locks to support multiprocessor synchronization.

Mach's virtual memory system completely replaces that of UNIX. It provides efficient and portable support for all of UNIX's VM needs as well as extensibility for future requirements.

A key point to remember is that Mach fosters continued improvements. In particular, as part of the *microkernel project*, all non-Mach portions of the system will be moved from the kernel to various user-level tasks, to produce a very simple, pure Mach kernel in which user tasks provide many of the operating-system functions.

An example of the easy extensibility obtained with Mach is the network memory server (discussed soon), which provides the abstraction of shared memory among threads running on different processes.

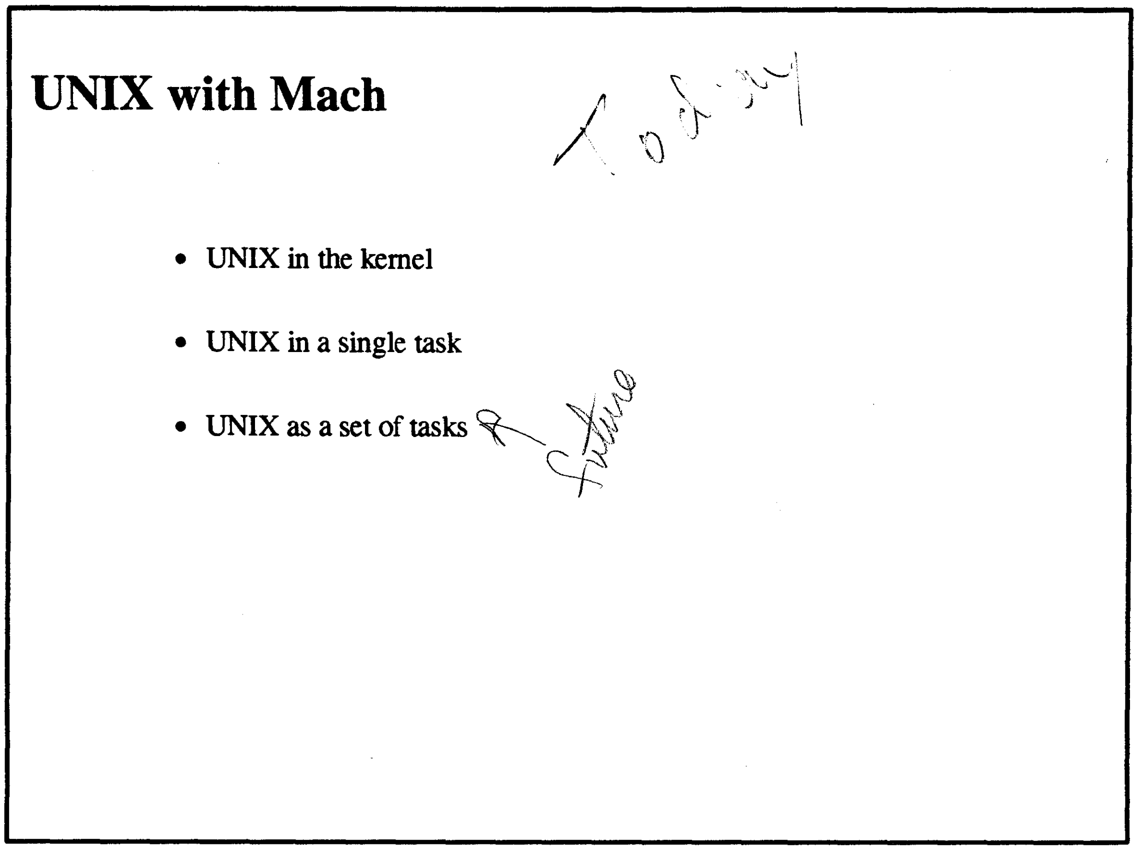
1–4. Organization of the OSF/1 Kernel



.H2596 1-4. 4 © 1991 Hewlett-Packard

OSF/1

1–5. Organization of the OSF/1 Kernel



UNIX with Mach

Today

- UNIX in the kernel
- UNIX in a single task
- UNIX as a set of tasks ← Future

H2596 1-5. 5 © 1991 Hewlett-Packard

UNIX with Mach

The OSF/1 implementation of UNIX coexists with Mach in the kernel, allowing the Mach-based technology to provide both performance and functional enhancements to standard UNIX technology. Unlike earlier versions of UNIX with Mach, the OSF/1 version is, except for a few infrequently executed subsystems, fully parallelized.

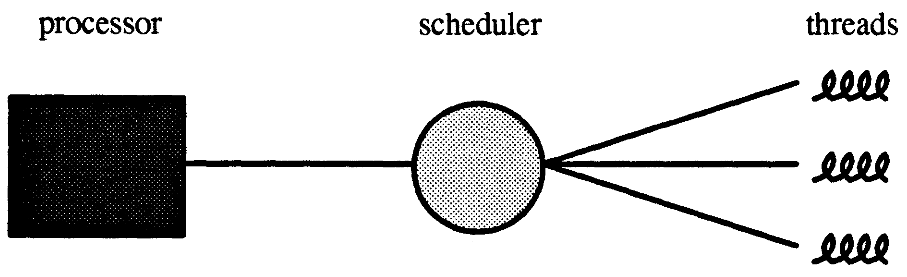
Current research at CMU, OSF, and elsewhere is directed towards providing UNIX functionality efficiently with user-level tasks supported by a "pure" Mach kernel. The first approach provided the UNIX functionality in a single task. This arrangement provided a pageable, interruptible, multithreaded UNIX, but it lacked much support for extensibility. Work at CMU and OSF is proceeding on a more extensible approach, the *microkernel architecture*, in which a set of tasks provides UNIX functionality. By breaking up UNIX along functional boundaries, various components can be replaced or UNIX components can be used to build other systems.

If this approach is successful, then not only UNIX but also other operating system interfaces can be implemented on top of the microkernel. By breaking up UNIX along functional boundaries, certain UNIX modules can be reused to implement other interfaces, and the UNIX interface can be improved or modified by substituting for certain modules.

For further discussion see Golub, 1990.

1–6. Threads and Parallelism

Concurrency



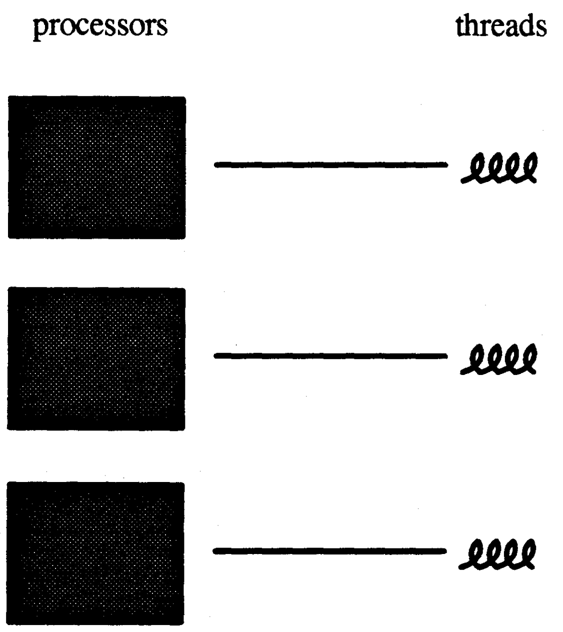
H2596 1-6. 6

Concurrency

Concurrency means that multiple threads are in progress at one time; on a single processor, their execution might be multiplexed.

1–7. Threads and Parallelism

Parallelism

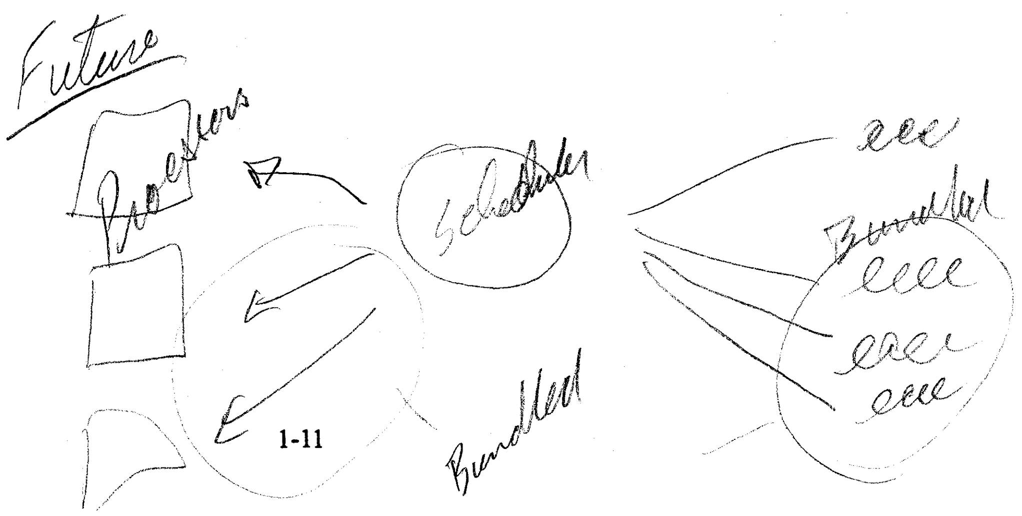


.H2596 1-7. 7 © 1991 Hewlett-Packard

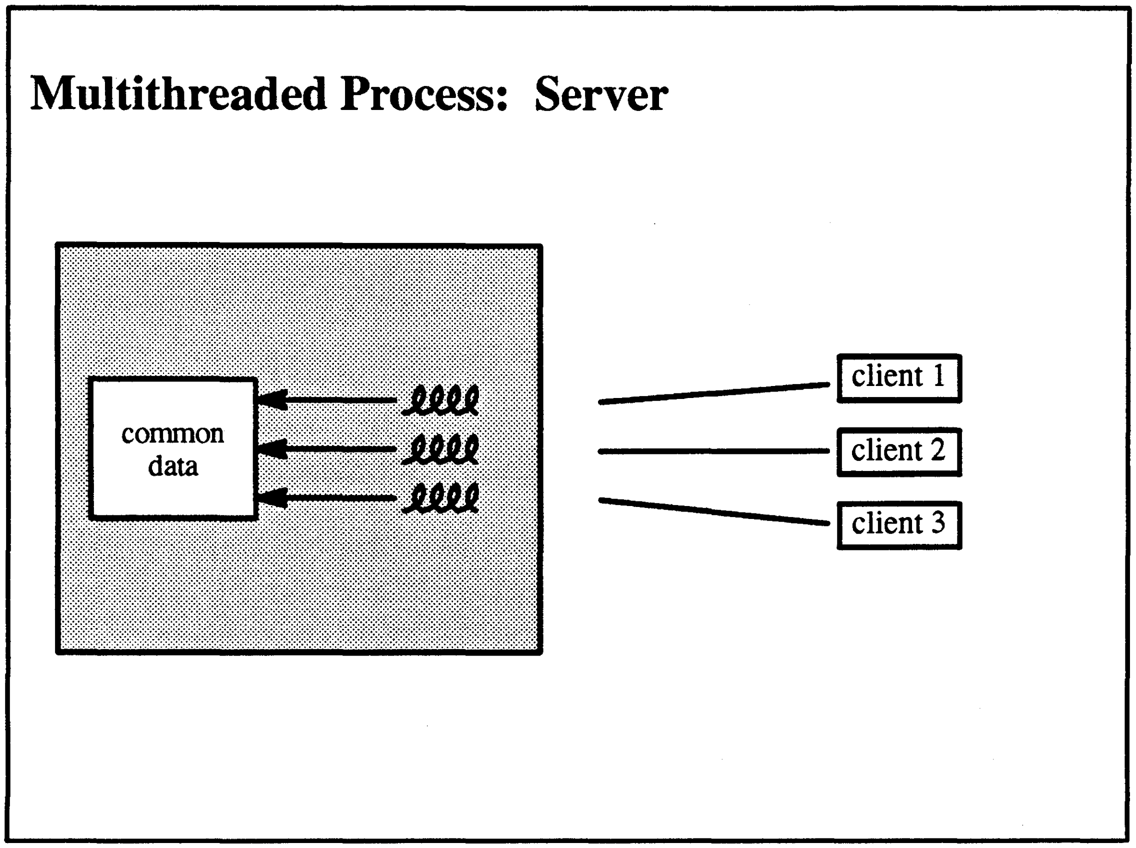
Parallelism

Parallelism means that multiple threads are executing simultaneously: parallelism requires multiple processors. The architecture assumed in OSF/1 is a shared-memory processor, i.e., all processors have equal access to memory.

would be possible



1–8. Threads and Parallelism



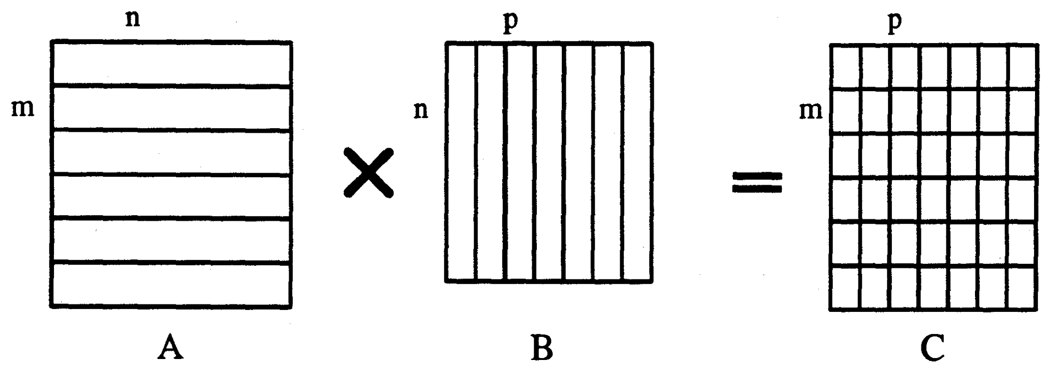
Multithreaded Process: Server

Multithreaded Process: Server

A typical example of the use of multithreaded processes in a uniprocessor environment is a server that deals with multiple clients concurrently. Rather than having to multiplex the clients explicitly, it can make use of the kernel's multiplexing of multiple threads.

1–9. Threads and Parallelism

Multithreaded Processes: Exploiting a Shared-Memory Multiprocessor



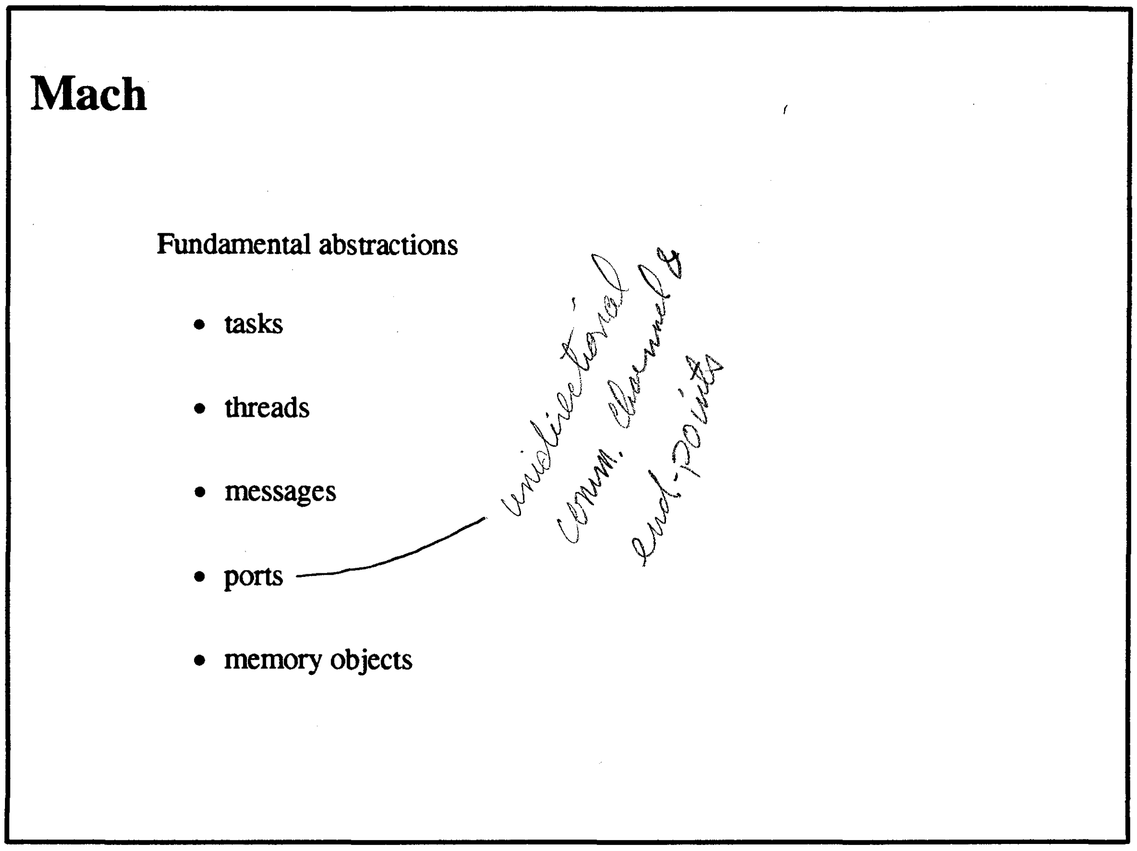
- $m \times p$ inner products to be computed
- t processors available

H2596 1-9. 9 © 1991 Hewlett-Packard

Multithreaded Processes: Exploiting a Shared-Memory Multiprocessor

An example of the use of a shared-memory multiprocessor is the computation of the product of two matrices. With a simple algorithm, this would involve computing a number of inner products. One can utilize all processors of shared-memory multiprocessors by creating a multithreaded process with one thread per processor. If $m \times p$ inner products need to be computed and we have t processors, then each thread would compute $(m \times p) / t$ inner products.

1–10. Introduction to Mach



Mach

Fundamental abstractions

- tasks
- threads
- messages
- ports — unidirectional comm. channels & end-points
- memory objects

H2596 1-10. 10 © 1991 Hewlett-Packard

Mach

A *task* is a holder of *capabilities,* such as address space and communication channels. These capabilities are represented as *ports,* and the kernel itself is viewed as a task.

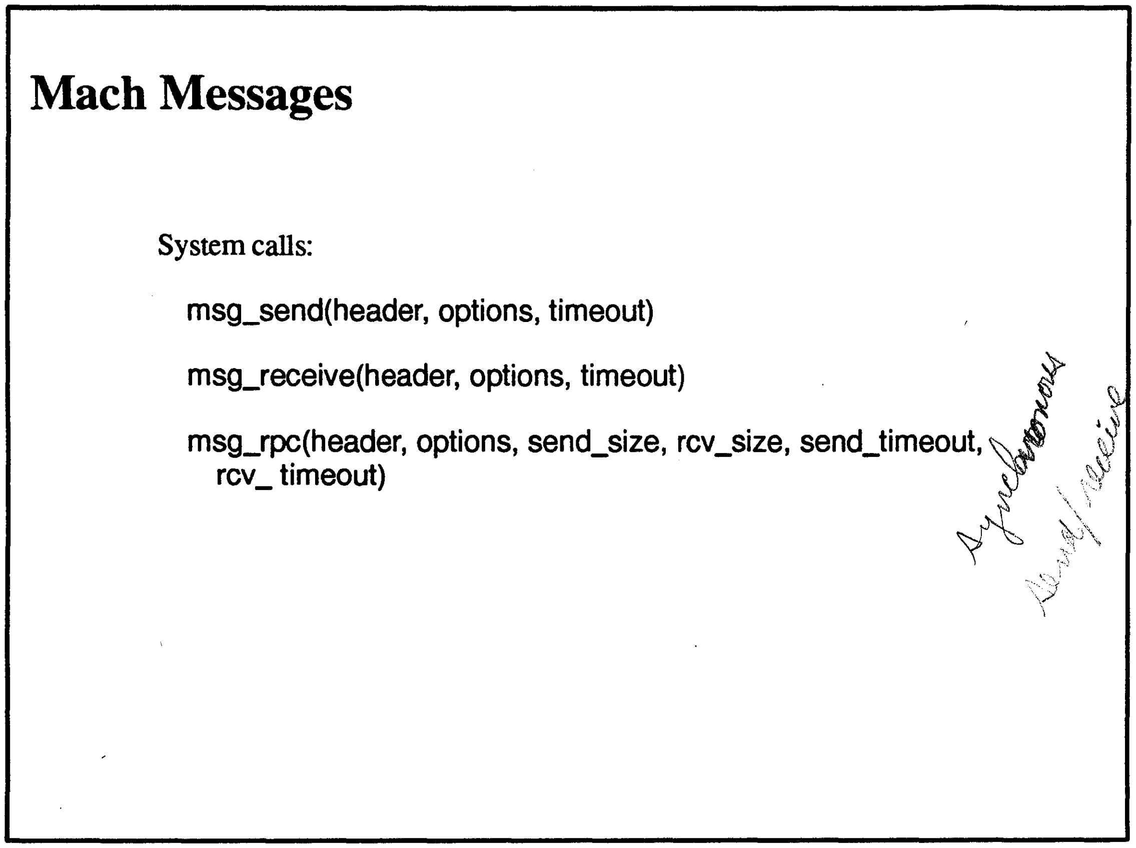
A *thread* is the usual notion of a *thread of control.* The equivalent of a UNIX process is one task containing a single thread. In Mach (and in OSF/1), however, a task may have multiple threads. Tasks may have disjoint address spaces or they may share memory with each other.

Threads can communicate by exchanging *messages.* (Any two threads can communicate this way, although it is more efficient for threads in the same task to communicate using *shared memory.*)

Ports have two purposes: they represent *communication channels* and they are *object references.* Unlike sockets in BSD, which are the endpoints of a communication channel, a port is the entire channel. An object holding a reference to the output end of a port is *securely named* by references to the input side.

Memory objects are "things" that can be mapped into a task's address space. These things might be temporary storage (e.g., UNIX's BSS and stack), files, or objects defined by user-provided servers.

1–11. Introduction to Mach



Mach Messages

System calls:

msg_send(header, options, timeout)

msg_receive(header, options, timeout)

msg_rpc(header, options, send_size, rcv_size, send_timeout,
  rcv_ timeout)

Synchronous send/receive

H2596 1-11. 11 © 1991 Hewlett-Packard

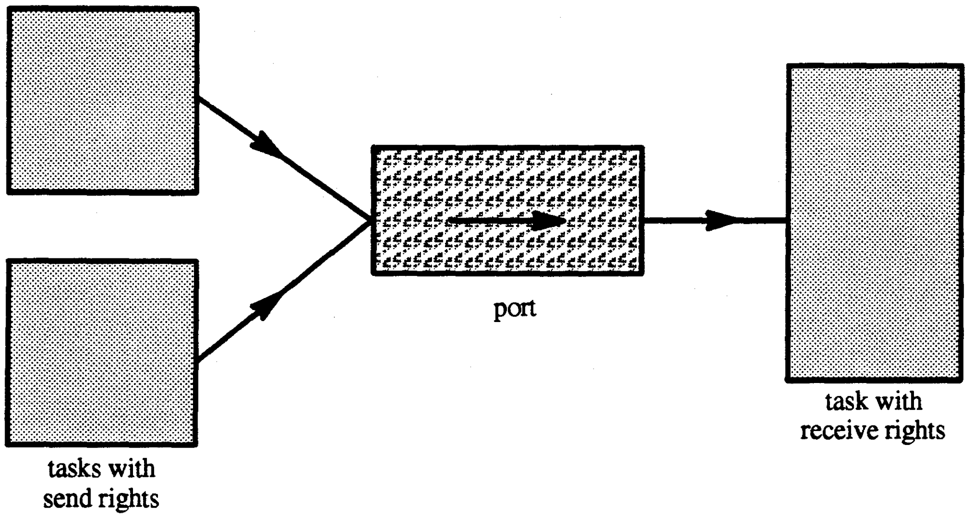
Mach Messages

A message is represented by a header that names the port, gives the type of the message (e.g. integer or real, or a port), and either contains a small amount of data or refers to a larger amount of data.

1–12. Introduction to Mach

3 or 4 special ports are "thread" based rights — all others are task based.

Ports



H2596 1-12. 12 © 1991 Hewlett-Packard

Ports

A task may have either send and receive rights to a port or just send rights. However, while only one task may have receive rights, any number may have send rights. Thus one task can provide a service to multiple clients.

Ports in OSF/1 are most commonly used as object references: send rights on a port represent the name of the associated object.

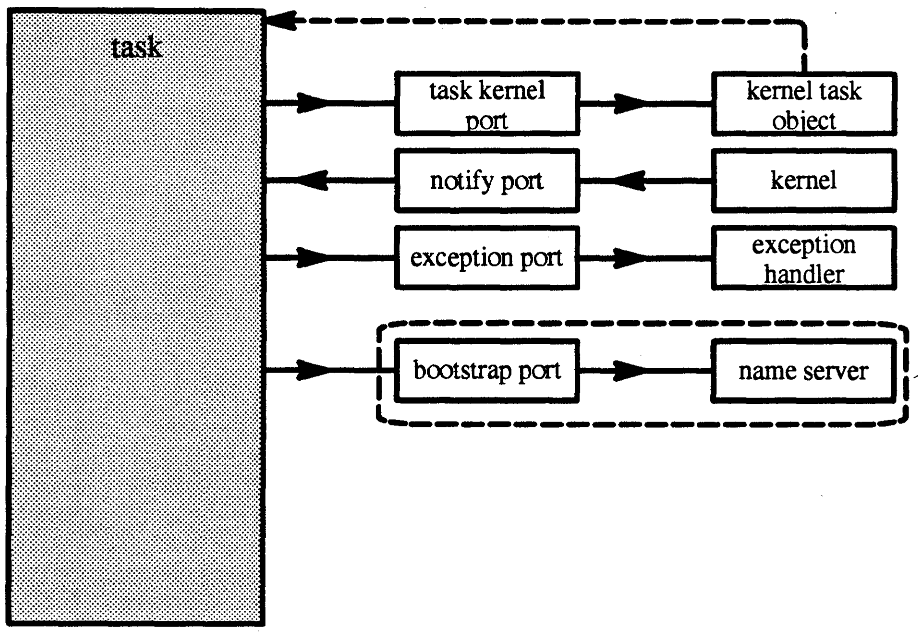
System calls:

- port_allocate(task, port_name): create a port, giving *task* both send and receive rights
- port_deallocate(task, port_name): eliminate *task*'s rights to the named port

} externally visible system call

1–13. Introduction to Mach

Tasks



name server does not exist in 1.1

.H2596 1-13. 13 © 1991 Hewlett-Packard

Tasks

Tasks are the basic unit of protection: threads within a task share all of the task's capabilities (ports) and thus are not protected from one another.

minimally default

Each task has four ports associated with it:

1. *Task kernel port*: essentially the name of the task. In order to perform a system call that affects a task, the calling thread must have *send rights* to the task kernel port of this task. Thus threads in other tasks may issue system calls on a task's behalf if their tasks have send rights to the target task's task kernel port. This ability is particularly useful for debuggers. The special call task_self returns send rights for the current task.

2. *Notify port*: the kernel sends messages through this port to notify the task of various kernel events, such as the destruction of ports. Each task is given *receive rights* on its own notify port.

kernel handler - converts into "unix" signal

exception port

3. *Exception port*: used to implement the exception mechanism (discussed in Module 2). Each task inherits from its parent send rights to an exception port.

4. *Bootstrap port*: used by the threads in a task to send requests (to a name server) to obtain other ports. A task is given send rights to a bootstrap port. This port is available in OSF/1 but not used.

System calls:

- task_create(parent_task, inherit_memory, child_task): the OSF/1 kernel does not currently export this call, although a pure Mach kernel would. Instead, one uses the UNIX fork system call, which creates both a task and a thread within that task.

- task_terminate(target_task): also not currently exported.

- task_suspend(target_task): suspends all threads within a task.

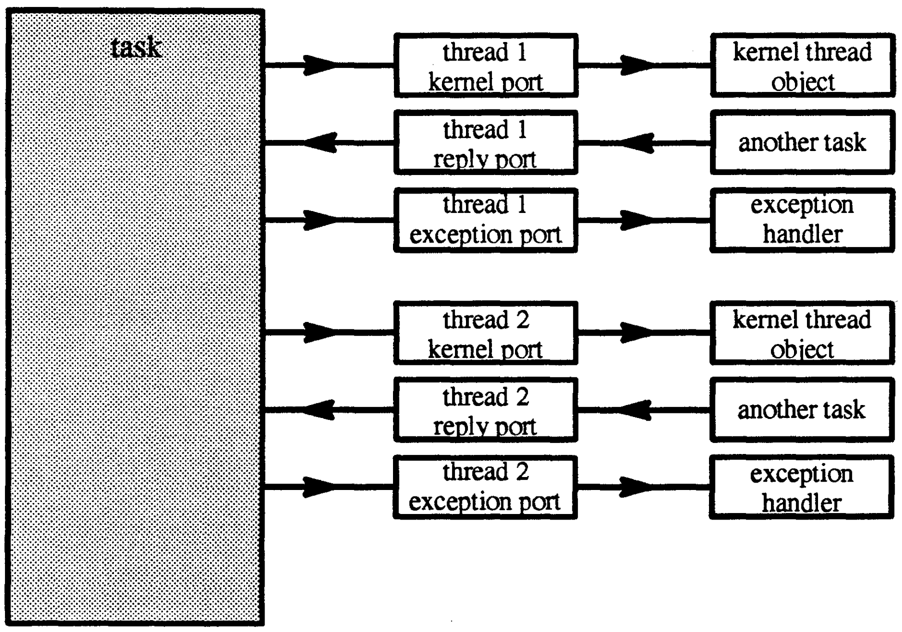
- task_resume(target_task): resumes all threads within a task.

exported

1–14. Introduction to Mach

w/ Addition to "task" Ports

Threads



Threads

Threads are the basic unit of scheduling.

Each thread has three ports associated with it:

1. *Thread kernel port*: represents the name of a thread. When a thread is created, its task is given send rights to the thread's kernel port. Threads in tasks holding send rights on this port may use these rights to issue system calls on the target thread's behalf. A thread can discover its own kernel port by calling thread_self.

2. *Thread reply port*: used for receiving initialization messages and responses from early RPC calls. When a thread is created, its task is given receive rights to this port.

3. *Thread exception port*: part of the implementation of exception handling (described in Module 2). When a thread is created, its task is given send rights to the task's exception port.

You can create reads on exception ports, (in another thread)

Ports, like threads, exist within a task: all of a task's ports are accessible by all of the task's threads.

System calls:

- thread_create(parent_task, child_thread)
- thread_terminate(target_thread)
- thread_suspend(target_thread)
- thread_resume(target_thread)

Suspend Counts

Signals Are NOT parallelized

"Oldest" thread take process schedule work

1–15. Introduction to Mach

VM-Map

Virtual Memory in Mach

System calls:

exported

- vm_allocate(target_task, address, size, anywhere)
- vm_deallocate(target_task, address, size)
- vm_read(target_task, address, data, data_count)
- vm_write(target_task, address, data, data_count)
- vm_protect(target_task, address, size, set_maximum, new_protection)
- vm_inherit(target_task, address, size, new_inheritance)

H2596 1-15. 15 © 1991 Hewlett-Packard

Virtual Memory in Mach

Virtual memory is a property of the task.

Each task has a (possibly sparse) virtual address space.

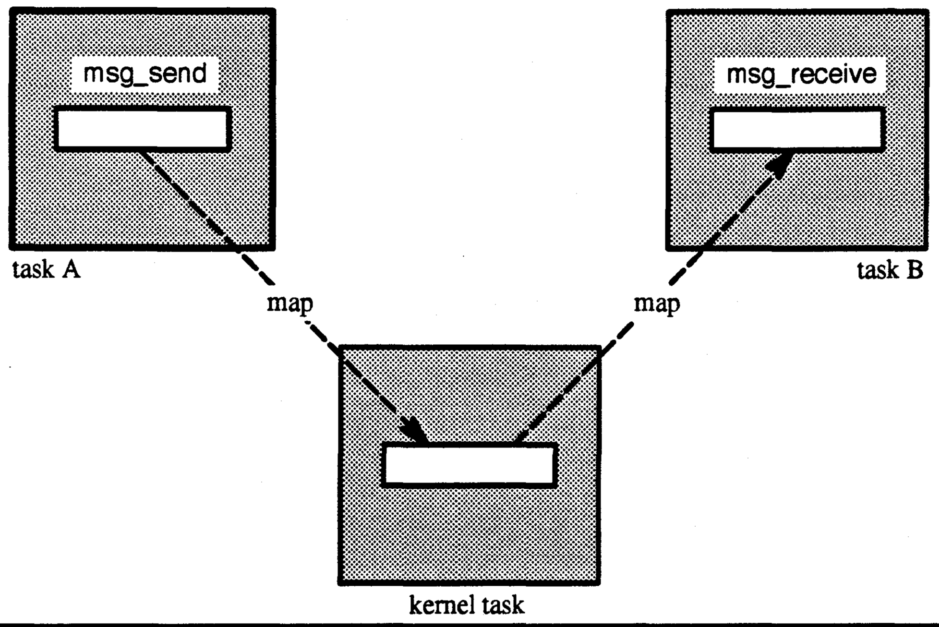
Tasks may inherit virtual memory from their parents, either shared or copied.

Pages are backed up by memory objects, which may be either temporary (traditional paging/swapping space) or permanent.

Lazy evaluation is the pervasive implementation technique.

1–16. Introduction to Mach

Messages Revisited

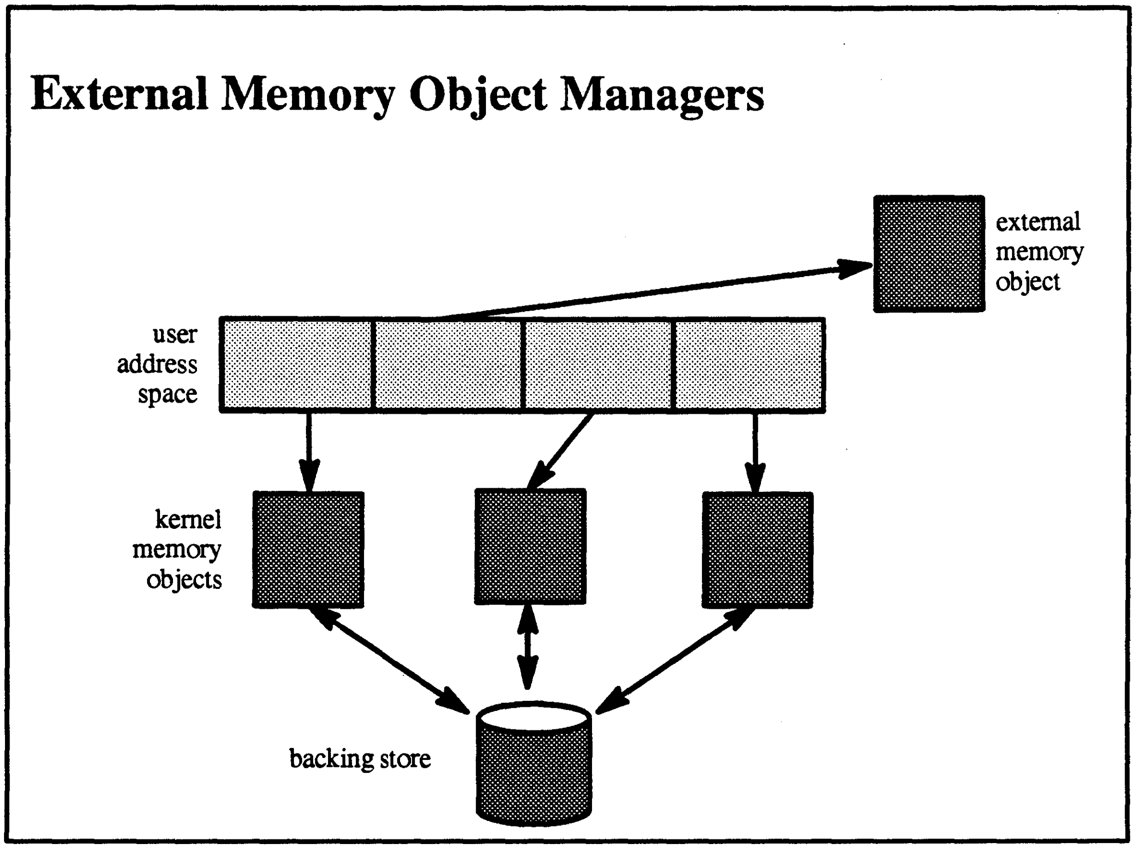


.H2596 1-16. 16 © 1991 Hewlett-Packard

Messages Revisited

Longer messages are first mapped (*copy-on-write*) into the kernel's address space. When the message is received, it is remapped into the receiver's address space. Thus the receiver gains not only the data of the message, but new valid locations in its address space. These locations may be deallocated using vm_deallocate.

1–17. Introduction to Mach



External Memory Object Managers

H2596 1-17. 17 © 1991 Hewlett-Packard

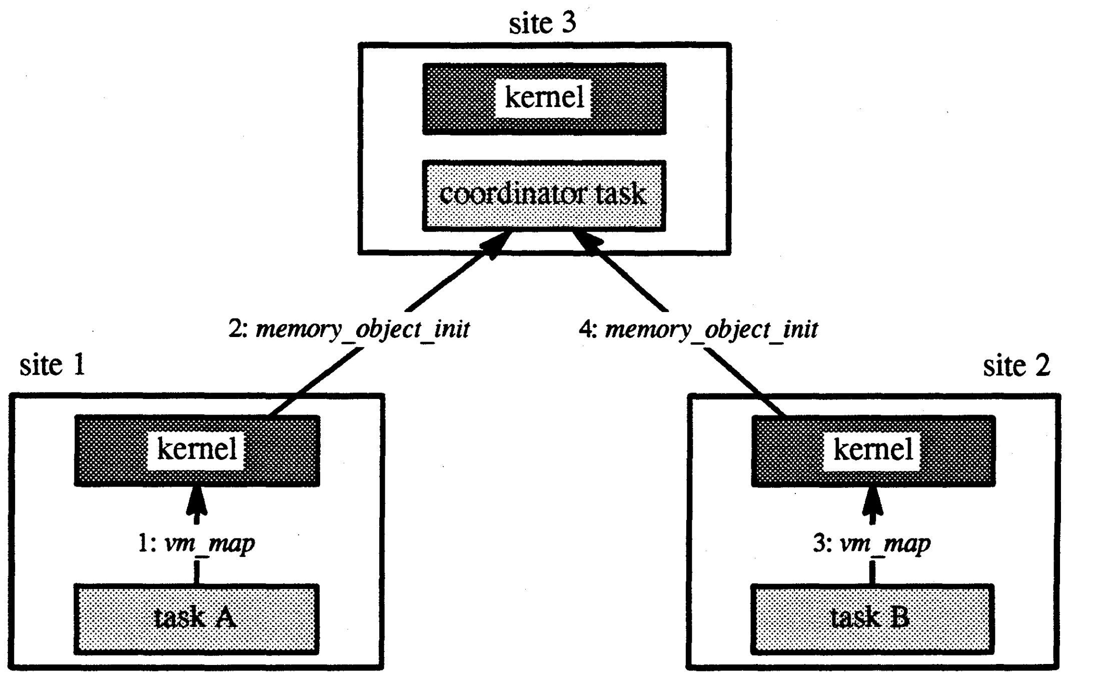
External Memory Object Managers

- The Mach kernel normally manages the backing store for virtual memory
- Users may supply *external memory object managers* to perform this chore

External object managers are responsible for supplying initial values for a range of virtual memory and for backing up virtual memory when the physical memory cache becomes full. Such managers may be used, for example, to map files into the address spaces of tasks, to provide shared memory in a distributed system, and to implement a transaction-management system.

1–18. The Extensible Kernel

Network Shared Memory, part 1



H2596 1-18. 18 © 1991 Hewlett-Packard

Network Shared Memory, part 1

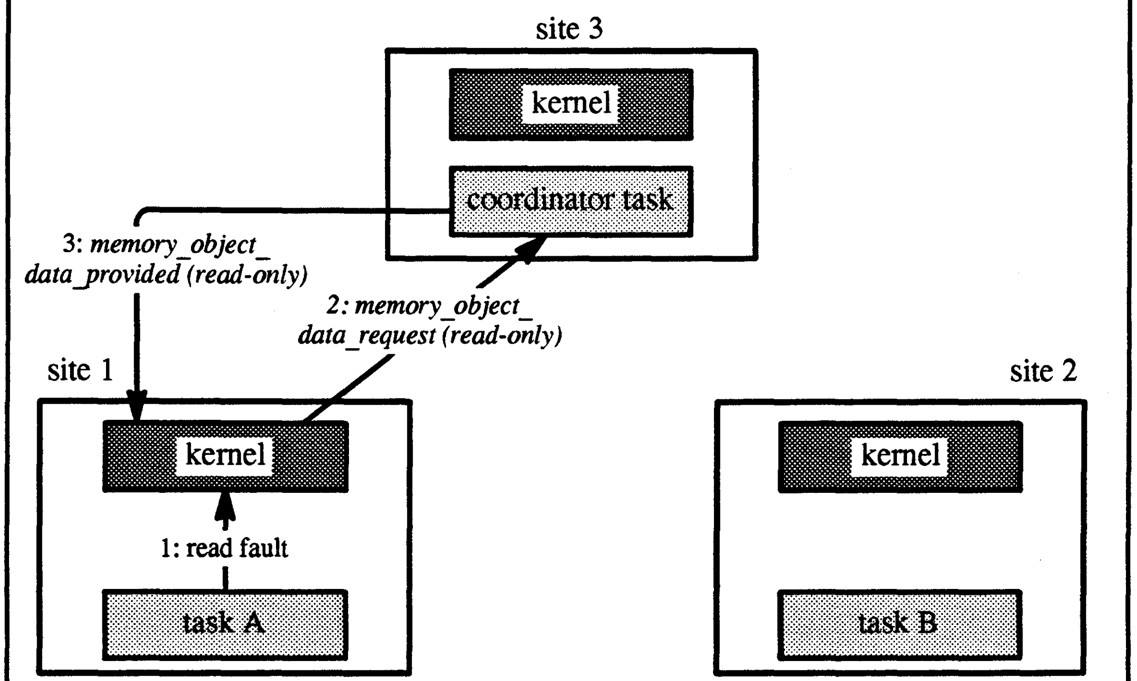
This example shows how the Mach facilities of OSF/1 might provide the abstraction of shared memory to threads running on different machines. Here, site 1 and site 2 are two different machines; the coordinator, the provider of the "shared memory," might be on a third machine.

Two sites share memory by mapping it from the coordinator. A thread uses the vm_map system call to inform its kernel that it wishes to map a particular object into its task's address space. The kernel, in turn, forwards a notification to the coordinator (a *memory_object_init* message), telling it that yet another site is using one of its objects.

Note that at this point no pages have been transferred.

1–19. The Extensible Kernel

Network Shared Memory, part 2


site 3
kernel
coordinator task
3: memory_object_ data_provided (read-only)
2: memory_object_ data_request (read-only)
site 1
kernel
1: read fault
task A
site 2
kernel
task B

.H2596 1-19. 19 © 1991 Hewlett-Packard

Network Shared Memory, part 2

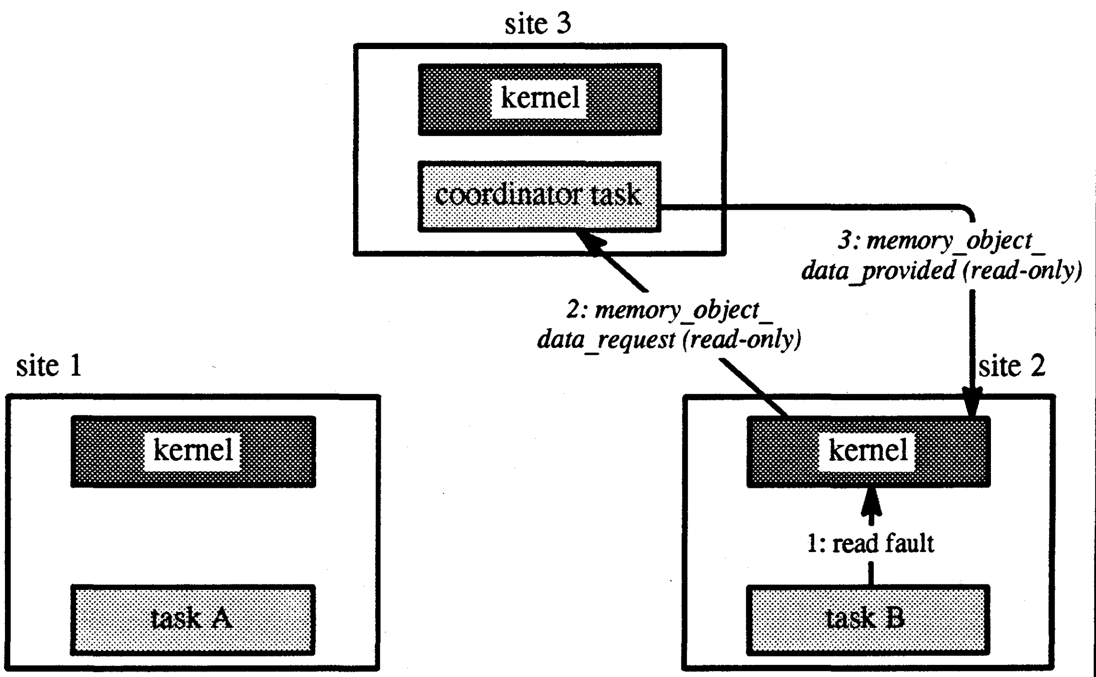
A thread running on site 1 attempts to read from one of the pages in the object maintained by the coordinator. Since the page is not resident at its site, a page fault occurs. The local kernel handles the fault and forwards it to the coordinator (by sending the coordinator an *memory_object_data_request* message).

The coordinator sends a copy of the page back to the kernel on site 1 (via a *memory_object_data_provided* message), but marks it read-only.

The kernel then puts this page in its memory cache and allows the original thread to resume execution.

1–20. The Extensible Kernel

Network Shared Memory, part 3



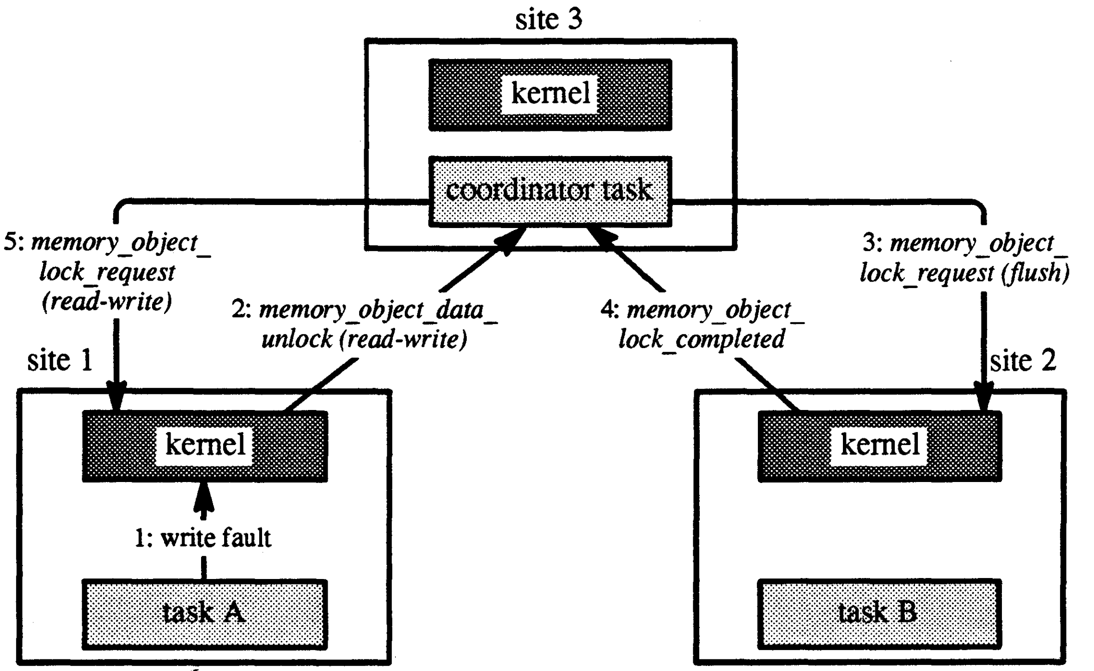
.H2596 1-20. 20 © 1991 Hewlett-Packard

Network Shared Memory, part 3

A thread running on site 2 attempts to read the same page that was just read on site 1. As before, the coordinator gives site 2 a read-only copy of the page. Thus threads on both sites effectively share this page, though at the moment they are only reading it.

1–21. The Extensible Kernel

Network Shared Memory, part 4


site 3
kernel
coordinator task
5: memory_object_ lock_request (read-write)
2: memory_object_data_ unlock (read-write)
4: memory_object_ lock_completed
3: memory_object_ lock_request (flush)
site 1
site 2
kernel
1: write fault
task A
kernel
task B

H2596 1-21. 21 © 1991 Hewlett-Packard

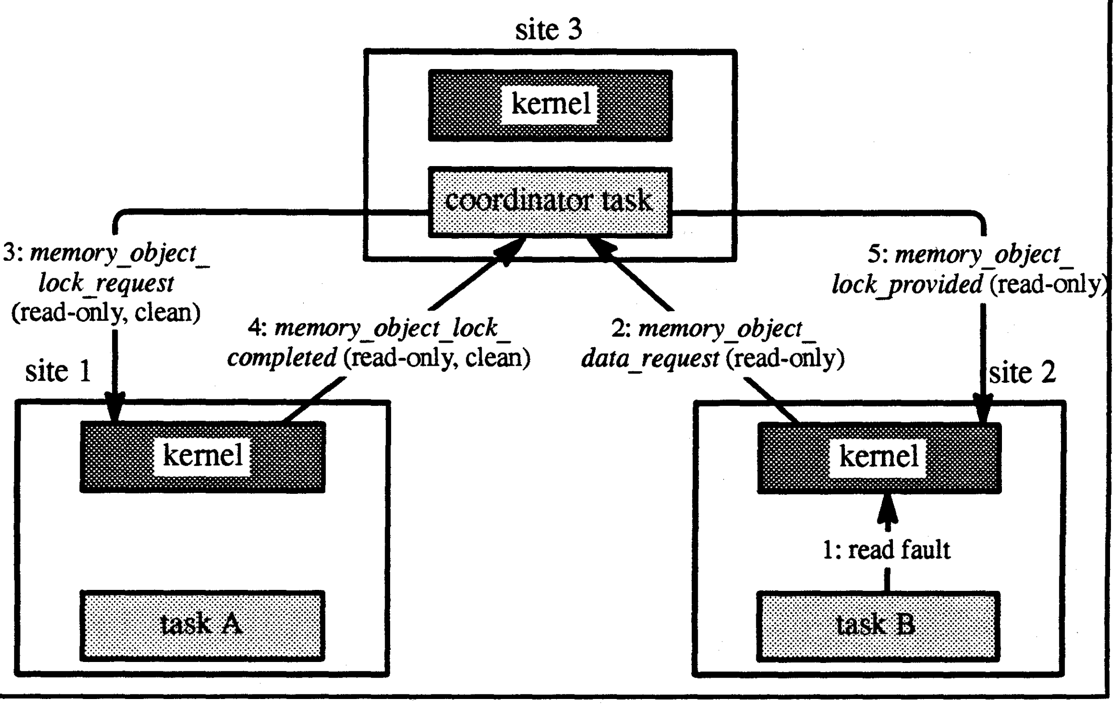
Network Shared Memory, part 4

A thread on site 1 now attempts to modify the page of which both sites have a read-only copy. The local kernel handles the resulting protection fault by sending a request to the coordinator (a *memory_object_data_unlock* message), asking to upgrade its permissions for this page from read-only to read-write.

The coordinator must arrange that all subsequent reads of this page by any site obtain the *modified version* of the page. To accomplish this, it sends a request to site 2 (a *memory_object_lock_request* message) asking it to flush the page from its cache. After it has done so, site 2 sends a *memory_object_lock_completed* message back to the coordinator. After the coordinator receives this message, it sends a message to site 1 (a *memory_object_lock_request* message) granting it read-write permission for the page. Thus threads on site 1 are now free to modify the page.

1–22. The Extensible Kernel

Network Shared Memory, part 5


site 3
kernel
coordinator task
3: memory_object_ lock_request (read-only, clean)
4: memory_object_lock_ completed (read-only, clean)
2: memory_object_ data_request (read-only)
5: memory_object_ lock_provided (read-only)
site 1
kernel
task A
site 2
kernel
1: read fault
task B

.H2596 1-22. 22 © 1991 Hewlett-Packard

Network Shared Memory, part 5

If a thread on site 2 attempts to access the page, a page fault occurs, since the page is no longer resident, and a request (*memory_object_data_request*) is sent to the coordinator for a copy of the page.

To obtain the current contents of the page, the coordinator must send a message to site 1 (*memory_object_lock_request*) asking for the latest version. To make certain that this version continues to be the latest version, this message tells site 1 to turn off write permission.

Module 1 — Background and Introduction

Exercises:

1. List the components of OSF/1.
2. Explain how Mach and UNIX coexist within the OSF/1 kernel.
3. Characterize the multiprocessor architectures supported by OSF/1.
4. a. List the fundamental abstractions of Mach.
   b. What is the difference between concurrency and parallelism?
   c. Give an example of how concurrency as provided by threads simplifies the design of an application even on a uniprocessor.
   d. Explain how threads may be used to exploit the multiprocessor.
   e. Explain how ports may be used for both object references and interprocess communication.
5. Can network-shared memory be implemented on other UNIX systems without kernel modifications?

Advanced Questions:

6. What functionality does Mach supply that Berkeley UNIX does not?
7. What functionality does OSF/1 supply that is supplied by neither Berkeley UNIX nor Mach?

Module 2 — The Process Abstraction

Module Contents

Module Objectives

In order to demonstrate an understanding of the differences between the OSF/1 process and the traditional UNIX process and the implementation of the process in OSF/1, the student should be able to:

- list the UNIX system calls that are difficult to adapt for use by threads within a multithreaded process and describe the difficulty

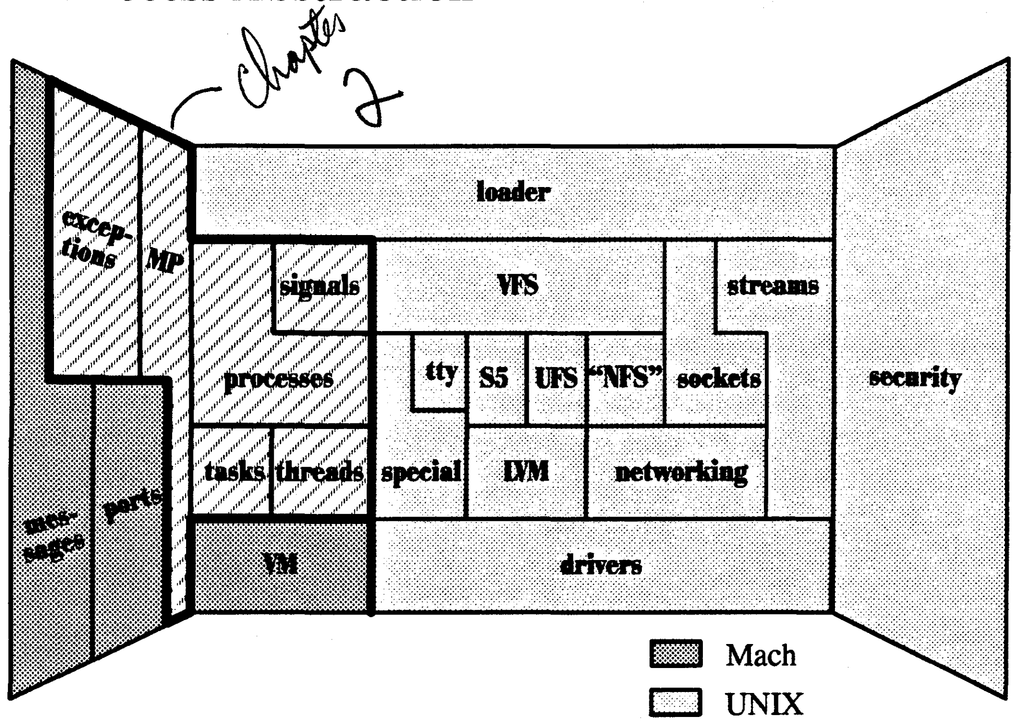
Module 2 — The Process Abstraction

- explain how the *proc* and *user* structures of older UNIX implementations must be modified for use with multithreaded processes
- explain the conceptual difference between UNIX and Mach system calls
- list and explain the need for the kernel synchronization routines
- for each of two types of signals explain how it is determined which thread receives the signal
- explain Mach implementation of exception handling and how exceptions are converted into signals
- explain the rationale for using the POSIX threads library
- describe the scheduling policies used in OSF/1
- explain the rationale of processor sets
- describe the purpose of thread pools and which subsystems use them
- explain the advantage of the zone memory allocation technique

Module 2 — The Process Abstraction

2–1. The Big Picture

The Process Abstraction

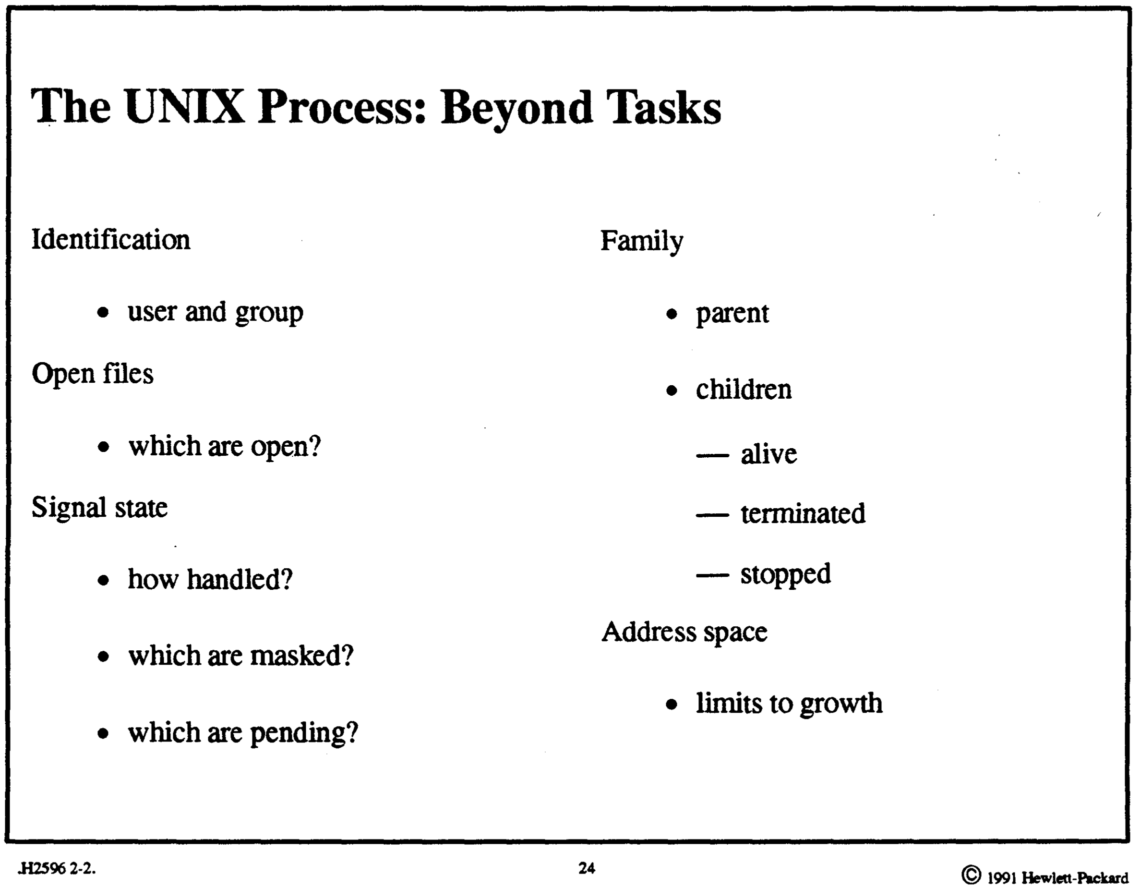


.H2596 2-1. 23 © 1991 Hewlett-Packard

The Process Abstraction

The material in this module is partially covered in Open Software Foundation, 1990a, chapters 4 and 5.

2–2. Processes



The UNIX Process: Beyond Tasks

Identification

- user and group

Open files

- which are open?

Signal state

- how handled?
- which are masked?
- which are pending?

Family

- parent
- children
  - — alive
  - — terminated
  - — stopped

Address space

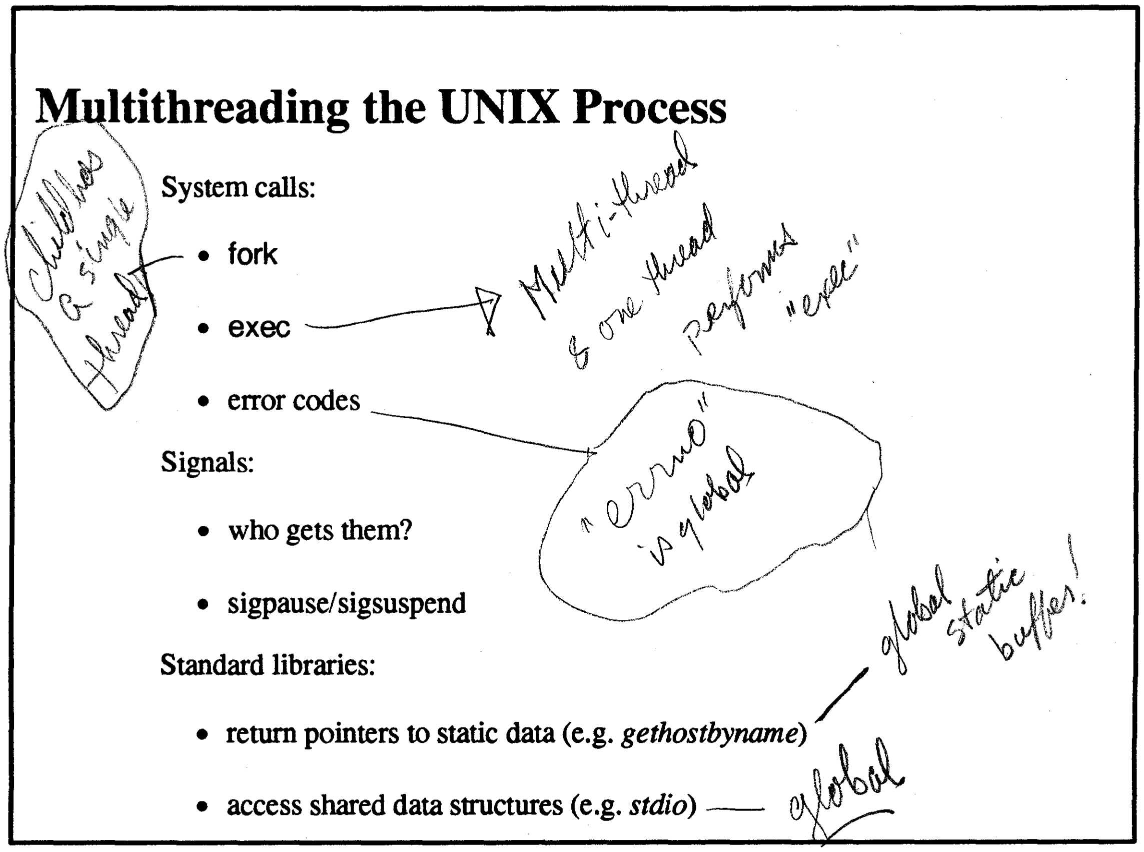
- limits to growth

H2596 2-2. 24 © 1991 Hewlett-Packard

The UNIX Process: Beyond Tasks

The UNIX process embodies much more than what is in a Mach task. Associated with a task is the address space and a collection of port rights. Associated with the UNIX process are additional concepts such as *userid* and *groupid*s, open files, signal information, and relationships between parents and children. Since this information is not part of the task concept, it must be represented separately.

2–3. Processes



Multithreading the UNIX Process

System calls:

- fork — child has a single thread
- exec — Multi-thread & one thread performs "exec"
- error codes — "errno" is global

Signals:

- who gets them?
- sigpause/sigsuspend

Standard libraries:

- return pointers to static data (e.g. *gethostbyname*) — global static buffers!
- access shared data structures (e.g. *stdio*) — global

H2596 2-3. 25 © 1991 Hewlett-Packard

Multithreading the UNIX Process

2–4. Processes

Threads, Tasks, and Processes

- The thread abstraction
  - — a single thread of control
  - — represented by a thread structure
- The task abstraction
  - — holds capabilities and an address space
  - — represented by a task structure
- The process abstraction
  - — combines thread and task abstractions: a UNIX concept

+ environment / file descriptors, ...

.H2596 2-4. 26 © 1991 Hewlett-Packard

Threads, Tasks, and Processes

A user thread normally executes in *user mode*. While it is in a system call, however, it executes in *system mode* (or *kernel mode* or *privileged mode*). The operating system must maintain a separate context for each mode (user and system modes): for example, a thread has both a *user stack* and a *system stack* (or *kernel stack*).

Threads and tasks are represented in Mach by their thread and task data structures. In traditional UNIX, each process is represented by two data structures: the *proc* structure, which is allocated from the kernel's address space, and the *user* structure, which is allocated at a fixed location in the private address space of the process.

OSF/1 represents a UNIX process with both the Mach data structures and most of the information contained in the traditional UNIX data structures. Ideally, the UNIX data structures should be basically unchanged from 4.4BSD so that no significant changes to the UNIX code are necessary.

However, extending the single-threaded UNIX process into a multithreaded process requires some significant changes. Most of the information in the *proc* structure is a property of the task, but the *user* structure contains both task information and thread information. Thus each thread within the task requires its own copy of the thread portion of the *user* structure: the *user* structure is divided into a *u_task* component and (multiple) *u_thread* components.

Both types of components are allocated from the system address space, not the process address space (the original UNIX scheme of allocating user structure at a fixed location in the process address space is not possible because of the multiple *u_thread* components).

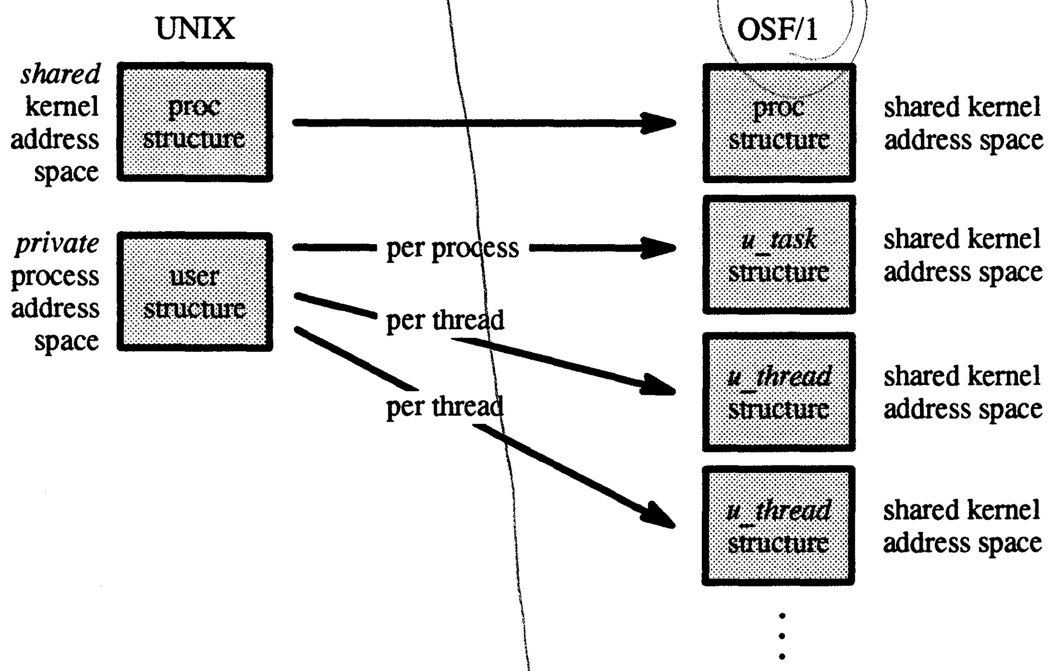
A slight problem arises here: UNIX kernel code refers to components within the *user* structure as, for example, *u.xxx*. OSF/1 copes with this simply by using the C preprocessor to convert such references into either *u_task->xxx* or *u_thread->xxx*.

These issues are discussed in A. Tevanian, 1987.

2–5. Processes

Traditional

Process Data Structures

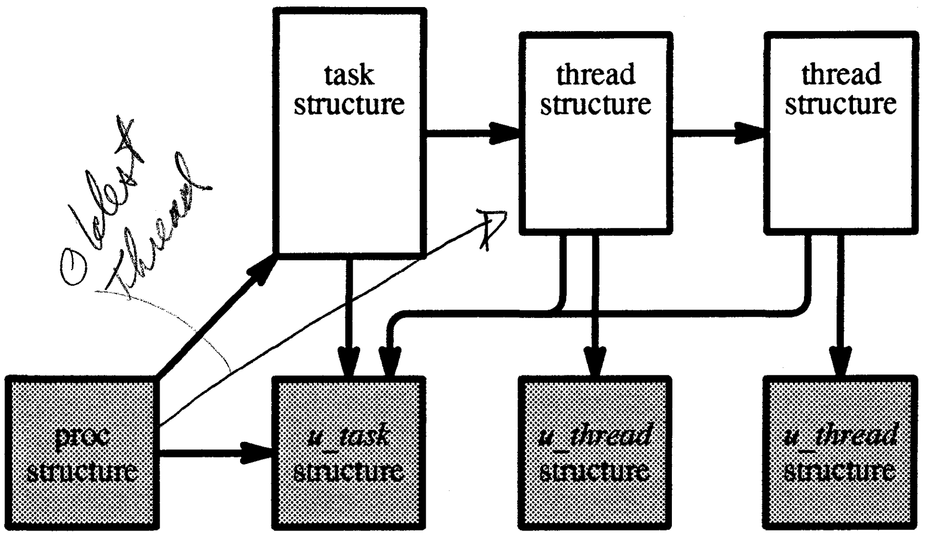

UNIX
OSF/1
shared kernel address space
proc structure
proc structure
shared kernel address space
private process address space
user structure
per process
u_task structure
shared kernel address space
per thread
u_thread structure
shared kernel address space
per thread
u_thread structure
shared kernel address space

.H2596 2-5. 27 © 1991 Hewlett-Packard

Process Data Structures

2–6. Processes

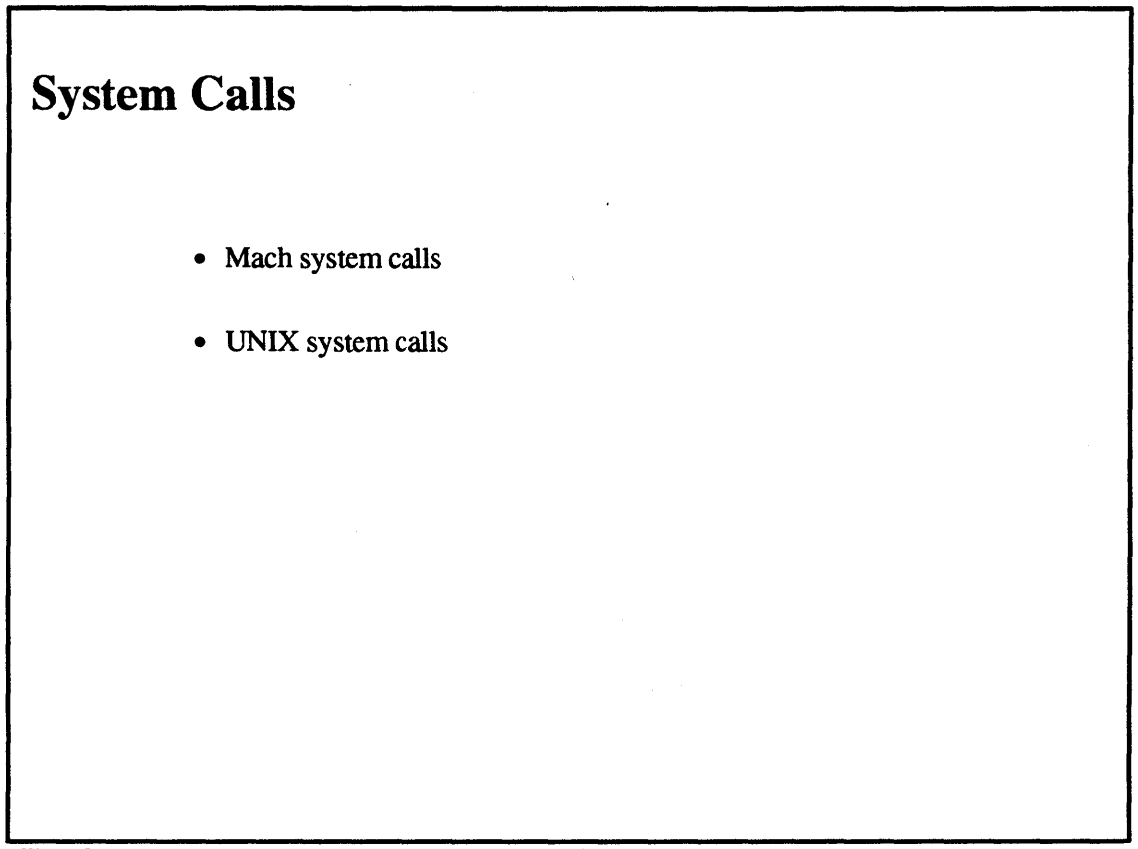
Thread and Task Structures



.H2596 2-6. 28 © 1991 Hewlett-Packard

Thread and Task Structures

2–7. System Calls in OSF/1



System Calls

- Mach system calls
- UNIX system calls

.H2596 2-7. 29 © 1991 Hewlett-Packard

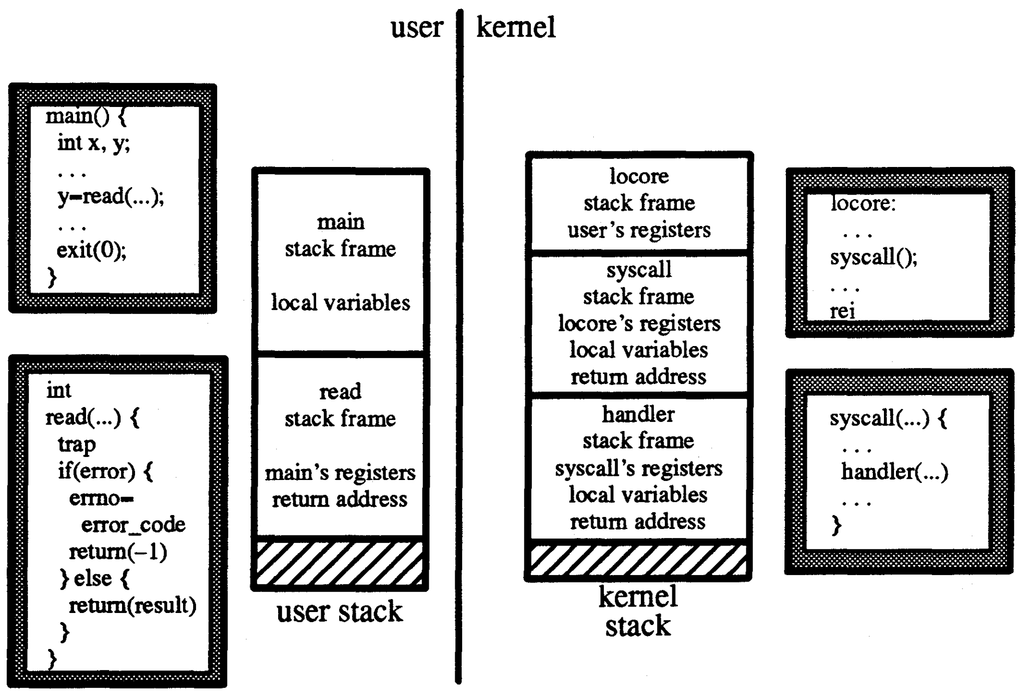
System Calls

System calls are the sole interface between the user and the operating system. From the user's point of view, a system call is a subroutine call, but the body of this subroutine call involves a switch from unprivileged user mode to privileged system mode. Accomplishing this switch requires an architecture-specific trap construction.

Most system calls issued on an OSF/1 system are UNIX system calls, but Mach system calls are used as well. The UNIX system-call interface is implemented differently from the Mach system-call interface.

2–8. System Calls in OSF/1

UNIX System Calls



```
main() {
  int x, y;
  . . .
  y=read(...);
  . . .
  exit(0);
}
```

```
int
read(...) {
  trap
  if(error) {
    errno=
      error_code
    return(-1)
  } else {
    return(result)
  }
}
```

```
locore:
  . . .
  syscall();
  . . .
  rei
```

```
syscall(...) {
  . . .
  handler(...)
  . . .
}
```

.H2596 2-8. 30 © 1991 Hewlett-Packard

UNIX System Calls

To the C programmer, system calls are calls to subroutines provided by the C library. The bodies of these library routines first execute whatever machine construction is required to generate a trap. The trap is handled in kernel mode (but in the context of the calling process) via a call to *syscall.*

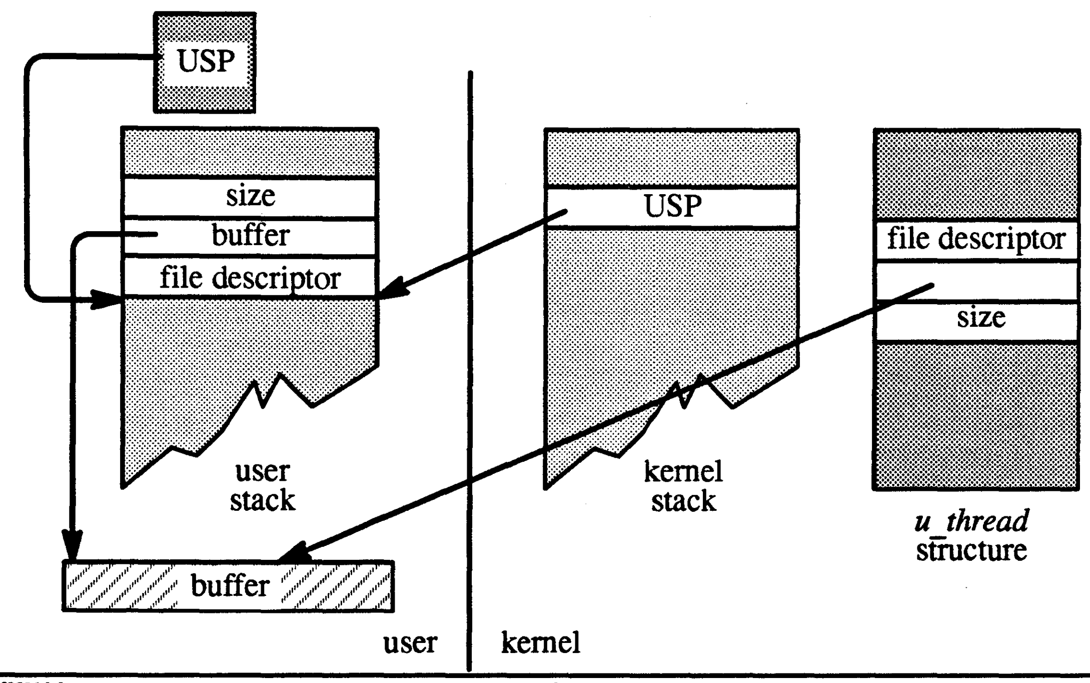
Syscall copies the arguments of the system call to the process's *u_thread* structure, then calls the appropriate system call handler in the kernel.

On return, *syscall* deals with errors, and arranges for results to be returned to user mode.

Finally, the original C library routine passes either an error indication or a result back to the caller.

2–9. System Calls in OSF/1

UNIX System Calls: Passing of Arguments



H2596 2-9. 31 © 1991 Hewlett-Packard

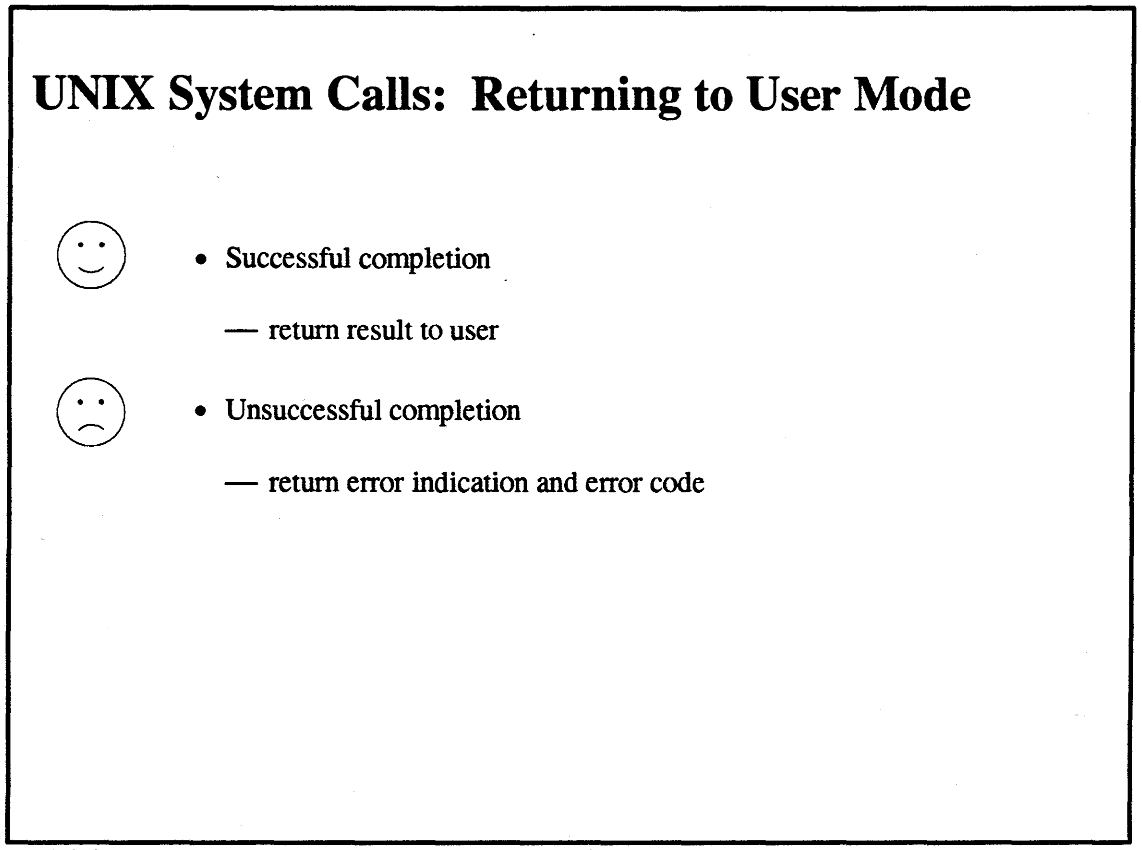
UNIX System Calls: Passing of Arguments

How arguments are passed from the user to the kernel depends upon the architecture, but what is described here is typical. We use the write system call as an example.

As part of calling the write library routine, the arguments are pushed onto the user stack (the end of the stack is pointed to by the *user stack pointer*—USP). When a trap occurs, all of the user's registers, including the USP, are saved on the kernel stack.

The *syscall* routine in the kernel determines which system call is being made and how many arguments it expects. Then, following the saved USP, it finds the arguments on the user stack and copies them to the *u_thread* structure. Copying must be done with care: the user supplies the value of the USP to the kernel. The kernel has no reason to believe that the user has supplied a legitimate value—it might point into the kernel. Thus the kernel must first validate locations pointed to by the USP before copying them to the *u_thread* structure.

2–10. System Calls in OSF/1



UNIX System Calls: Returning to User Mode

- Successful completion
  - — return result to user
- Unsuccessful completion
  - — return error indication and error code

.H2596 2-10. 32 © 1991 Hewlett-Packard

UNIX System Calls: Returning to User Mode

Again, the details here depend upon the architecture; what follows is typical. Assume a calling convention in which functions return their results in register 0.

On return from the trap instruction, if the system call completed successfully, the system arranges for the value to be returned to appear in the user's register 0. This is accomplished by copying this value into the saved copy of register 0 in the kernel stack before returning to user mode. The C library code leaves this value where it is and returns to its caller, which sees the system call returning the appropriate value.

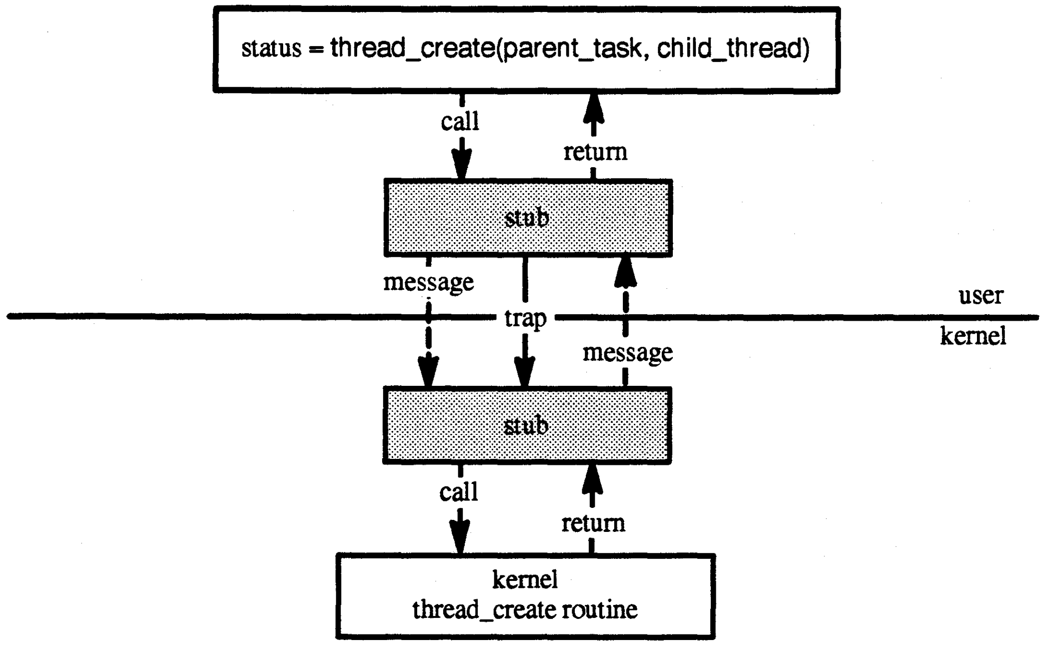
If an error occurred in the system call, UNIX requires that the library routine (e.g. write) return –1 and that the error code be found in the global variable *errno*. This is achieved via cooperation between the operating system and the library routine: the carry bit of the program status word is used to indicate whether the system call succeeded or not. This word is saved on the kernel stack as part of the trap; the operating system sets the carry bit in it accordingly before control is returned to user mode, and the program status word is restored from the kernel stack.

If there was an error, the carry bit is set to 1 and the error code is placed in register 0. When control returns to the library code, if it finds the carry bit is set it copies the value of register 0 to *errno*, puts a –1 into register 0, and then returns.

Note that this does not work well with multithreaded processes!

2–11. System Calls in OSF/1

Mach System Calls



H2596 2-11. 33 © 1991 Hewlett-Packard

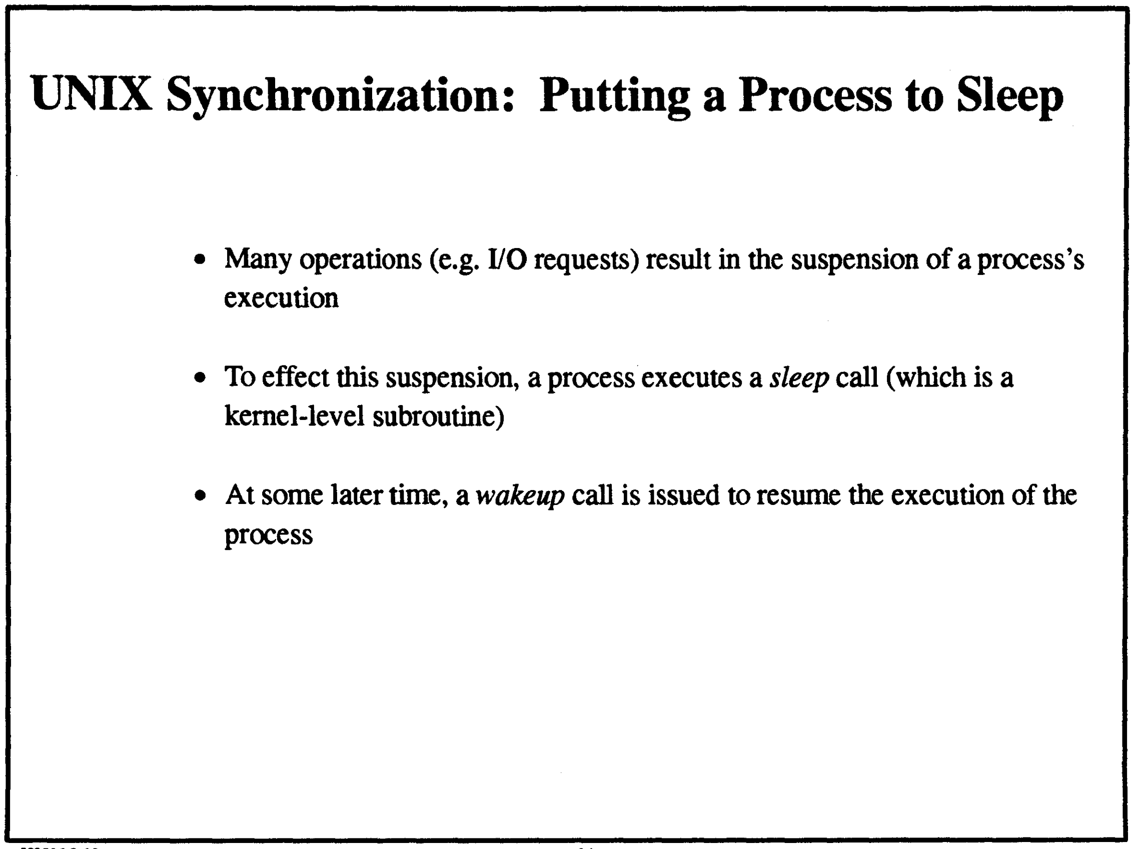
Mach System Calls

In the UNIX system call interface, the functional value returned by the system call is overloaded with either an error indication or the result of a call. Mach avoids the clumsy type problems associated with this style by using an output parameter to return the result of the system call, so that the functional value of the call is solely an indication of success or error (and if an error, what sort of error).

A Mach system call is essentially a remote procedure call to a procedure provided in a kernel task. These procedural requests are actually transmitted to the kernel as messages. The result obtained by the kernel is sent back to the user as another message. Thus a system call, from the user's point of view, is implemented as a msg_rpc call. The stub routines that convert the procedure calls into messages are produced by MIG (the Mach Interface Generator).

The implementation of the Mach system call is optimized: it is not the case that special threads exist in the kernel for the purpose of receiving these system call messages. Instead, the user thread generating the Mach system call traps into the kernel (i.e., switches to kernel mode) and receives its own message and processes its own system call.

2–12. Synchronization and Thread Management



UNIX Synchronization: Putting a Process to Sleep

- Many operations (e.g. I/O requests) result in the suspension of a process's execution
- To effect this suspension, a process executes a *sleep* call (which is a kernel-level subroutine)
- At some later time, a *wakeup* call is issued to resume the execution of the process

.H2596 2-12. 34 © 1991 Hewlett-Packard

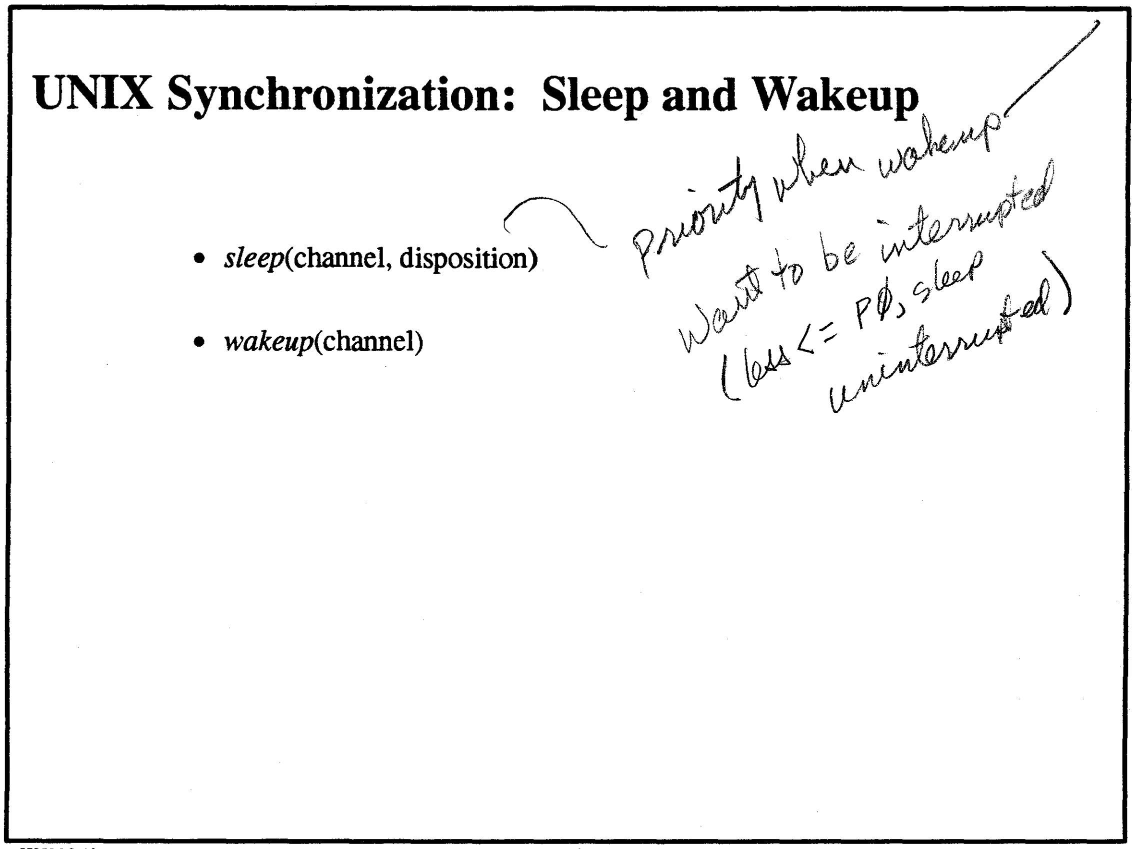
UNIX Synchronization: Putting a Process to Sleep

UNIX synchronization is a very simple, event-driven mechanism. A process (in kernel mode) puts itself to sleep by calling *sleep*. Its state is then set to sleeping, and control is passed to *swtch*, which finds another runnable process and resumes its execution.

A call to *wakeup* resumes the execution of all processes waiting on a particular event. Such processes' states are changed to runnable, and when the scheduler chooses them they resume execution.

the lower the ~~priority~~ value the better the priority

2–13. Synchronization and Thread Management



UNIX Synchronization: Sleep and Wakeup

- *sleep*(channel, disposition)
- *wakeup*(channel)

priority when wakeup
Wants to be interrupted
(less <= P∅, sleep uninterrupted)

.H2596 2-13. 35 © 1991 Hewlett-Packard

UNIX Synchronization: Sleep and Wakeup

The integer *channel* specifies the awaited event. By convention, this channel is the address of some relevant data structure.

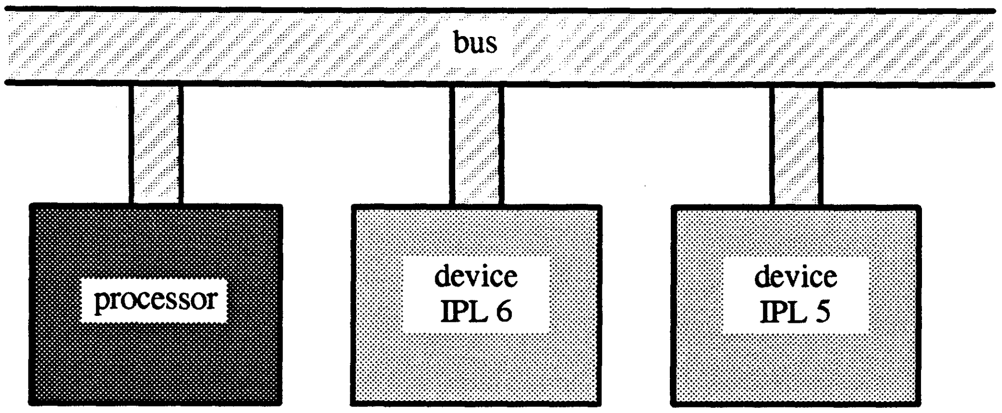
A number of things are overloaded on top of *disposition*:

1. It represents the scheduling priority to be taken on by the process when it is awakened (low values are good priorities, high values are poor priorities).
2. It indicates whether or not the sleep is interruptible by a signal. If it is less than or equal to the fixed value PZERO, then the sleeping process may not be awakened by a signal.
3. It indicates what happens if the sleep is interrupted by a signal. The call to sleep either *longjmp*s back to an exception handler or returns a value indicating that there was a signal.

(N.B.: This is done differently in OSF/1, as will be seen.)

2–14. Synchronization and Thread Management

Protection from Interrupts



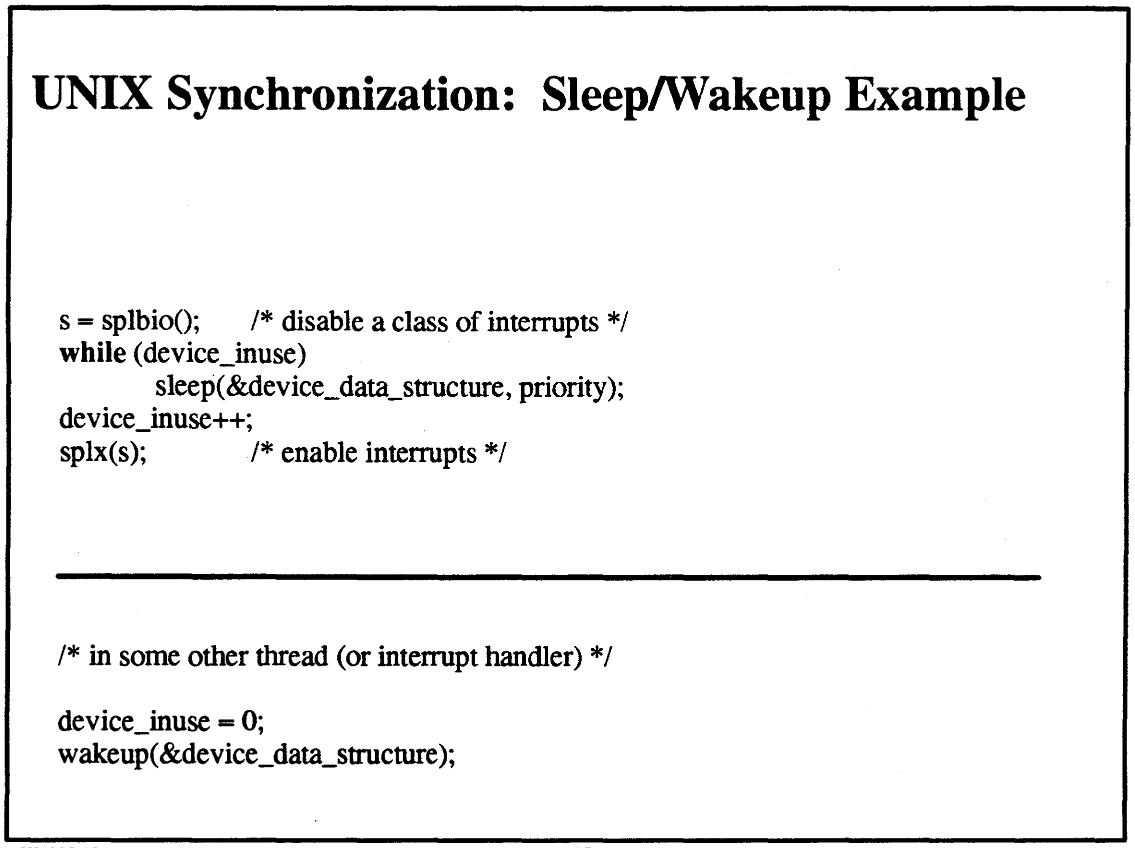
H2596 2-14. 36

Protection from Interrupts

Interrupt protection is very architecture-dependent. The UNIX (and OSF/1) model is based on the PDP-11 architecture: the devices and processor connect via a bus. A device interrupts a processor by raising a line on the bus corresponding to a particular *interrupt priority* (or *bus request level*). If the processor's current *interrupt priority level* (IPL) is less than the bus request level, then the request interrupts the processor's current computation. The processor receives the interrupter's interrupt priority; after the processor returns from the interrupt, it regains its previous priority level.

A call to *splnnn,* where *nnn* identifies the interrupt priority level, disables an entire class of interrupts by raising the processor's IPL. This call returns the previous priority, which can be restored via a call to *splx.*

2–15. Synchronization and Thread Management



UNIX Synchronization: Sleep/Wakeup Example

```
s = splbio();      /* disable a class of interrupts */
while (device_inuse)
        sleep(&device_data_structure, priority);
device_inuse++;
splx(s);           /* enable interrupts */
```

```
/* in some other thread (or interrupt handler) */

device_inuse = 0;
wakeup(&device_data_structure);
```

.H2596 2-15. 37

UNIX Synchronization: Sleep/Wakeup Example

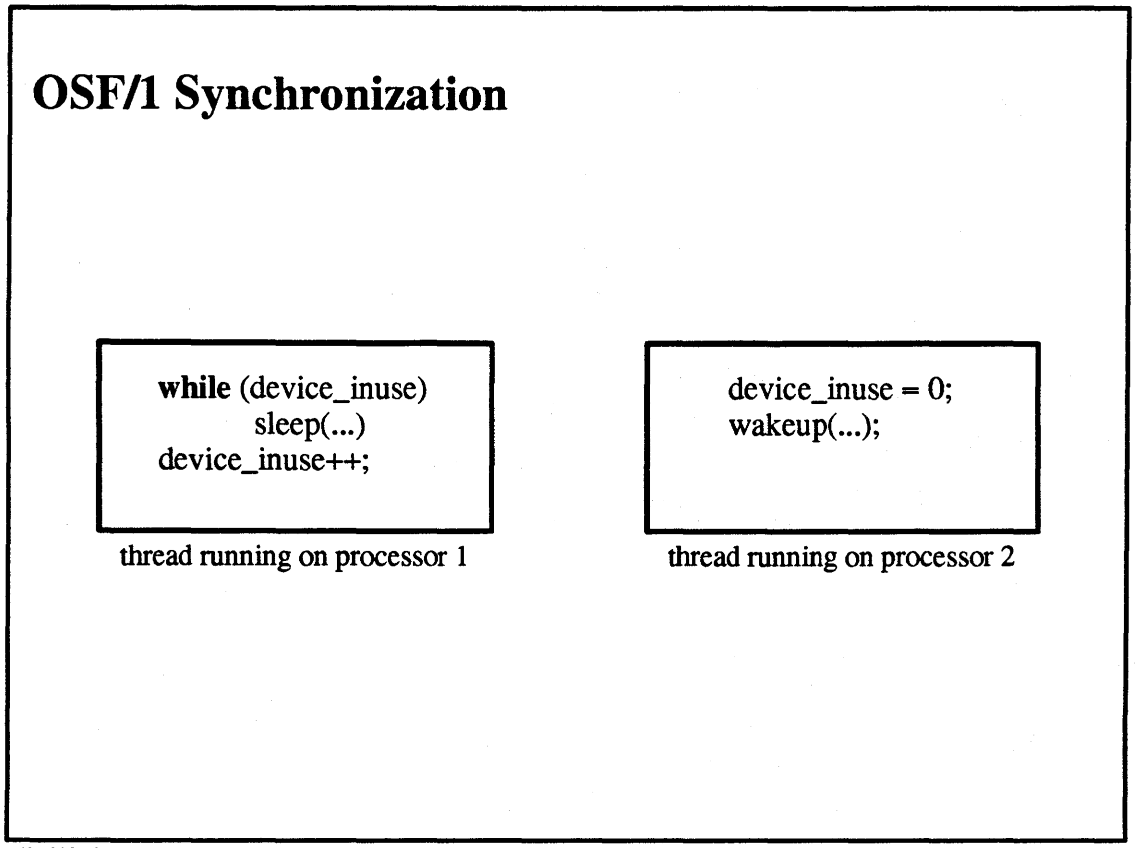
In this example, various threads wish to obtain mutually exclusive access to a device. They check *device_inuse* to see if the device is in use. If it is, they then put themselves to sleep. After the thread using the device finishes with it, the thread clears the *device_inuse* flag and then wakes up all threads waiting for the device.

A potential race condition must be guarded against: between testing the *device_inuse* flag and calling *sleep*, an interrupt handler might issue a *wakeup.* (Since *wakeup*s are not remembered, if no thread is sleeping when a *wakeup* occurs, nothing happens.) Thus, if a thread goes to sleep and the one and only *wakeup* that would ever wake it up has already been issued, the thread sleeps forever.

The solution is straightforward: interrupts must be disabled while the flag is tested and the thread is put to sleep. The call to *splbio* disables (disk) interrupts and returns the previous IPL. Thus interrupts are disabled through the call to sleep.

Inside of sleep, after the thread has been effectively put to sleep, the IPL is reduced back to zero so that interrupts may occur. However, the IPL set by *splbio* is remembered as part of the thread's context. When the thread is woken up and returns from sleep, this IPL is restored and the thread can then make the test (and possibly put itself to sleep again, immune from interrupts). Once it has taken the device, it can restore the original IPL (probably 0) by a call to *splx*.

2–16. Synchronization and Thread Management



OSF/1 Synchronization

```
while (device_inuse)
        sleep(...)
device_inuse++;
```

thread running on processor 1

```
device_inuse = 0;
wakeup(...);
```

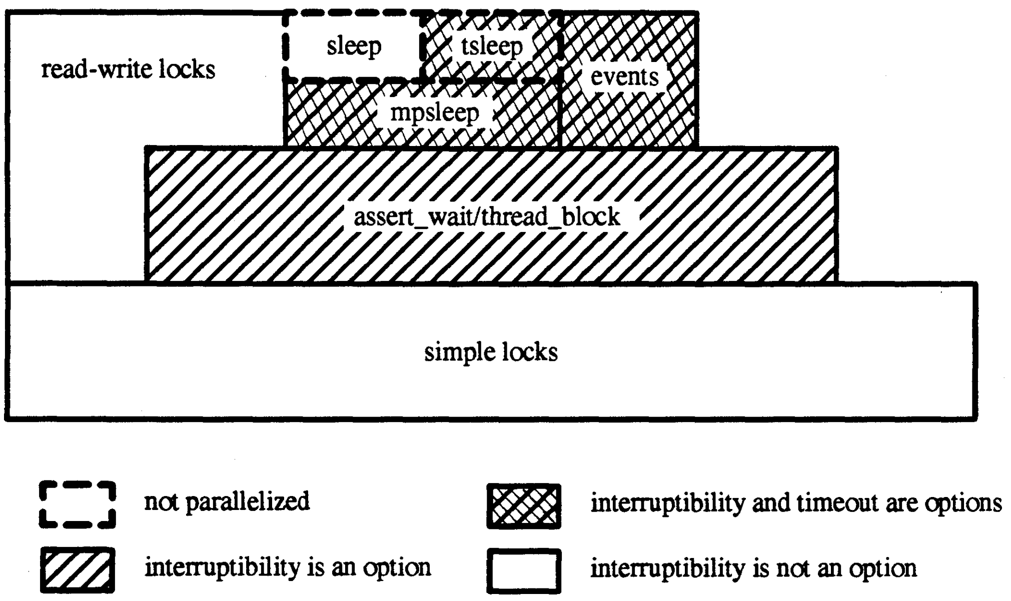
thread running on processor 2

OSF/1 Synchronization

OSF/1 synchronization must be able to cope with the effects of multiprocessors. Masking interrupts is not sufficient protection on a (shared-memory) multiprocessor. Any operation that might be affected by actions of other processors must be protected.

2–17. Synchronization and Thread Management

Synchronization Primitives in OSF/1


read-write locks
sleep
tsleep
events
mpsleep
assert_wait/thread_block
simple locks
not parallelized
interruptibility and timeout are options
interruptibility is an option
interruptibility is not an option

.H2596 2-17. 39 © 1991 Hewlett-Packard

Synchronization Primitives in OSF/1

A good discussion of synchronization in OSF/1 can be found in Open Software Foundation, 1990b, chapter 8.

2–18. Synchronization and Thread Management

Simple Locks

```
while (test_and_set (lock) == WAS_LOCKED)
        ;
```

simple_lock_init(lock)

simple_lock(lock)

simple_unlock(lock)

simple_lock_try(lock)

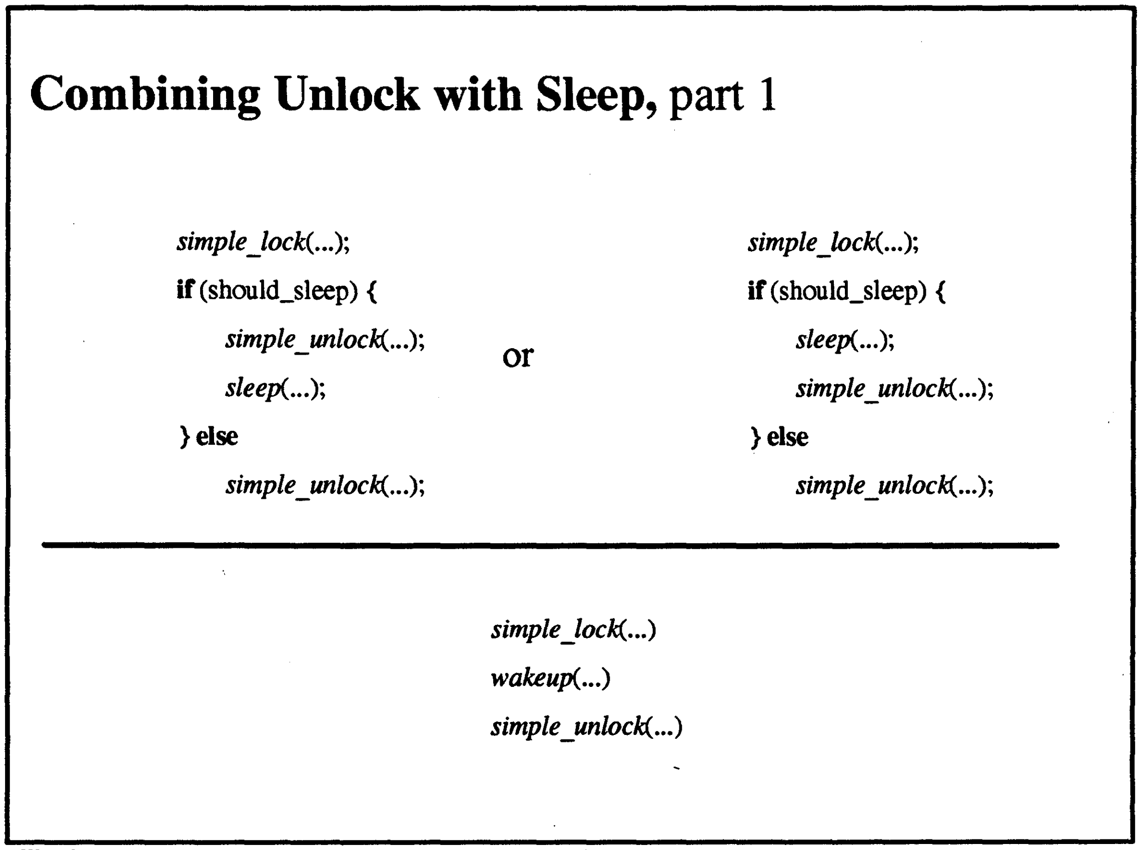
.H2596 2-18. 40

Simple Locks

Simple locks are used in many cases where mutual exclusion is required. They are implemented as *spin locks;* i.e., a thread or interrupt handler sets a lock by setting a bit and waits for a lock by repeatedly testing the bit until the holder of the lock clears it. Because of this active involvement on the part of the processor, simple locks should be held only briefly.

Although simple locks are normally acquired in a synchronous manner, an additional request is provided in which the lock is taken if it is not already taken and otherwise returns failure.

2–19. Synchronization and Thread Management



Combining Unlock with Sleep, part 1

```
simple_lock(...);                          simple_lock(...);
if (should_sleep) {                        if (should_sleep) {
    simple_unlock(...);        or              sleep(...);
    sleep(...);                                simple_unlock(...);
} else                                     } else
    simple_unlock(...);                        simple_unlock(...);
```

```
simple_lock(...)
wakeup(...)
simple_unlock(...)
```

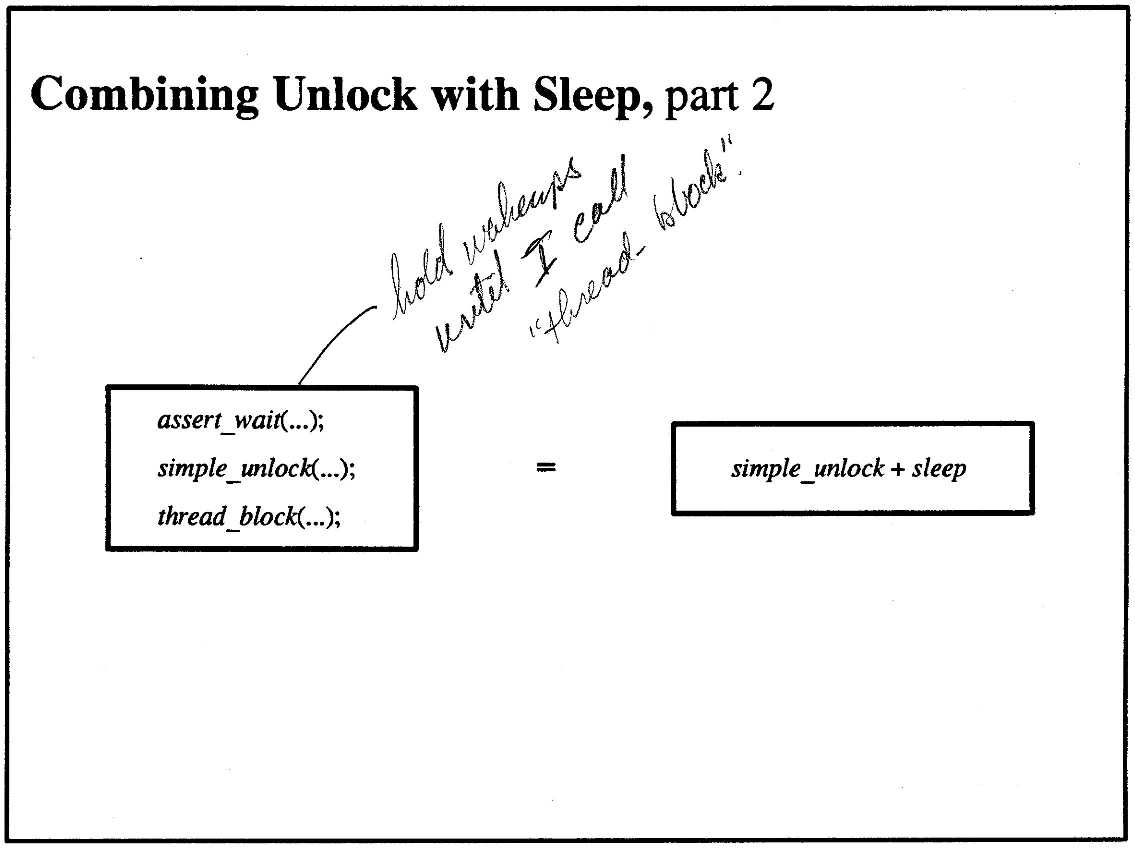
H2596 2-19. 41 © 1991 Hewlett-Packard

Combining Unlock with Sleep, part 1

To avoid a race between one thread doing a *sleep* and another thread doing a *wakeup*, it is necessary to use a lock. However, as illustrated in the slide, it is not clear when the thread calling *sleep* should unlock the lock. In the code fragment in the top left, a thread first takes a lock to guarantee that no thread does a *wakeup* while there is the possibility that the first thread may go to sleep. It then discovers that it indeed should go to sleep, so it unlocks the lock and then calls *sleep*. However, another thread running on another processor might call a *wakeup* at the instant that the lock is unlocked (before the first thread calls *sleep*). Thus we still have the race condition we are trying to eliminate.

Another approach, as illustrated in the upper right, might be to switch the calls to *sleep* and *simple_unlock*. But now, though we eliminate the race condition, we introduce a deadlock. A thread attempting to do a *wakeup* won't be able to do so until the lock is released, but the thread holding the lock won't release it until after the *wakeup* has been performed.

2–20. Synchronization and Thread Management



Combining Unlock with Sleep, part 2

hold wakeups until I call "thread-block".

```
assert_wait(...);
simple_unlock(...);
thread_block(...);
```

= *simple_unlock + sleep*

H2596 2-20. 42 © 1991 Hewlett-Packard

Combining Unlock with Sleep, part 2

The solution is to find a way to combine *sleep* and *simple_unlock*. One approach might be to add an extra argument to *sleep* indicating which lock to unlock after the calling thread is effectively asleep. The approach taken, however, is to split *sleep* into two parts. The first part, *assert_wait*, announces that the thread is about to go to sleep. The second part, *thread_block*, actually puts the thread to sleep. A call to *simple_unlock* may be safely placed between the calls to *assert_wait* and *thread_block*.

2–21. Synchronization and Thread Management

Blocking Threads

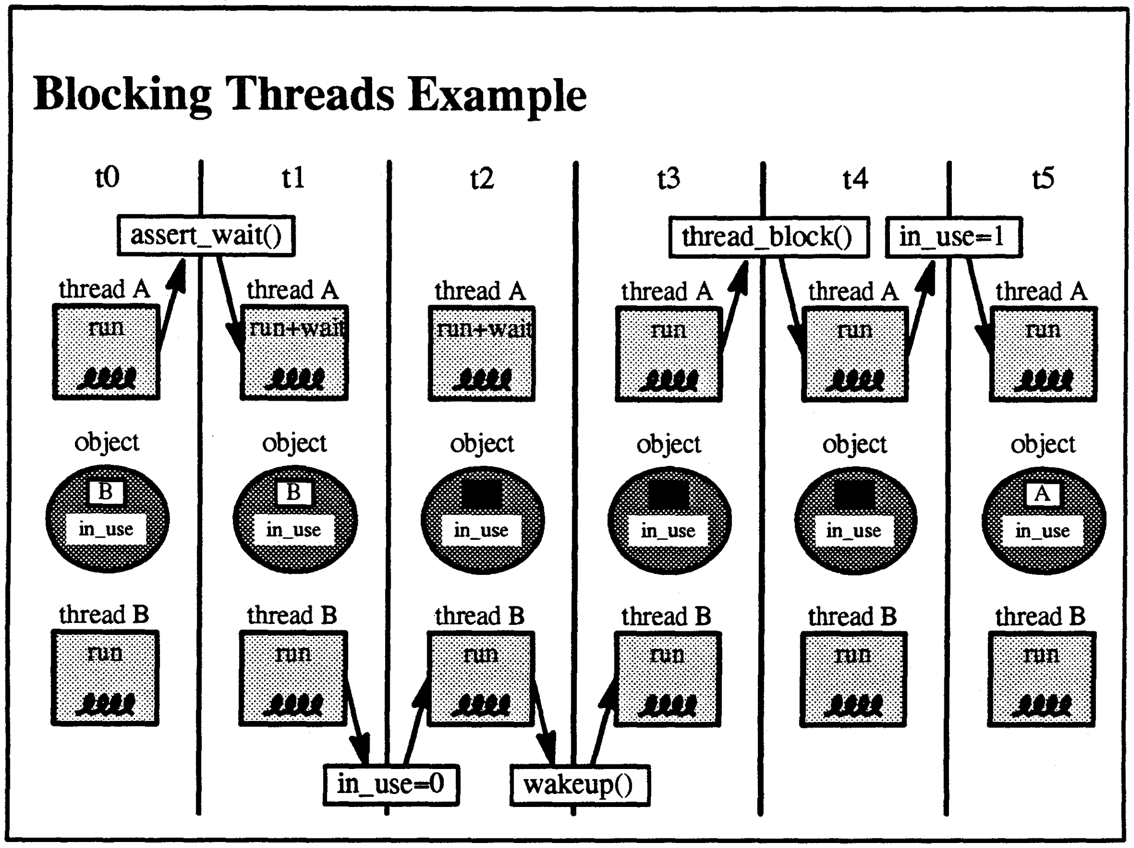
```
simple_lock(&object.lock);
while (object.in_use) {
   assert_wait(&object.wait);
      /* indicate intent to wait */
   simple_unlock(&object.lock);
   thread_block();
      /* give up the processor—however, the thread might return immediately
         if a wakeup has already happened */
   simple_lock(&object.lock);
}
object.in_use = 1;
simple_unlock(&object.lock);
```

.H2596 2-21. 43

Blocking Threads

In this example, threads desire mutually exclusive access to *object*. Associated with the object is a simple lock, which a thread takes so that it can safely determine if another thread is using the object. If the object is in use, then the thread attempting to take the object declares its intention to block by calling *assert_wait*; it then unlocks the simple lock and calls *thread_block* to yield the processor. If this thread is woken up before it yields the processor, the call to *thread_block* does not put the thread to sleep but, at worst, puts the thread on the run queue.

2–22. Synchronization and Thread Management



time

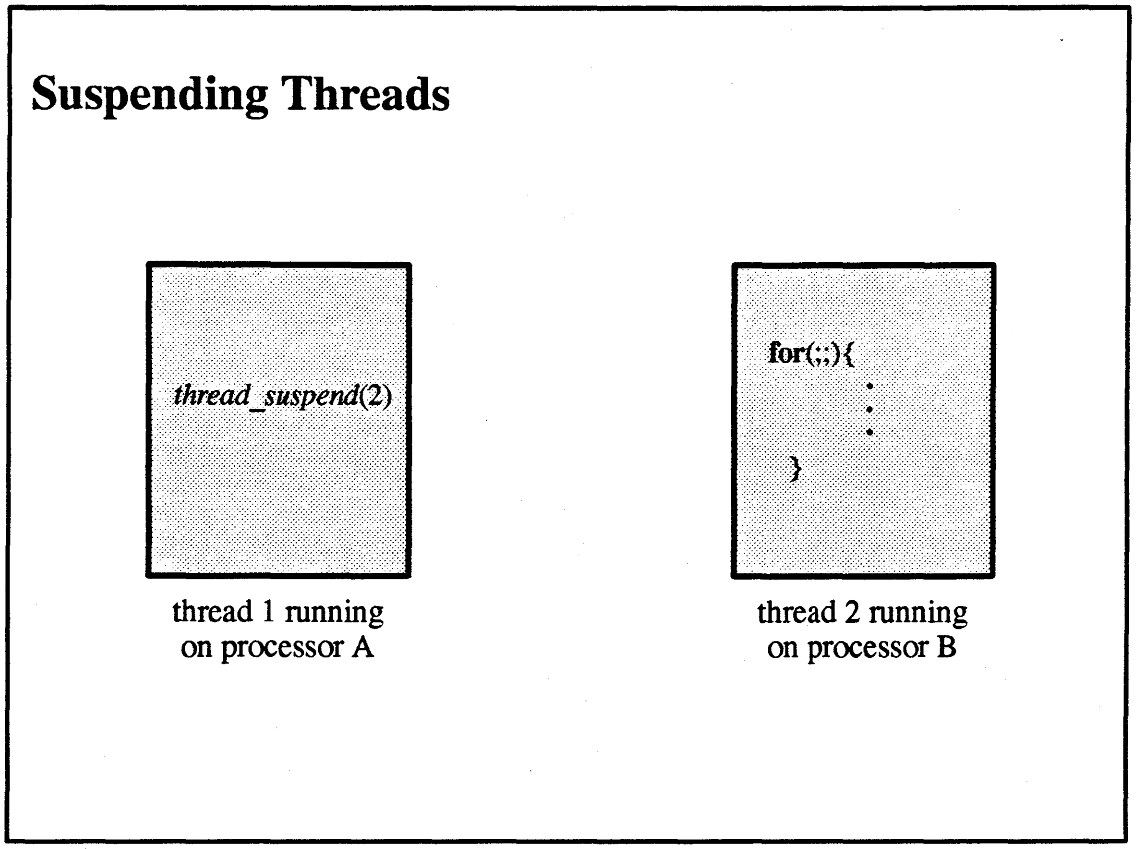
Blocking Threads Example

Initially thread B is using the object (has set its *in_use* field) and both thread A and thread B are running (or runnable). Thread A is attempting to use the object but finds by examining the *in_use* field that the object is being used by another thread. It indicates its intention to wait for the object by calling *assert_wait.* This sets the wait bit in thread A's state vector and queues thread A on the list of those threads waiting for the object. Since thread A has not called *thread_block,* it continues to run.

In the meantime, thread B finishes with the object, so it clears the *in_use* field and then calls *wakeup* to wake up those threads waiting for the object.

The effect of thread B's call to *wakeup* is to wake up thread A. However, thread A has not gone to sleep yet, so the wait bit is cleared in its state vector. Thread A subsequently calls *thread_block.* Since the wait bit is no longer set, thread A returns from *thread_block* immediately (if the call to wake up thread B had not taken place, then the call by thread A to *thread_block* would have put thread A to sleep—the run bit of its state vector would have been cleared, leaving only the wait bits set). Thread A now can test the *in_use* bit, see that it is clear, and set it itself.

2–23. Synchronization and Thread Management



Suspending Threads

thread_suspend(2)
for(;;){
}
thread 1 running on processor A
thread 2 running on processor B

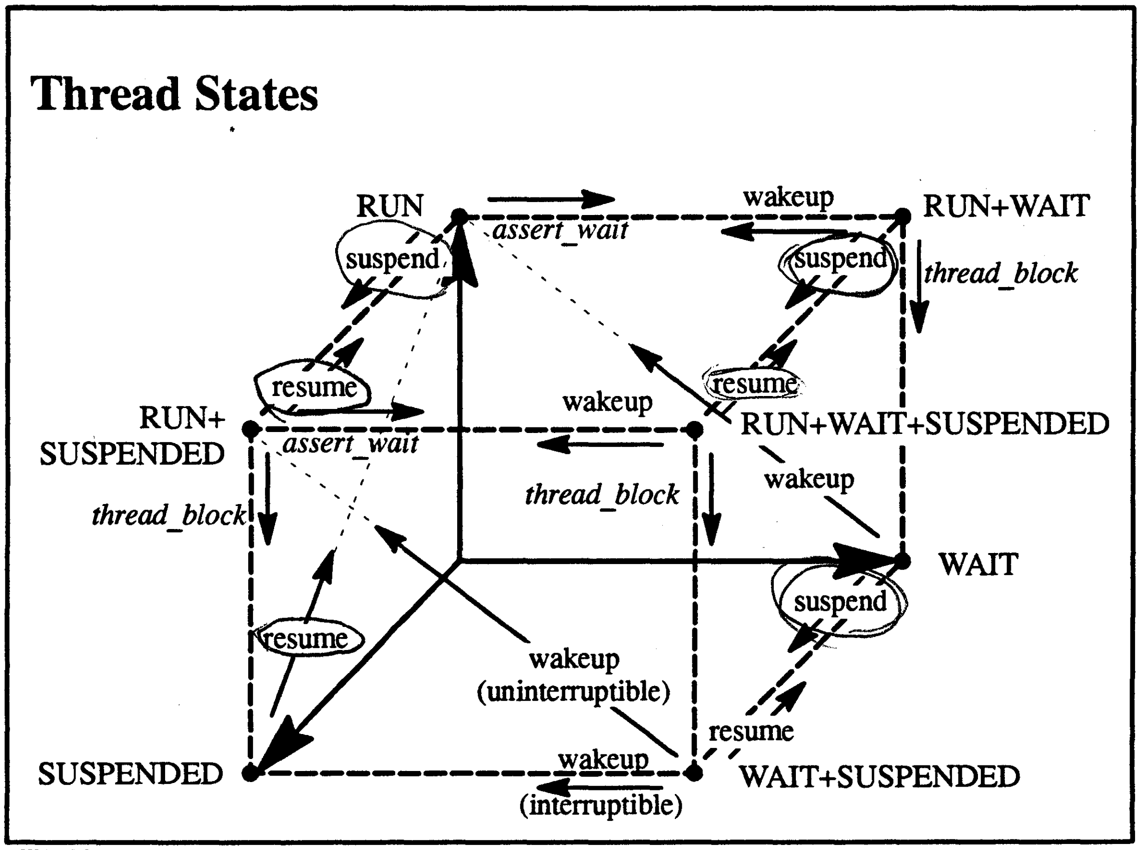
.H2596 2-23. 45 © 1991 Hewlett-Packard

Suspending Threads

The effect of suspending a thread is not necessarily immediate. In this picture, threads 1 and 2 are running on different processors, and thread 1 issues a *thread_suspend* call on thread 2. This call marks thread 2 to be suspended, but nothing is done to make this suspension happen immediately. Thread 2's processor will eventually switch to kernel mode, because of an interrupt or trap. The thread will then notice that it is marked to be suspended and suspend it accordingly. In general, the kernel will notice that a thread is to be suspended when that thread calls *thread_block* (which is done when the thread is about to return from kernel mode to user mode).

run = runnable

2–24. Synchronization and Thread Management



.H2596 2-24. 46 © 1991 Hewlett-Packard

Thread States

Represented as three bits in the thread structure: RUN, WAIT, and SUSPENDED

- RUN
  - — thread is either runnable or running
- WAIT
  - — thread is blocked, waiting for an event (it is on a wait queue)
  - — both interruptible and noninterruptible waits are supported (represented by another bit)
- SUSPENDED
  - — thread is suspended and thus not on any queue

— nested suspends are supported via a suspend count

— usually the result of a thread_suspend system call

- RUN+WAIT

— thread has just performed an *assert_wait*; either it will do a *thread_block* and switch to WAIT, or another thread will wake it up (before the *thread_block*) and switch it to RUN

- RUN+SUSPENDED

— thread has been set to be suspended; it will switch to SUSPENDED as soon as it either calls *thread_block* or returns to user mode

- WAIT+SUSPENDED

— a call to *thread_resume* switches thread to WAIT; if the wait is interruptible, the thread switches to SUSPENDED when it wakes up; otherwise it switches to RUN+SUSPENDED (i.e., the effect of the *thread_suspend* is delayed)

- RUN+WAIT+SUSPENDED

— a call to *thread_block* switches thread to WAIT+SUSPENDED, a *thread_resume* switches it to RUN+WAIT, a *wakeup* switches it to RUN+SUSPENDED

2–25. Synchronization and Thread Management

UNIX-Style Sleep

sleep(chan, disposition) Always INTERRUPTABLE

tsleep(chan, disposition, wmesg, timeout) ignored except for timeout

mpsleep(chan, disposition, wmesg, timeout, lockp, flags)

H2596 2-25. 47 © 1991 Hewlett-Packard

UNIX-Style Sleep

UNIX-style *sleep*s involve waiting for "one-shot" events. Traditionally, the kernel provided only a *sleep* call, but starting with 4.4BSD a *tsleep* (timed sleep) call was added as well. *Sleep*s instigated by calls to *sleep* are not interruptible (i.e. by signals). Calls to *tsleep* can be interruptible; interruptibility is specified by setting the PCATCH flag in the disposition argument. Unlike other UNIX implementations, the disposition argument has no other use in OSF/1.

As discussed earlier, a lock is necessary on multiprocessors to prevent a race between a *wakeup* and a *sleep* or *tsleep*, but, since there is no clear position for the unlock, these routines can only be used in unparallelized code, i.e., only in situations where all relevant activities are guaranteed to take place on the same processor. *Mpsleep* is a multiprocessor-safe version of *sleep* and *tsleep* that takes a pointer to a lock as an argument. *Mpsleep* contains calls to *assert_wait* and *thread_block*, and the lock is unlocked between these calls.

The *wmesg* argument to *tsleep* and *mpsleep* is a character string indicating why the thread is sleeping. Its only purpose is for display when a user types *control-T* to see the states of the foreground processes.

2–26. Synchronization and Thread Management

Waking Up

clear_wait(thread, result, interruptible_only)

thread_wakeup_one(event)

thread_wakeup_with_result(event, result)

Possible results:

THREAD_AWAKENED

THREAD_TIMED_OUT

THREAD_INTERRUPTED

THREAD_SHOULD_TERMINATE

THREAD_RESTART

.H2596 2-26. 48

Waking Up

Wakeup routines:

- *clear_wait*: wakes up a particular thread. If the *interruptible-only* flag is set, then the thread is awakened only if it is in an interruptible sleep (this flag is used, for example, to wake up a thread conditionally in response to a signal).

- *thread_wakeup_one*: wakes up the first thread waiting for a particular event, and sets the *wait_result* to THREAD_AWAKENED.

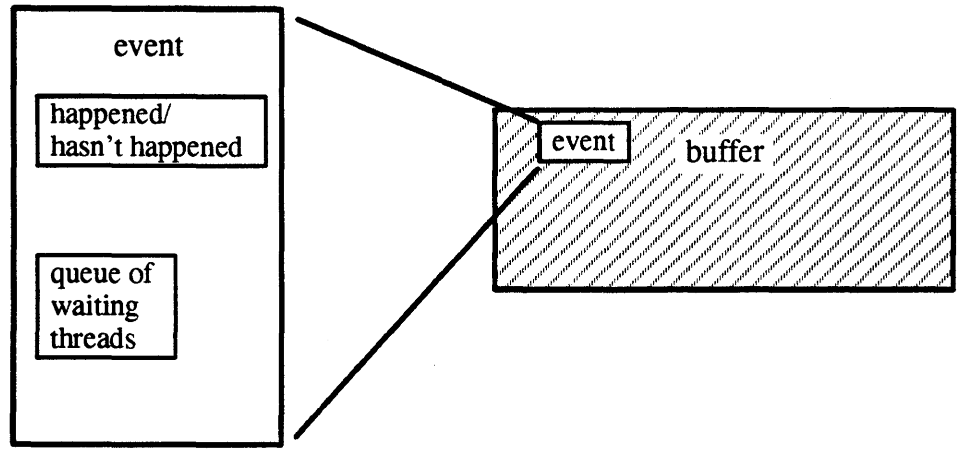
- *thread_wakeup_with_result*: wakes up all threads waiting for a particular event, and sets their *wait_result*s to the second argument.

Whenever a thread is woken up, the cause of the *wakeup* is put in the *wait_result* field of the thread's *thread* structure. The five standard results are as follows:

1. THREAD_AWAKENED: returned if the event for which the thread was waiting actually occurred.
2. THREAD_TIMED_OUT: returned if the timeout period expired (e.g. as set in *tsleep*).
3. THREAD_INTERRUPTED: returned if the thread was interrupted by a signal, and this caused the *wakeup*.
4. THREAD_SHOULD_TERMINATE: returned if a signal forces the termination of the thread.
5. THREAD_RESTART: returned if a thread was waiting for some event that turns out to be no longer relevant, e.g. a thread is waiting on a condition involving a leaf of a tree, but a structural change occurs higher up in the tree. This result notifies the thread that it should reevaluate its circumstances.

2–27. Synchronization and Thread Management

Events



.H2596 2-27. 49 © 1991 Hewlett-Packard

Events

Events provide an improvement of the common case of the UNIX *sleep* call. A thread can test whether an event has been *posted* and, if it has not, then *wait for the event* to be posted. When the event is posted, it stays posted until explicitly cleared.

Kernel subroutines:

- *event_clear*(event): mark an event as *hasn't happened*
- *event_posted*(event): return whether the event has happened
- *event_wait*(event): wait until the event has happened
- *event_post*(event): mark the event as *has happened*

Note that the implementation guarantees that there will not be a race between *event_post* and *event_wait*: a thread calling *event_wait* returns soon (if not immediately) after *event_post* is called.

As an example, consider operations on a buffer. One thread starts I/O to fill the buffer, but before doing so, *clears* the event that would indicate the buffer is filled. Other threads might test for this event, find that the event has not been posted, and thus wait (inside of *event_wait).* When the first thread finishes filling the buffer, it then posts the event, which both wakes up all threads waiting for the buffer and marks the buffer as filled for any subsequent thread that needs its contents.

2–28. Synchronization and Thread Management

Read-Write Locks - exported

lock_init(lock)

lock_read(lock)

lock_write(lock)

lock_done(lock)

lock_read_to_write(lock)

lock_write_to_read(lock)

lock_try_write(lock)

lock_try_read(lock)

lock_set_recursive(lock)

lock_clear_recursive(lock)

H2596 2-28. 50

Read-Write Locks

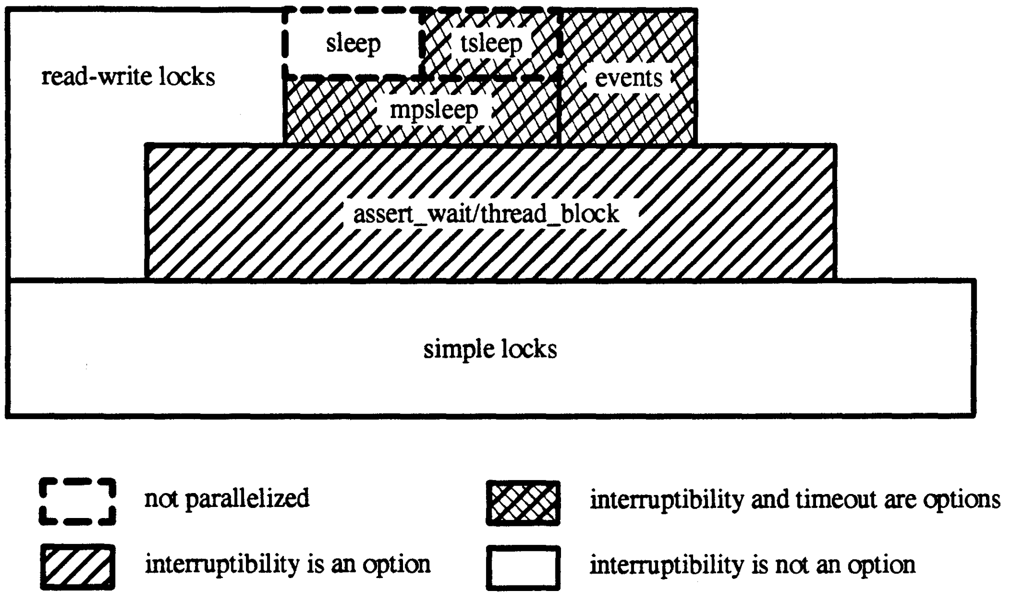
Read-write locks provide reader-writers-type synchronization, i.e., any number of threads may hold a lock for reading, but if a thread holds a lock for writing, no other thread may hold it for either reading or writing. A read-write lock may be configured to be either a blocking lock or a spin lock. In most cases, it is a blocking lock, i.e., threads waiting for the lock will yield their processor. But, particularly when it is used in the interrupt context, it may be a spin lock.

In some situations, it may be convenient to use a read-write lock *recursively,* i.e., a thread may "take" a lock even if it already has it. This notion is useful in situations in which a thread possesses a lock but is calling a routine that causes it to take the lock again (if the lock is not set to be recursive, this produces an immediate deadlock situation).

On multiprocessors, threads do not immediately block while waiting for a lock. Instead, they test the lock for a number of times equal to the value of *lock_wait_time* (a global variable whose value is typically set to 100) and then yield their processor, if necessary.

2–29. Synchronization and Thread Management

Synchronization in OSF/1: Summary


sleep
tsleep
read-write locks
events
mpsleep
assert_wait/thread_block
simple locks
not parallelized
interruptibility and timeout are options
interruptibility is an option
interruptibility is not an option

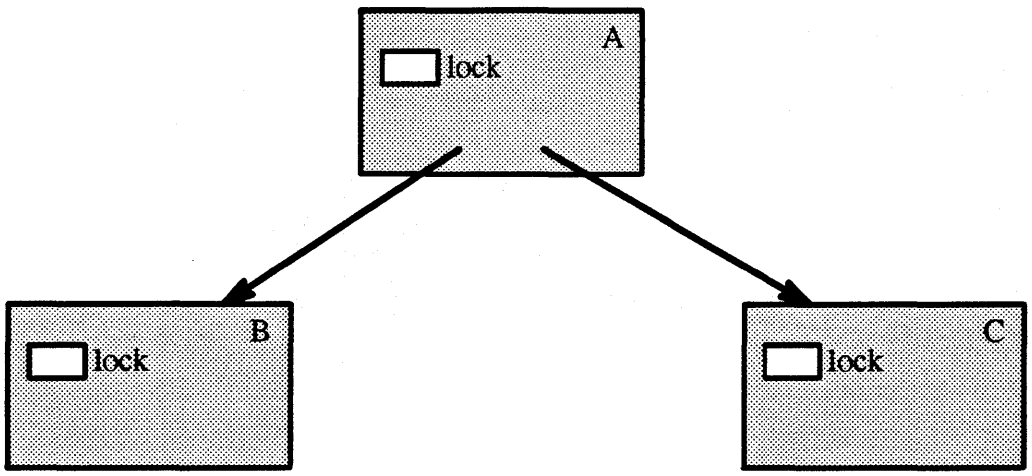
H2596 2-29. 51 © 1991 Hewlett-Packard

Synchronization in OSF/1: Summary

This diagram summarizes synchronization in the OSF/1 kernel and shows the layering. Note that simple locks are not used on uniprocessors.

2–30. Synchronization and Thread Management

Avoiding Deadlock



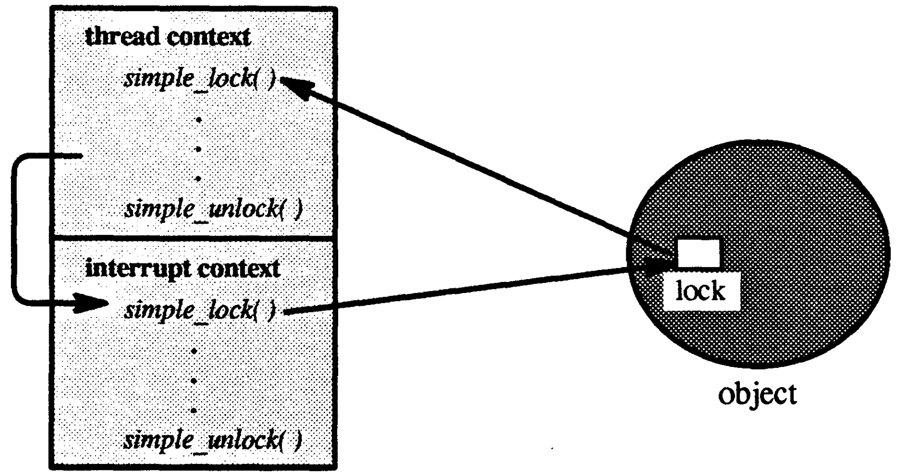
H2596 2-30. 52 © 1991 Hewlett-Packard

Avoiding Deadlock

In many cases it is necessary to hold two or more locks. Unless these locks are taken with care, there is a potential for deadlock. To avoid deadlock, locks are usually taken in a prescribed order by all threads (typically "downwards"). However, it is occasionally necessary to take locks out of order. Deadlock is avoided in this case by using conditional requests for locks. For example, if the prescribed order is "take lock A, then take lock B," but one has lock B and desires lock A, then one should make a conditional request for lock A. If the request fails, then one should release lock B (thus avoiding deadlock) and try again.

2–31. Synchronization and Thread Management

Taking Locks in the Interrupt Context



H2596 2-31. 53

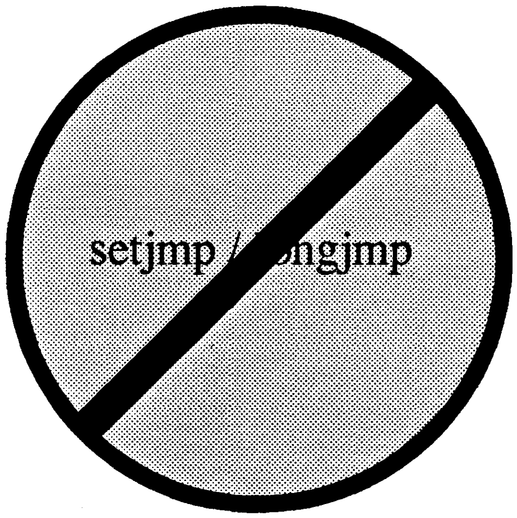
Taking Locks in the Interrupt Context

Locks can be taken in the interrupt context, but only with some care. The picture illustrates a situation to be avoided. A thread is interrupted after it has taken a lock. The interrupt handler (executing in the interrupt context) then attempts to take the same lock, and deadlock results: the interrupt handler cannot return from the interrupt context until it takes the lock, and the thread cannot release the lock to the interrupt handler until the interrupt handler returns and lets the thread have the processor.

The solution to this problem is straightforward. If a lock can be taken in the interrupt context at an interrupt-priority level of *n*, then whenever the lock is taken in any other context, the interrupt-priority level must be at least as high as *n*.

2–32. Signals and Exception Handling

Signals and Blocked Threads



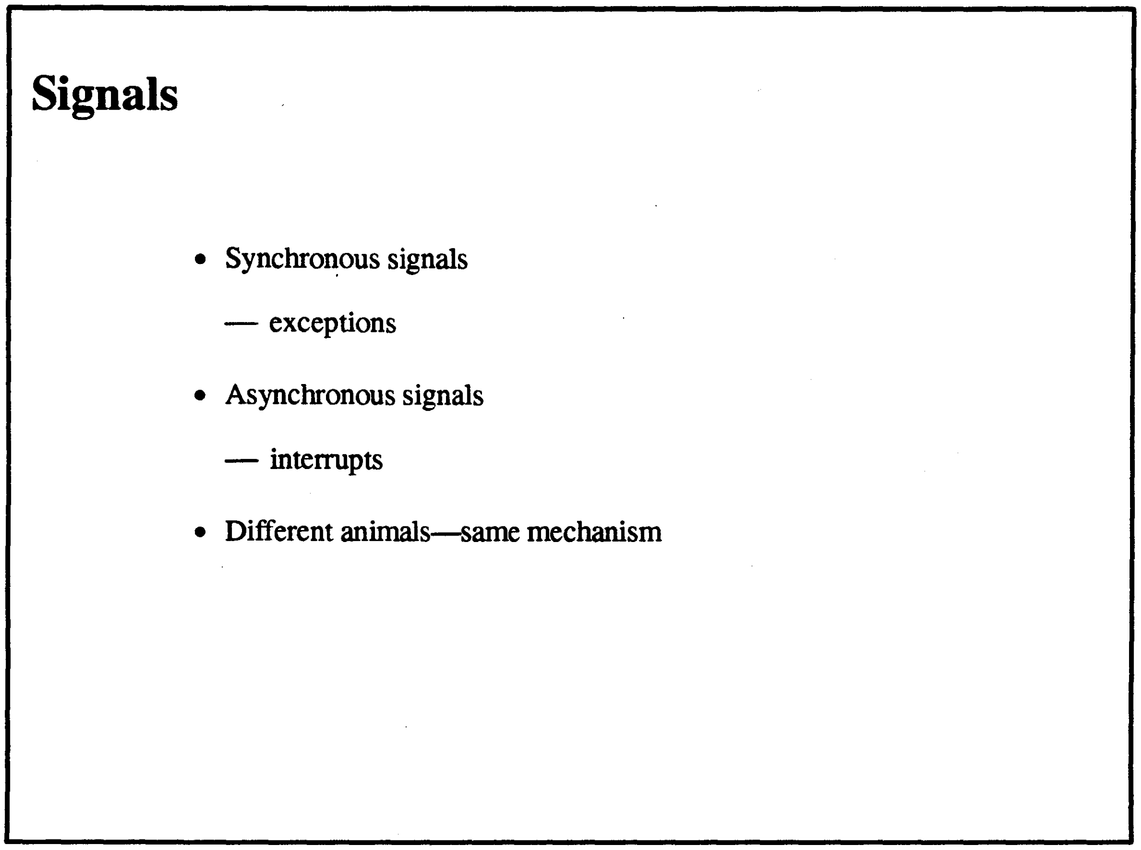
H2596 2-32. 54

Signals and Blocked Threads

All UNIX synchronization calls (*tsleep*, etc.) return to their caller even if they have been interrupted by a signal. The *tsleep* call (which is interruptible only if PCATCH is set) returns one of four possible values:

VALUE	EVENT
0	normal wakeup
EINTR	interrupted and the system call should return the EINTR error code
ERESTART	interrupted and the system call should be restarted
EWOULDBLOCK	the *sleep* timed out

2–33. Signals and Exception Handling



Signals

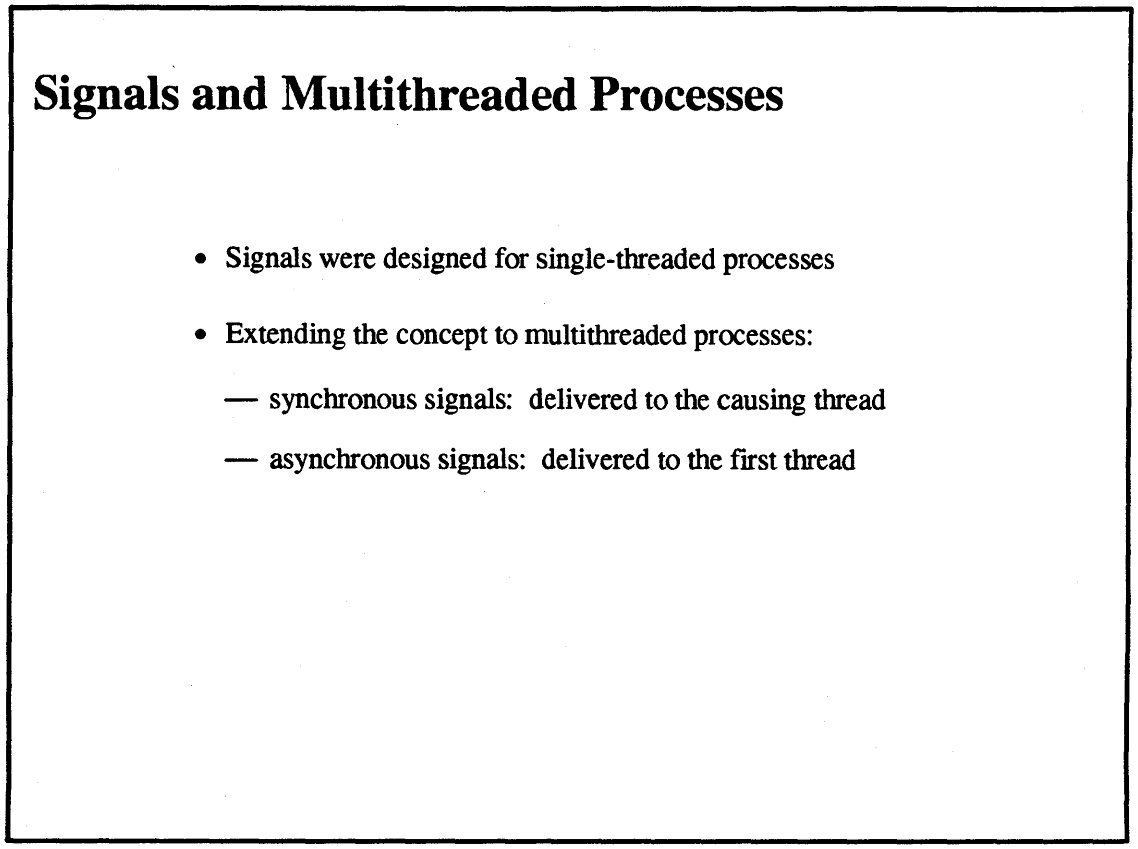
- Synchronous signals
  - — exceptions
- Asynchronous signals
  - — interrupts
- Different animals—same mechanism

.H2596 2-33. 55 © 1991 Hewlett-Packard

Signals

Signals serve a dual purpose in UNIX. They are used to inform processes about *exceptions* (e.g. addressing errors), and they are used to inform processes about *external events* (e.g. the typing of an interrupt character, a signal sent from another process). For each signal, a process may set up a *handler* (*catch* the signal), *ignore* the signal, or chose the *default* action (which may be to abort the process, stop the process, resume the process, or ignore the signal).

2–34. Signals and Exception Handling



Signals and Multithreaded Processes

- Signals were designed for single-threaded processes
- Extending the concept to multithreaded processes:
  - — synchronous signals: delivered to the causing thread
  - — asynchronous signals: delivered to the first thread

H2596 2-34. 56

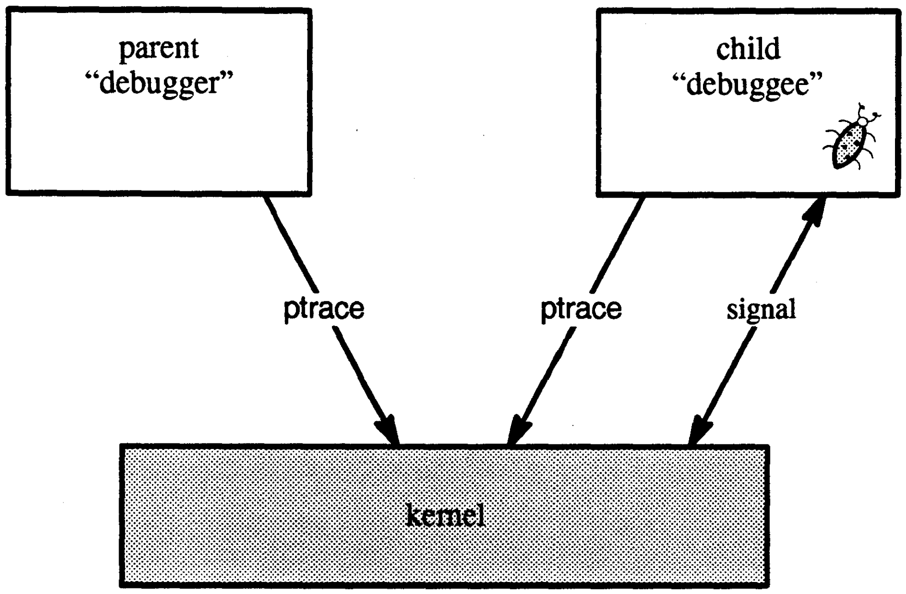
Signals and Multithreaded Processes

Most of the signal-handling state is kept with the process (as opposed to the thread): signal mask, signal disposition, and vector of pending signals. Per-thread signal disposition is kept for synchronous signals. There is no universally accepted semantics for generalizing signals for multithreaded processes.

It is clear to whom a synchronous signal (i.e. exception) should be sent. What is not so clear is to whom an asynchronous signal (i.e. interrupt) should be sent. In OSF/1, such signals are delivered to the first thread that was created within the process (if this thread has terminated, then to the second thread, etc.).

2–35. Signals and Exception Handling

Debugging with Ptrace



H2596 2-35. 57 © 1991 Hewlett-Packard

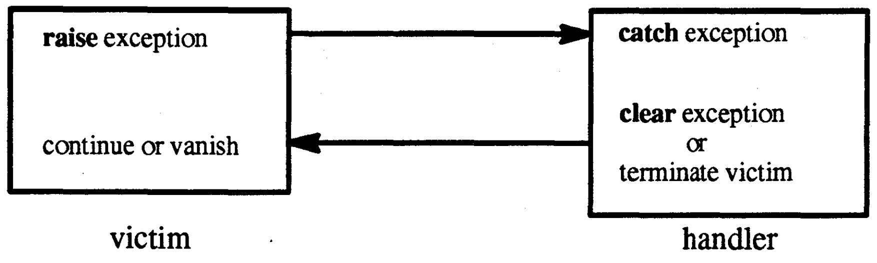
Debugging with Ptrace

A process may "allow" its parent to debug it by the use of the ptrace system call. A child issues a ptrace with an argument of zero, thereby turning on the *trace* bit in its *proc* structure. From that point on, whenever it receives a signal, it stops so that its parent (the debugger) may examine and possibly modify it.

The parent debugger process may wait for a child to stop via the wait system call. The parent may send requests to the child by issuing ptrace calls with positive arguments. With ptrace, it may examine and modify the child's memory and registers, and control the child's response to signals. The data transfer is performed using Mach facilities for reading and writing to another task's address space.

2–36. Signals and Exception Handling

Exception Handling in Mach



H2596 2-36. 58 © 1991 Hewlett-Packard

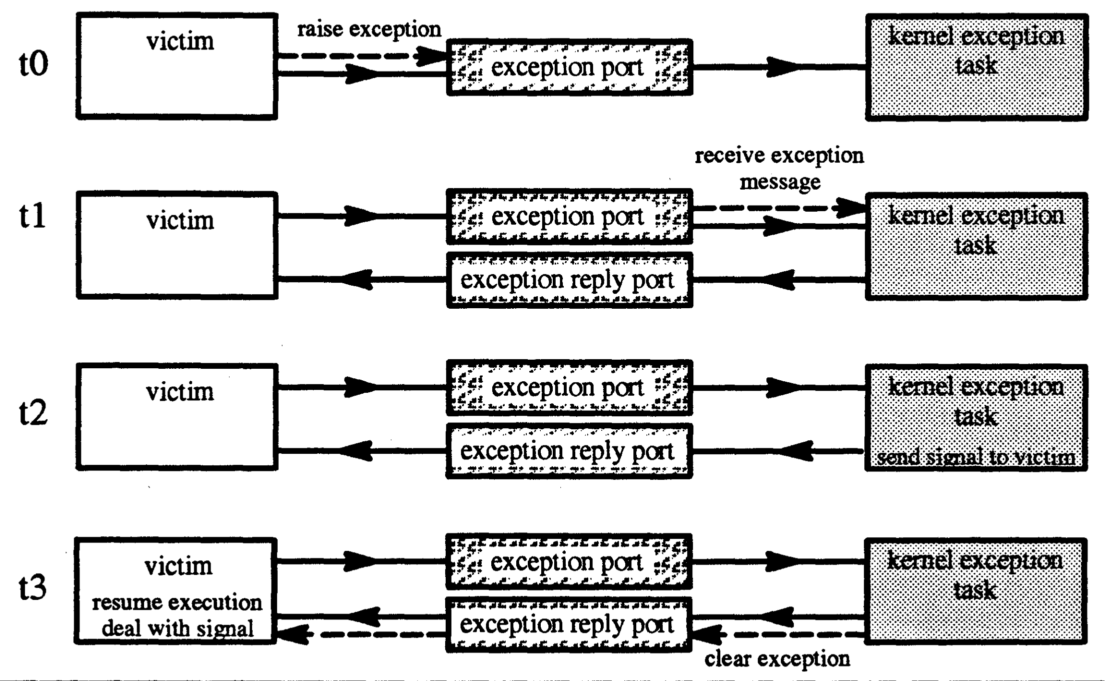
Exception Handling in Mach

Components of exception handling:

1. victim thread: *raise* the exception
2. victim thread: *wait* for handler to complete
3. handler: *catch* the exception, i.e. receive notification of the exception and perform appropriate actions
4. handler: either *clear* the exception, i.e. resume the waiting victim, or *terminate* the victim thread

2–37. Signals and Exception Handling

Exceptions: Exporting the Interface



.H2596 2-37. 59 © 1991 Hewlett-Packard

Exceptions: Exporting the Interface

Each task has send rights to an *exception port* that it inherits from its parent. The receive rights for the default task's exception port are held in the kernel by a routine that converts exceptions into UNIX signals. Associated with a thread may be a *thread exception port*, to which the task has send rights. By default, there is no such port, but a thread may establish one. If the thread exception port exists, it is used instead of the task's exception port.

The slide illustrates the sequence of events during exception handling with the default task exception handler:

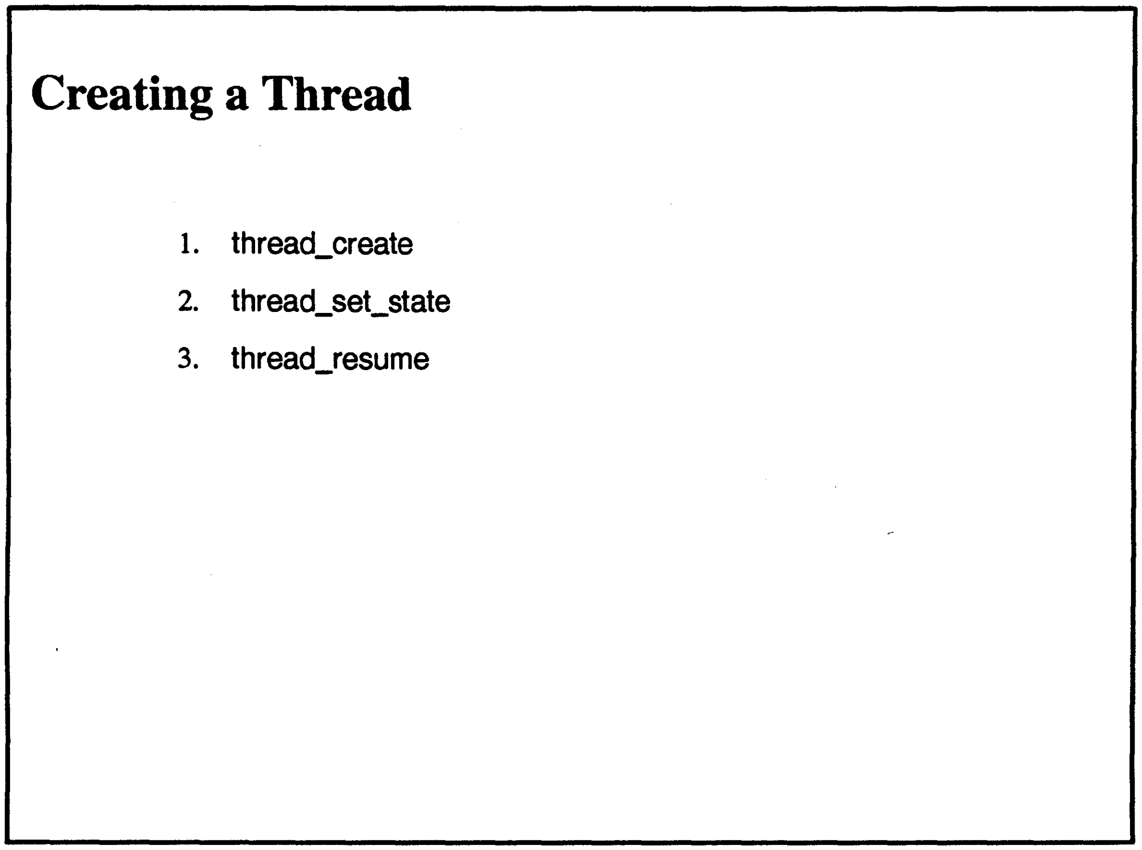
t0. the victim raises an exception (e.g., divides by zero); a message is sent through the exception port.

t1. an exception reply port is created if one does not already exist, and an exception message is received by the (single) thread in the kernel exception task, which is a subtask of the kernel task.

t2. the thread in the kernel exception task translates the exception into the UNIX signal and marks this signal as *pending* in the victim.

t3. the thread in the kernel exception task sends a *clear exception* message through the exception reply port; this has the effect of waking up the victim, which then discovers that it has a signal and deals with it in its own context.

2–38. Threads



Creating a Thread

1. thread_create
2. thread_set_state
3. thread_resume

H2596 2-38. 60 © 1991 Hewlett-Packard

Creating a Thread

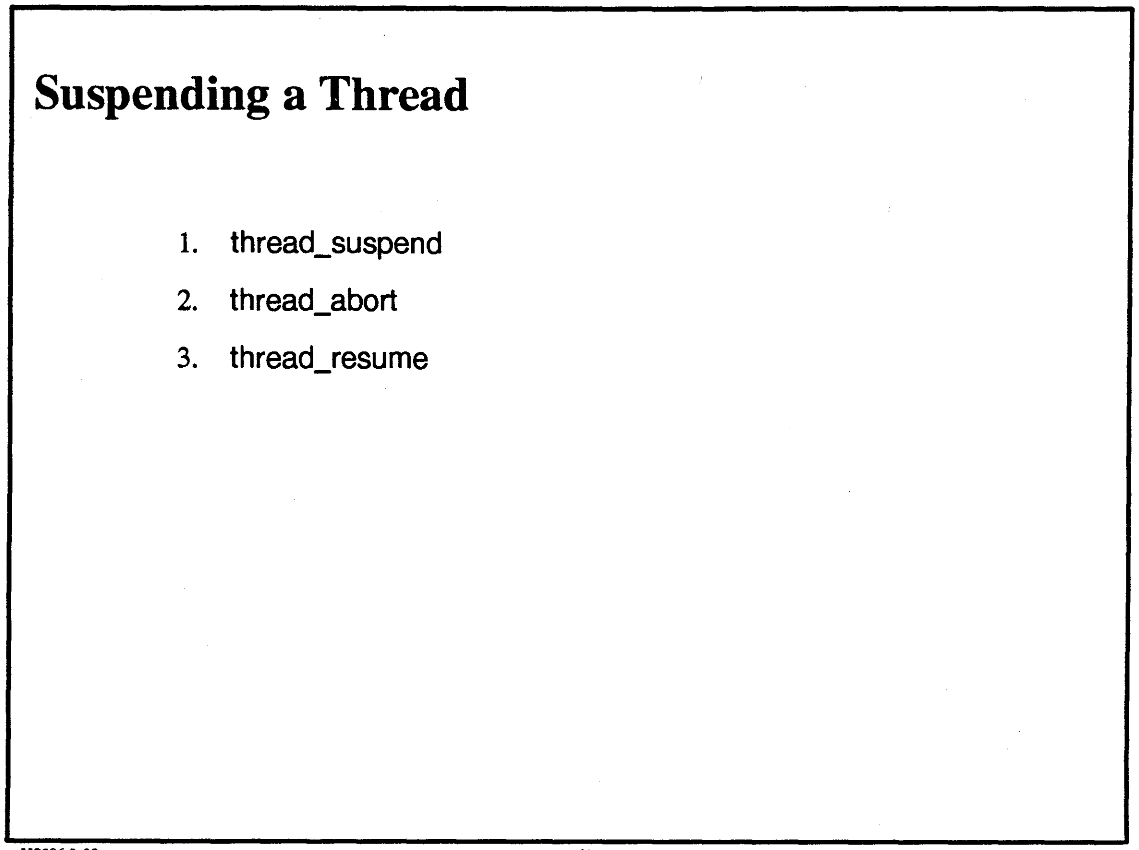
Creating a thread takes a surprising number of system calls.

1. The thread_create call establishes the kernel context of a thread, but leaves the establishment of the user context to the caller. Thus the new thread is created in the *suspended* state.

2. The user then establishes a user context for the thread with the thread_set_state call. This may involve giving initial values for all of the general-purpose registers in the thread's user context, which has the effect of giving the thread a user stack and an initial value for its program counter. Thus the management of stack space, and the semantics of what a new thread should do, are left to the user.

3. The final step is for the user to put the thread into a runnable state by calling thread_resume.

2–39. Threads



Suspending a Thread

1. thread_suspend
2. thread_abort
3. thread_resume

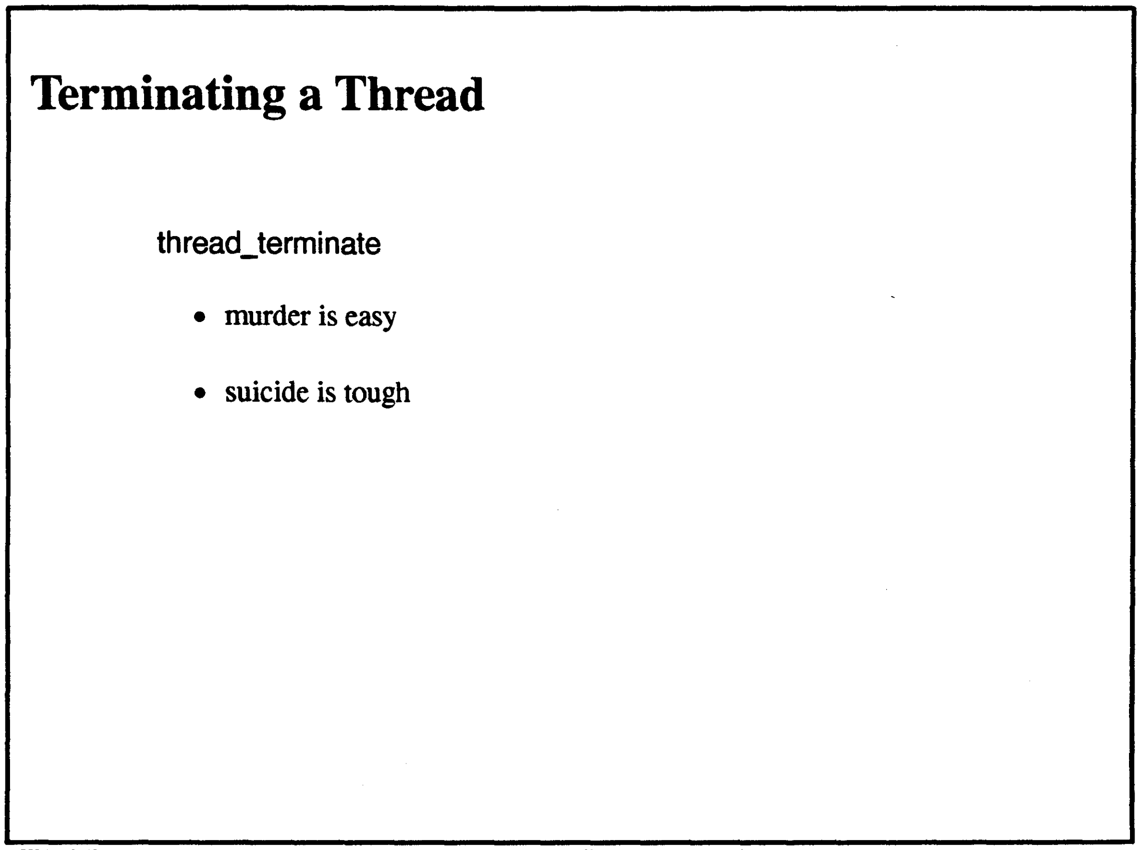
H2596 2-39. 61

Suspending a Thread

Simple suspension and resumption are straightforward: the user just calls the appropriate system calls. Changing the suspended thread's behavior is more difficult: the thread might be suspended in the kernel (in mid-system call or in some other sort of trap), but the user can only directly modify the thread's user context. When the thread is resumed, the user state might be modified as part of completing the trap, thus overriding any changes made to the user state.

To allow the deterministic modification of a thread's user context, the system must suspend the thread at the point at which it is about to return to the user, i.e., after any modifications to its user context have been made within the trap. However, if the thread is blocked, i.e. in the WAIT state, at the time at which it is suspended, then the thread must be forced to go to the point at which it is just about to return to user mode. This forcing is accomplished by the thread_abort system call. The effect of this call is to wake the target thread up if it is waiting interruptibly. This thread will then do any necessary cleanup and then effectively abort the system call.

2–40. Threads



Terminating a Thread

thread_terminate

- murder is easy
- suicide is tough

.H2596 2-40. 62 © 1991 Hewlett-Packard

Terminating a Thread

Terminating another thread is straightforward: the victim thread is stopped at a clean point, i.e., a point at which it is holding no locks, and then eliminated.

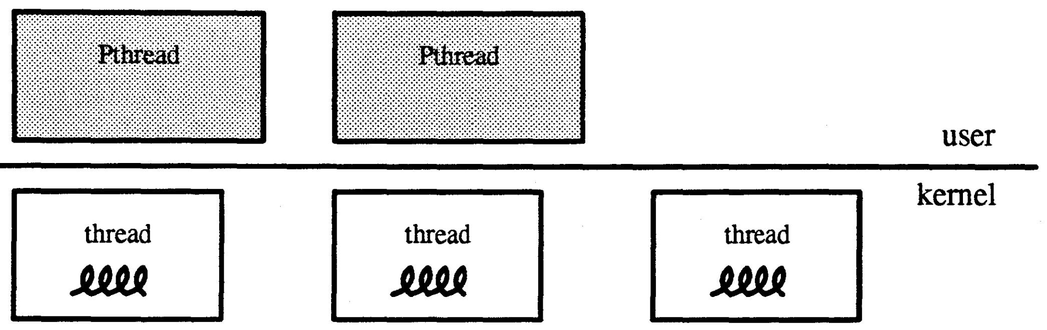
Terminating oneself presents a problem. Part of termination involves freeing a thread's kernel stack and thread structure. However, doing so requires a call to a subroutine, and calls to subroutines in the kernel involve the use of the caller's kernel stack. On a multiprocessor, the instant that a stack is freed it may be allocated to some other thread. The suicidal thread is still using its stack as it returns from the stack liberation routine, but now a new thread is using the same stack, and total chaos ensues.

Thus a thread cannot terminate itself directly. Instead. the thread is put on a queue that is examined by the special-purpose kernel *reaper* thread, which cleans up the suicidal thread *after* that thread has yielded the processor.

system API

2–41. Threads

Pthreads



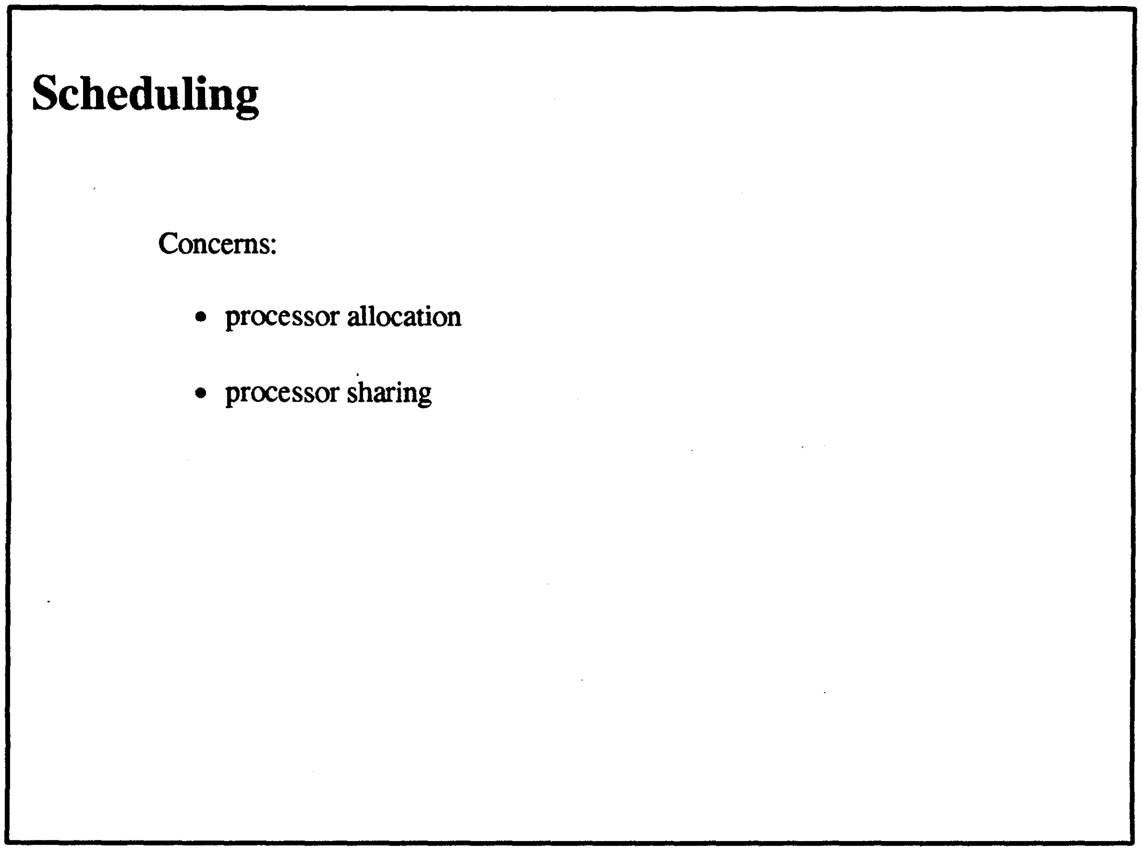
H2596 2-41. 63 © 1991 Hewlett-Packard

Pthreads

The intended programmer interface to multithreaded processes is provided by the POSIX threads (Pthreads) package, which is implemented as a user-level library. Though the OSF/1 kernel interface for threads may or may not become standard, it is used to support the Pthreads interface, which is standard.

The intent is that the programmers manage threads using Pthreads. Pthreads maintains a cache of kernel threads. When a Pthreads thread is terminated, the underlying kernel thread is merely suspended, and can be reused to support the next Pthreads thread.

2–42. Scheduling



Scheduling

Concerns:

- processor allocation
- processor sharing

H2596 2-42. 64

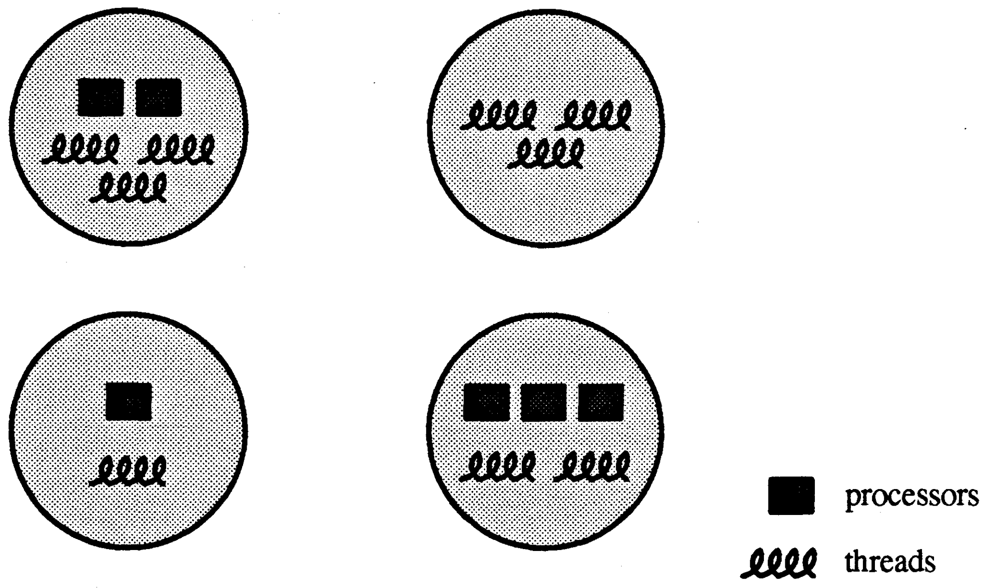
Scheduling

Processor allocation involves user-controlled partitioning of the processors to satisfy application requirements.

Processor sharing deals with two concerns: processors must be shared equitably among the running threads, but preferential treatment must be given to "important" threads.

2–43. Scheduling

Processor Sets



H2596 2-43. 65 © 1991 Hewlett-Packard

Processor Sets

Processor sets are a mechanism for processor allocation supplied in the OSF/1 kernel. The intent is that a (privileged) user-level server should supply the policy for processor allocation. The user-level server will establish processor sets and manage their contents in response to requests from ordinary threads.

Processors are partitioned into *containers* called *processor sets*: each container holds zero or more processors, and each processor is in exactly one container. Threads are also assigned to these containers: threads may run only on a processor and its container (processor set). By default, there is exactly one processor set containing all processors and threads.

Examples of use:

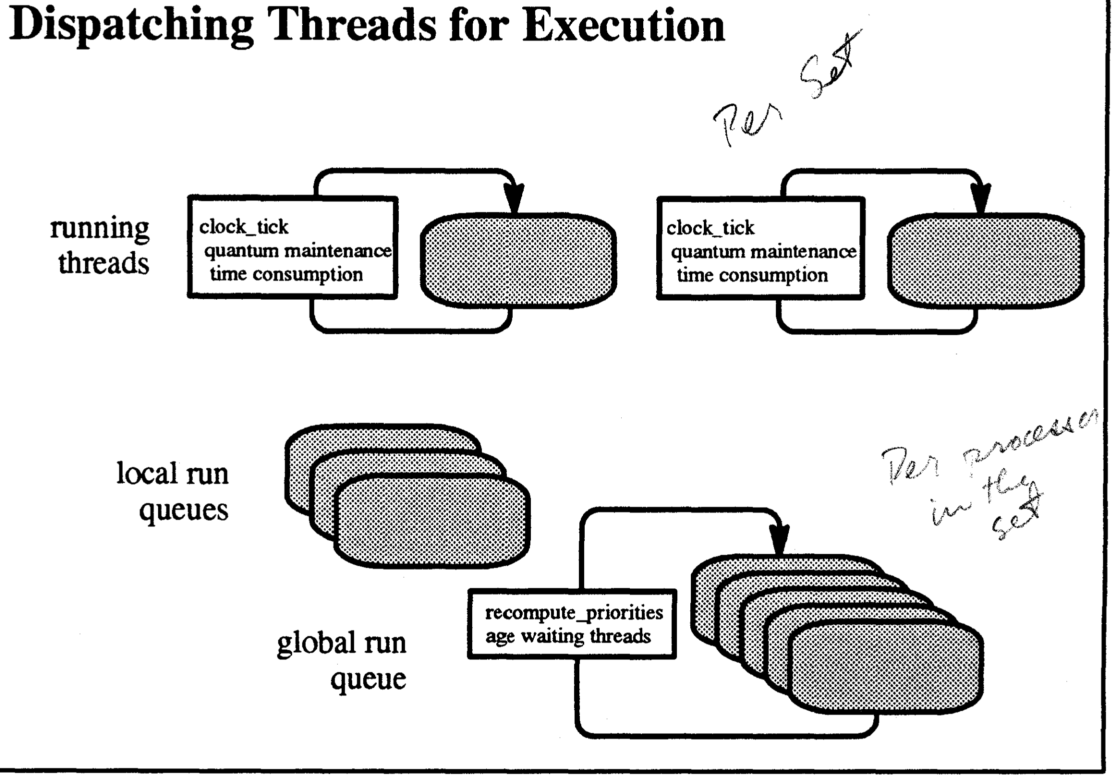
- *Gangs*. A set of cooperating threads can be given a set of processors for their exclusive use.

- *Non-homogeneous multiprocessors.* Multiprocessor might have two classes of processors, one with floating-point hardware, one without. Processor sets could be used to run those threads with extensive floating-point requirements on the appropriate processors.

For further discussion, see Black, 1991.

2–44. Scheduling

Dispatching Threads for Execution



H2596 2-44. 66 © 1991 Hewlett-Packard

Dispatching Threads for Execution

OSF/1 maintains two types of run queues: a *global run queue* (one per processor set) for threads with no processor affinity (the usual case), and *local run queues* for threads with processor affinity (e.g., threads involved in unparallelized UNIX system calls and in device handling on unsymmetric hardware). Currently, the processor known as the *UNIX master* has the only local run queue. This queue is used solely to support those few parts of the kernel that have not been parallelized.

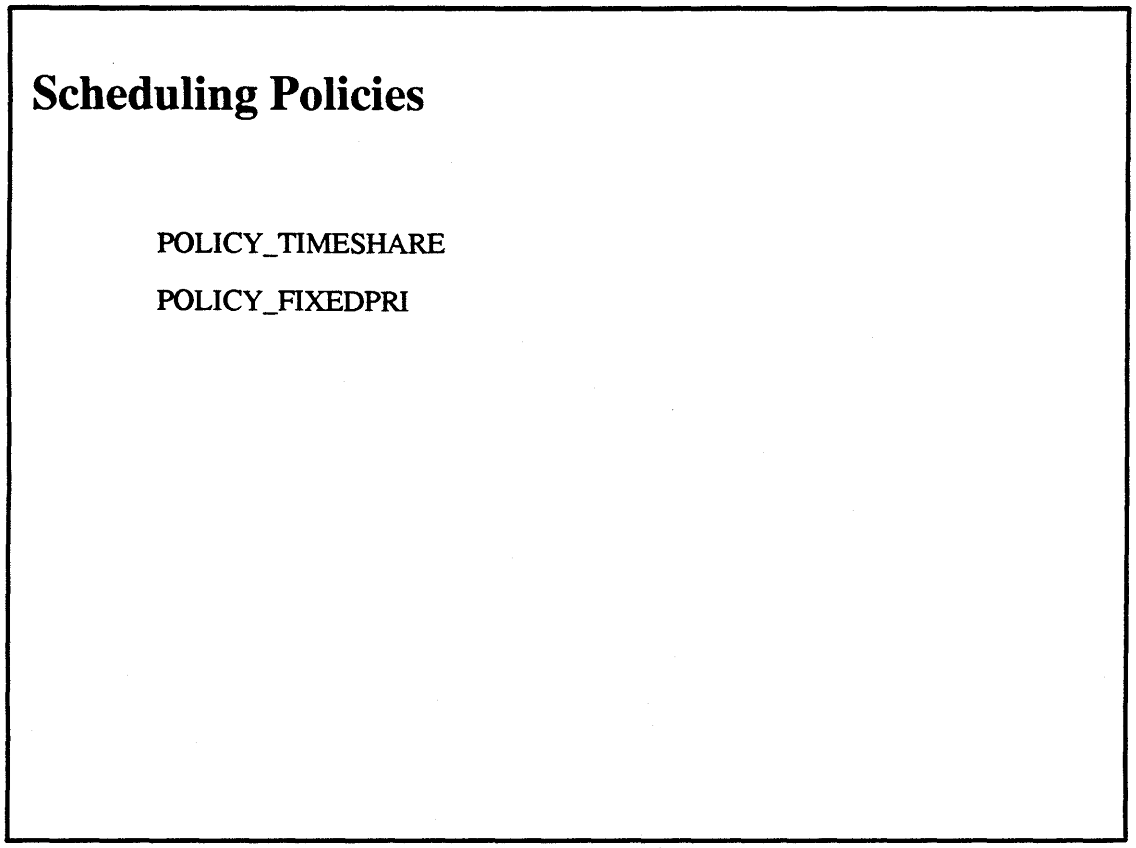
When a processor needs work, it first checks its local run queue (if any) and then its global run queue; finally, if it finds no work to do, it runs a special kernel *idle thread.*

An important case is the dispatching of a runnable thread when there are idle processors. To speed this dispatch, the system maintains a list of the idle processors. If this list is not empty when a thread is made runnable, then the agent making the thread runnable selects the first processor in the idle list and quickly dispatches that processor to the newly runnable thread.

To call & release the UNIX Master processor { unix_master()
unix_release()

A further optimization applies to those architectures in which it is advantageous that a newly runnable thread resume execution on the processor on which it last ran. (This technique is conditionally compiled into the kernel: it is used only when architecturally relevant.) Associated with each thread is a reference to its last processor; this processor is chosen, if available, when the thread runs again.

2–45. Scheduling



Scheduling Policies

POLICY_TIMESHARE

POLICY_FIXEDPRI

.H2596 2-45. 67

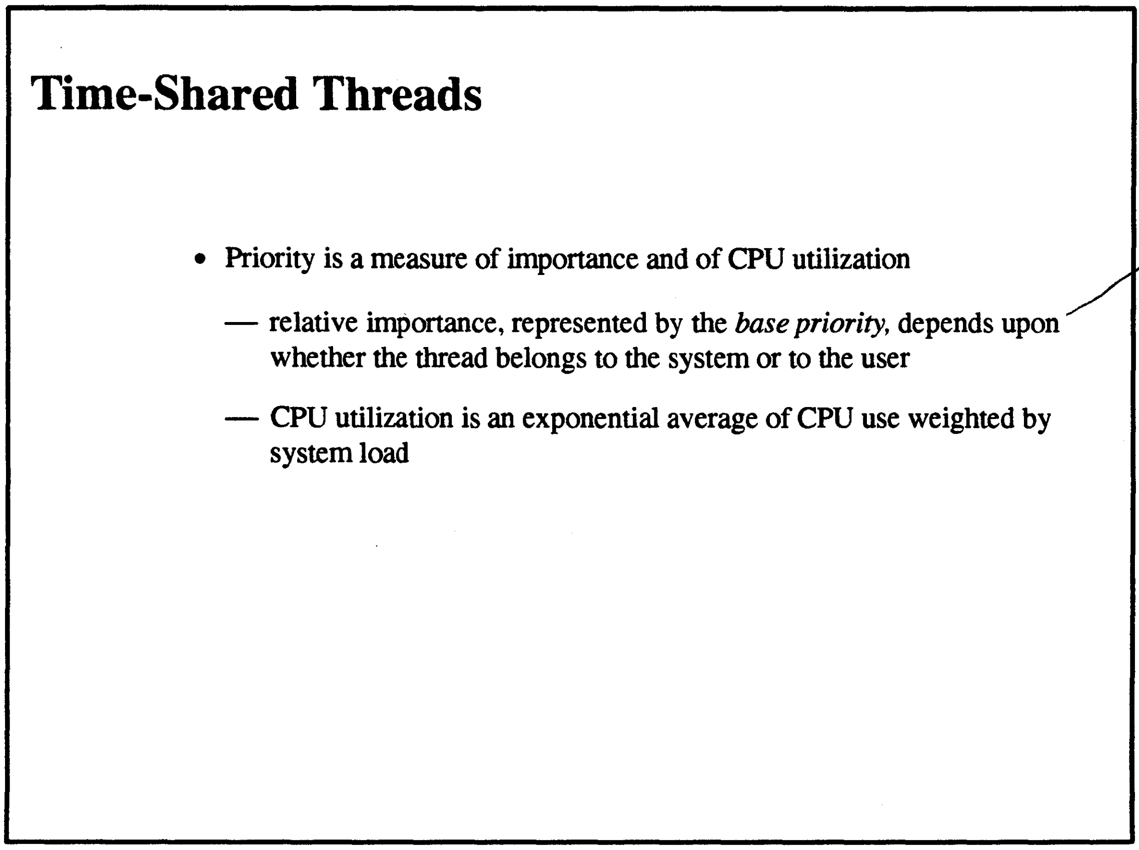
Scheduling Policies

Two scheduling policies are supported: a *time-shared policy* and a *fixed-priority policy*. These are properties both of the thread, i.e., how it is scheduled, and of the processor set, i.e., which policies are allowed. The primary goal of the time-shared policy is the equitable sharing of the processors among the various threads. The goal of the fixed-priority policy is to provide preferential treatment to particular threads.

Each thread has a *base priority* and a *scheduler priority*, both in the range between 0 and 31. The base priority is fixed for each thread—it represents the thread's "importance" (as is usual in UNIX, numerically low priorities are "better" than numerically high priorities). The *scheduler priority* is equal to the base priority for fixed-priority threads. However, for time-shared threads, the scheduler priority is computed from the base priority by adding a (positive) value based on processor usage.

UNIX's *nice* routine (which uses the (UNIX) getpriority and setpriority system calls) affects the calling thread's base priority.

2–46. Scheduling



Time-Shared Threads

- Priority is a measure of importance and of CPU utilization
  - — relative importance, represented by the *base priority*, depends upon whether the thread belongs to the system or to the user
  - — CPU utilization is an exponential average of CPU use weighted by system load

0-31

H2596 2-46. 68 © 1991 Hewlett-Packard

Time-Shared Threads

The basis for computing the *weighted average* of a thread's CPU usage is the following formula:

*sched_average = current_usage * load + (5/8) * sched_average*

where *current_usage* is the CPU time used in the past second and *load* is the current (averaged) measure of load (based on the length of the run queue).

The effect of the weighted average is that CPU seconds are more costly the more they are in demand.

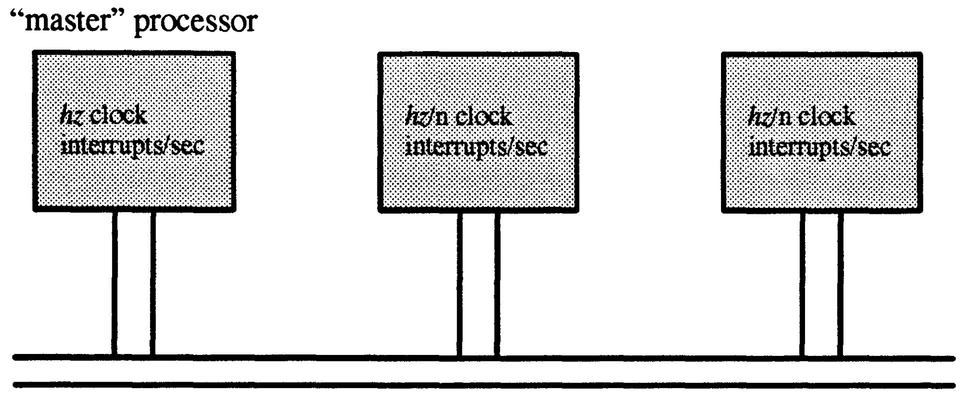
OSF/1 uses a distributed approach to compute this average efficiently: the *sched_average* computation is done in the clock-interrupt context for the currently running threads. Every two seconds all threads in the global run queues are "aged" by multiplying their *sched_average*s by $(5/8)^n$, where n is the number of seconds since this computation was last performed (each thread has a private count of seconds that is compared with the system count of seconds, maintained in the global variable). Threads joining the run queue have their priorities recomputed so as to "catch up."

The *sched_average* decays to 0 after 30 seconds of no processor use. Thus a thread's scheduler priority reverts to the thread's base priority after the thread has been idle for more than 30 seconds.

No floating-point arithmetic is involved in these computations: numbers are scaled and arithmetic is performed using shifts and adds. No floating point is ever used in the kernel; thus floating-point registers need not be saved across system calls.

2–47. Scheduling

Measuring Time



.H2596 2-47. 69 © 1991 Hewlett-Packard

Measuring Time

The basic unit of time is given as *hz*: number of clock ticks per second. The number of clock ticks per second is architecture-dependent, but is typically 100.

- On a uniprocessor: *hz* clock interrupts/second
- On a multiprocessor: the master processor's clock interrupts *hz* times a second; the other processors' clocks may be set to interrupt at an integral multiple slower (but their clock interrupt rates in the Encore Multimax reference port is identical to that of the master processor)
- On some architectures, hardware timers are used to measure per-thread processor time accurately
- On the others, per-thread processor time is a count of clock ticks

2–48. Scheduling

Time Slicing

- A thread is assigned a processor for a particular time period (or time quantum)
- During this period, it is not preempted unless a thread with a better scheduler priority is made runnable
- Threads are not preempted while executing in kernel mode
- For a multiprocessor, an adjustable quantum is used

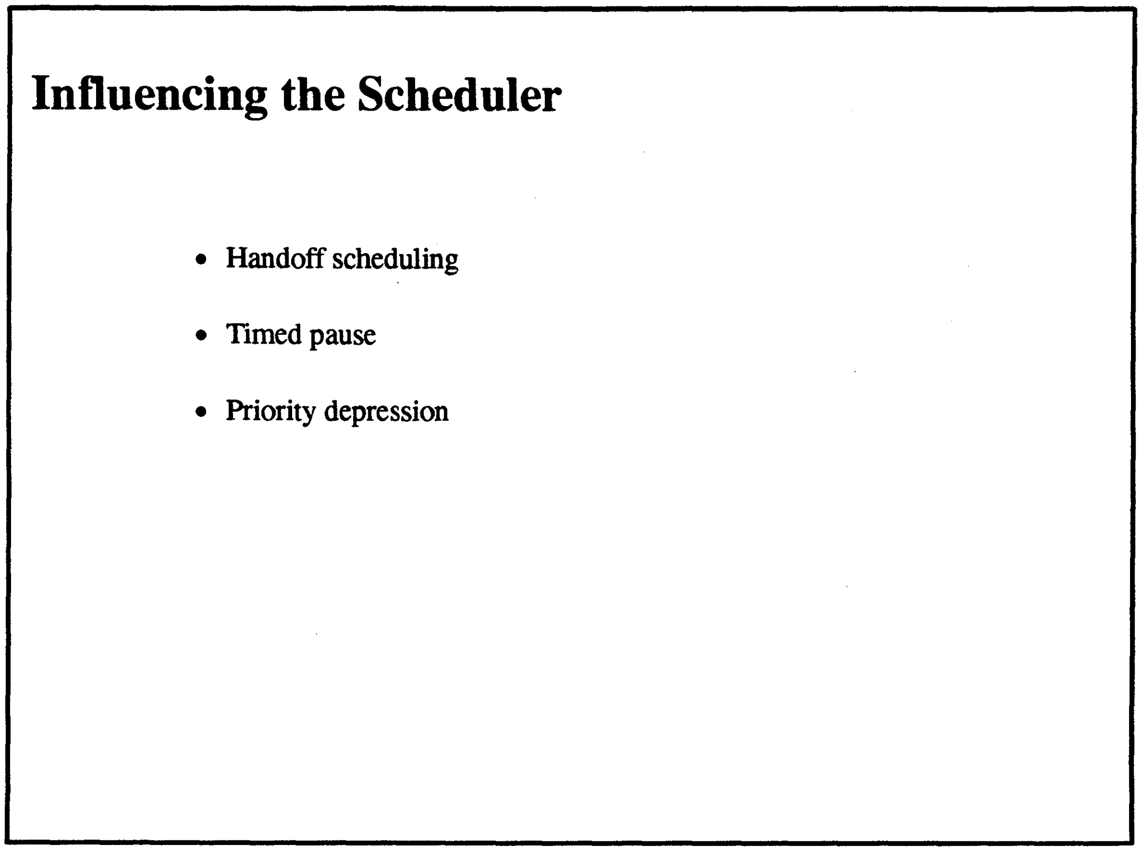
H2596 2-48. 70

Time Slicing

The quantum for fixed-priority threads is settable for each thread. However, for time-shared threads, the time quantum is typically $^1/_{10}$ of a second. While a thread is running, it cannot be preempted by threads that have been on the run queue since the beginning of the quantum. However, if a thread with a better priority becomes runnable, then it preempts the currently running thread. Currently, preemption does not take place immediately for threads running in kernel mode: a thread is not preempted unless it is running in user mode (or "voluntarily" gives up the processor by a call to *thread_block*). The effect of a quantum expiration in kernel mode is delayed until the running thread returns to user mode (or blocks).

For a multiprocessor, an adjustable time quantum is used for time-shared threads. If there are more processors than runnable threads then there is no preemption—it is not needed. If, however, there are more runnable threads than processors, then the individual time quanta are set so that the average time between quantum ends, over all processors, is $^1/_{10}$ of a second. E.g., for 11 threads competing for 10 processors, the per-thread time quantum is set to one second. Thus there is an average of $^1/_{10}$ of a second between quantum expirations. The scheduler adjusts the quanta so that quantum expirations are never in sync.

2–49. Scheduling



Influencing the Scheduler

- Handoff scheduling
- Timed pause
- Priority depression

H2596 2-49. 71

Influencing the Scheduler

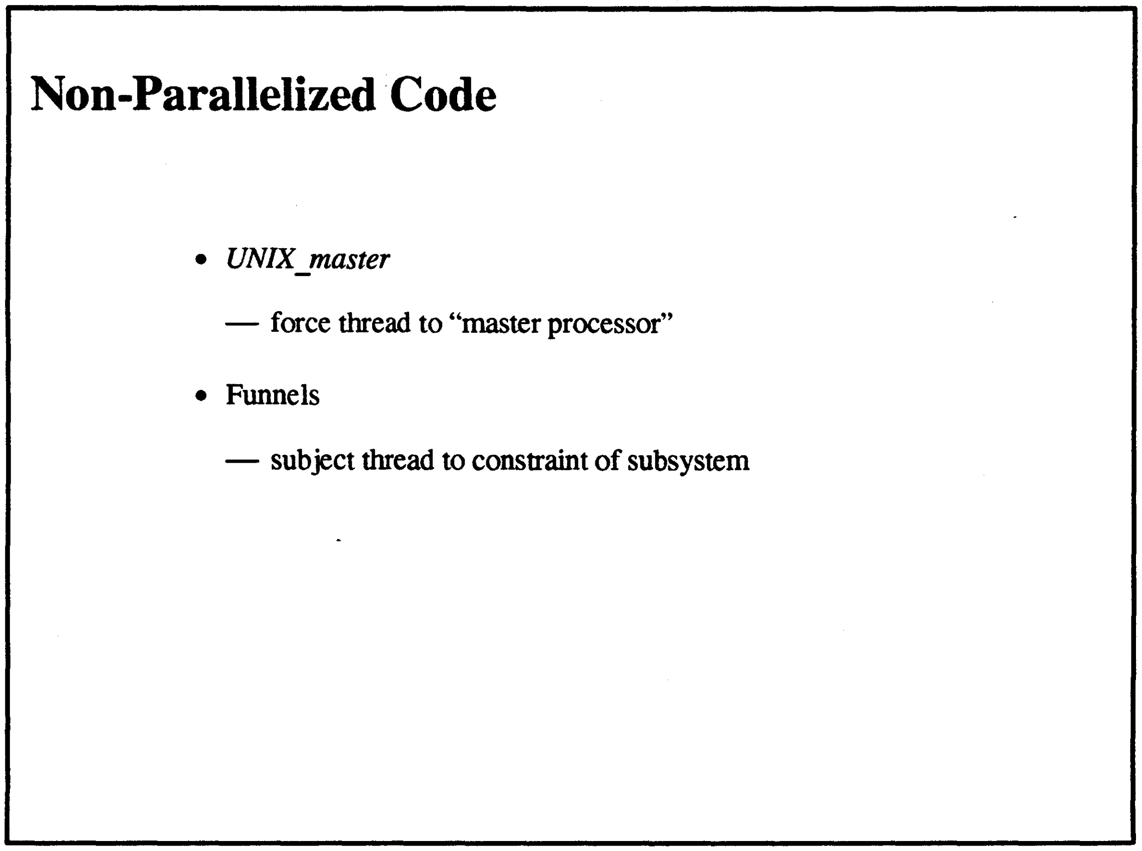
An application can exert some local influence over scheduling decisions through the thread_switch system call. One application of thread_switch is when a thread is in effect making a synchronous request of some other thread. To avoid delays, it may "give" its processor to this other thread (as long as the thread is within its processor set). If both threads are time-shared, the new thread receives the remainder of the current time quantum. Otherwise, the new thread gets a new quantum.

The other two options of thread_switch arise when, for example, a thread is spinning on a lock in user mode, waiting for another thread to release that lock. To avoid this perhaps wasteful use of processor time, it might be advisable to yield the processor by blocking. However, in user mode, this would require at least two system calls: one call executed by the thread itself to put itself to sleep, and another executed by another thread to wake it up. In certain situations, we can reduce this system call overhead to just one system call. If the duration of the wait is known, a thread can issue the thread_switch system call with the wait option, requesting that it be suspended for a fixed period of time and then automatically woken up.

Another approach to the same problem uses the *priority depression* option to thread_switch. This system call "depresses" the calling thread's priority to the worst possible value for a given period of time and then restores it. After depressing its priority, the caller might then start spinning on a lock. If there is no competition for its processor, then it uses otherwise idle processor cycles by spinning. Otherwise, if there is competition for the processor, then the thread yields to the competition because of its depressed priority.

The swtch system call returns an indication of whether another runnable thread is waiting to use the caller's processor. The swtch_pri system call is a special case of the thread_switch system call in which priority depression is requested with a fixed time period (set to the time quantum for time shared threads—$^1/_{10}$ of a second).

2–50. Scheduling



Non-Parallelized Code

- *UNIX_master*
  - — force thread to "master processor"
- Funnels
  - — subject thread to constraint of subsystem

H2596 2-50. 72 © 1991 Hewlett-Packard

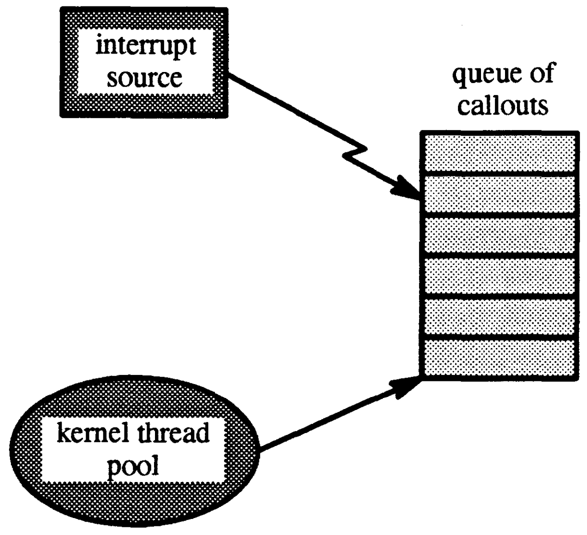
Non-Parallelized Code

When a thread enters an unparallelized subsystem within the kernel, it calls *UNIX_master* to force itself to run on the master processor (i.e. it joins that processor's local run queue). When it completes its execution of the unparallelized subsystem, it calls *UNIX_release* to allow itself to run on other processors. The *signal subsystem* is one of the few such unparallelized subsystems.

The notion of *funnels* is intended as a generalization of the *UNIX_master* concept. Associated with a subsystem, for example a device driver, might be a *funnel data structure* that describes the constraints of that subsystem. E.g., for an asymmetric I/O architecture, it might indicate to which processors a particular I/O device is accessible. Calls to the driver for that device would then be "funneled" to a processor of that set. Currently, funnels are used only to force processing to take place on the *UNIX_master.*

2–51. Thread Pools

Kernel Thread Pools



Kernel Thread Pools

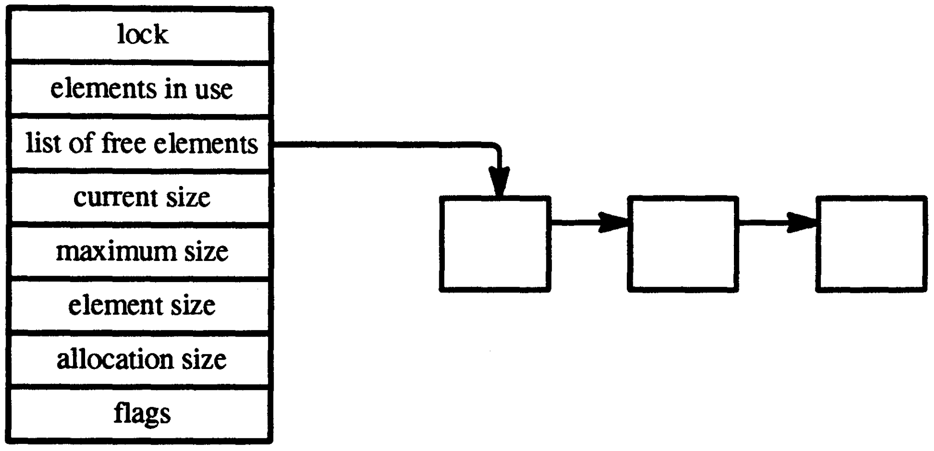
Kernel thread pools are used in a number of places to perform actions in a thread context that would otherwise be performed in the interrupt context. These pools are particularly useful for multiprocessors but may be used for uniprocessors as well.

In the interrupt context, the interrupt handler places a request for action on a *callout queue* and directs a *wakeup* call to a pool of kernel threads. One of these threads pulls the request off the queue and services it.

This technique is used in the logical volume manager, in the networking subsystem, and for device drivers (for multiprocessors).

2–52. Zoned Memory Allocation

Kernel Memory Allocation: Zones



struct zone

H2596 2-52. 74 © 1991 Hewlett-Packard

Kernel Memory Allocation: Zones

Zones provide a technique for fast allocation and liberation of storage in the kernel. A zone is a collection of fixed-size blocks: a separate zone is created for each kernel data structure that is so managed (e.g., task and thread structures, etc.).

A zone is initialized with a pre-allocated free list, an allocation size, and a maximum size. Allocations are taken from the free list until it is exhausted; then additional memory (of *allocation size*) is allocated from the virtual memory system and added to the free list. (Zones may be paged or wired: currently they are always wired.)

Module 2 — The Process Abstraction

Exercises:

1. a. Which UNIX system calls can be adapted simply for use by threads within a multithreaded process?

   b. Which UNIX system calls are difficult to adapt for use by threads within a multithreaded process?

   c. Explain how the *proc* and *user* structures of older UNIX implementations must be modified for use with multithreaded processes.

   d. Why is it not sufficient to represent a multithreaded process with the Mach *task* and *thread* structures?

2. Explain the conceptual difference between UNIX and Mach system calls.

3. a. Explain why two routines in OSF/1, *assert_wait* and *thread_block*, are needed in place of the typical *sleep* routine in older UNIX systems.

   b. What is the difference between a simple lock and a read-write lock?

   c. Why are conditional lock requests (e.g., *simple_lock_try*) necessary?

   d. Explain what is meant when a thread is in the state RUN+WAIT+SUSPENDED.

4. a. What is the difference between a synchronous signal and an asynchronous signal?

   b. When an asynchronous signal is sent to a process, which thread within the process receives the signal?

   c. Which aspects of *signal handling* state information are kept with the thread and which are kept with the process as a whole?

   d. How is an exception converted into a signal?

5. a. How does a user-level program create a thread?

   b. What is the function of the thread_abort system call?

   c. What is the difference between a thread as supported by the POSIX library and a thread as supported by the OSF/1 kernel?

6. a. What scheduling policies are used in OSF/1?

   b. What is the difference between a thread's scheduling priority and its base priority?

   c. Explain the meaning and use of *handoff scheduling* and *priority depression.*

   d. What might processor sets be used for?

   e. Why might there be threads in a processor set but no processors?

7. a. What are *thread pools* used for?

b. Which subsystems use them?

8. Why is zoned memory allocation used instead of dynamic storage allocation techniques such as the "buddy system"?

Advanced Questions:

9. Since the original UNIX *user* structure is now split into two structures, *u_task* and *u_thread*, and both are now located in the kernel address space, why is it necessary to maintain separate *u_task* and *proc* structures?

10. In what ways are OSF/1's kernel threads cheaper than UNIX's kernel processes?

11. a. Why can't OSF/1 be preemptible in kernel mode?

    b. Some versions of UNIX have added *preemption points* in the kernel at which a thread in kernel mode may yield to more important threads. If such preemption points were added to OSF/1, what would be the constraints on where they might be placed?

Module 3 — Messages and Ports

Module Contents

Module Objectives

In order to demonstrate an understanding of the use of messages and ports in OSF/1, the student should be able to:

- describe how a message header would be set up to represent a C structure and differentiate between the header created for a msg_send system call and the header created for a msg_rpc system call

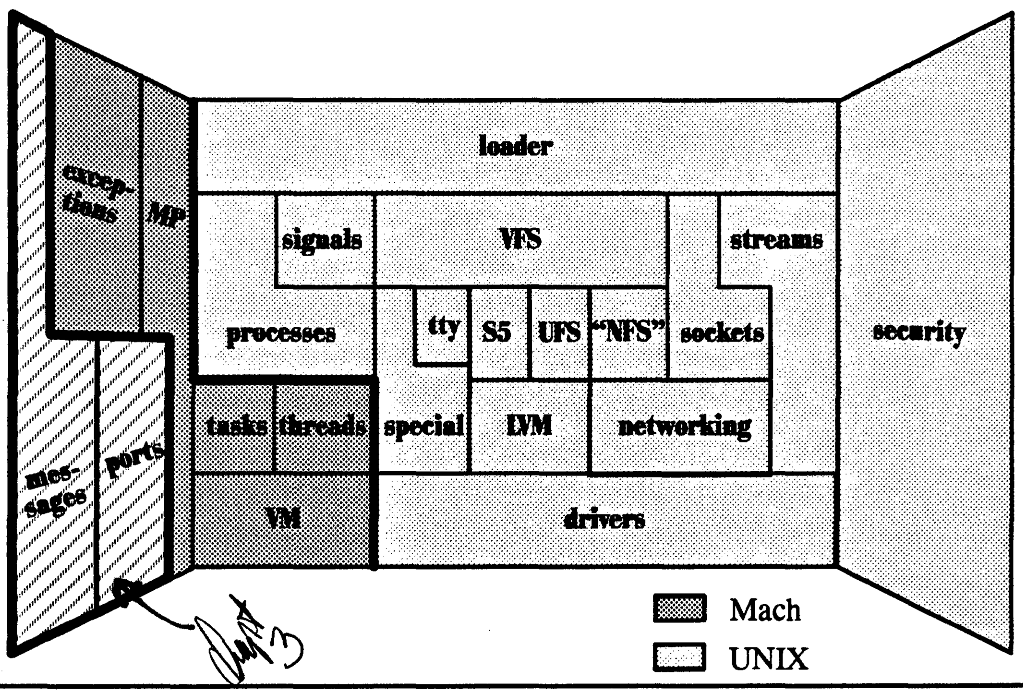
- describe how port rights are represented both within a user task and within the kernel

- describe the flow of control and data within the msg_send and msg_receive system calls.

Module 3 — Messages and Ports

3–1. The Big Picture

Messages and Ports

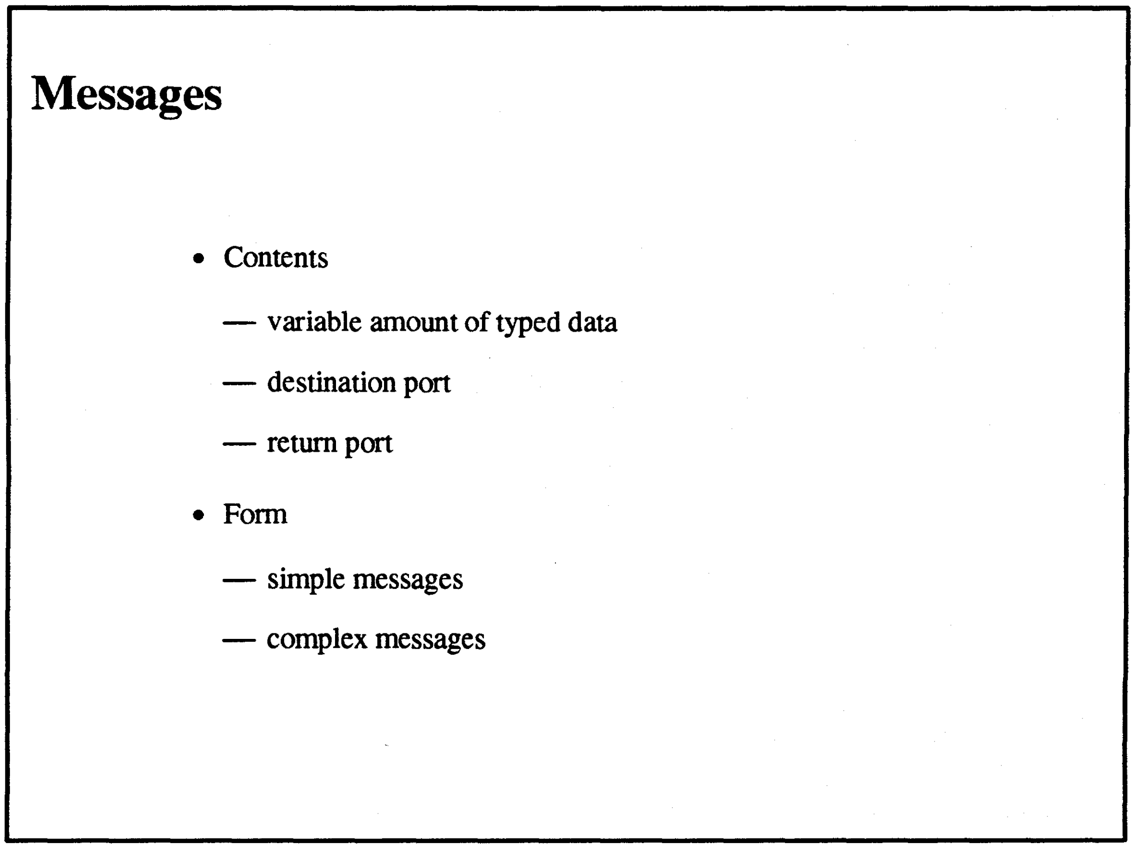


.H2596 3-1. 75 © 1991 Hewlett-Packard

Messages and Ports

The material in this module is covered in Open Software Foundation, 1990a, chapter 3.

3–2. Messages



Messages

- Contents
  - — variable amount of typed data
  - — destination port
  - — return port
- Form
  - — simple messages
  - — complex messages

.H2596 3-2. 76 © 1991 Hewlett-Packard

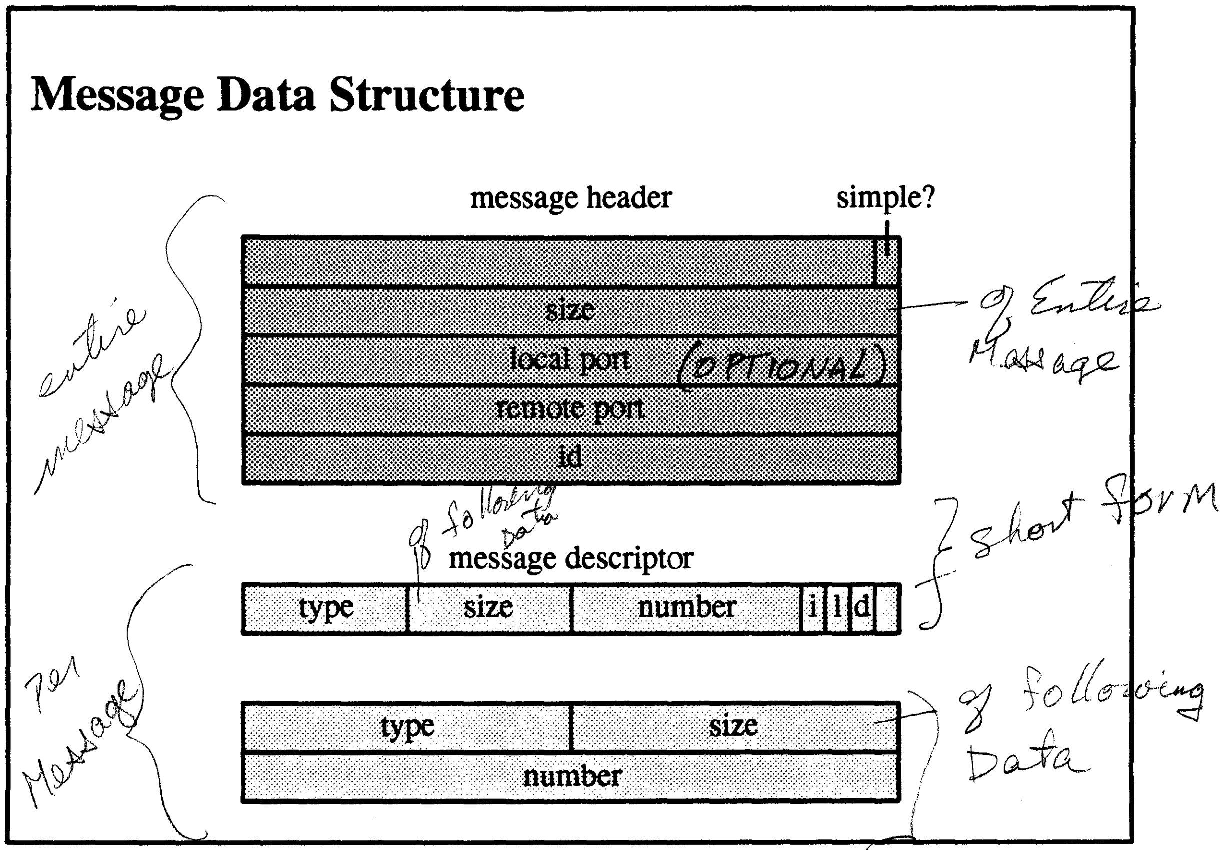
Messages

A message is a collection of data to be sent through a *port* to the task that has *receive rights* for the port. The data is typed, allowing the kernel or intermediate tasks to interpret it as necessary. For example, the kernel must know if a data item is a port right (send or receive) so that it can deal with it accordingly. If the data is to be transferred from one machine to another in a heterogeneous environment, then the kernel must know the type of application data, so that it can convert the data to the target machine's representation. (The use of ports for inter-machine communication is not supported in OSF/1.)

The message must contain a reference to the *destination port*, which is the port through which the message is transferred, and may contain a reference to a *return port* through which a reply can be sent.

Simple messages contain no *out-of-line* data, and they are copied directly into and out of the kernel. This technique is used if the message is small and does not contain port rights. Otherwise the message is deemed to be *complex* and requires additional processing by the kernel. Port rights must be interpreted by the kernel, as discussed later. The transfer of out-of-line data is optimized using copy-on-write techniques.

3–3. Messages



H2596 3-3. 77 © 1991 Hewlett-Packard

Message Data Structure DATA

A message consists of a header followed by zero or more data items, each headed by a descriptor.

Message header:

- *simple?* no ports or out-of-line data?
- *size*: total bytes (except for out-of-line data) total message
- *local port*: optional port through which a reply might be sent
- *remote port*: port for sending message

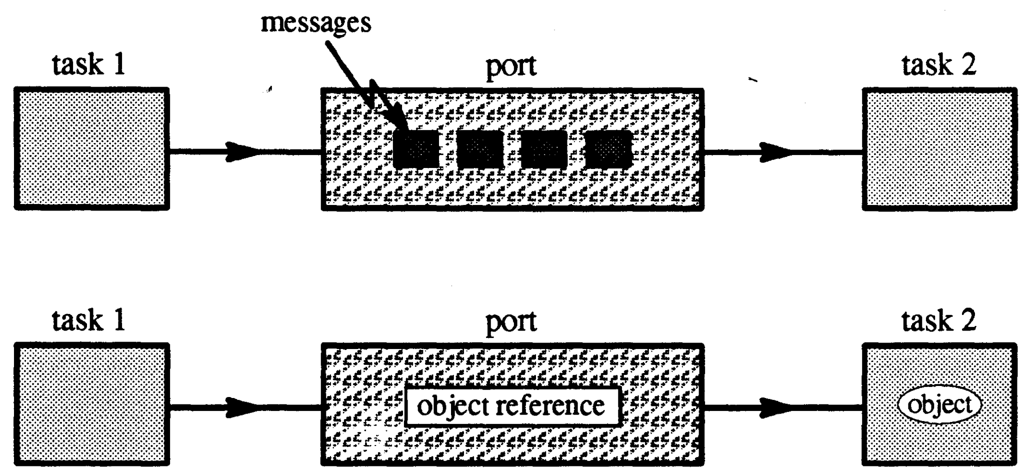
- *id*: application-specific id

Message descriptor:

- *type*: send right, receive right, int
- *size*: bits per item
- *number*: number of items
- *i*: inline (data follows) or out-of-line (pointer follows)
- *l*: longform—type, size, number follow
- *d*: deallocate port right or memory

3–4. Ports

Ports



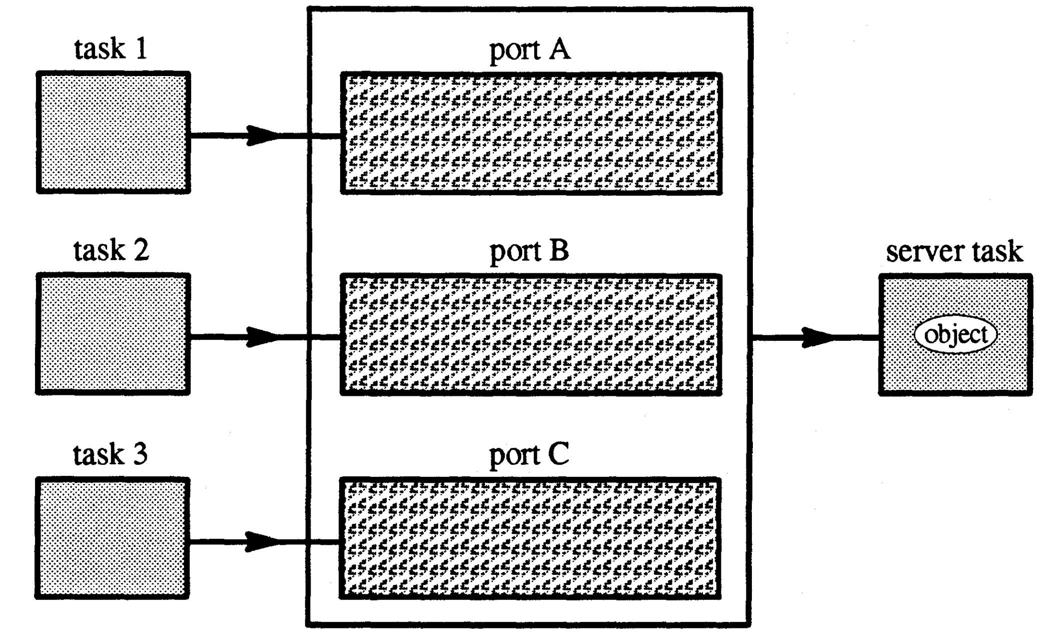
.H2596 3-4. 78 © 1991 Hewlett-Packard

Ports

A port may be used either as a protected queue of messages or as an object reference. When a port is used as an object reference, the task with receive rights manages the object, and send rights to the port are effectively references to the object.

3–5. Ports

Port Sets



.H2596 3-5. 79

Port Sets

The server task, which has receive rights for ports A, B, and C, can consolidate them into a port set. The effect of this is to merge the message queues of all the ports into a single queue. The server can then receive messages from the port set, and thus receive messages from any of ports A, B, or C.

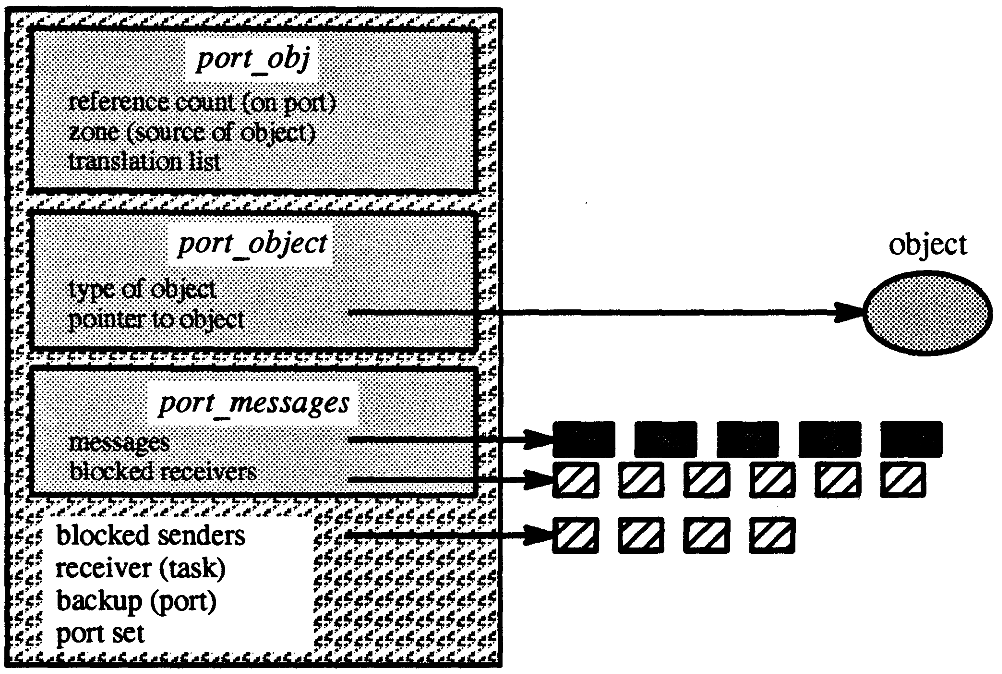
a port can only belong to a single port set.

NO BROADCAST

Single Write & Multiple Readers.

3–6. Ports

The Kernel Port Structure



.H2596 3-6. 80 © 1991 Hewlett-Packard

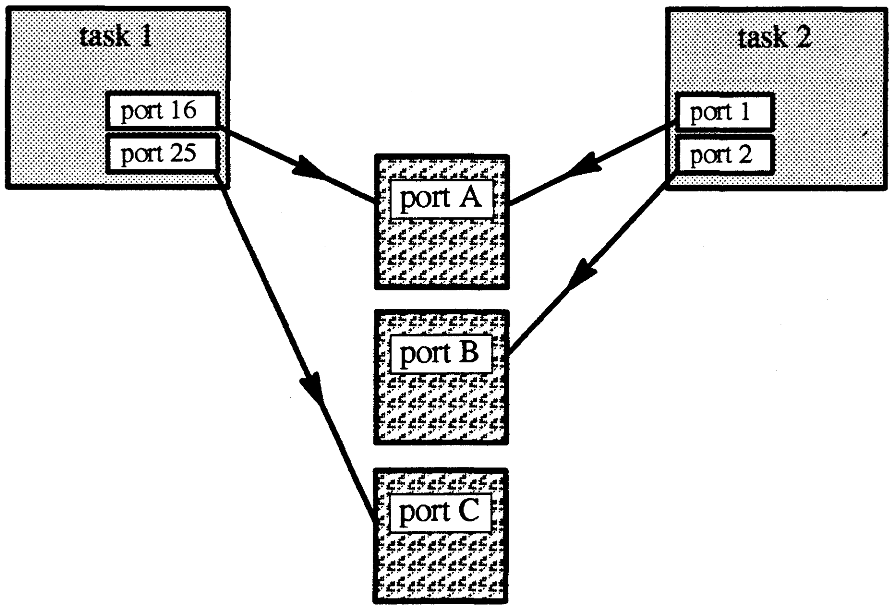
Kernel Port Structure

A port is represented in the kernel by a structure of type *kern_port_t*. The first parts of this structure, of type *port_obj* and *port_object*, refer to a kernel object if this port represents such a reference. (The source code often uses the types *port_obj* and *kern_port* interchangeably. This works only because the *port_obj* is the first component of the *kern_port* structure.) The next portion of the structure, the *port_messages* structure, represents the queue of messages for the port and the queue of receivers waiting for a message to arrive (of course, only one of these queues can be non-empty at a time). Even though only one task can hold receive rights to the port, there may be multiple blocked receivers, since this task may have multiple threads.

The remaining important fields of the *kern_port* structure include a queue of blocked senders, a reference to the task holding receive rights to the port, and a reference to the port's backup port.

3–7. Ports

Port Names



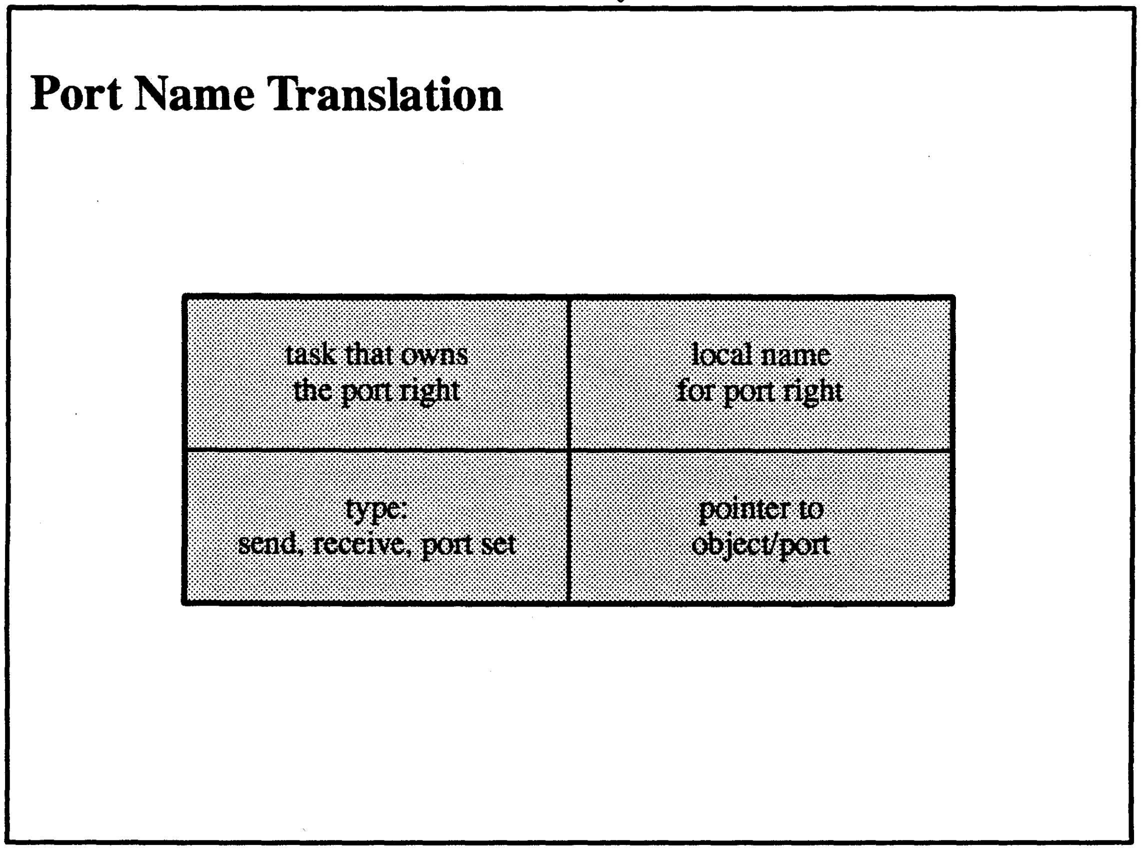
.H2596 3-7. 81 © 1991 Hewlett-Packard

Port Names

Internally, a port is named by the address of its *kern_port* structure. Externally (i.e., in user tasks), a strictly local name is used. These local names (of type *port_t*) are just integers. They are analogous to file descriptors in UNIX: one task's port names mean nothing to another task; when these names are passed to the kernel, they must be converted to the internal form (analogous to the address of a file-table entry). However, unlike UNIX file descriptors, if two local port names within a task are different, then they necessarily refer to different ports.

3–8. Ports

in Kernel



Port Name Translation

task that owns the port right	local name for port right
type: send, receive, port set	pointer to object/port

H2596 3-8. 82 © 1991 Hewlett-Packard

Port Name Translation

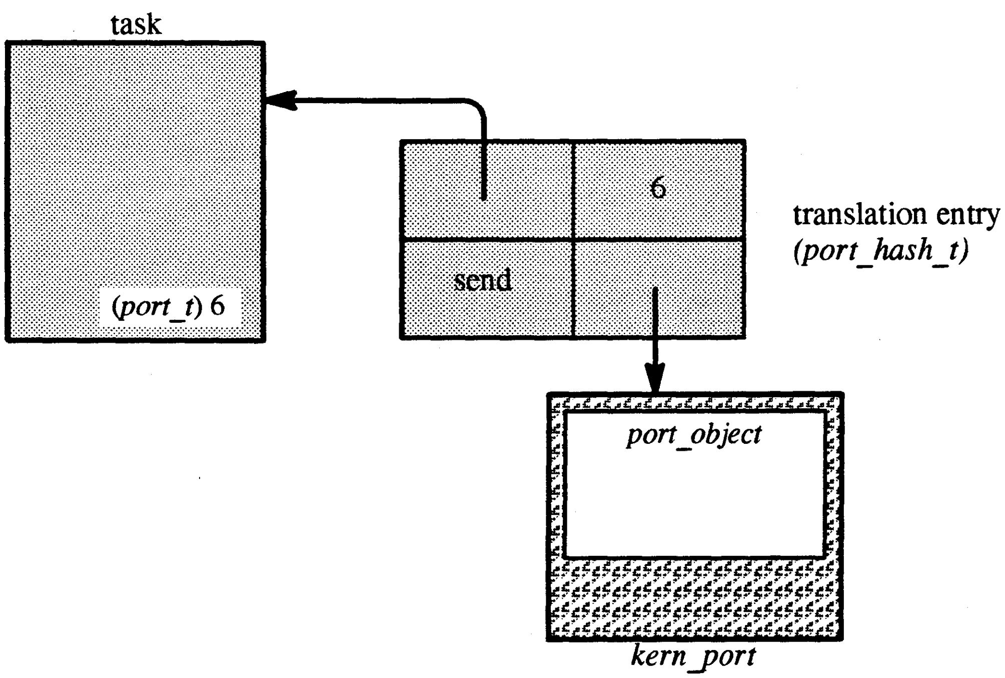
The first two lists are doubly linked and are used just to keep track of the *port rights* associated with each object. The second two lists are actually hash tables and are used for efficient translation from external to internal names and vice versa.

Each time a *port right* is added to a task's name space, a *translation entry* is created. Each such entry is put on four lists:

- The task's *translation entry chain*
- The port's *translation entry chain*
- The *task/local name table* (TL table)
- The *task/port table* (TP table)

3–9. Ports

Port Name Interpretation

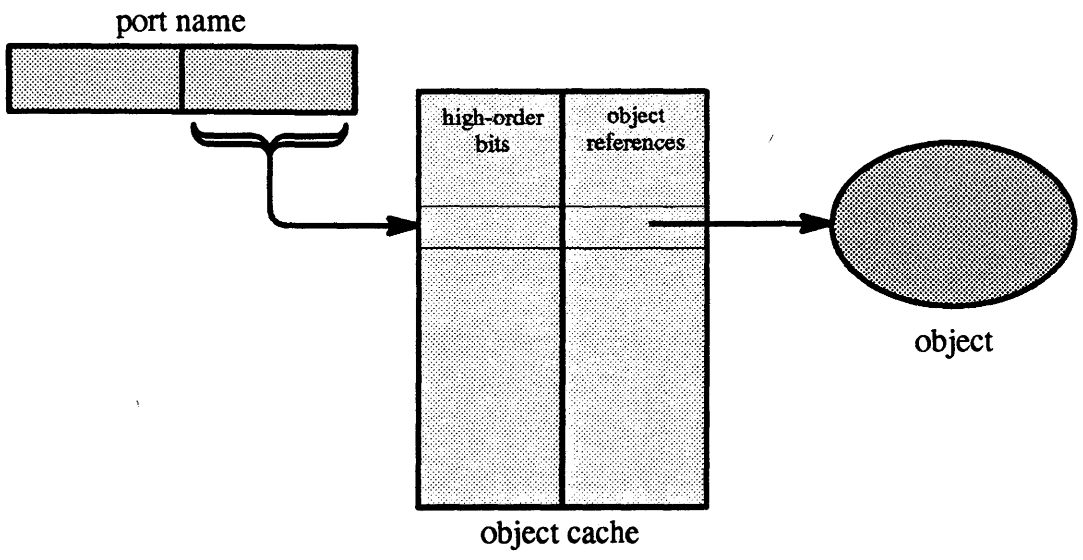


.H2596 3-9. 83 © 1991 Hewlett-Packard

Port Name Interpretation

3–10. Ports

Object Cache



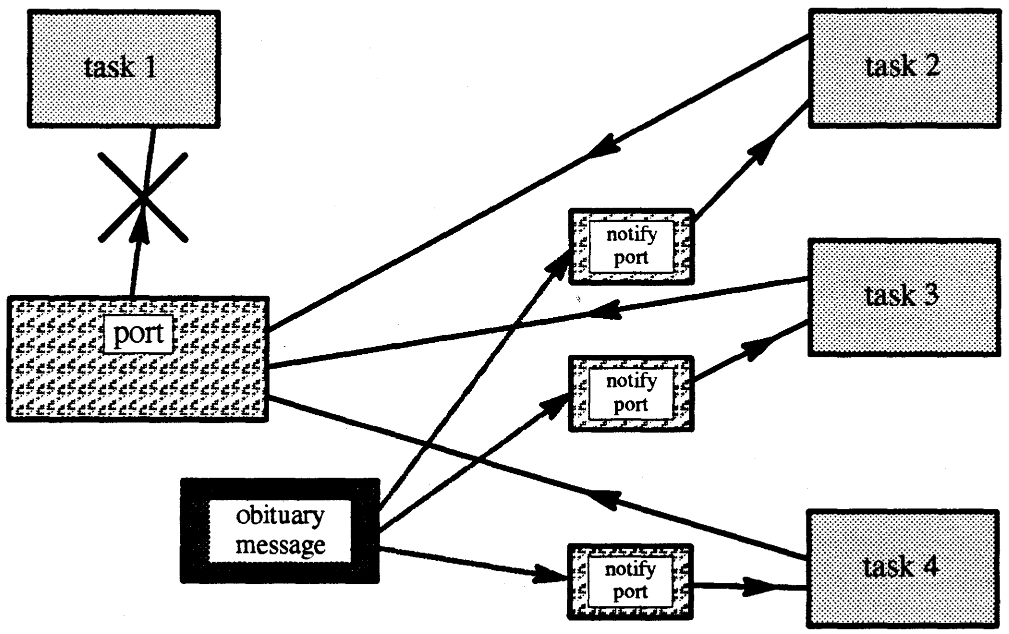
.H2596 3-10. 84 © 1991 Hewlett-Packard

Object Cache

To speed the translation from the local name of a port to the object it identifies, each task has an *object cache.* This cache is a simple array indexed by the low-order bits of the local name of the port. If the translation is in the table, it is found immediately.

3–11. Ports

Port Destruction



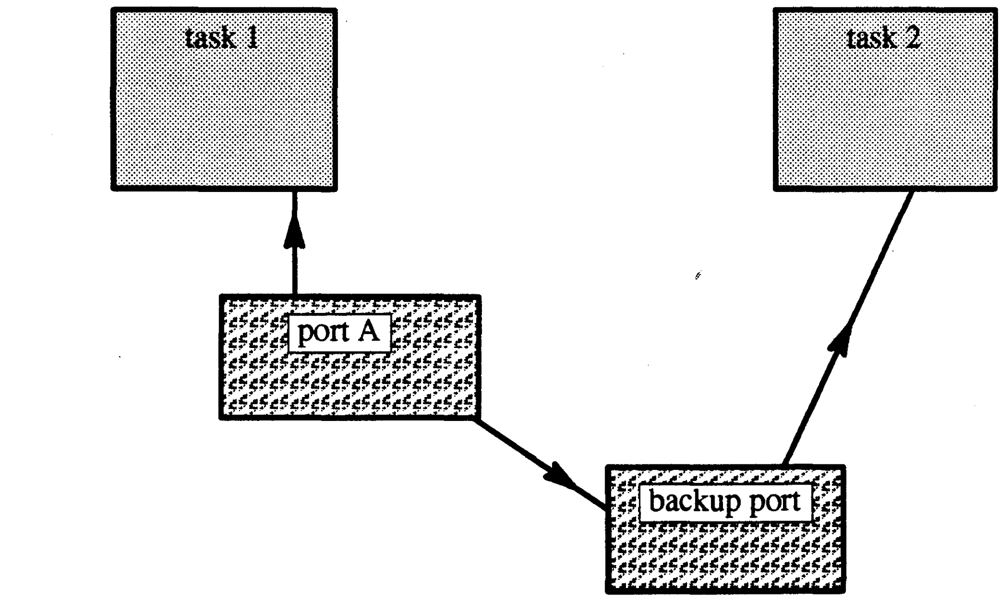
.H2596 3-11. 85 © 1991 Hewlett-Packard

Port Destruction

If a port with no backup port has its receive rights deallocated, then the port is marked *dead.* All tasks with send rights to this port receive a *port_deleted* message and lose their rights. All threads that were blocked and queued on the port's queue of blocked senders are woken up and their msg_send system calls return with an error status.

3–12. Ports

Backup Ports, part 1

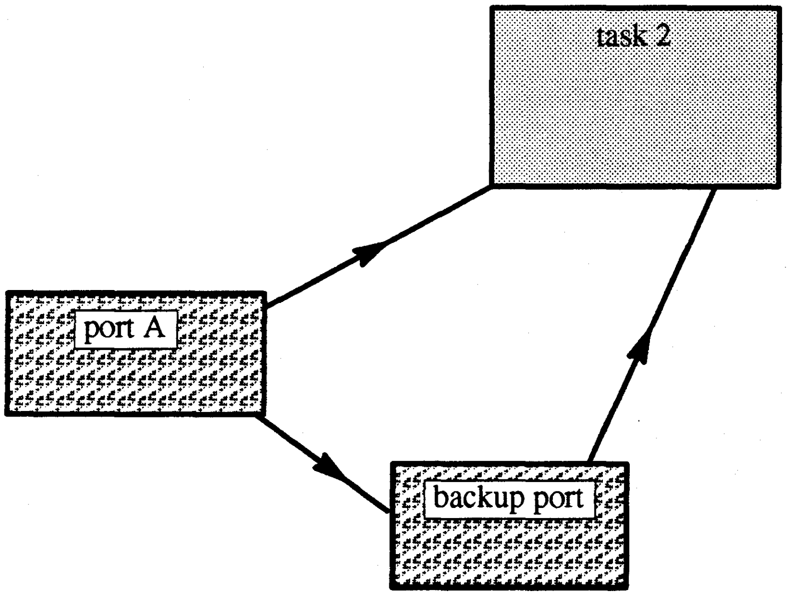


Backup Ports, part 1

If a task deallocates a port's receive rights, then these rights are transmitted to some other task through the port's *backup port.* Send rights to the backup port are effectively held by the port itself. Receive rights are held by the task that is to provide the backup function.

3–13. Ports

Backup Ports, part 2



.H2596 3-13. 87

Backup Ports, part 2

Task 1 has vanished; its receive rights to port A have been transferred to task 2 via the backup port.

3–14. Flow of Control

msg_send

- *msg_copyin*
  - — transfer message to kernel
    - ♦ use *copyin* to transfer header and inline data
    - ♦ use *vm_map_copyin* to map the out-of-line data into *copy objects*
  - — convert to internal form
    - ♦ use *object_copyin* to deal with ports in messages

send rights ~~data~~ to port

- *msg_queue*
  - — if kernel is the receiver, call *mach_msg*, which transfers to appropriate kernel routine
  - — handle flow control
  - — if a thread is waiting for a message, then transfer control to it immediately (*handoff scheduling*)
  - — otherwise, queue message

.H2596 3-14. 88 © 1991 Hewlett-Packard

msg_send

Sending a message involves two steps. First, the message has to be transferred into the kernel. Then it has to be disposed of: either queued on the port's message queue or immediately handed off to a waiting receiving thread.

Out-of-line data is not directly mapped into the kernel's address space, but instead is represented by a *copy object*, which has the effect of a *vm_map_entry* but does not occupy kernel address space. The purpose of the copy object is to maintain a copy of the out-of-line data that can be later mapped into a receiving task's address space.

3–15. Flow of Control

msg_receive possible block

- *msg_dequeue*
  - — check queue for message, possibly block
  - — when a message is consumed, wake up blocked senders
    - ♦ wake up one blocked sender for each message received
    - ♦ generate *notify message* if necessary
- *msg_copyout*
  - — convert ports from internal to external representation with *object_copyout*
  - — use *vm_map_copyout* to transfer out-of-line data
  - — use *copyout* to copy header to user

.H2596 3-15. 89 © 1991 Hewlett-Packard

msg_receive

Receiving a message is also a two-step process. The first step is to remove the next message from the port's message queue. If there is no message, then the following thread is queued on the queue of waiting procedures. If there is a message and there are blocked senders (i.e., the message queue was full), then the first blocked sender is woken up. If the sender of the message requested it, a *notify* message is sent to inform it of the message's consumption.

The second step in message reception is to transfer the message from the kernel to the user task.

How do you find a port in another task? - same in Unix- static pre-designed arrangement

w/o Bootstrap port (name server)

Module 3 — Messages and Ports

Exercises:

1. a. How does the contents of a message header created for a msg_send system call differ from that created for a msg_rpc system call?

   b. Describe how a message header would be set up to represent a C structure.

2. a. What is contained in the kernel port structure?

   b. How are a task's rights to a particular port represented within the task?

   c. How are such external references to a port converted by the kernel into the address of the kernel port structure?

   d. When the task receives a port right via a message, how is it added to the task's port space?

   e. What happens when a task deallocates its send rights to a port?

   f. What happens when a task deallocates its receive rights to a port?

3. a. Explain how out-of-line data is passed from one task to another.

   b. Explain how flow control is implemented as part of the msg_send and msg_receive system calls.

Advanced Question:

4. Messages and ports are not heavily used in OSF/1, in part because the UNIX standards that dictate the user/system interface make no mention of message- and port-like constructs. However, messages and ports could be used to aid the implementation of a number of UNIX system calls. For example, in the write system call, the buffer could be transferred from the user's address space to the kernel's address space as part of a message, allowing copy-on-write techniques to be used to minimize the actual copying of data. What problems would be associated with doing this? What other UNIX system calls and facilities could benefit from the use of messages and ports?

Module 4 — Virtual Memory

Module Contents

Module Objectives

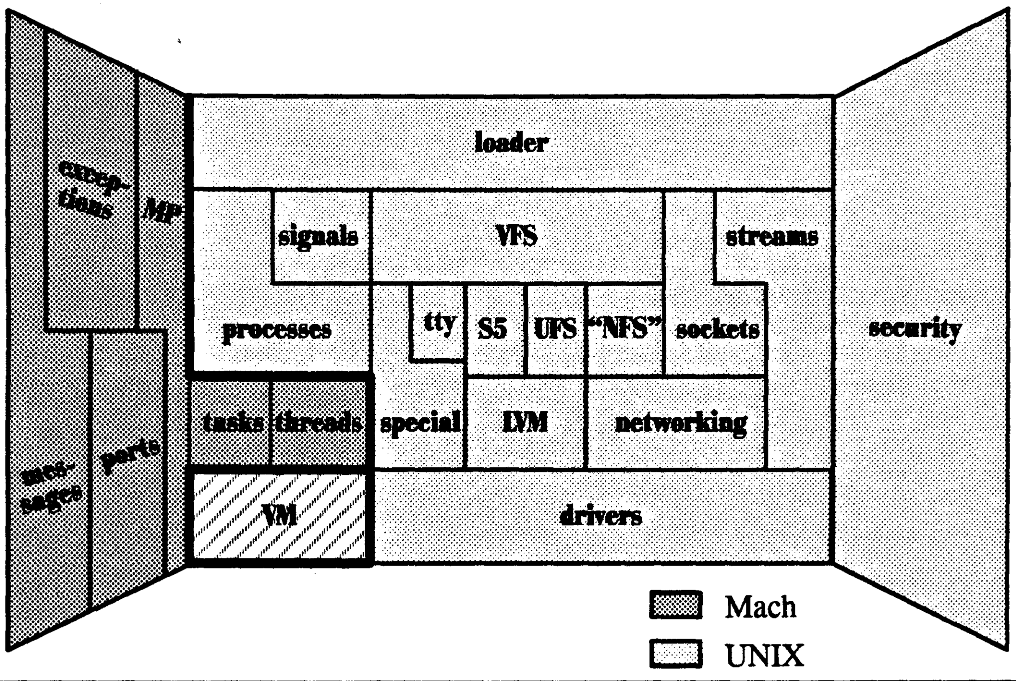
In order to demonstrate an understanding of virtual memory within OSF/1, the student should be able to:

- explain the concept of lazy evaluation and give four examples of how it is used in OSF/1
- list the data structures in the architecture-independent portion of OSF/1 and explain their purpose
- describe the interface between the memory object manager and the virtual memory kernel
- describe the implementation of memory objects within the vnode pager
- explain the use of shadow objects and copy objects in optimizing virtual copy operations
- explain what must be done at the architecture-dependent level to implement the virtual copy operation

Module 4 — Virtual Memory

4–1. The Big Picture

Virtual Memory



H2596 4-1. 90 © 1991 Hewlett-Packard

4-2.
Virtual Memory

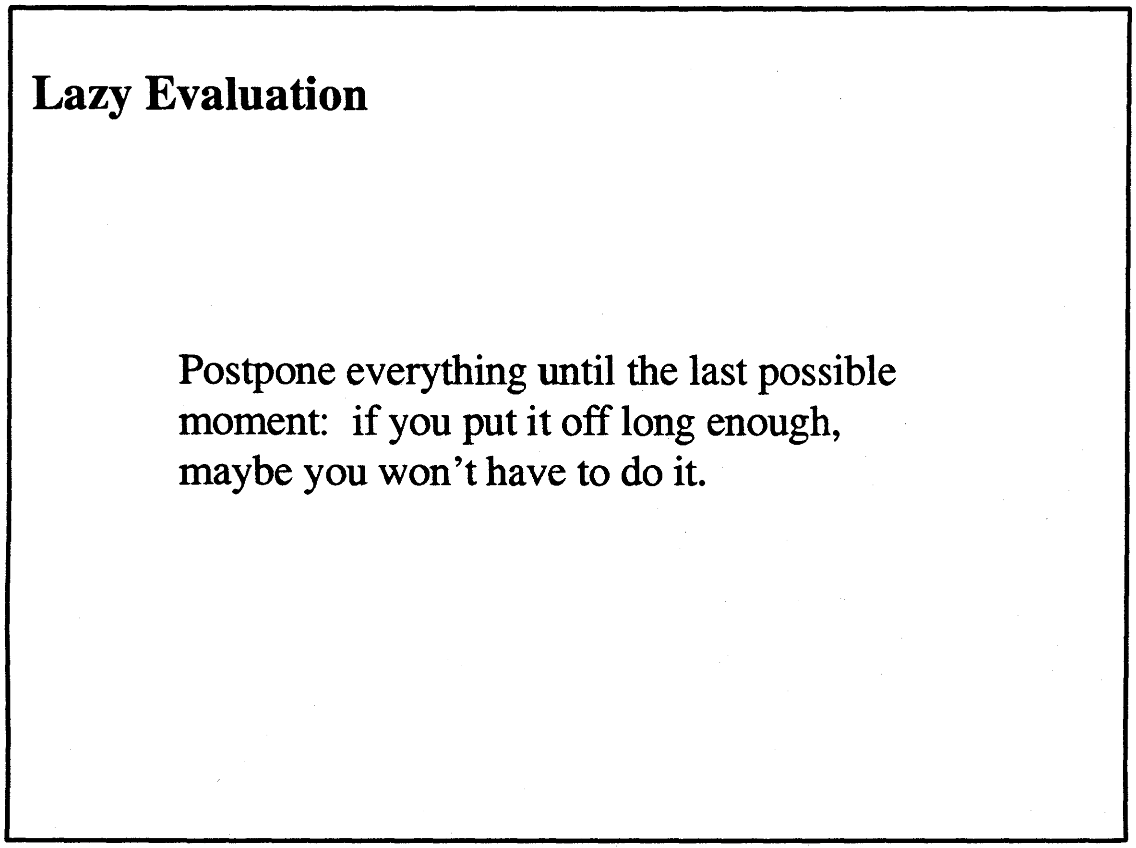
A task's *address space* consists of a number of either private or shared *virtual memory objects*. The address space may be large and sparse. Objects such as files can be mapped into the address space.

The VM model is independent of the underlying architecture; primary storage is a cache of pages belonging to the VM objects. The architecture's address-translation mechanism maps references to cached pages. The architecture-dependent code and the architecture-independent code are separate.

User code can access the interface between the kernel's page cache and memory object managers, and thus memory object management can be performed outside of the kernel.

The material in this module is covered in Open Software Foundation, 1990a, chapters 7, 8, 9, and 10.

4–2. Lazy Evaluation



Lazy Evaluation

Postpone everything until the last possible moment: if you put it off long enough, maybe you won't have to do it.

.H2596 4-3. 91 © 1991 Hewlett-Packard

Lazy Evaluation

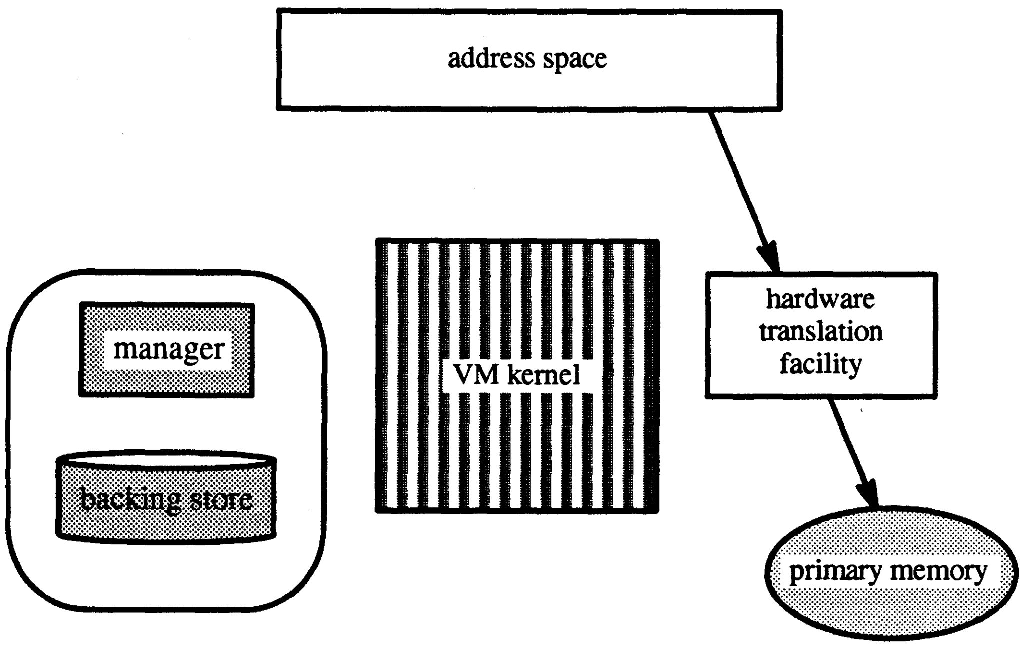
The technique of *lazy evaluation* is pervasive; it's used throughout the VM system. It is an effective optimization, since many operations, such as copying, often turn out not to be really necessary.

Examples of the use of lazy evaluation:

- no physical address maps are created until they are needed to satisfy a reference
- no pages are allocated until they are needed
- no page is copied until two copies are necessary
- no backing store is allocated until it is needed

4–3. VM Components

VM Rough Sketch



.H2596 4-4. 92

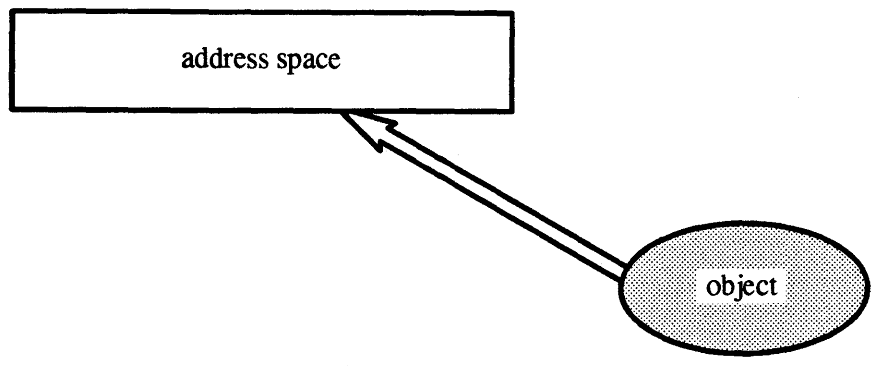
VM Rough Sketch

A process's address space is managed by the kernel, which is responsible for setting up the hardware address translation facilities as required, for responding to page faults. and for determining which pages should be kept in primary memory. This kernel functionality is divided into two pieces, a machine-independent piece and a machine-dependent piece. The former is by far the larger, and it is responsible for maintaining a description of each process's address space. The size of the latter depends upon the architecture, but is typically much smaller than the former.

There is a third component of the VM subsystem, which is the manager of the backing store. It is responsible for supplying the initial values of pages and for holding on to pages that have been paged out. Two possibilities are available to the programmer. A special subtask of the kernel known as the vnode pager is the default manager of backing store. It uses the file system for its backing storage. An alternative is to provide a user-level backing store manager (known as an *external memory object manager* or an *external pager*). It can be used to back objects that the user has mapped into its address space via the vm_map system call; what it does with the pages is entirely up to the application.

4–4. VM Components

Mapping Objects into an Address Space



Mapping Objects into an Address Space

Mapping an object into an address space involves a number of issues. First of all, what is the nature of the object? It might be:

1. *temporary*: it has no name and hence no permanent existence.
2. *a file*: it has a name and hence a permanent existence, but should changes made to the address space be reflected to the file?
3. *a user-provided object*: a user process provides the contents of the object and manages its modifications.

Second, how might the pages obtained from the object be shared among multiple processes? For example:

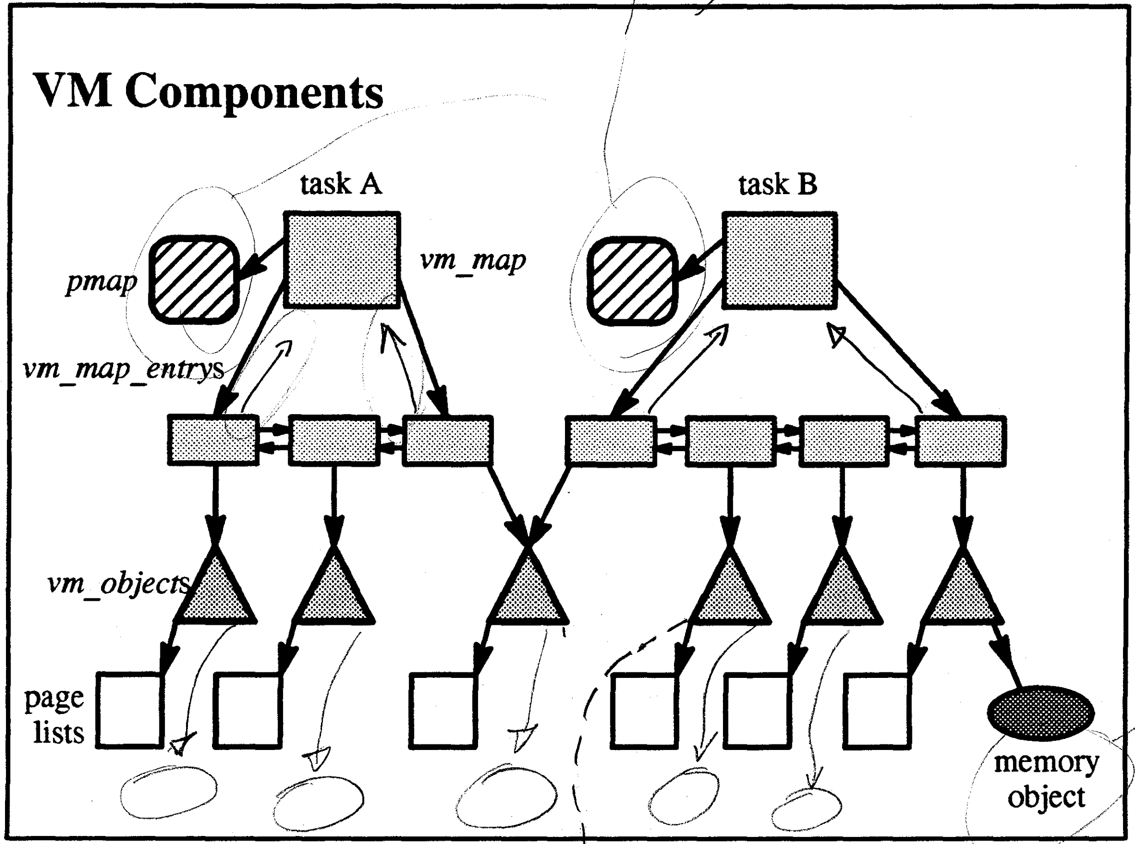
1. the pages are not shared

2. the pages are shared read-only (and if they are modified, copies are made)
3. the pages are shared read-write

4–5. VM Components

architect Dependent

Everything else is Arch. INDEPENDENT



H2596 4-6.

© 1991 Hewlett-Packard

stored elsewhere not primary Memory

VM Components

could be vm_map → vm_map_entry → △

Each task has a map to represent its address space, consisting of a header, the *vm_map* structure, and a linked list of structures, each a *vm_map_entry* representing a continuous range of addresses. Collectively they do NOT have to be contiguous

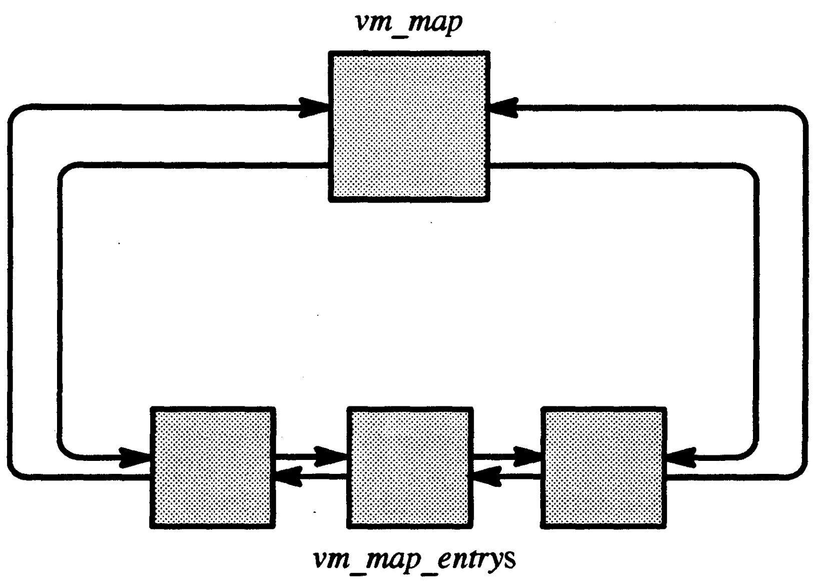
Each such range is mapped to a memory object represented internally as a *vm_object*. Each *vm_object* may refer to other *vm_objects* (via the *shadow chain*, which will soon be discussed). In addition, it represents a set of virtual pages, some of which may currently be in primary storage. Those pages in primary memory are represented by *vm_page* structures and are linked to the *vm_object*. The *vm_object* may refer to a *memory object* (via a port reference), which represents those pages stored elsewhere (in backing store) and managed by a memory object manager. This object manager may be supplied by either the kernel or a user task.

The *pmap* data structure encapsulates of the architecture-dependent portion of the VM subsystem. It represents the architecturally required memory mapping structures and related information.

↳ Virtual to physical mapping system

4–6. VM Components

VM Maps



H2596 4-7. 95 © 1991 Hewlett-Packard

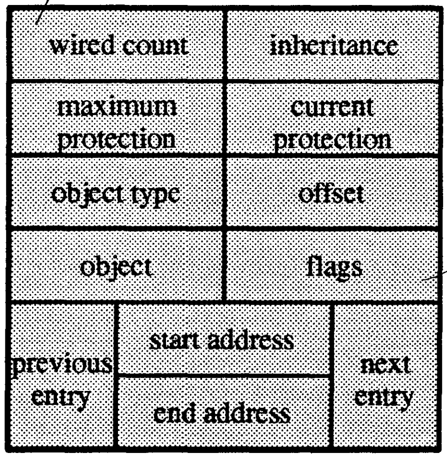
VM Maps

A *VM map*, mapping a range of virtual addresses to *vm_objects*, is represented as a doubly linked list of *vm_map_entry* structures headed by a *vm_map* structure. The mapping may be *sparse,* i.e., many if not most addresses in the range may not be represented. The *VM map* represents either the address space of a task or a range of addresses shared by a number of tasks.

4–7. VM Components

vm_map_entry

are they wired? number?



Copy on write?

vm_map_entry

.H2596 4-8. 96 © 1991 Hewlett-Packard

vm_map_entry

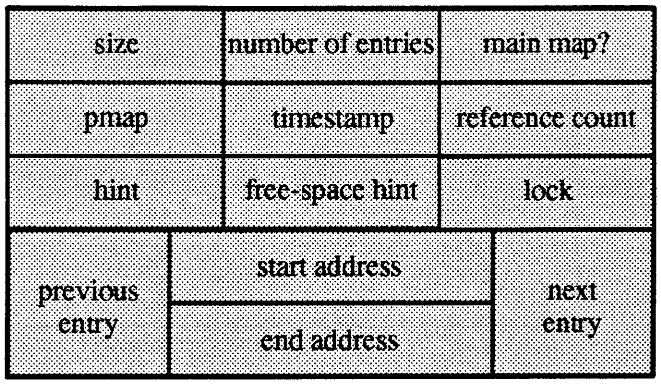
- Previous entry, next entry:
  - — links in chain of *vm_map_entry*s
- Start address, end address:
  - — range of addresses represented in this entry
- Inheritance:
  - — how this range should be inherited by a child (i.e., via a fork)
  - — share, copy, or none (not inherited at all)

can we coallesce adjacent entries?

- Maximum protection, current protection:
  - — specifies maximum and currently permitted accesses
  - — some combination of read, write, and execute; not all combinations may be possible
  - — (RWX) > (RX) > ( )
- Object:
  - — reference to an object, which may be a *vm_object* or a *vm_map*
- Object type:
  - — share map, submap, and *vm_object*
- Offset:
  - — offset into object
- Flags:
  - — copy-on-write information
- Wired count:
  - — this is incremented by one to indicate that the range of addresses must not be paged out; thus pages in this range may only be paged out if the wired count is 0 (which is the usual value)

4–8. VM Components

vm_map



vm_map

.H2596 4-9. 97 © 1991 Hewlett-Packard

vm_map

- Size:
  — virtual size of mapped region
- Number of entries:
  — number of *vm_map_entry*s in list
- Main map?
  — whether it is the top-level map of a task
- Pmap:

— pointer to *pmap* for this mapping

- Lock:

— a blocking lock protecting this data structure

- Timestamp:

— time of last change to map (used to determine if anything has changed since the object was unlocked)

- Reference count

- Hint:

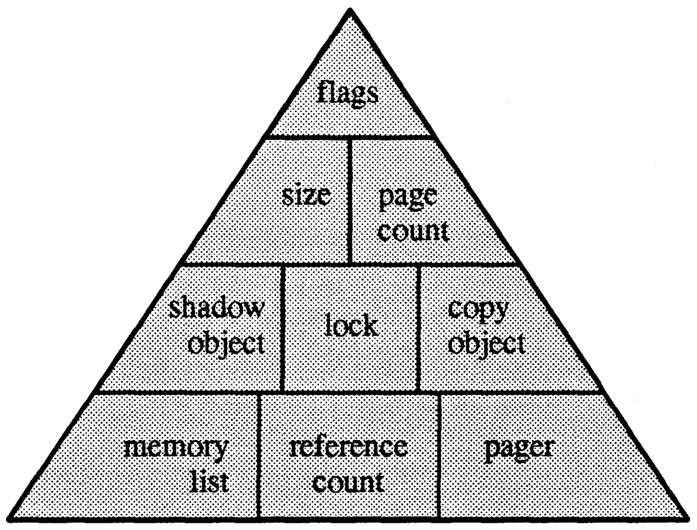
— pointer to last *vm_map_entry* that was encountered in a lookup (a good place to start for the next lookup)

- Free-space hint:

— pointer to the first hole in the address space

4–9. VM Components

vm_object



vm_object

.H2596 4-10. 98

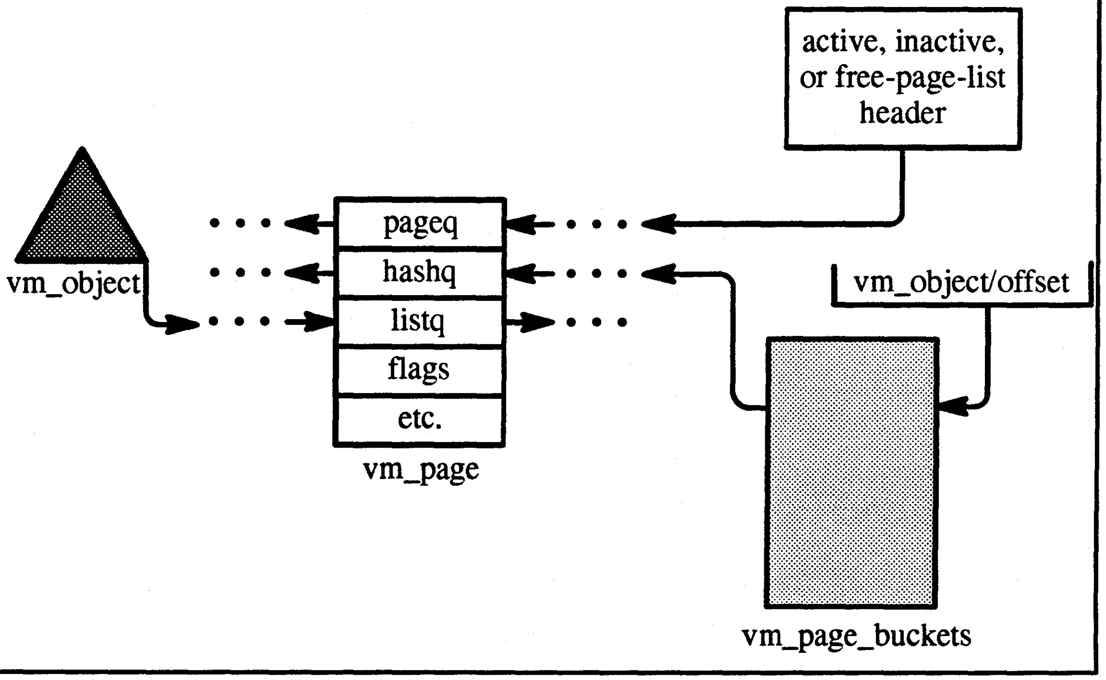
vm_object

- Memory list:
  - — list of incore pages assigned to this object
- Reference count
- Pager:
  - — the memory object manager
    - ♦ send rights to memory object port
    - ♦ offset into the memory object

- Shadow object:
  - — link to backing object for copy-on-write
- Copy object:
  - — link to object that should receive copies of the modified pages (used for copy-on-write with permanent memory objects)
- Size:
  - — object's size if it's an internal object
- Page count:
  - — number of incore pages
- Lock:
  - — a simple lock for mutual exclusion
- Flags:
  - — various

4–10. VM Components

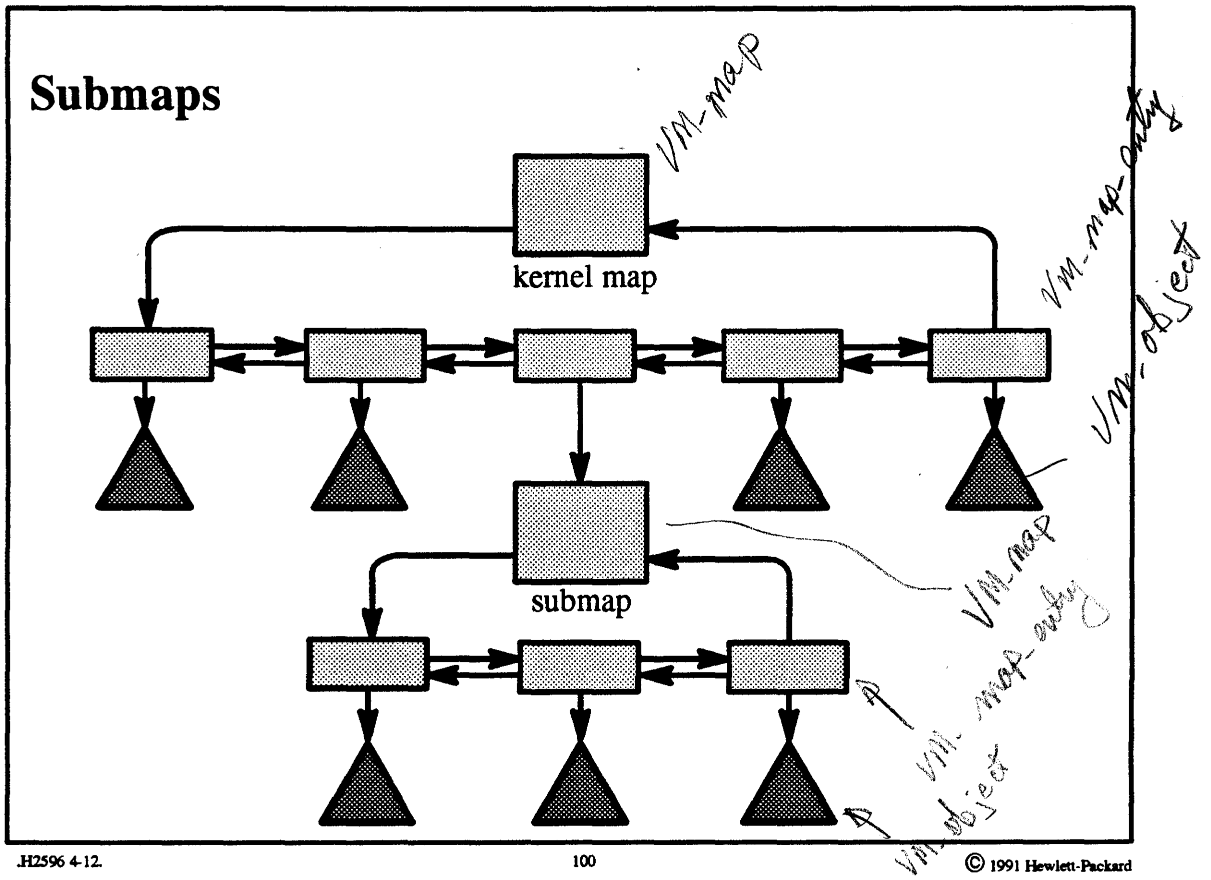
Representing Pages in Primary Memory



Representing Pages in Primary Memory

Each page in primary storage is represented by a 56-byte *vm_page* data structure, which is used to represent the page in a number of lists. Attached to each *vm_object* is a list of all the *vm_page* structures for incore pages associated with the object. If the underlying page is pageable, then the *vm_page* structure is attached to one of three lists managed by the *pageout daemon* (the active, the inactive, or the free-page list). In order to find a particular page, there is a system-wide hash table headed by the array *vm_page_buckets*. This hash table is keyed by the address of the *vm_object* and the page's offset within the virtual memory represented by the object.

4-11. VM Components



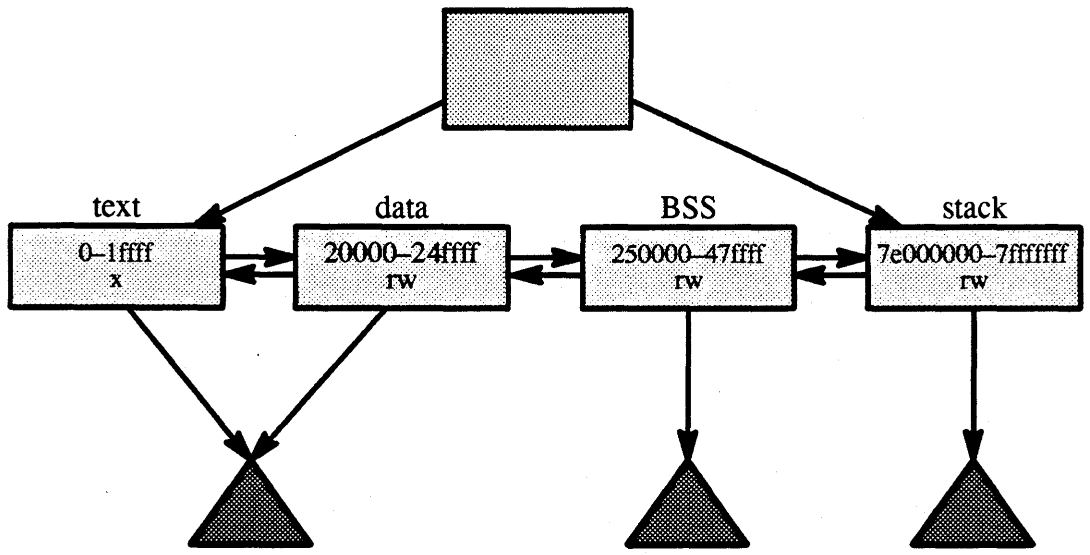
Submaps

Since the list of *vm_map_entrys* is typically not very long, sequential search is reasonable for user tasks. However, the kernel task's address space representation can become fairly complicated. To simplify searching, special submaps are used (only in the kernel) to represent a range of addresses.

Note that this representation is used only in the kernel.

4–12. VM Components

UNIX VM on OSF/1



N 32 Meg

.H2596 4-13. 101 © 1991 Hewlett-Packard

UNIX VM on OSF/1

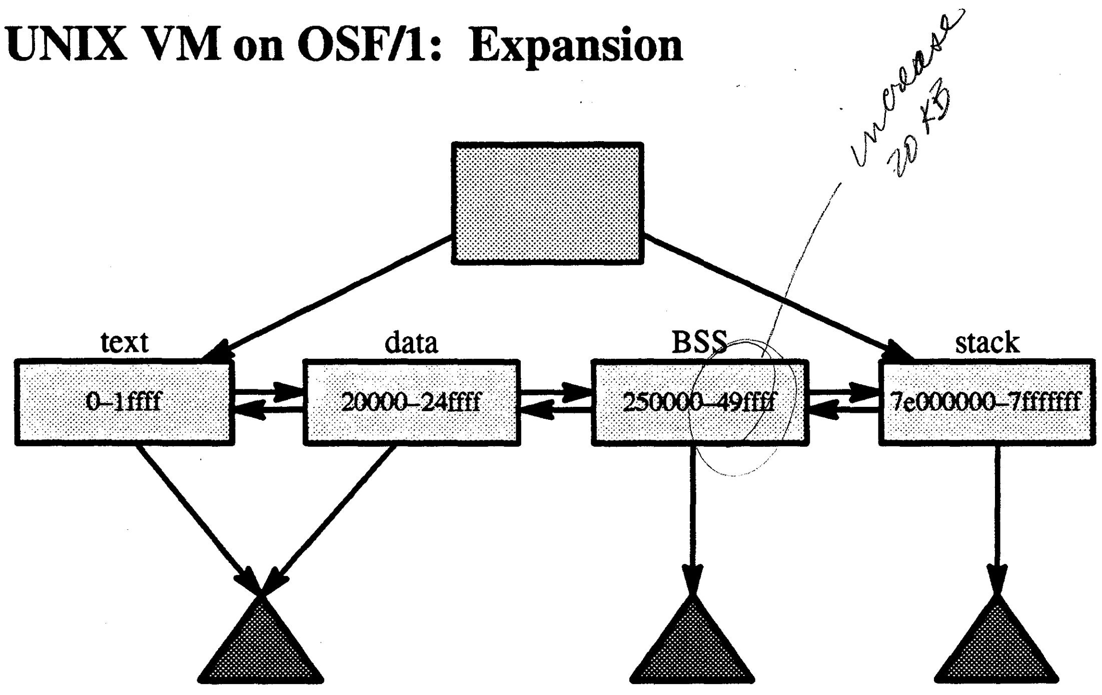
This picture shows how the address space is initially set up for a UNIX process. A *vm_object* for an executable file is mapped into both the text and the data sections. The executable portion of the file is mapped in the text region, and then the initialized data from the file is mapped on the data region.

Since the data region of the process may be modified but the file should not be, the file pages representing data are mapped copy-on-write. The kernel creates a temporary memory object to back up the modified copies of data pages. Since threads in the process cannot modify the text region, the kernel need not allocate any additional backing store for the text.

Two *vm_objects* representing temporary storage are set up for the BSS *(block skip section* or, less cryptically, uninitialized data) and the stack regions.

4–13. VM Components

UNIX VM on OSF/1: Expansion

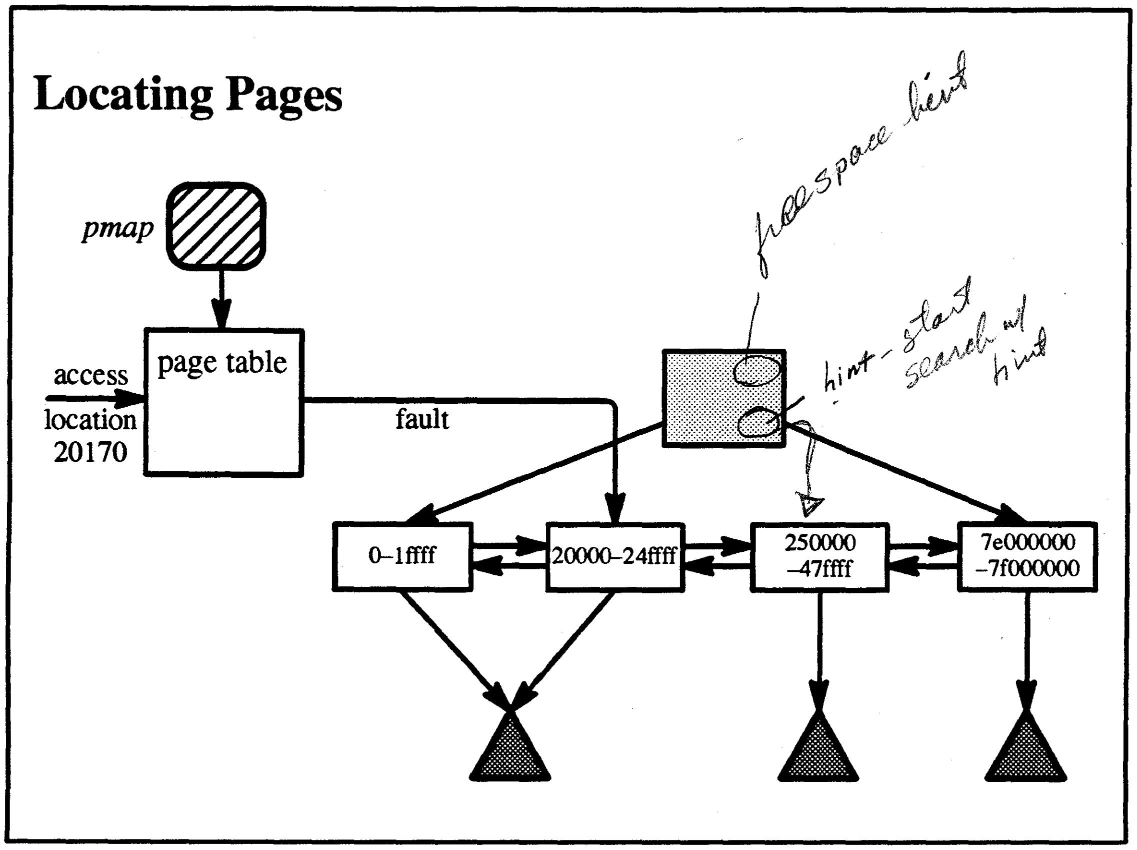


H2596 4-14. 102 © 1991 Hewlett-Packard

UNIX VM on OSF/1: Expansion

This picture shows the effect of growing the UNIX address space. The UNIX process issues an sbrk system call to increase the size of BSS by 20K bytes. Internally, this is converted into a *vm_allocate* request, which determines that an existing *vm_map_entry* can simply be extended to include the new address space.

4–14. VM Components



H2596 4-15. 103 © 1991 Hewlett-Packard

Locating Pages

A page fault occurs when a page is referenced that is not mapped by the hardware. The page-fault handler must determine if this is a legitimate reference; if so, it must allocate primary storage for the page and put data into the real memory.

1. The page-fault handler first scans the list of *vm_map_entrys* for an entry whose range includes this page. To speed this search, the ***hint*** field of the *vm_map* structure points to the last *vm_map_entry* referenced by this task; successive page faults often occur on pages within the same *vm_map_entry.*
   a. If a containing *vm_map_entry* is not found, then the reference is invalid and an *exception* is generated.
2. If a containing *vm_map_entry* is found, then the page-fault handler checks to see whether the desired access is allowed.

   a. If access is not allowed (e.g., an attempt to modify a page in a read-only region), then, again, an exception is generated.

3. If the access is allowed, then the page-fault handler follows the pointer to the *vm_object.* Associated with the object is a hash table representing virtual pages belonging to the object.

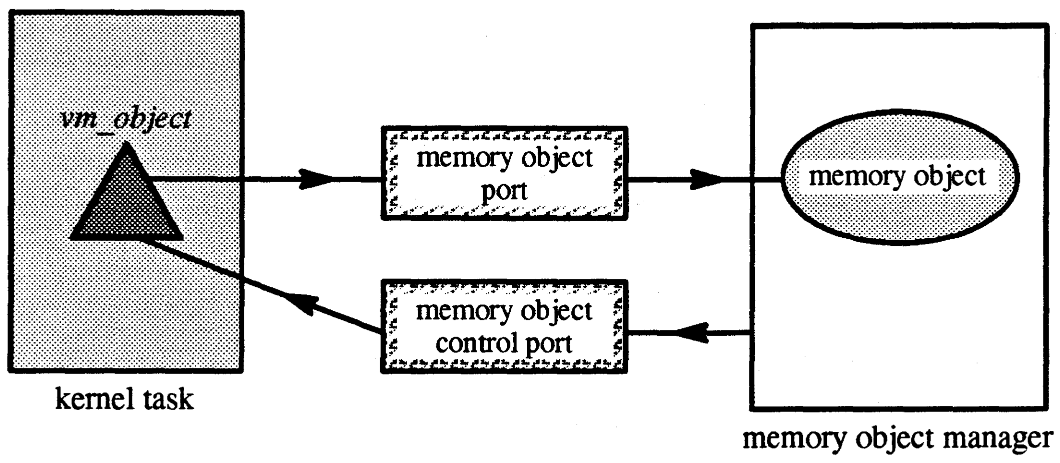
   a. If the desired page belongs to the object, then its contents are fetched from the associated memory object (as described later).

   b. If the page is not present in this *vm_object*, then the page-fault handler checks the next *vm_object* (which this one shadows). (We discuss what this means and why it occurs in the following pages.)

   c. If no *vm_object* claims ownership of the page, then the page is created, filled with zeros, and given to the topmost *vm_object.*

4–15. Memory Objects

Memory Objects



Vnode pager

H2596 4-16. 104 © 1991 Hewlett-Packard

Memory Objects

A *memory object* is an abstraction representing what is mapped into virtual memory. The object might be a file, temporary storage, or something implemented by a user task (such as the *network memory server*).

A memory object is implemented (managed) either in the kernel or in a user task. It is represented by three ports:

- *memory object port*: effectively the name of the object—used to transmit requests to the manager. The *memory object manager* holds the receive rights to this port.

- *memory object control port*: a path from the manager to the *vm_object* used to transmit requests from the (external) memory object manager. The kernel holds the receive rights to this port.

- *memory object name port*: created by the kernel and used to name an object in the kernel's response to the vm_regions system call (it provides a means for showing that an object exists without giving away rights to it).

Memory object managers (also known as *pagers*) manage the objects that may be mapped into tasks' address spaces. A pager's duties are to respond to a kernel's requests for pages (in response to page faults) and to store pages on some sort of backing store in response to *pageout* requests.

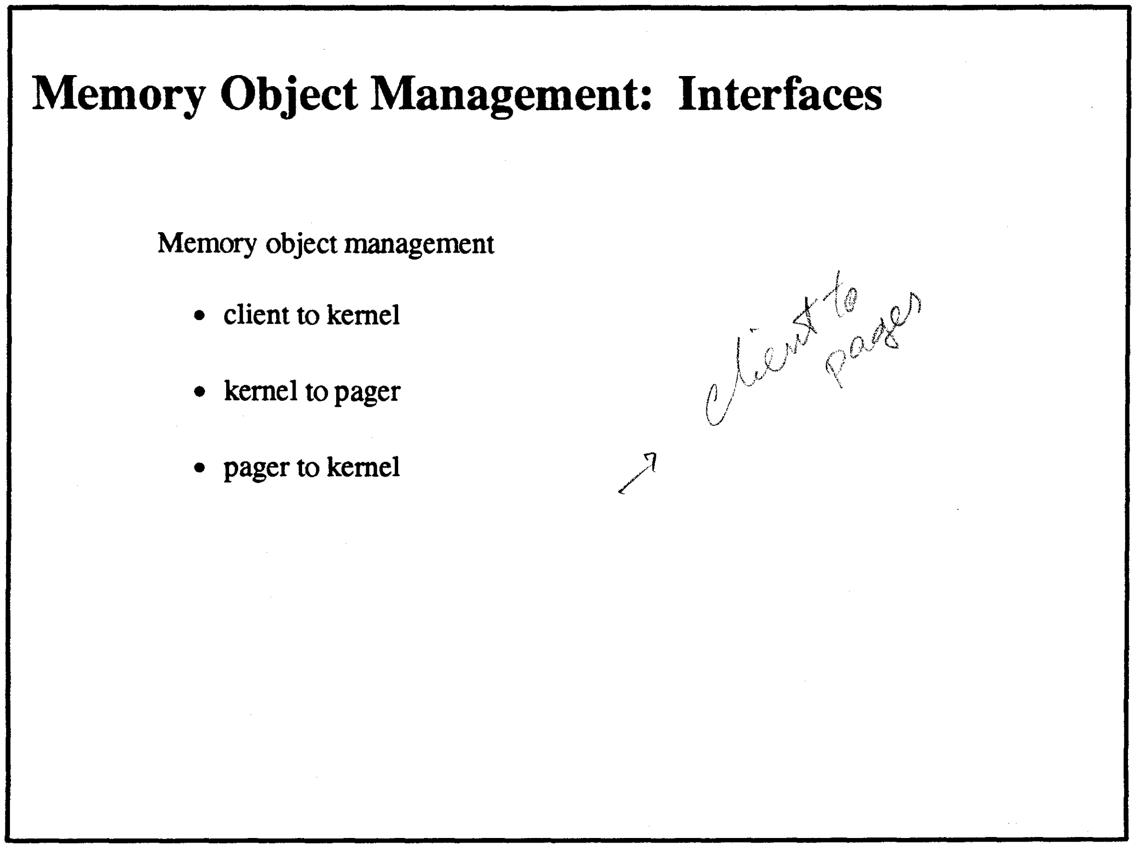
The *default memory object manager*, known as the *vnode pager*, is implemented as a separate task running in kernel mode. (Its address space is implemented as a submap of the kernel map, as we will discuss later.) The vnode pager supports both temporary memory objects and permanent memory objects. The former are used to back up virtual memory that will exist only as long as tasks have it mapped into their address spaces. This is used, for example, to back up BSS and stack, as well as to back up a process's private modifications to permanent objects that the process has mapped *copy-on-write*, such as initialized data.

Permanent memory objects have names in the file system and thus can continue to exist even if no process has them mapped (i.e., permanent object are files). Examples are text, initialized data, and memory-mapped files.

Memory object interactions typically involve three parties:

- the memory object manager (pager)—manages one or more memory objects
- the kernel—maintains the page cache and responds to page faults
- the client—one or more user threads; maps memory objects into its address space

4–16. Memory Objects



Memory Object Management: Interfaces

Memory object management

- client to kernel
- kernel to pager
- pager to kernel

client to pager

H2596 4-17. 105 © 1991 Hewlett-Packard

Memory Object Management: Interfaces

- Client to pager
  - — obtain memory object (i.e. a port); no formal interface
- Client to kernel
  - — map memory object into address space; use either mmap (for files) or vm_map (for Mach objects).
- Kernel to pager
  - — initialize memory object
  - — request a data page
  - — write back a modified data page

— upgrade permissions

- Pager to kernel

— provide a data page (either in response to a request or gratuitously)

— indicate that a page is not available (will be zero-filled)

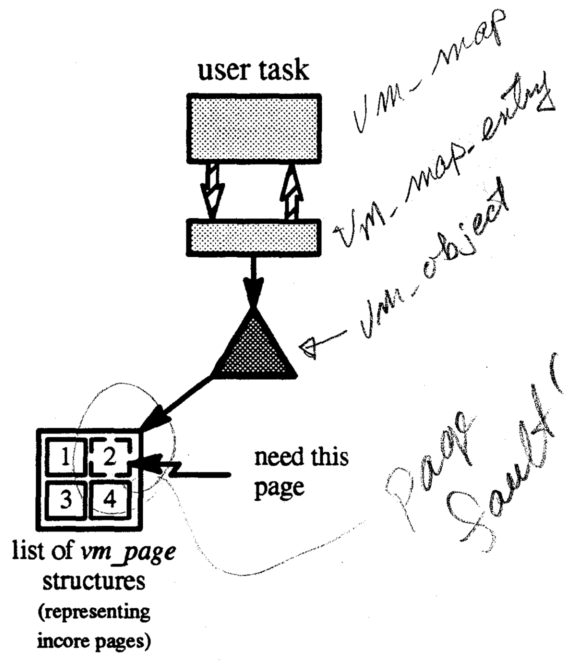
— restrict access to cached data (e.g. write-protect or read-and-write protect)

— clean or flush cached data

— set persistence and virtual copy attributes

4–17. Memory Objects

Pagein, part 1



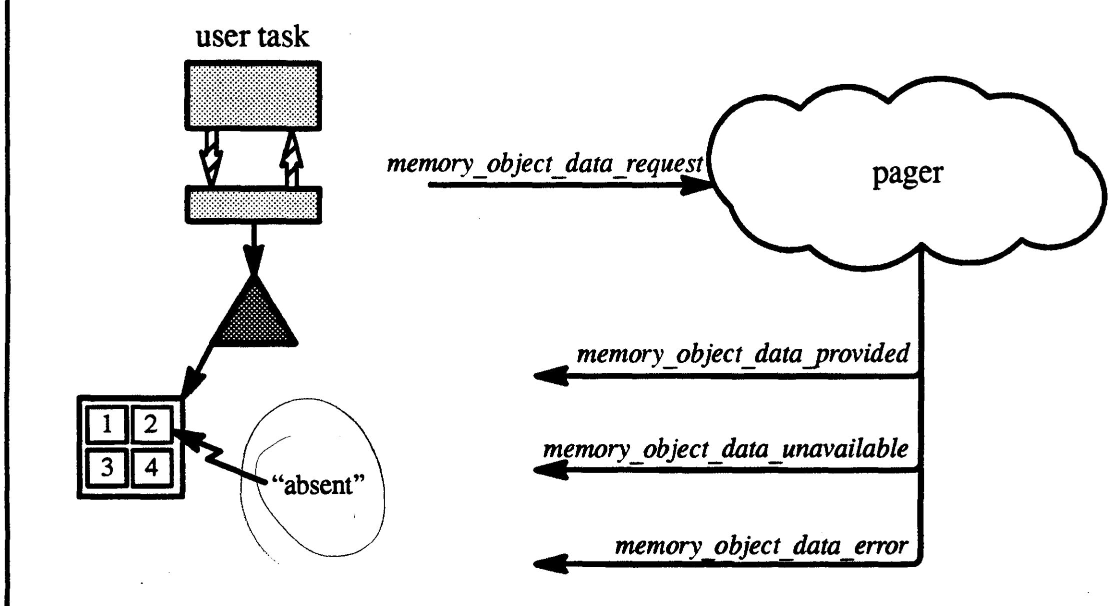
.H2596 4-18. 106 © 1991 Hewlett-Packard

Pagein, part 1

When a page fault occurs, the kernel's page fault handler first determines that the desired page is not incore. It consults the *vm_page* hash table to check if a *vm_page* structure for the desired page exists. If one does not, it must fetch the page from the associated pager.

4–18. Memory Objects

Pagein, part 2


user task
memory_object_data_request
pager
memory_object_data_provided
memory_object_data_unavailable
memory_object_data_error
1
2
3
4
"absent"

H2596 4-19. . 107 © 1991 Hewlett-Packard

Pagein, part 2

1. The kernel creates a *vm_page* structure for the desired page (page 2 in the example) and marks this page *absent* (indicating that a value for the page has not been found yet) and *busy* (indicating that an operation on the page is in progress).

2. The faulting thread requests the desired page by sending a *memory_object_data_request* message to the pager through the pager's memory object port (send rights for which are found in the *vm_object*).

3. The faulting thread blocks, awaiting a response.

4. The pager, using the memory object control port, either:

   a. returns the desired page (via a *memory_object_data_provided* message) and then turns off the *absent* indication in the *vm_page* structure.

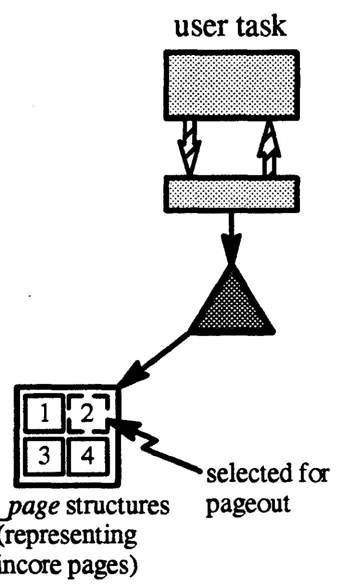
b. indicates that it does not have the desired page (by sending a *memory_object_data_unavailable* message), marks the page no longer busy, and wakes up the waiting threads.

c. indicates that an error occurred while fetching the page (by sending a *memory_object_data_error* message), marks the page no longer busy, and wakes up the waiting threads.

A "short-circuit" approach is used with the default pager, i.e., the vnode pager. Since this pager exists in the kernel, it does not need to be sent a message; it can be called directly. Thus a call is made to it in the context of the faulting thread and the page I/O occurs in this thread's context. Instead of sending a return message, the thread merely returns.

4–19. Memory Objects

Pageout, part 1

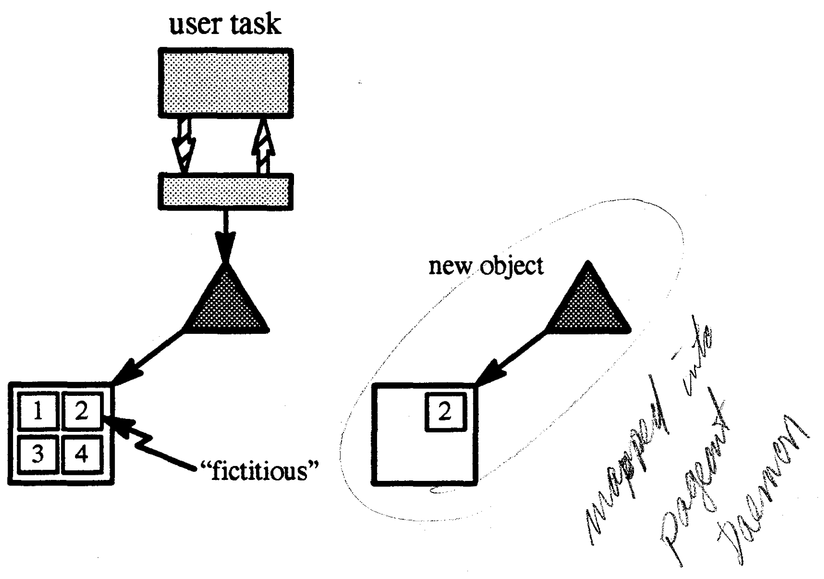


Pageout, part 1

Pageouts are performed in the context of a special kernel thread called the *pageout daemon* (as will be discussed). It selects a page to be freed and then contacts the appropriate pager.

4–20. Memory Objects

Pageout, part 2

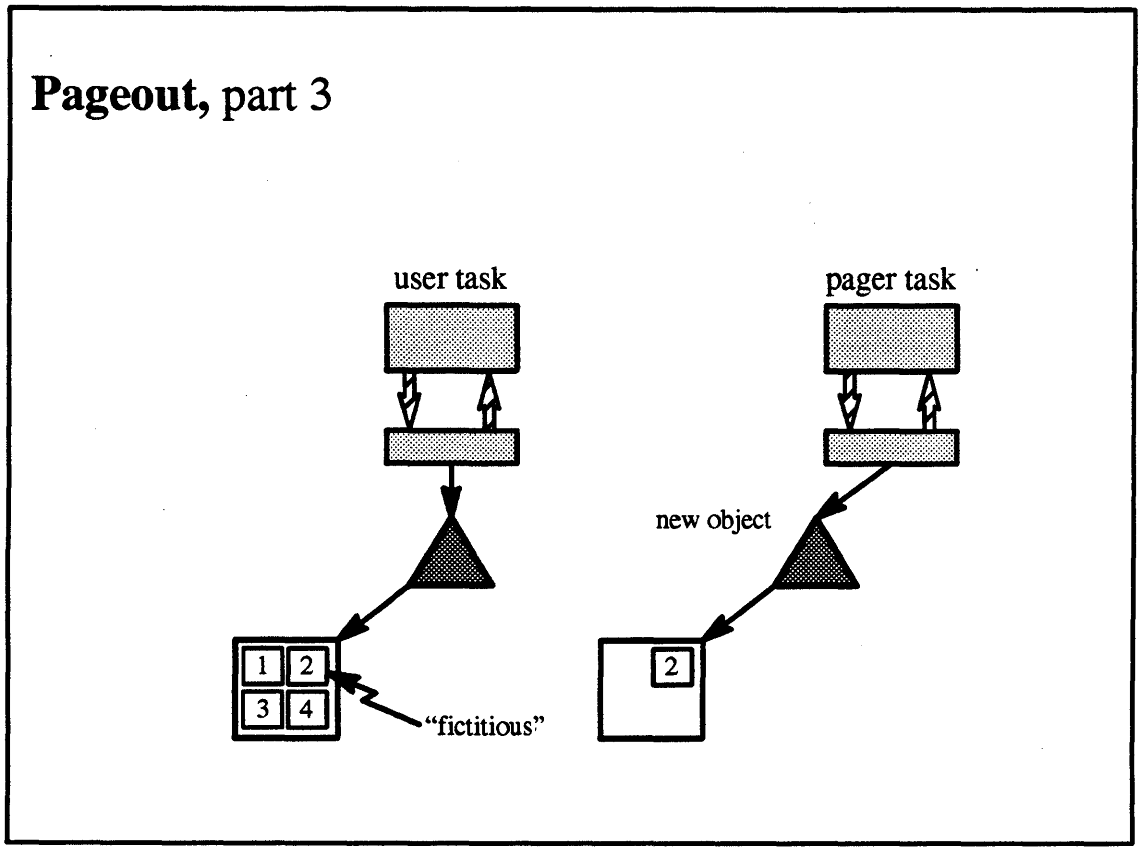


Pageout, part 2

The pageout daemon then:

1. locks the *vm_object* to prevent any other thread from manipulating the page in question
2. creates a new object
3. assigns to this new object the *vm_page* structure for the page to be paged out
4. assigns a new *vm_page* structure (marked "fictitious") to the original *vm_object* in place of the page being paged out
   a. this structure blocks any attempt to page the page in while it is being paged out
5. unlocks the original *vm_object*

4–21. Memory Objects



H2596 4-22. 110 © 1991 Hewlett-Packard

Pageout, part 3

The new object is sent to the pager as part of a *memory_object_data_write* message and is mapped into the pager task on receipt. The pager is now responsible for copying the page to some permanent storage. After it has done so, it issues a vm_deallocate system call to deallocate the page and thereby indicate that it has dealt with it.

Once the *memory_object_data_write* message has been successfully queued, the fictitious page is removed. The purpose of this page is to serialize *pagein*s with *pageout*s: we must make certain that pageins are dealt with after the pageout has been completed so that the most recent version of the page will be fetched. The fictitious page is placed in the original *vm_object*, forcing any thread that faults on this page to block until the fictitious page is removed. At this point the faulting thread sends a *memory_object_data_request* message that is queued after the *memory_object_data_write* message, thus serializing the messages and leaving it to the pager to maintain serialization.

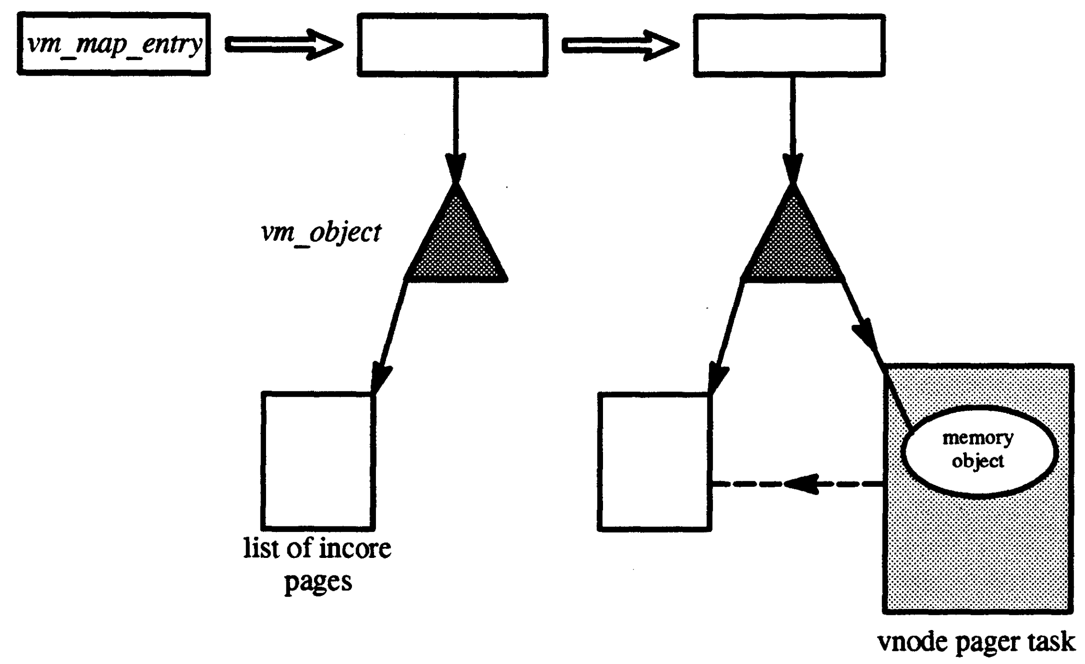
The original page is put into a new object just in case the pager does not complete the pageout quickly enough. The pager for this new object is set to be the vnode pager and the outgoing page is returned to the domain of the *pageout* daemon. If the page is not deallocated soon enough, then the *pageout* daemon will give the page to the vnode pager for a sure pageout.

The interface to the vnode pager is identical to that of other pagers. However, the *pageout* daemon is assured that the vnode pager will always complete a pageout. Thus the page being paged out is not returned to the domain of the *pageout* daemon, but instead is "wired," assuring that the *pageout* daemon will keep its hands off of it until the vnode pager has paged it out and deallocated it.

The vnode pager needs additional synchronization for serialization with a concurrent pagein request. Since pagein requests are short-circuited (are done in the context of the faulting thread as opposed to being handled by sending a message to the pager), this serialization, based on message order as described above, doesn't happen here. Instead, the vnode pager maintains a hash table of pageouts in progress. When a pagein of an outgoing page is attempted, the *pagein* thread must block until the pageout has been completed so that it does not page in stale data. The vnode pager normally pages to ordinary files (via the buffer cache). Thus a pageout effectively completes as soon as the page is copied into the buffer cache.

4–22. Memory Objects

Lazy Evaluation of Object Creation



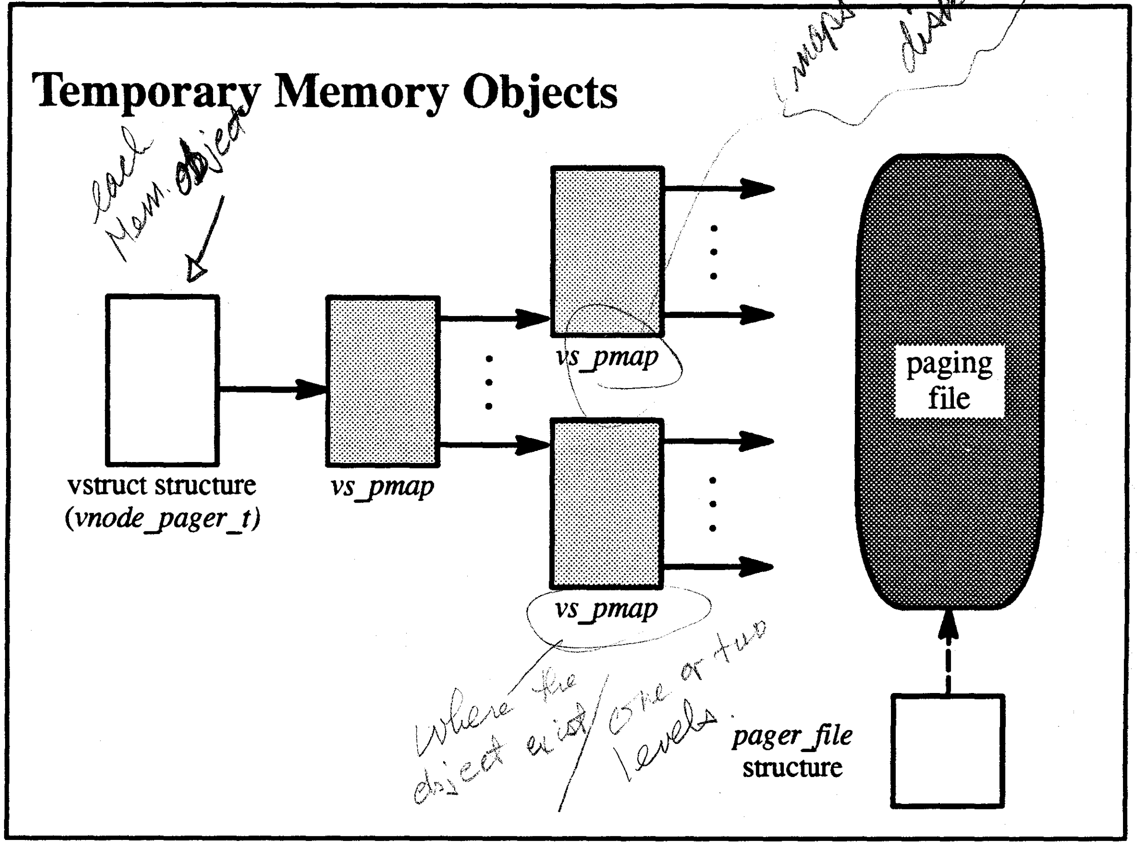
Lazy Evaluation of Object Creation

Memory objects and *vm_objects* are both created using lazy evaluation techniques. A vm_allocate system call creates a *vm_map_entry*, not a *vm_object*. The *vm_object* is created only when a page is actually accessed. Only then does the system set up the *vm_object* and link to it a *vm_page* structure for the accessed page. The vnode pager does not allocate a memory object until the pageout daemon issues a pageout request.

I/O can ~~Only~~ Occur through Buffer Cache

maps Virtual Mem to disk Address

4–23. Memory Objects



H2596 4-24. 112

Temporary Memory Objects

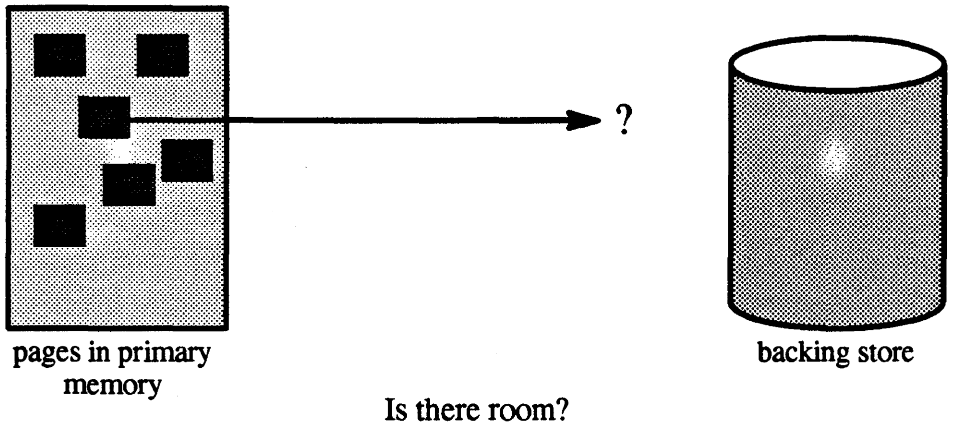
In a typical configuration, a fairly small number of paging files is set up for use by the vnode pager to back the pages of temporary memory objects. Each of these files is represented by a *pager_file* structure, which, among other things, gives the vnode for the file and a limit on its size. Each memory object is represented by a *vstruct* structure that indicates on which paging file the object is backed.

vs_pmap structures are used to indicate where pages of the object have been stored in the paging file. If the object's size is no more than 512 pages, then a single *vs_pmap* is used to map each of the pages to the paging file. For larger objects a two-level scheme is used: the first *vs_pmap* points to up to 512 *vs_pmaps*, each of which contains up to 512 pointers to where pages have been backed in the paging file.

Note that lazy evaluation is used as much as possible, so that the *vs_pmap*s (and space in the paging files) are allocated only when necessary to back up a page.

4–24. Memory Objects

Allocating Backing Storage



Is there room?

Allocating Backing Storage

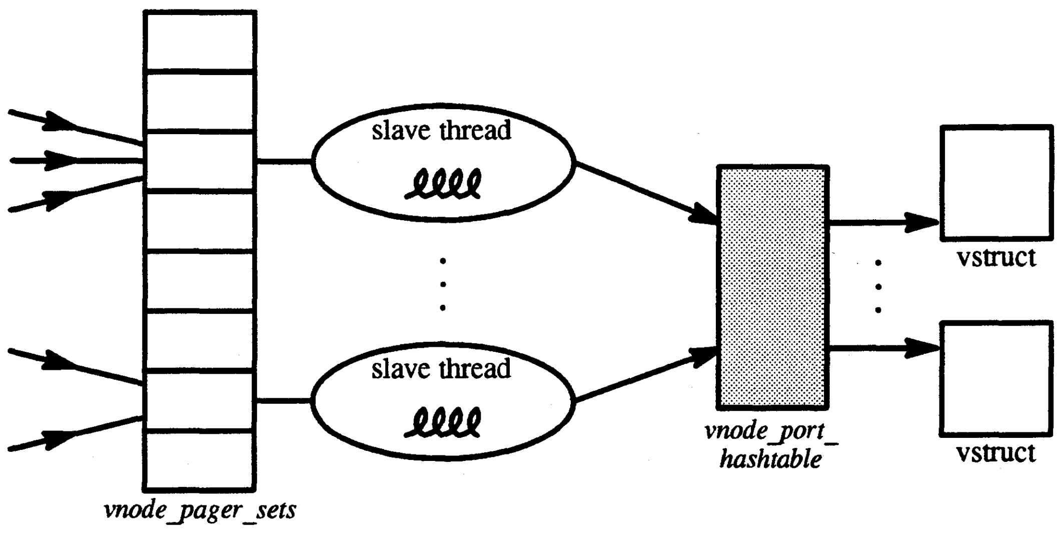
OSF/1 takes a liberal approach to the allocation of the backing store: backing store is allocated only when necessary, i.e., when a page must be written out. This approach differs from the conservative approach used in earlier versions of UNIX, in which backing store and virtual memory are allocated at the same time.

To see the difference between the two approaches, consider an extreme example: a system has 100 Mb of primary storage and 10 Mb of backing store. With the conservative approach, since all virtual memory must have backing store allocated for it, at most 10 Mb of the primary store can be used. With the liberal approach, a total virtual address space of 110 Mb can be used: 100 Mb in primary memory and 10 Mb on backing storage.

Unfortunately, with the liberal approach one may find out at a rather inopportune moment that there is no more backing store. Recovery from running out of backing store is not currently handled gracefully.

4–25. Memory Objects

Vnode Pager Task: Slave Threads



.H2596 4-26. 114 © 1991 Hewlett-Packard

Vnode Pager Task: Slave Threads

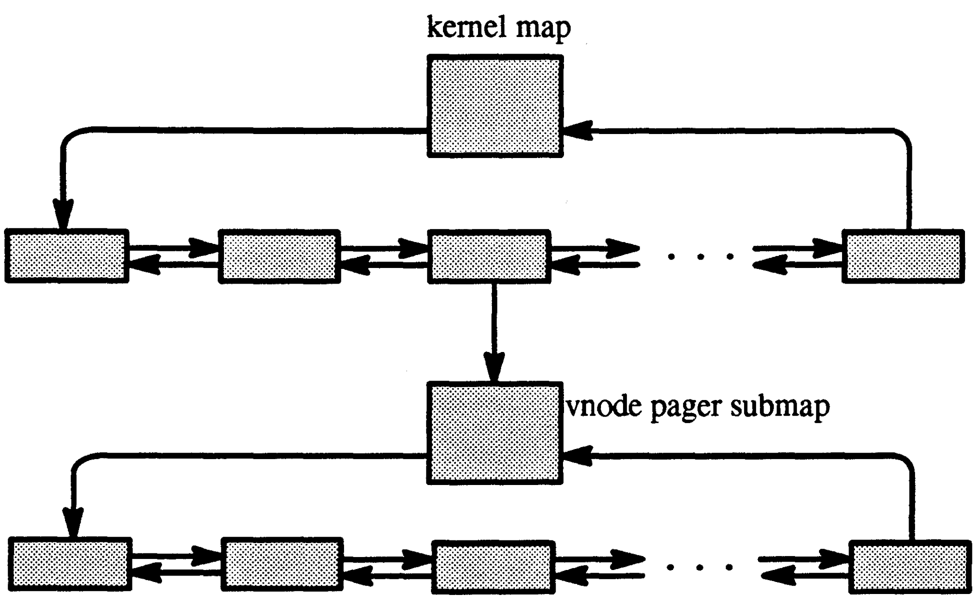
A number of threads (termed "slave threads") exists within the vnode pager task. Each such thread is responsible for a set of memory objects and deals with all requests coming on the memory objects ports for its memory objects. Given a request from a particular port, it looks this port up in the *vnode_port_hash_table* to determine the memory object's associated *vstruct* structure.

When a memory object is created it must be assigned to a slave thread. This is done by randomly chosing an index into the array *vnode_pager_sets*. Each entry of this array contains a *port set* (described in the ~~next~~ module) to which the memory object's memory object port is added. Using the port set mechanism, the slave thread receives messages sent through any of its *memory_object_ports*.

previous!

4–26. Memory Objects

Vnode Pager Task: Address Space



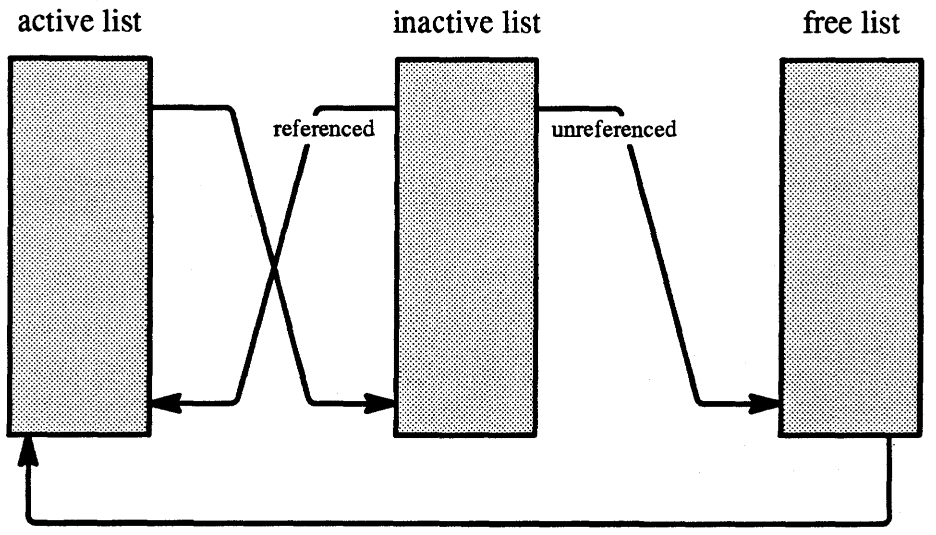
.H2596 4-27. 115 © 1991 Hewlett-Packard

Vnode Pager Task: Address Space

The vnode pager is implemented as a very special task. It has a *task* structure, a *u_task* structure, and contains threads, but its address space is the kernel address space. It has access to all of the kernel address space, though its private data structures are segregated within a special submap.

4–27. Memory Objects

Page Replacement



H2596 4-28. 116 © 1991 Hewlett-Packard

Page Replacement

Unlike the architecture-independent address space representation, page replacement is very simple. Each unwired page is on one of three lists, each maintained in FIFO order:

- free list
- inactive list
- active list

Whenever there is a memory shortage, a single kernel thread, the *pageout daemon*, is woken up by whichever thread in the kernel notices the memory shortage. It transfers enough pages from the inactive list to the free list to increase the size of the free list to a threshold. It examines each page in turn on the inactive list: if the page's *reference bit* is set, it transfers the page to the end of the active list; otherwise it transfers it to the end of the free list. If a page's *modified* bit is set, then the pageout daemon writes the page out to its memory object before it transfers the page to the free list.

After transferring pages to the free list, the pageout daemon checks to see if the inactive list has enough pages. If it does not, it transfers pages from the active list to the inactive list. As it does so, it turns off the pages' reference bits. The intent is that, once a page has been placed on the inactive list, it must be proved that this page is needed. The proof comes when a thread accesses the page, thus turning on the reference bit. Thus non-referenced pages eventually go to the free list; referenced pages go back to the end of the active list.

Pages on the free list may be reclaimed if they are referenced by a page before they are used for some other purpose.

If the hardware does not support a reference bit, a slightly different strategy is used. The translation entries for pages on the inactive list are marked invalid, thus forcing a page fault to occur when these pages are referenced. The page fault thus proves that the page is needed. These faulted pages are then moved to the end of the active list.

4–28. Memory Objects

mark a kernel stack as being unwired.

Swapping

Swapout

- *unwire* the kernel stack

pageout Daemon will page it out.

Swapin

- *fetch* and *wire* the kernel stack

Swapping

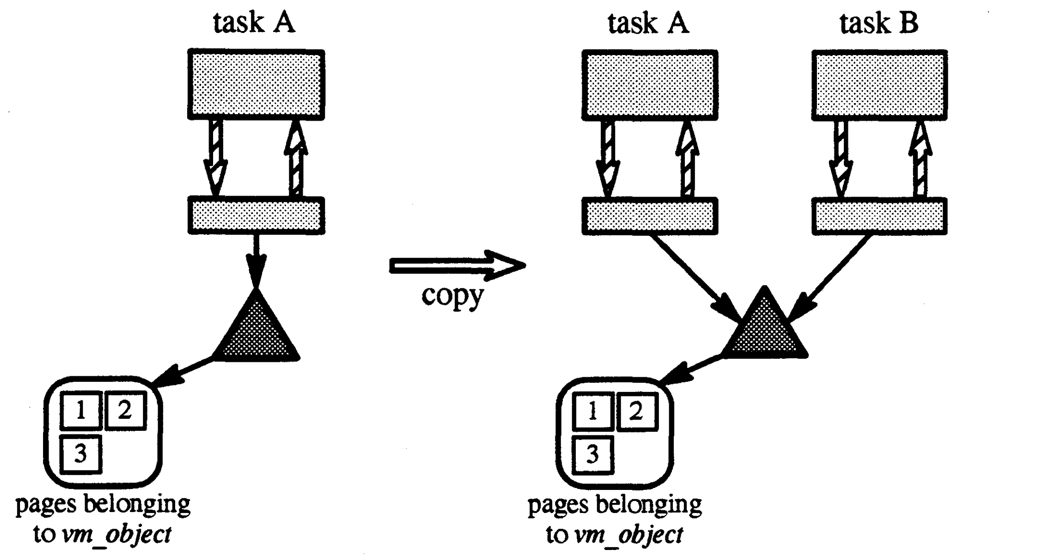
Swapping is handled by two kernel threads: a *swapout thread* and a *swapin thread.* The pageout daemon wakes up the swapout thread in response to memory shortages (but no more than once a minute). The swapout thread scans the list of all threads and swaps out those nonrunnable, interruptible threads that have been idle for more than 10 seconds.

The OSF/1 notion of "swapping out" is somewhat unusual: the thread is marked "swapped out" and its kernel stack is unwired. Nothing else happens to the thread immediately. Eventually, however, the pageout daemon will claim the pages of this thread. Since its kernel stack is unwired, these pages will be freed as well.

The swapin thread is responsible for swapping in threads. The swapin thread "swaps in" threads by wiring the thread's kernel stack. A swapped-out thread becomes a candidate to be swapped in when it is made runnable. Runnable swapped-out threads are placed on a *swapin list*; the swapin thread is woken up whenever a thread is placed on this list.

4–29. Copying and Sharing

Virtual Copy


task A
1
2
3
pages belonging to *vm_object*
copy
task A
task B
1
2
3
pages belonging to *vm_object*

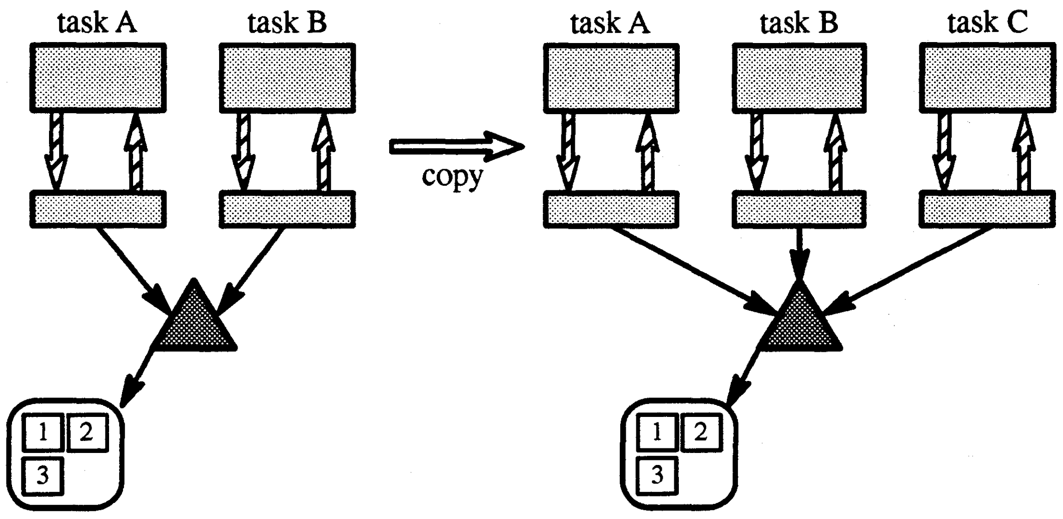
Virtual Copy

There are many situations in which it is necessary to make a logical copy of a range of pages. For example, after a UNIX fork system call, the child process has a copy of the parent's address space. When a task sends a message to another task, the recipient receives a copy of the message. A very useful optimization is *copy-on-write*, in which the "copying" is lazily evaluated, i.e., postponed in hope that it will not be necessary. Two tasks holding logical copies of a page can share the same physical page until one of them modifies it, at which point the modifier obtains a copy of the page to modify.

A thread in task A has just executed a fork system call, creating task B. We focus our attention on a range of addresses represented by a single *vm_map_entry* in task A, which is "copied" into task B. As long as neither task modifies any of the pages in this range, the pages are shared.

4–30. Copying and Sharing

Virtual Copy Redux



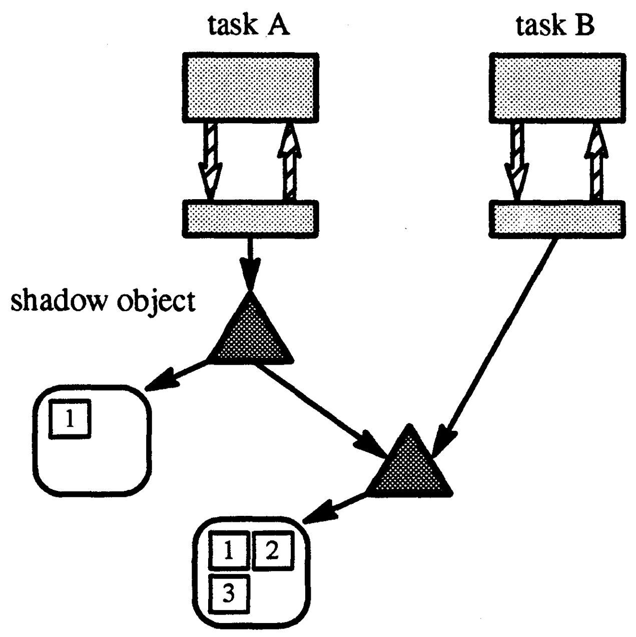
.H2596 4-31. 119

Virtual Copy Redux

Suppose that task B from page NO TAG executes a fork system call, creating task C. If none of tasks A, B, and C has modified any pages within the range, then they will continue to share all the pages.

4–31. Copying and Sharing

Virtual Copy and Modified Pages, part 1



H2596 4-32. 120

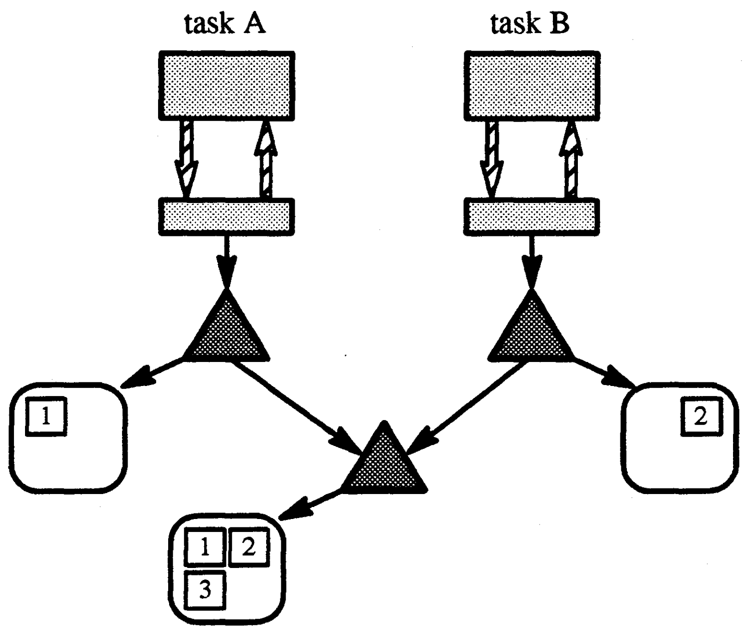
Virtual Copy and Modified Pages, part 1

This picture shows the situation of page NO TAG after a thread in task A has modified page 1. To represent the pages that a task has modified, and thus those pages that are now private to the task, the system creates a *shadow object.*

The picture shows the architecture-independent representation of the address spaces for tasks A and B: task A has its private version of page 1, but uses the original versions of pages 2 and 3; task B uses the original versions of pages 1, 2, and 3.

4–32. Copying and Sharing

Virtual Copy and Modified Pages, part 2



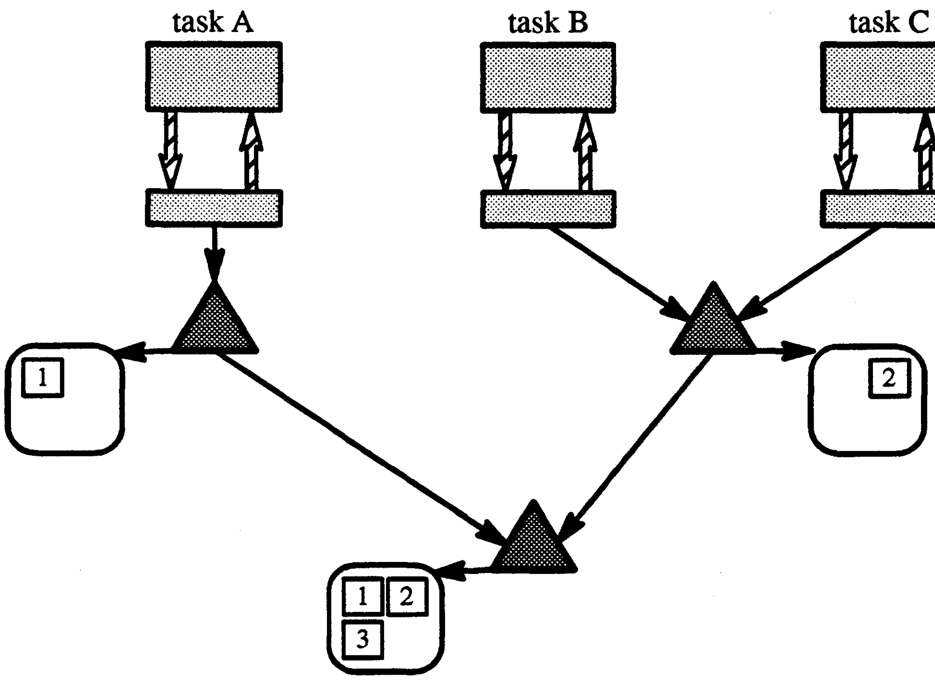
.H2596 4-33. 121 © 1991 Hewlett-Packard

Virtual Copy and Modified Pages, part 2

This picture shows the situation in the previous picture after a thread in task B has modified page 2. Another shadow object has been created, this time to represent those pages which are private to task B.

4–33. Copying and Sharing

Virtual Copy and Modified Pages, part 3



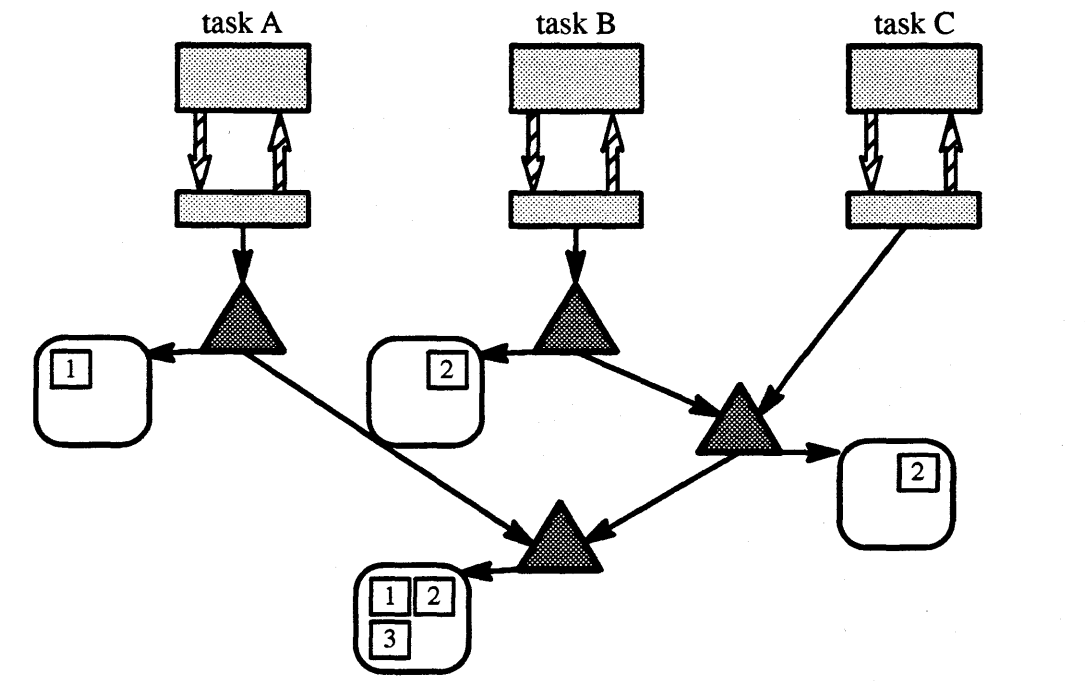
4-34. 122 © 1991 Hewlett-Packard

Virtual Copy and Modified Pages, part 3

In this picture, task B has executed a fork system call, creating task C. As long as neither task B nor task C modifies any pages, the situation will be as shown here: the two tasks share the version of page 2 referred to by the shadow object and use the versions of pages 1 and 3 referred to by the original object.

4–34. Copying and Sharing

Virtual Copy and Modified Pages, part 4



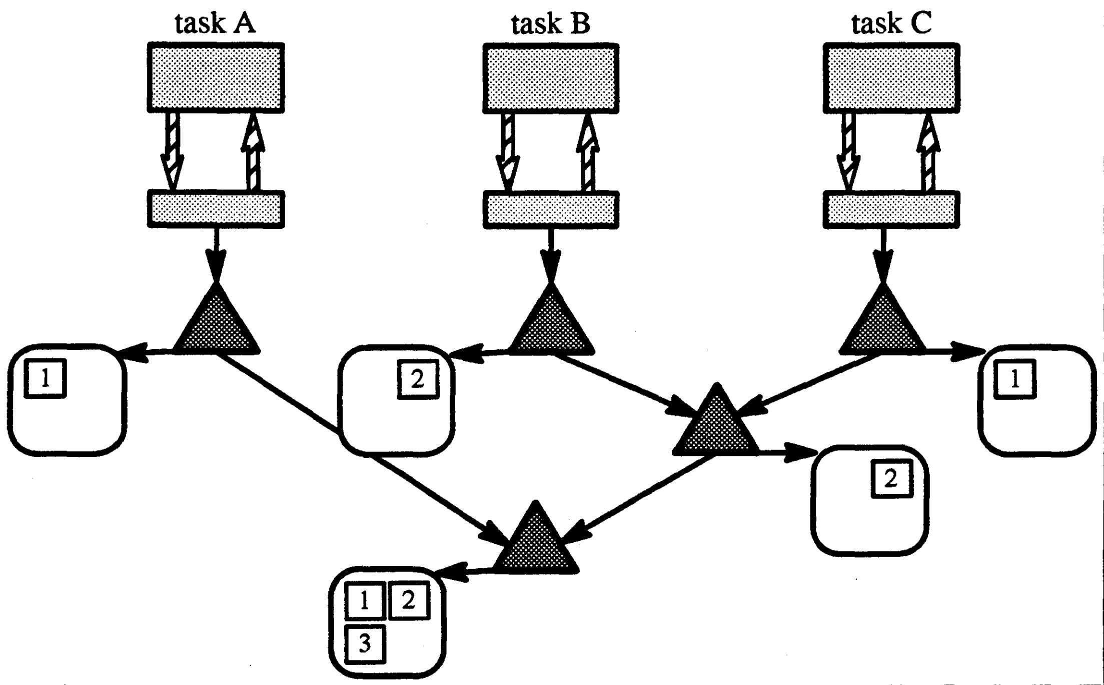
.H2596 4-35. 123 © 1991 Hewlett-Packard

Virtual Copy and Modified Pages, part 4

A thread in task B has further modified page 2, necessitating the creation of still another shadow object to represent what are now task B's private pages.

4–35. Copying and Sharing

Virtual Copy and Modified Pages, part 5



.H2596 4-36. 124 © 1991 Hewlett-Packard

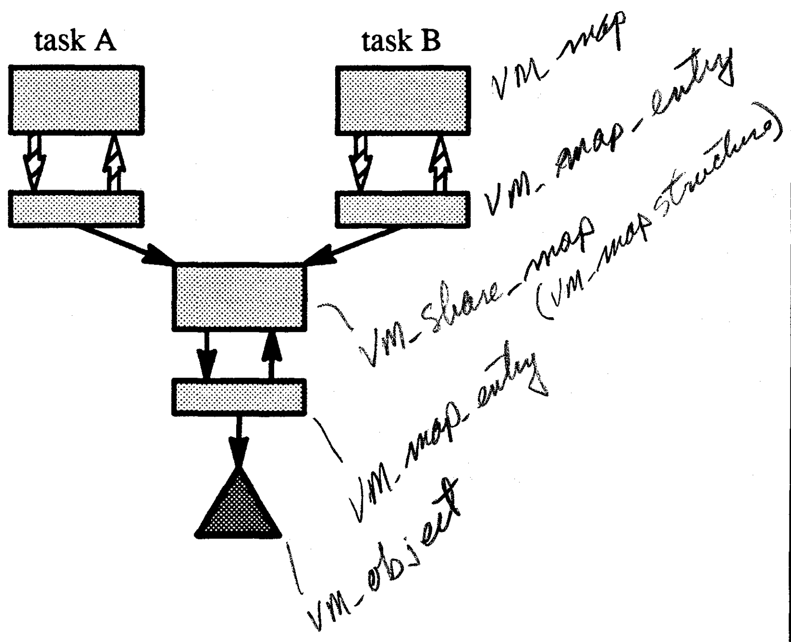
Virtual Copy and Modified Pages, part 5

A thread in task C has modified page 1, resulting in the creation of yet another shadow object.

This ~~is~~ (these last couple Pages showing Copying & Sharing) is for Temp. object only — not Permanent (files) objects.

4–36. Copying and Sharing

Sharing



H2596 4-37. 125 © 1991 Hewlett-Packard

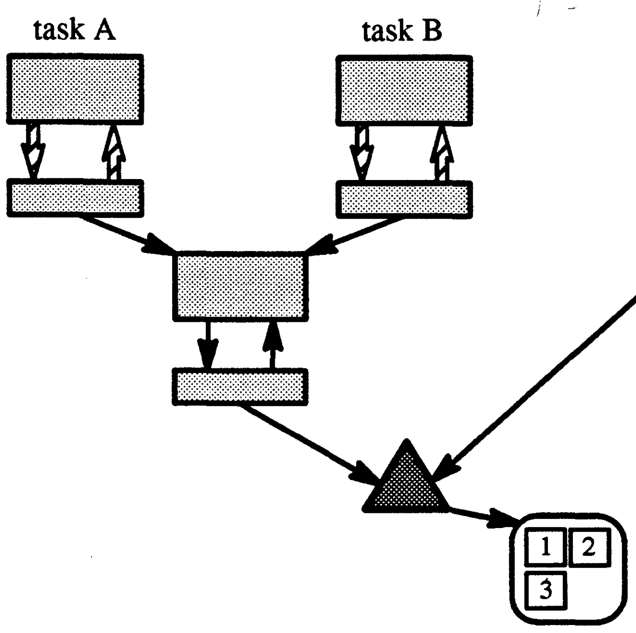
Sharing

Multiple tasks occasionally share portions of their address spaces with one another. The most straightforward representation of this would be for the appropriate *vm_map_entry* of each task to point to the one *vm_object* representing the shared memory. However, this representation is already taken: it is used for copy-on-write. A separate memory map, called the *share map*, represents the shared memory. This map consists of a *vm_map* structure heading a linked list of *vm_map_entrys*, each of which points to a *vm_object*. The *vm_map_entrys* of the tasks sharing this memory point to the share map (not to *vm_objects*).

4–37. Copying and Sharing

Share Then Copy, part 1

fork

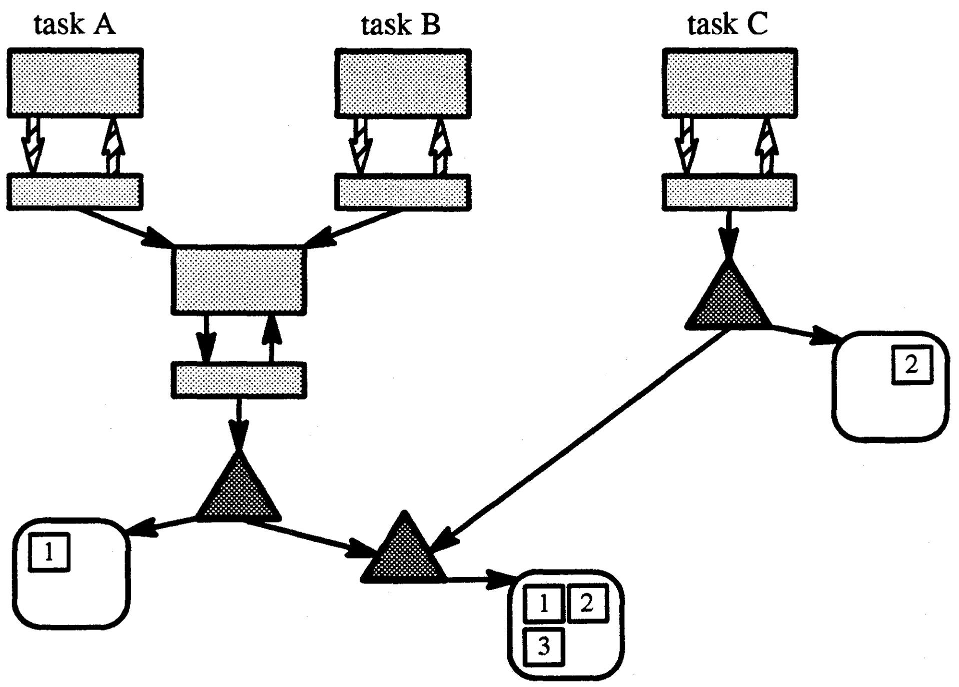


Share Then Copy, part 1

Fairly complex memory representations can be achieved by performing numerous copy and share operations. In the picture, the original object is shared by tasks A and B. Task B has created a child task C, but this portion of the address space is (virtually) copied into task C.

4–38. Copying and Sharing

Share Then Copy, part 2



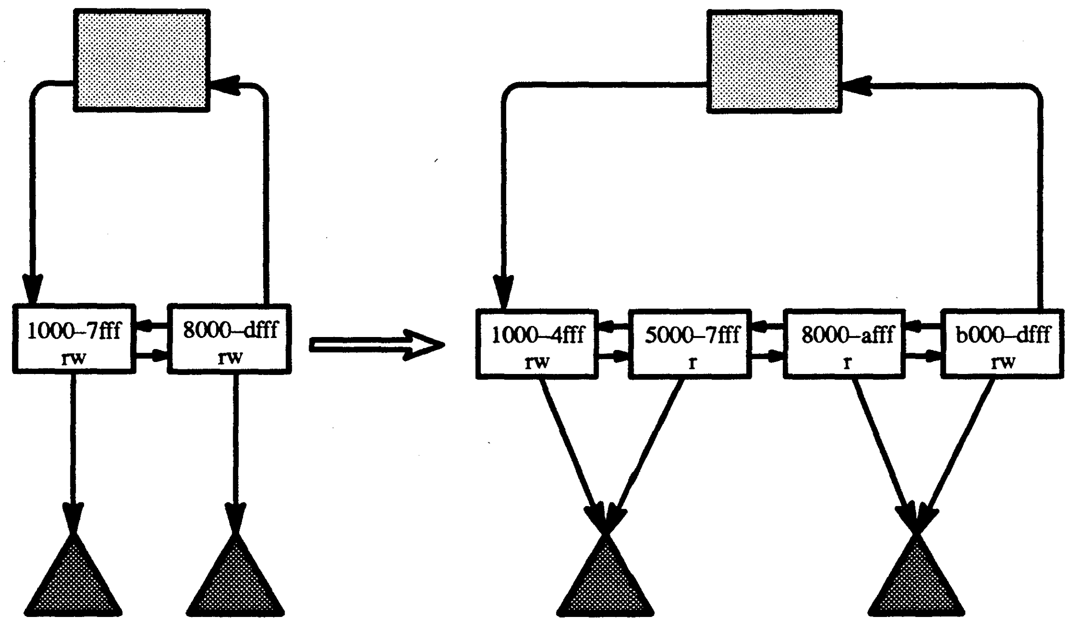
Share Then Copy, part 2

Task C has modified page 2 and either task A or task B has modified page 1.

This picture illustrates why a separate map is needed when tasks share a portion of their address space. An alternative representation might be for the *vm_map_entrys* of tasks A and B to point directly to the *vm_object*. However, this would complicate the creation of the shadow object needed in this model to represent the modified copy of page 1. Without a share map, it would be necessary to track down all of the *vm_map_entrys* that point to a *vm_object* and then change them to point to the new shadow object.

4–39. Copying and Sharing

Clipping: Changing Attributes



H2596 4-40. 128 © 1991 Hewlett-Packard

Clipping: Changing Attributes

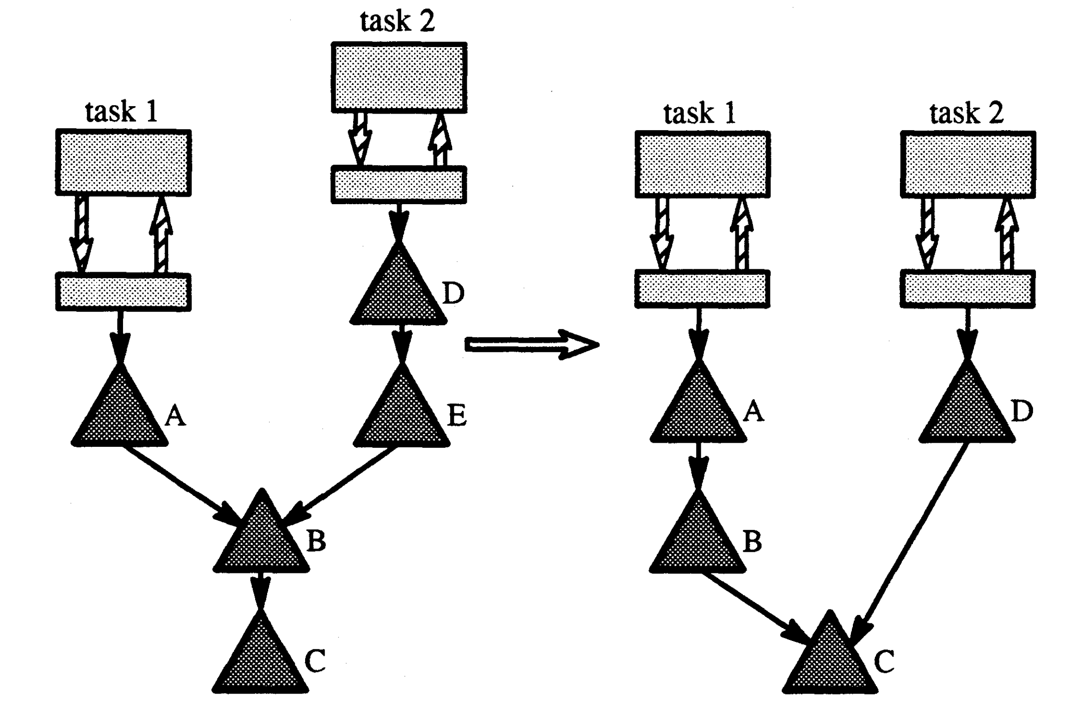
Programmers do not see the organization of the address space imposed by the *vm_map_entrys*. Instead, they see the address space as a collection of pages; i.e., the only important boundary is the page boundary (a system call is available to determine the page size).

In particular, programmers can adjust the protection on arbitrary ranges of pages by using the *vm_protect* system call. The use of this call might well result in the creation of new *vm_map_entrys* to represent the new view of the address space.

This picture illustrates the effect of using *vm_protect* to set a range of pages, previously read-write, to be read-only. The affected pages span two *vm_map_entrys*, each of which must be split in two to allow a read-write portion and a read-only portion.

4–40. Copying and Sharing

Collapsing Objects



H2596 4-41. 129 © 1991 Hewlett-Packard

Collapsing Objects

Shadow chains can become fairly lengthy after a series of virtual copy operations. A couple of simple rules are employed to reduce their length.

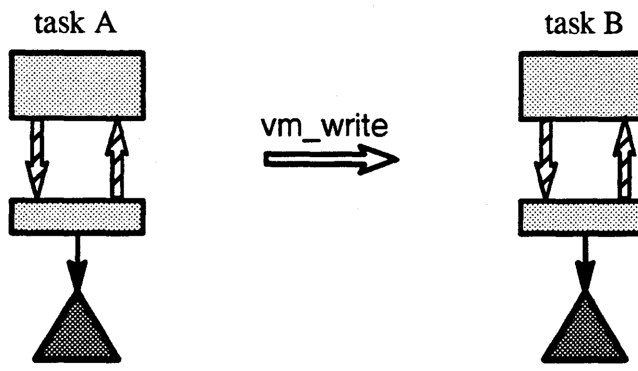
The first rule is that if a *vm_object* is pointed to by only a single *vm_object* via a shadow link, then it is not necessary to have both objects: they may be combined into a single object.

The second rule is a bit more complicated. If a shadow chain links three *vm_objects* and all pages of the middle object are shadowed by the objects above it, then the middle object is unnecessary and can be eliminated: the top object's shadow link is changed to point directly at the bottom object. To apply this rule, no pages in either *vm_object* can be paged out (otherwise it is too cumbersome to determine if the upper object completely shadows the lower).

Due to complications with locking, these optimizations can be performed only in the context of one task at a time. They are done when the shadow links are being traversed anyhow, for instance while a page fault is being handled.

4–41. Copying and Sharing

The Virtual Copy Operation: Permanent Objects



.H2596 4-42. 130 © 1991 Hewlett-Packard

The Virtual Copy Operation: Permanent Objects

As mentioned previously, the virtual copy operation is extremely important. It is used as part of:

- fork
- message passing
- vm_read, vm_write

It is essential that a virtual copy be quick. However, if a permanent object is involved, the standard copy-on-write optimization must be performed with care: all changes to the associated virtual memory must be reflected back into the permanent object.

Suppose that a thread in task A uses vm_write to copy data from its address space into a portion of task B's address space, into which a permanent object (e.g., shared mappings of memory-mapped files or an external memory object) has been mapped. If it weren't for the fact that the object was permanent, the vm_write could be easily optimized using copy-on-write techniques. For example, task B's *vm_map_entry* could be set to point directly to the *vm_object* of task A. However, the copy-on-write optimization will not work in this case because it would effectively unmap the permanent object from task B. The system must ensure that changes to this portion of task B's address space get back to the permanent memory object.

4–42. Copying and Sharing

Optimizing the Virtual Copy Operation

- Three parties are involved:
  - — the server (i.e. the memory object manager)
  - — the client (i.e. the task that maps the permanent memory object)
  - — the copier (i.e. the task that is the target of the virtual copy)
- From the copier's viewpoint, the mapped object should be a snapshot of its state taken at the time of the virtual copy

Optimizing the Virtual Copy Operation

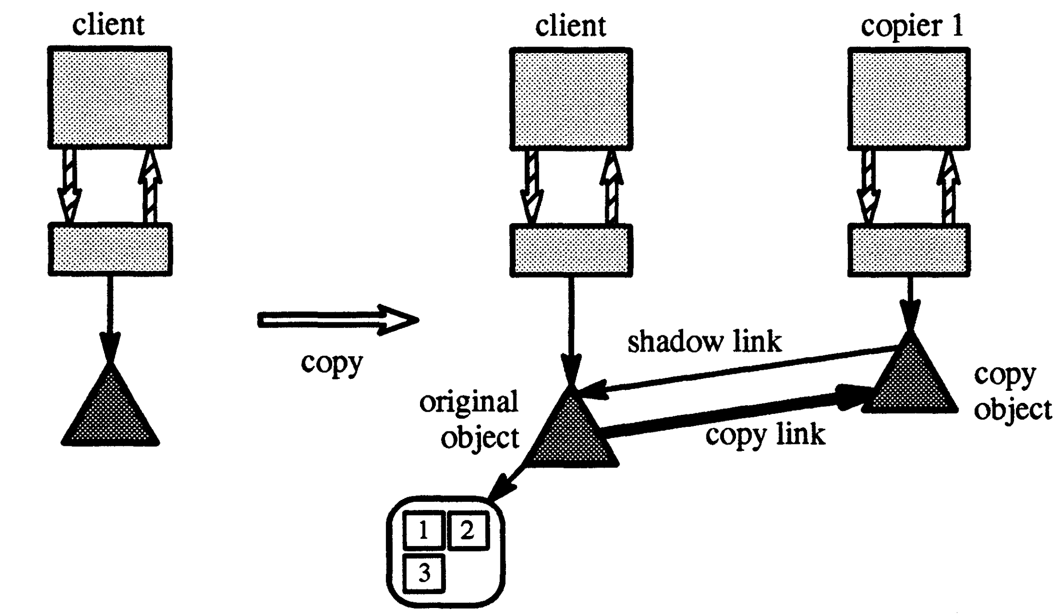
Suppose now that we are making a virtual copy of a portion of an address space into which a permanent object has been mapped; the virtual copy is a temporary object, not a permanent object.

Immediately after a virtual copy operation takes place, both the client and the copier should "see" the original value of the object. However, the copier's changes to the object should have no effect on the object itself, but should change only the copier's private view of the object. The client's changes to the object, however, must affect the object, so that all other clients that have mapped the same memory object see the changes. Furthermore, any changes made by any of these other clients, even if they reside on other computers, will be seen by this client. The major problem is determining whether such changes occurred before or after the virtual copy, and thus whether or not they should affect the copier.

4–43. Copying and Sharing

Virtual Copy from Permanent Objects:

COPY_DELAY, part 1



.H2596 4-44. 132 © 1991 Hewlett-Packard

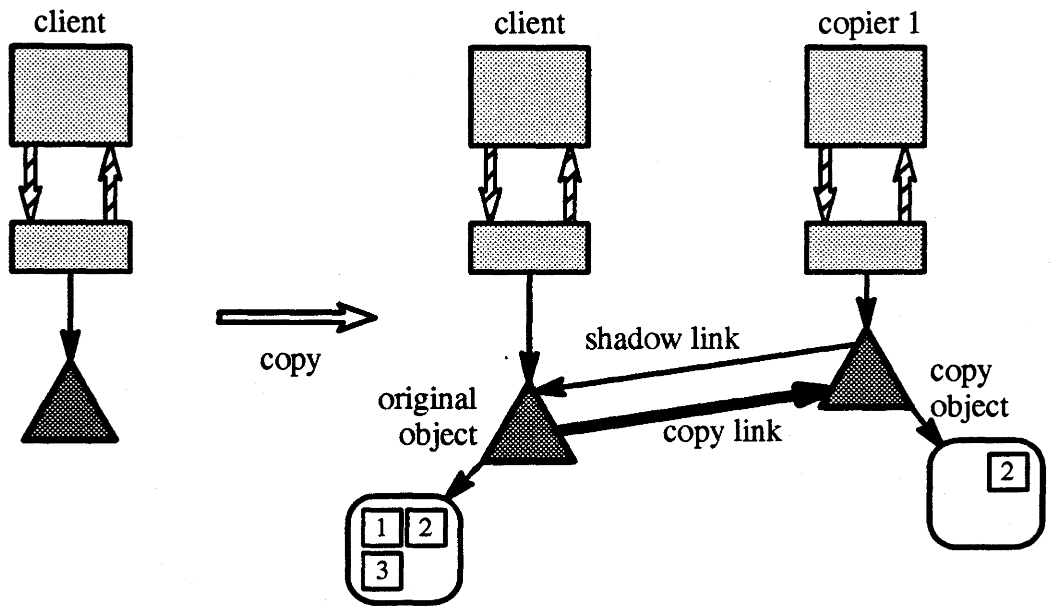
Virtual Copy from Permanent Objects: COPY_DELAY, part 1

In the simplest case, the memory object manager and all of its clients are on the same machine. Thus the kernel is immediately aware of any change made to the object. The major concern here is to make certain that all changes clients make to the object are reflected in the object itself. The representation of memory after a virtual copy is necessarily asymmetric, since the copier's changes to the object are reflected only in the copier's view and backed up by a temporary memory object, while the client's changes to the object are sent to the original object.

After a virtual copy, the client's view of the object is unchanged except that, whenever it modifies a page of the original object, the kernel must first copy the original version of the page to a *copy object* in the copier's view. The kernel finds the copy object by following a special *copy link*. Thus the copier is always assured of seeing the original version of the object.

4–44. Copying and Sharing

Virtual Copy from Permanent Objects: COPY_DELAY, part 2



.H2596 4-45. 133

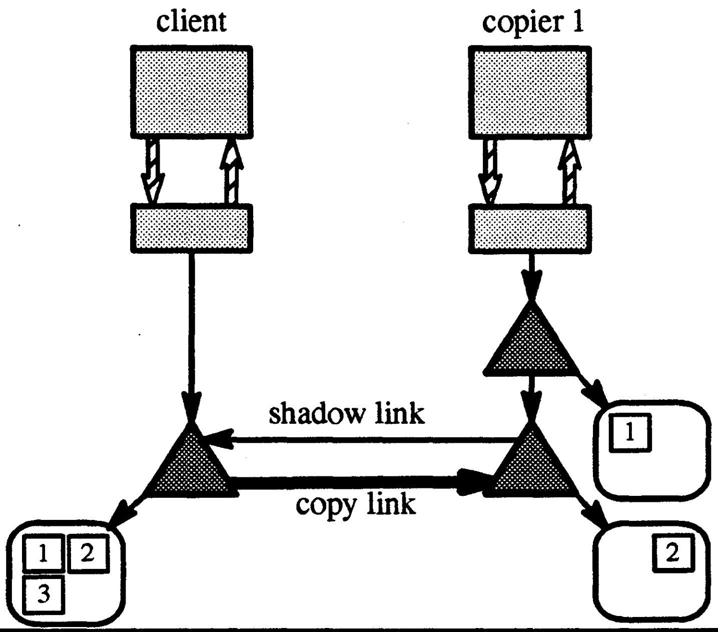
Virtual Copy from Permanent Objects: COPY_DELAY, part 2

In this picture the client has modified page 2, so the original value of page 2 is first copied to the copy object and the client now modifies the original.

4–45. Copying and Sharing

Virtual Copy from Permanent Objects:

COPY_DELAY, part 3



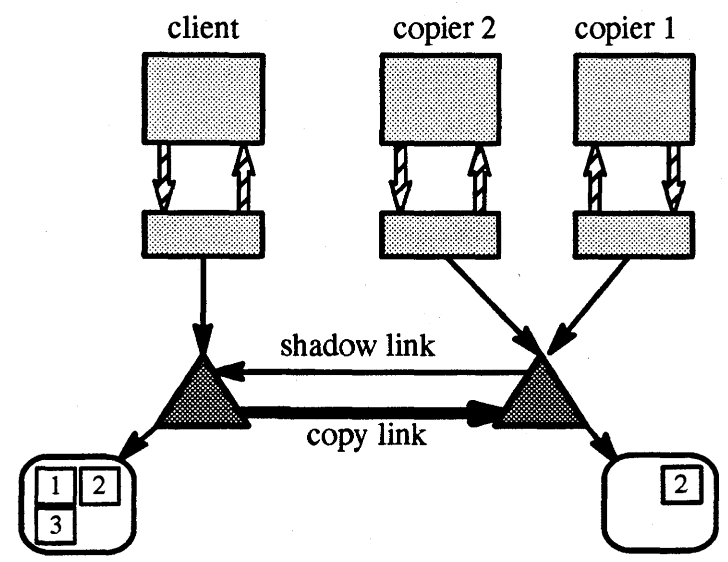
.H2596 4-46. 134 © 1991 Hewlett-Packard

Virtual Copy from Permanent Objects: COPY_DELAY, part 3

The copier now modifies page 1. A new shadow object is created for the copier and a copy of page 1 is attached to it.

4–46. Copying and Sharing

Virtual Copy from Permanent Objects: COPY_DELAY, part 4



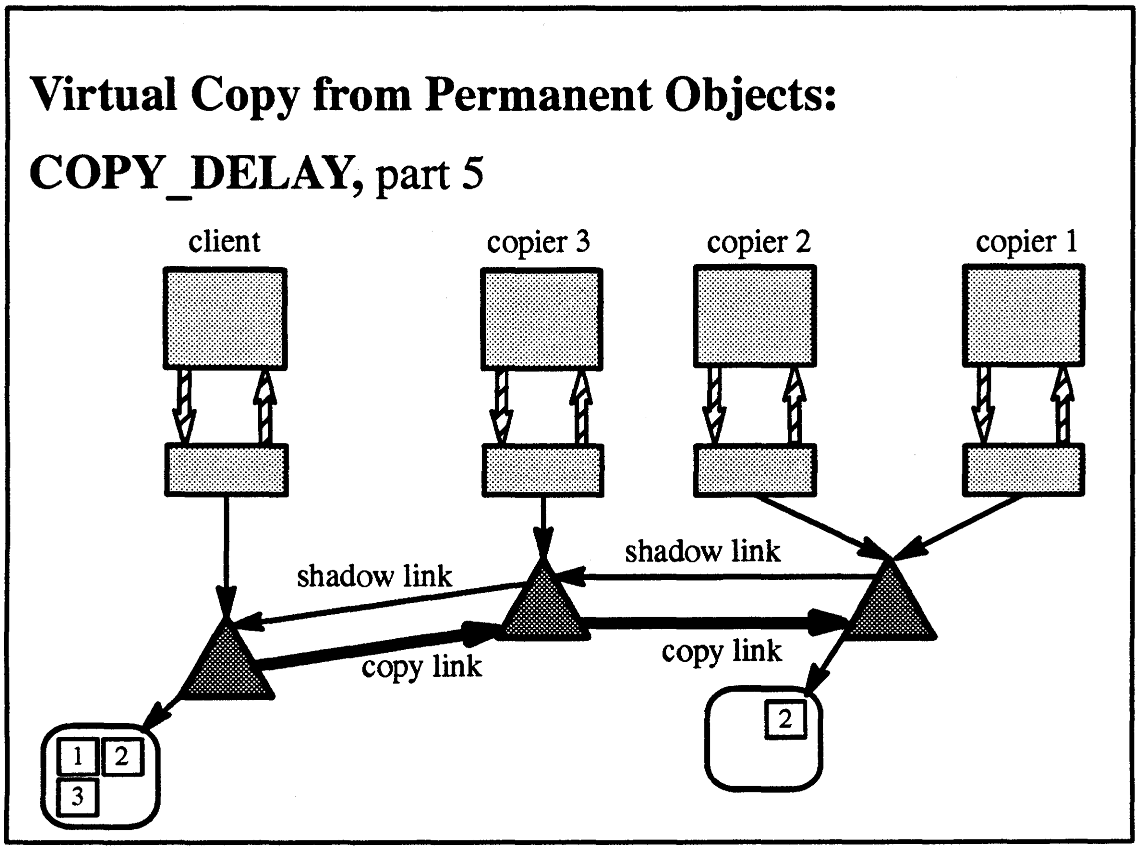
.H2596 4-47. 135

Virtual Copy from Permanent Objects: COPY_DELAY, part 4

The copier of page NO TAG has executed a fork system call, creating copier 2.

4-44

4–47. Copying and Sharing



Virtual Copy from Permanent Objects: COPY_DELAY, part 5

.H2596 4-48. 136 © 1991 Hewlett-Packard

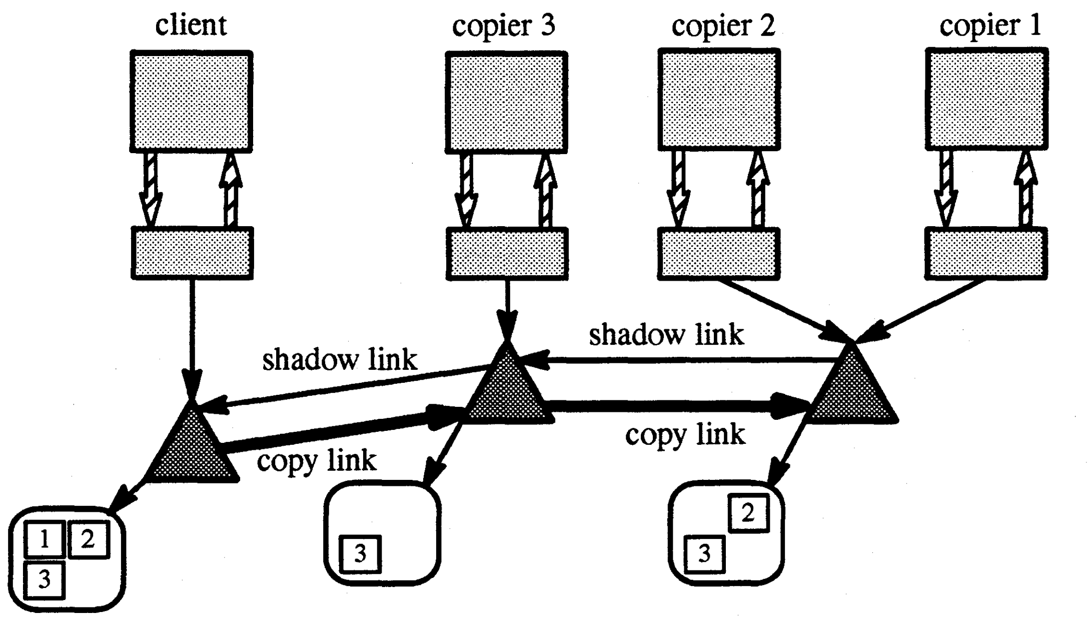
Virtual Copy from Permanent Objects: COPY_DELAY, part 5

Starting from the previous picture, the client has fork'd once again, creating copier 3, which must start with the same view of the object as that of the client.

4–48. Copying and Sharing

Virtual Copy from Permanent Objects:

COPY_DELAY, part 6



.H2596 4-49. 137

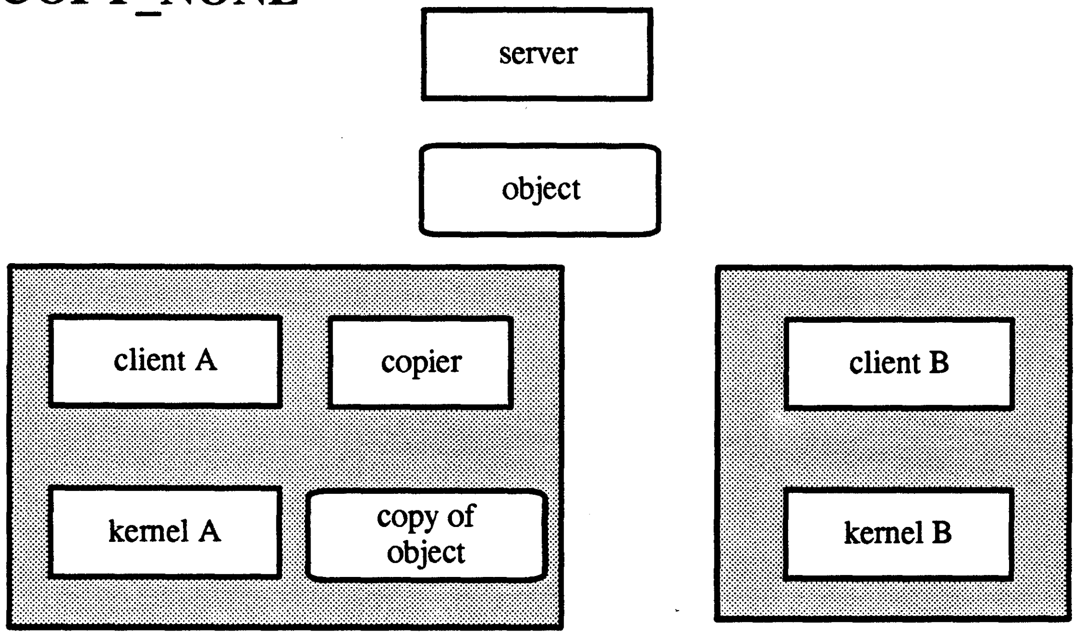
Virtual Copy from Permanent Objects: COPY_DELAY, part 6

The client now modifies page 3. In order to preserve the views of all of the copiers', copies of page 3 are propagated across the copy links to each of the copy objects.

4–49. Copying and Sharing

Virtual Copy from Permanent Objects:

COPY_NONE



Virtual Copy from Permanent Objects: COPY_NONE

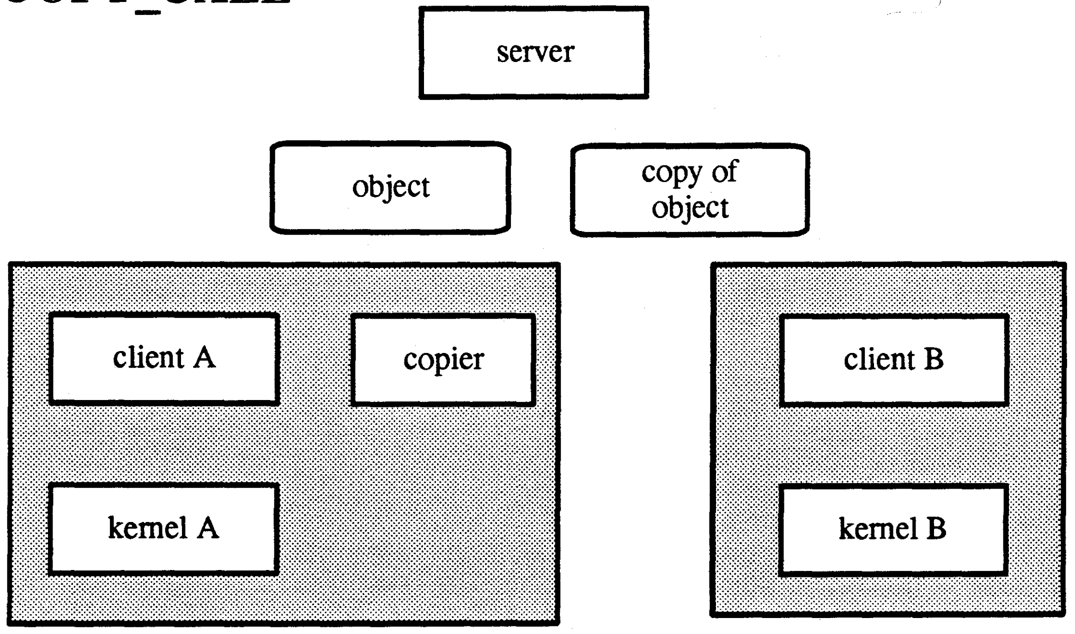
The COPY_DELAY technique does not work if clients on other machines are modifying the objects. The problem is that the local kernel does not know whether such changes took place before or after the virtual copy. Only the server knows for sure. If the server is not prepared to deal with this uncertainty, then the virtual copy must be implemented as a physical copy. That is, we ensure that the copier sees a snapshot of the object taken at the time of the copy by physically copying all of its pages at that moment.

4–50. Copying and Sharing

DOES NOT WORK in OSF/1 Ver. 1.1

Virtual Copy from Permanent Objects:

COPY_CALL



.H2596 4-51. 139 © 1991 Hewlett-Packard

Virtual Copy from Permanent Objects: COPY_CALL

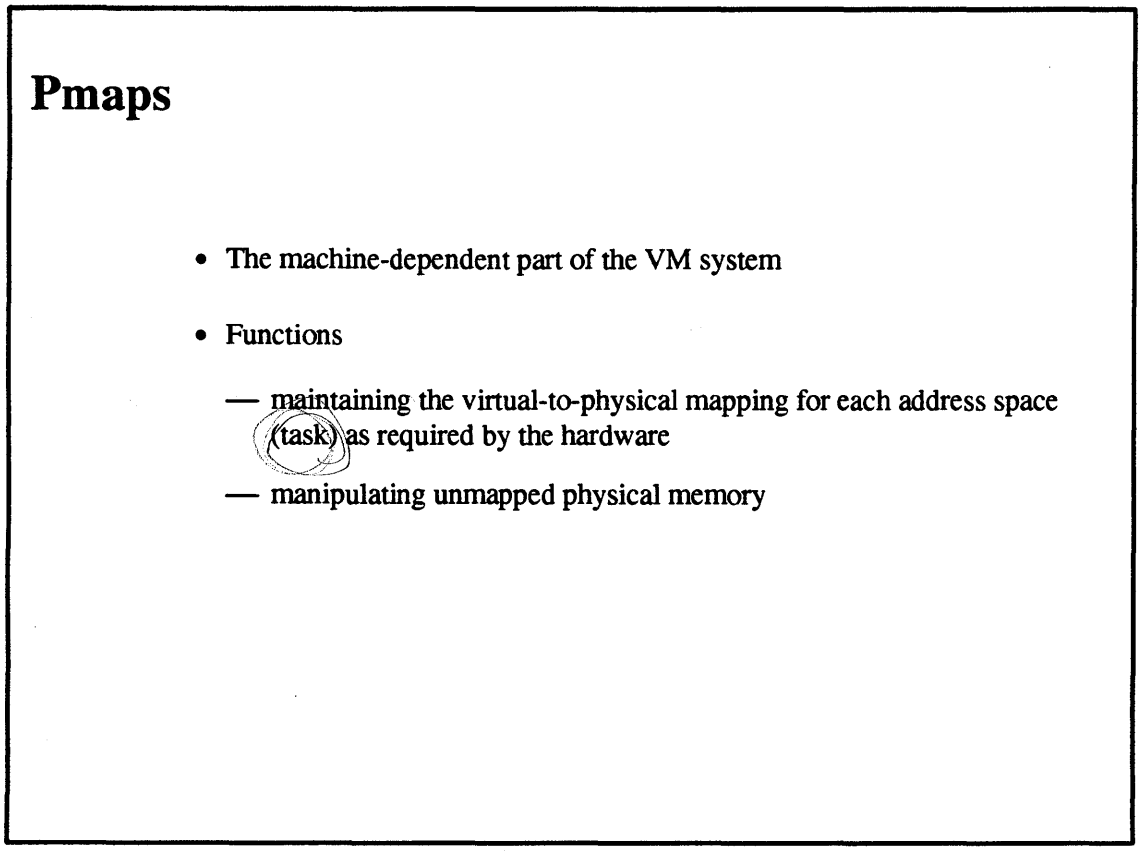
This technique augments the interface between the kernel and the server so that the server can manage the snapshot views of the various copiers of the object. Each time a virtual copy is performed, the kernel notifies the server. The server then receives rights to a port that it uses to represent the snapshot. All pages of the object that are in primary memory are marked read-only so that the server can handle each write-fault. Thus the server is given enough information to allow it to perform the job that the kernel performs with the COPY_DELAY option.

VM

PMAP →

MMU

4–51. The Pmap Module



Pmaps

- The machine-dependent part of the VM system
- Functions
  - — maintaining the virtual-to-physical mapping for each address space (task) as required by the hardware
  - — manipulating unmapped physical memory

H2596 4-52. 140 © 1991 Hewlett-Packard

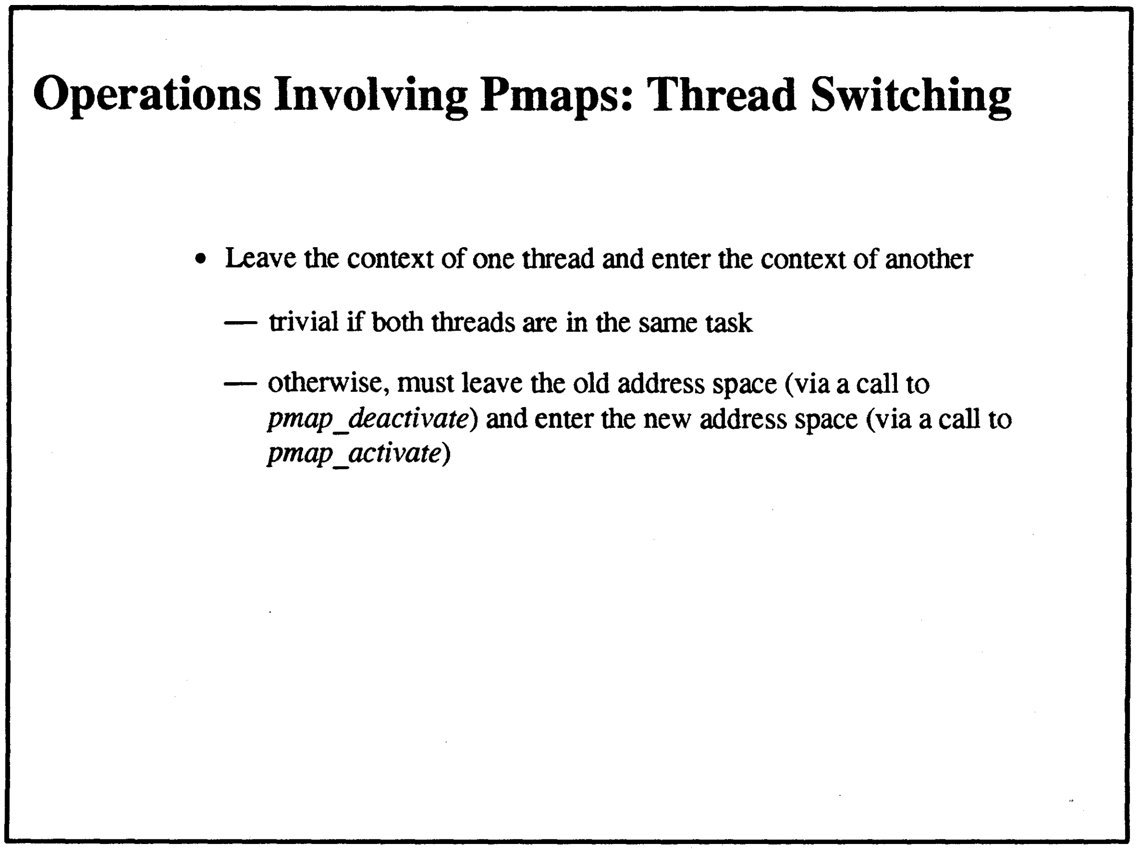
Pmaps

The *pmap* module maintains whatever hardware-mandated data structures are required to map virtual to physical addresses. These mappings need not be complete: all that is required is that enough mapping information be available to the hardware to satisfy the current reference.

Following the principle of lazy evaluation, physical mapping information is typically set up on demand, i.e., when it is needed to satisfy a reference. As threads within a task reference virtual memory, physical mapping information continues to be built up. However, this information may be deallocated when necessary, for example, to cope with shortages of memory.

The other function of the *pmap* module is to manipulate physical memory directly. For example, if the kernel must copy into an unmapped address space, it must call upon the *pmap* level to perform this operation.

4–52. The Pmap Module



Operations Involving Pmaps: Thread Switching

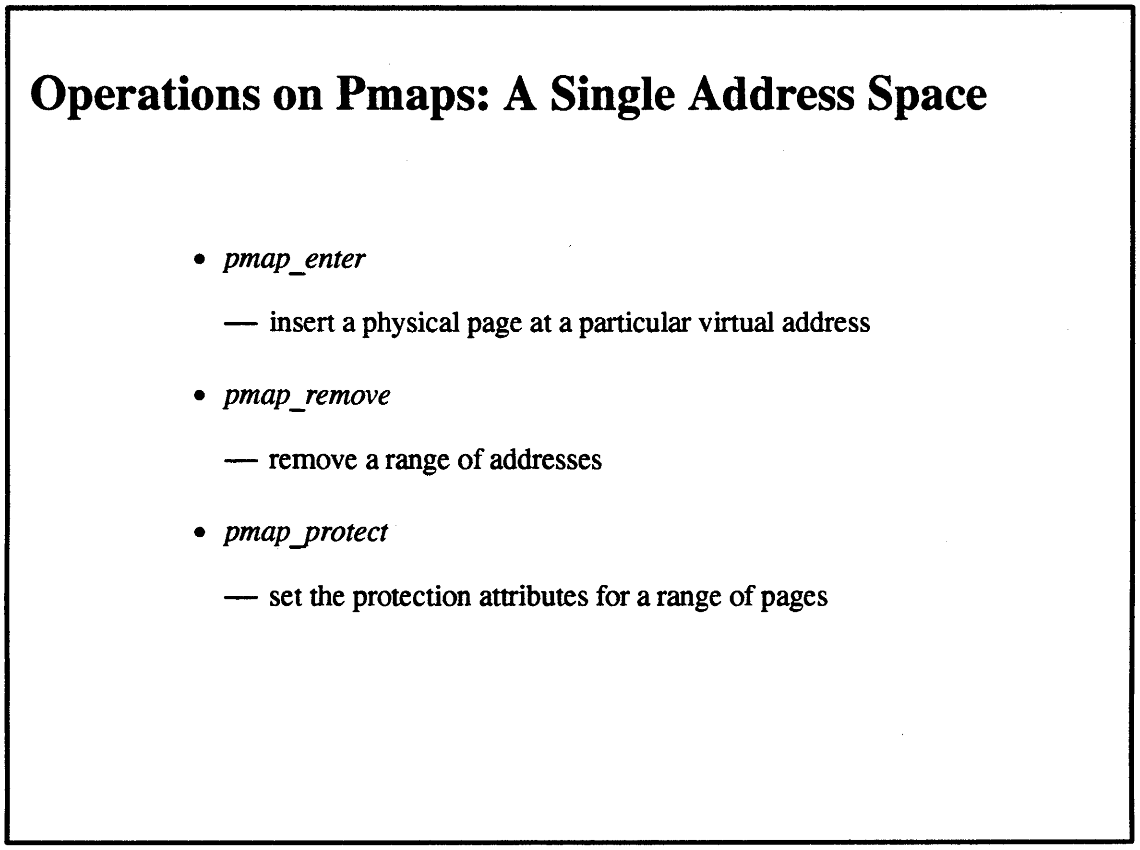
- Leave the context of one thread and enter the context of another
  - — trivial if both threads are in the same task
  - — otherwise, must leave the old address space (via a call to *pmap_deactivate*) and enter the new address space (via a call to *pmap_activate*)

H2596 4-53. 141

Operations Involving Pmaps: Thread Switching

Calls to *pmap_activate* and *pmap_deactivate* must be implemented for each particular architecture. The *pmap_deactivate* call might involve saving some context and, for multiprocessors, removes this processor from the list of processors using this *pmap*. The *pmap_activate* call might involve setting a hardware register to point to a new page table and, for multiprocessors, puts this processor on the list of processors using the new *pmap*.

4–53. The Pmap Module



Operations on Pmaps: A Single Address Space

- *pmap_enter*
  - — insert a physical page at a particular virtual address
- *pmap_remove*
  - — remove a range of addresses
- *pmap_protect*
  - — set the protection attributes for a range of pages

H2596 4-54. 142 © 1991 Hewlett-Packard

Operations on Pmaps: A Single Address Space

This set of operations affects the address space of a single task.

- *pmap_enter:* called as part of the response to a page fault. A new page allocated for the task must be entered into the address map immediately, so that a reference to this page can now be completed.

- *pmap_remove:* called as part of a vm_deallocate request to ensure that address faults result if the given range of addresses is accessed.

- *pmap_protect:* called as part of a vm_protect request to set the desired protection at the hardware level.

4–54. The Pmap Module

Operations on Pmaps: Physical Pages

- *pmap_copy_on_write*
  - — remove write permission on all maps to a particular page
- *pmap_remove_all*
  - — remove a page from all maps and indicate whether the page has been modified

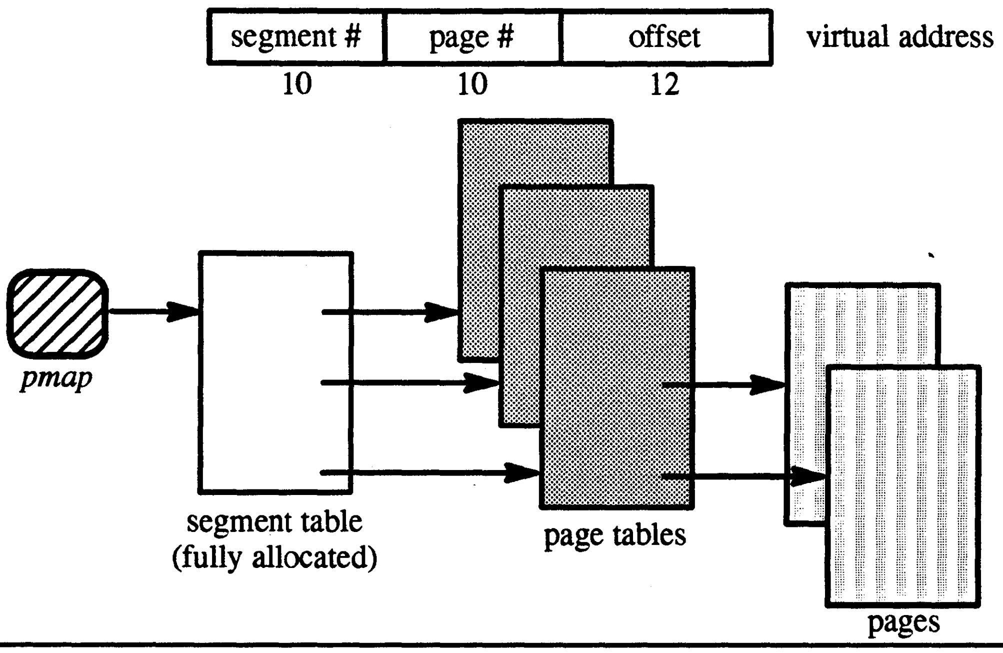
Operations on Pmaps: Physical Pages

This set of operations affects a physical page and all of the *pmap*s in which it appears. A *pmap_copy_on_write* message would be called as part of a virtual copy operation to implement copy-on-write semantics. It makes certain that write permission is not allowed for this page in all of the maps in which it appears.

A *pmap_remove_all* message might be called as part of a pageout operation. The page is to be removed from all *pmap*s but *pmap_remove_all* must check to see if the page has been modified via any of these *pmap*s. If it has, the modification is indicated by setting a bit in a global array.

4–55. The Pmap Module

Forward-Mapped Segmented-Paged Architecture



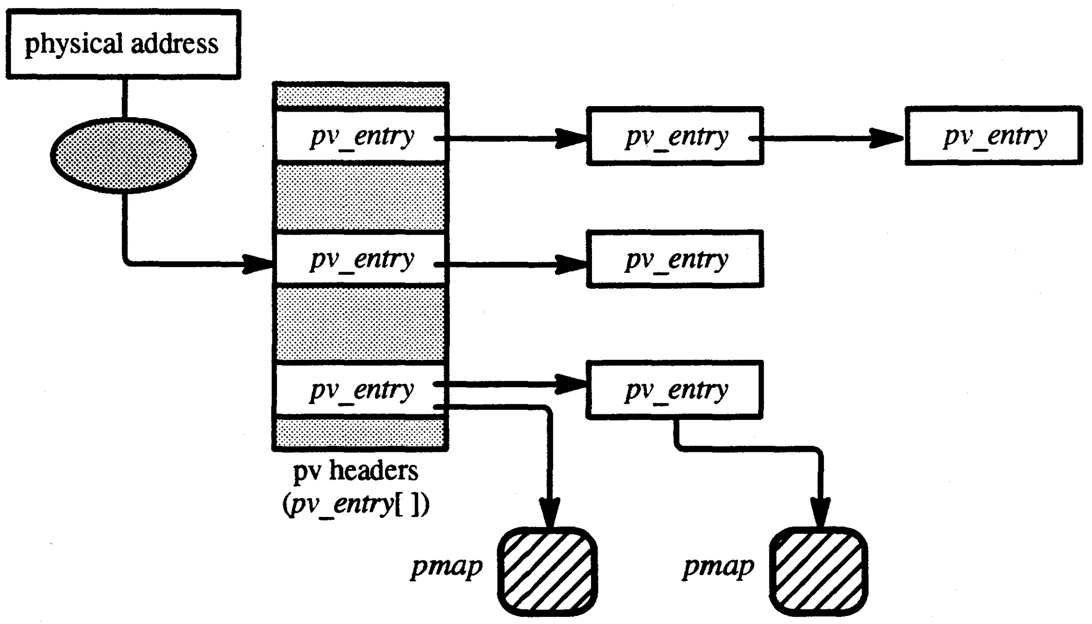
H2596 4-56. 144 © 1991 Hewlett-Packard

Forward-Mapped Segmented-Paged Architecture

As an example of a *pmap* module, we look at a forward-mapped segmented-paged architecture. Virtual addresses are divided into three parts: a 12-bit offset within a page (i.e., a page size of 4K), a 10-bit page number (i.e., a page table size of 1K entries), and a 10-bit segment number (i.e., a segment table size of 1K entries). Thus the hardware-required memory-mapping structures for an address space are headed by a segment table. Each *pmap* points to a unique segment table that is fully allocated when the *pmap* (and hence address space) is created. Page tables and pages are allocated as needed.

4–56. The Pmap Module

PV List



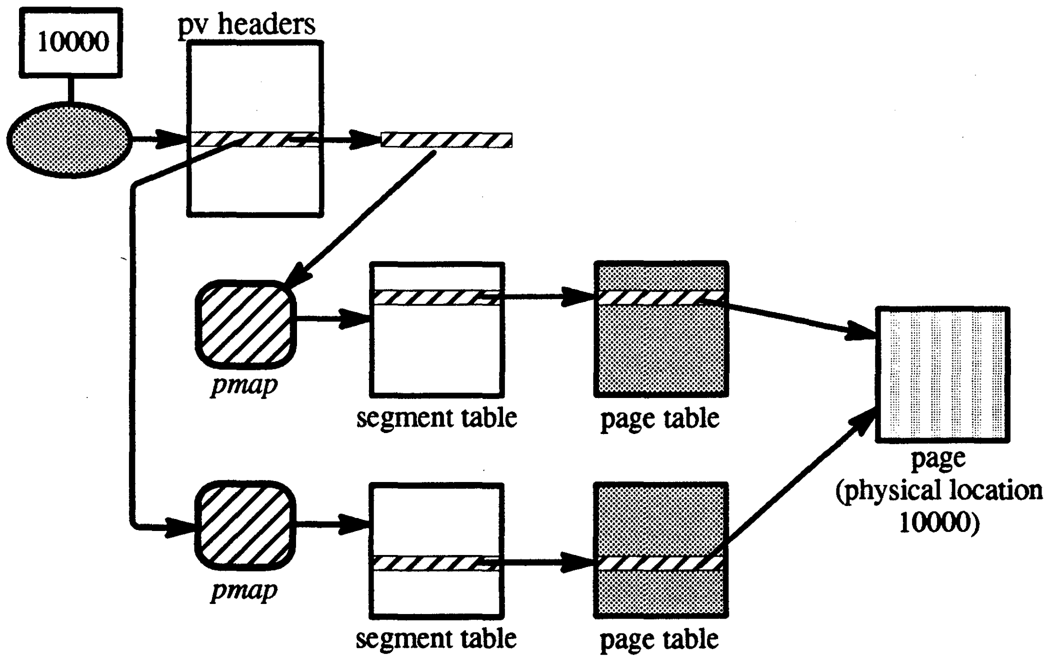
.H2596 4-57. 145 © 1991 Hewlett-Packard

PV List

Operations on physical pages need to be able to find all of the *pmaps* mapping each page. The *pv list* contains the location of each page's *pmaps*. Given a physical address, an index into the *pv_headers* array is computed. This array is an array of *pv_entrys*. Each *pv_entry* points to a *pmap* (i.e., one mapping the associated physical page), contains the virtual address of this physical page within the *pmap*-described address space, and points to the next *pv_entry* (if any) referring to another *pmap* that maps this physical page.

4–57. The Pmap Module

Pmaps: Sharing Pages



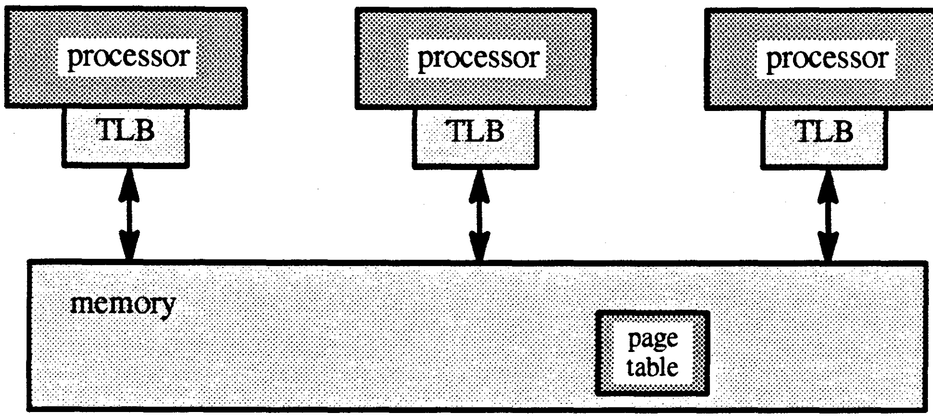
H2596 4-58. 146 © 1991 Hewlett-Packard

Pmaps: Sharing Pages

In our example architecture, page tables are not shared. Thus a page shared by two or more *pmaps* has multiple page-table entries pointing to it. If this page is being shared using copy-on-write semantics, then each of the page-table entries specifies read-only permission.

4–58. The Pmap Module

TLB Shootdown



TLB Shootdown

Most architectures employ translation-lookaside buffers (TLBs) to speed the translations from virtual address to physical address. If the architecture also uses a primary-memory resident data structure (e.g., page table) as the source of TLB entries, then the operating system must take care to keep the TLB and this mapping structure consistent. In a typical architecture, one might change a memory map by modifying the primary-memory data structure, but since the hardware accesses the TLB first, one must also arrange that the TLB be changed as well. This is usually accomplished by invalidating all or part of the TLB, thus forcing a miss when the hardware accesses this translation in the TLB and hence forcing a lookup in the mapping structure.

On a shared-memory multiprocessor, one must also be concerned about the consistency of the TLBs on other processors. This is an issue when threads of the same task are running simultaneously on different processors or when threads of tasks sharing memory are running simultaneously on different processors. The problem is that each of these threads may be modifying the memory map, and such changes must be propagated to all TLBs.

With most such multiprocessors, this is not easy: there is usually no notion of interprocessor TLB access. Thus to propagate changes to other TLBs one must use interprocessor interrupts to notify software to make these changes.

Two potential race conditions must be avoided when TLBs are modified across a multiprocessor:

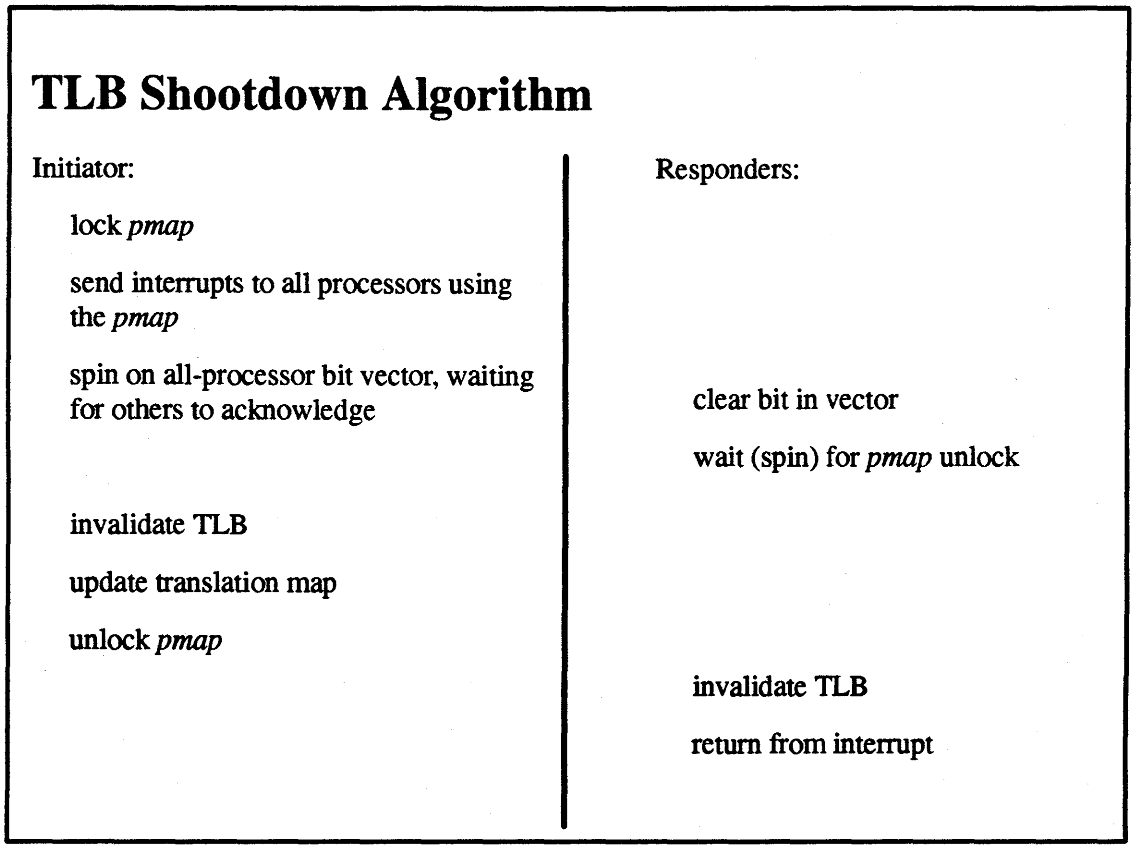
- if one invalidates a processor's TLB before changing the global page table, and if a thread continues to run on that processor, the hardware might reload the TLB from the (unmodified) page table before the page table is updated.

- if the page table is modified first, and if the unmodified affected entry is in the TLB of some other processor, a thread accessing another page might force the writeback of the unmodified TLB entry to the page table, thus undoing the modification to the page table.

These race conditions are avoided by "stalling" the other processors long enough to make the changes.

A detailed discussion of the TLB shootdown algorithm can be found in Black, 1989.

4–59. The Pmap Module



TLB Shootdown Algorithm

Initiator:	Responders:
lock *pmap*	
send interrupts to all processors using the *pmap*	
spin on all-processor bit vector, waiting for others to acknowledge	clear bit in vector
	wait (spin) for *pmap* unlock
invalidate TLB	
update translation map	
unlock *pmap*	
	invalidate TLB
	return from interrupt

.H2596 4-60. 148 © 1991 Hewlett-Packard

TLB Shootdown Algorithm

The algorithm is actually very simple: first the *pmap* is locked. This prevents any other thread from making changes to the translation map and it prevents any thread using this *pmap* from entering the running state. Attached to the *pmap* is the list of processors that are currently using the associated address space—these processors are running a thread from the *pmap*'s task. Each of these processors (the "responders") is sent an interrupt, which it acknowledges by clearing a bit; then they spin, waiting for the *pmap* to be unlocked (they don't lock the *pmap* themselves, but merely wait for it to be unlocked). Once the first processor (the initiator) determines that all responders have responded, then it can safely invalidate its own TLB and modify the translation map (referred to by the *pmap*). It then unlocks the *pmap*, notifying the responders that they can invalidate their TLBs.

Note that if multiple tasks are sharing the affected page, this procedure must be repeated for each *pmap*.

Module 4 — Virtual Memory

Exercises:

1. a. Under what circumstances is "lazy evaluation" a viable technique?

   b. Give four examples of how lazy evaluation is used in OSF/1.

2. a. List the components of the VM system.

   b. What are the three uses of a *vm_map*?

   c. Why might two *vm_map_entrys* point to the same *vm_object*?

   d. What is the purpose of the *pmap* data structure?

3. a. How is an internal memory object represented?

   b. When is it created?

   c. In whose context are pages written to a paging file?

   d. In whose context are pages fetched from the paging file?

   e. Is there any difference between the interface to the vnode pager and the interface to an external pager?

   f. What optimizations are employed to improve the performance of the vnode pager (as opposed to external memory object managers)?

   g. What is an "inactive" page?

   h. Explain what happens when a thread is swapped out and when it is swapped in.

4. a. How is lazy evaluation used in conjunction with the fork system call?

   b. When a *copy-on-write* page is modified, the copy is assigned to the topmost *vm_object*. Why is it not assigned to a lower *vm_object*?

   c. Why is it necessary to have *share maps*, e.g., why not represent read/write sharing by having multiple references to the same *vm_object*?

   d. Which virtual copy technique is used with objects set up by the mmap system call? Why?

   e. Under what circumstances does COPY_DELAY not work?

5. a. Explain what must be done at the *pmap* level in response to a virtual copy operation.

   b. Explain what must be done at the *pmap* level in response to a pageout operation.

   c. Give a detailed answer for the above two questions in terms of the architecture-dependent data structures used for the forward-mapped segmented-paged architecture discussed in the notes.

d. Suppose that we have a shared-memory multiprocessor that employs TLBs and forward-mapped segmented-paged virtual address translation. Explain what must happen in response to a vm_deallocate system call.

Advanced Questions:

6. What is the correct response to running out of backing store?

7. What difficulties would be encountered in replacing the vnode pager with an external pager?

8. Select an architecture different from the forward-mapped segmented-paged architecture discussed in the notes. Sketch the implementation of its *pmap* module.

Module 5 —File Systems

Module Contents

Module 5 — File Systems

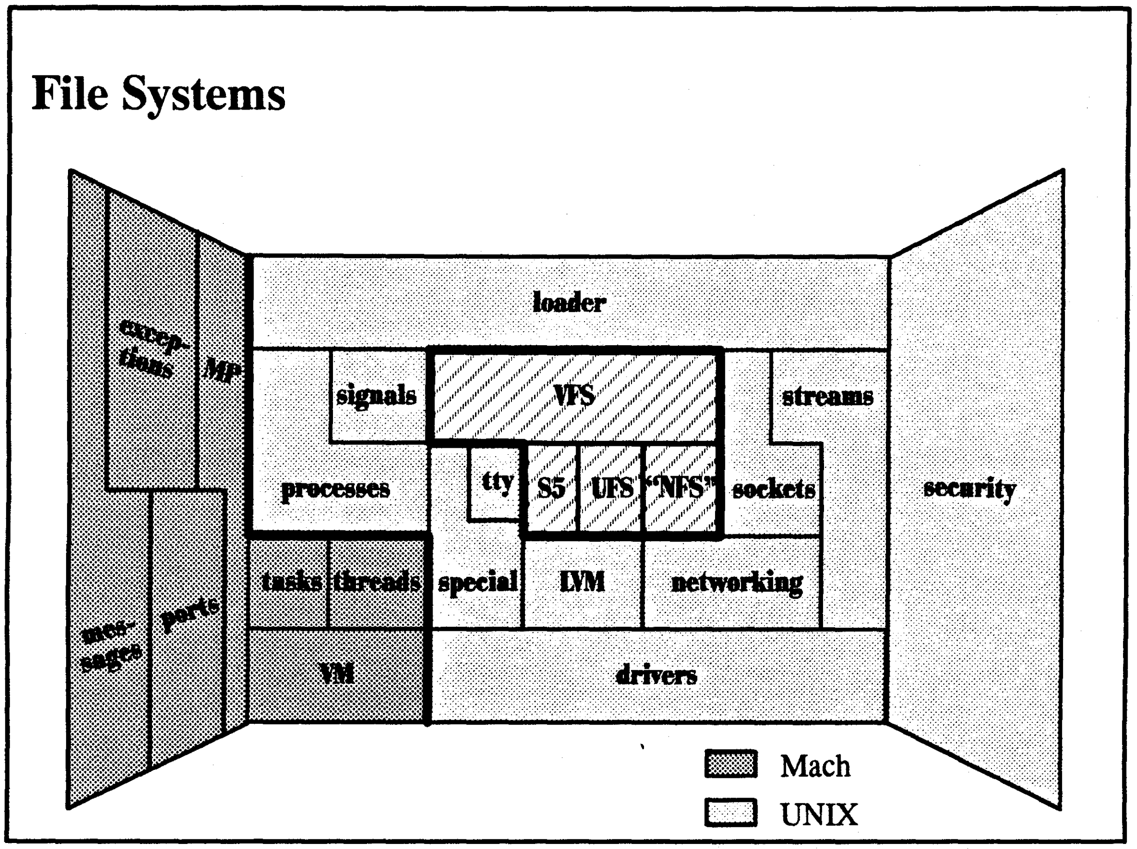
Module Objectives

In order to demonstrate an understanding of the virtual-file-system interface and of OSF/1's implementations of the S5, UFS, and NFS file systems, the student should be able to:

- explain the use of the reference count in the system file table entries
- explain the roles of the *vfsops* and *vnodeops* data structures and the abstraction of the file system concept
- describe how the buffer cache has been parallelized
- describe how directories are protected from concurrent updates
- give the size constraints on files in the S5 and UFS file systems
- explain how two threads may simultaneously extend the size of two different files within the same UFS file system
- explain why it is necessary for a NFS server to maintain a queue of recent NFS requests

Module 5 — File Systems

5–1. The Big Picture



H2596 5-1. 149 © 1991 Hewlett-Packard

File Systems

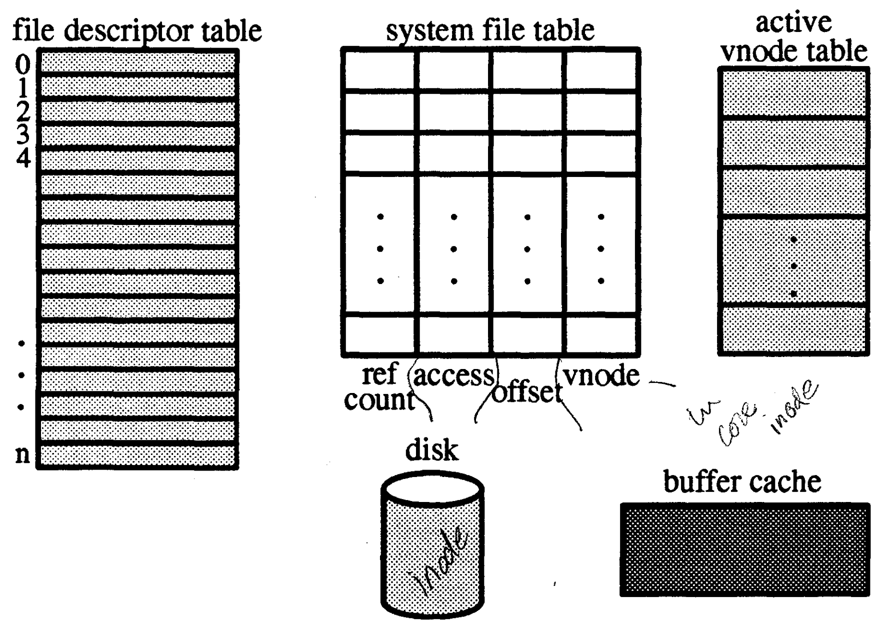
The file subsystem is part of the UNIX portion of OSF/1. The user interface to the file subsystem is that of UNIX. The implementation is primarily based on that of 4.4BSD. What has been added in OSF/1 is the *parallelization* of the file system.

The VFS implementation is from BSD but has been parallelized. The S5 implementation is from SVR3 and has not been parallelized; it is included mainly for compatibility purposes. The UFS implementation is, of course, from BSD and has been parallelized. The NFS implementation was originally done at the University of Guelph in Canada. It was modified by Berkeley and has been parallelized.

Some of the material of this module is discussed in chapter 11 of Open Software Foundation, 1990a.

5–2. Representing an Open File

Representing an Open File, part 1



.H2596 5-2. 150 © 1991 Hewlett-Packard

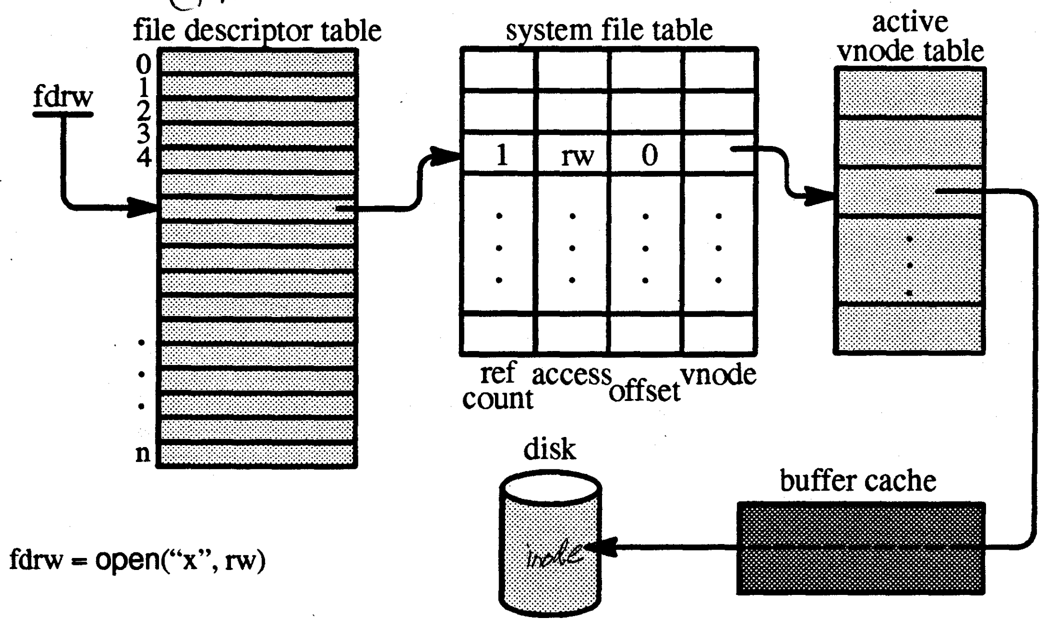
Representing an Open File, part 1

The set of open files is a property of the process as a whole. Thus, while in traditional UNIX the file descriptor table appears in the *user* structure, in OSF/1 it appears in the *u_task* structure. This structure is used to map file descriptors representing open files to system file table entries. Each system file table entry represents an open file. As discussed later, each active file (i.e., a file that is open or otherwise being used) is represented by a *vnode* that is entered in the active vnode table. Files are accessed via a kernel-supported *buffer cache* and the file itself is, of course, kept on disk.

5–3. Representing an Open File

Representing an Open File, part 2

(TASK)!



H2596 5-3. 151 © 1991 Hewlett-Packard

Representing an Open File, part 2

In this picture we illustrate what happens when a file is opened.

The lowest-numbered available file descriptor is allocated from the file descriptor table. Next, an entry in the system file table is allocated, and the file descriptor table entry is set to point to the system file table entry. A vnode for the file is allocated (or found if it already exists) and the system file table entry is set to point to it. Additional fields of the system file table entry are initialized, including:

- a reference count
- the allowed access (i.e., how the file was opened—read-only, read-write)
- the offset (i.e., the location within the file at which the next transfer will start)

In a multithreaded environment like OSF/1's, the reference count takes on particular importance. A reference count of 0, of course, means that the entry is no longer being used. Race conditions, for instance one thread closing the file while another thread within the same task accesses the file, must be guarded against. In addition, data structures such as the system file table entry must not be deallocated while they are in use.

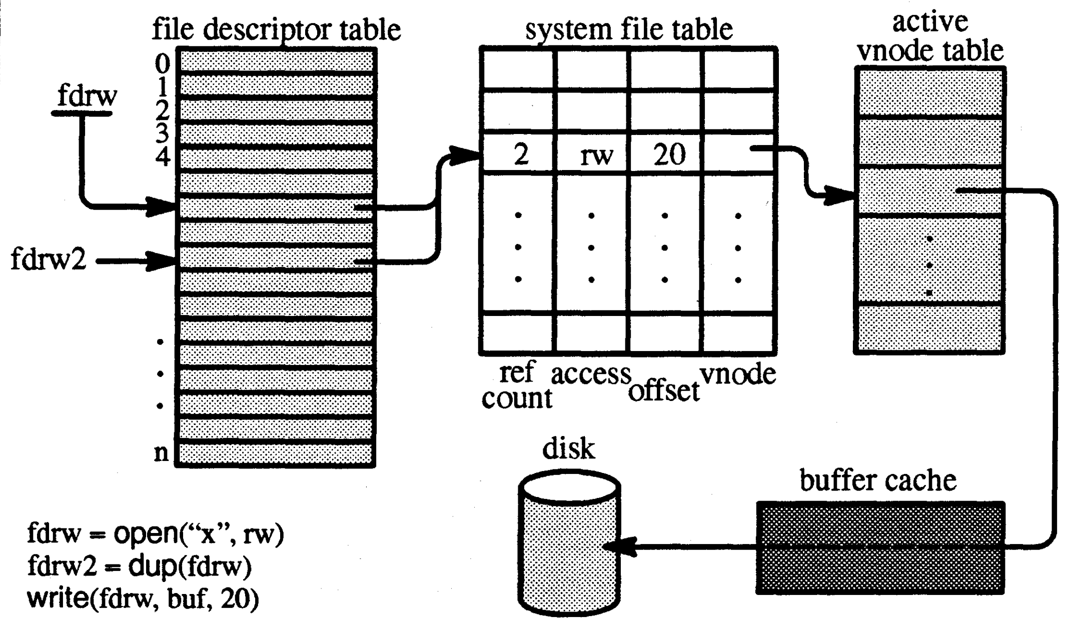
To avoid these problems, when the file is open the reference count is set to 2 (1 for the file descriptor table entry and 1 for the thread performing the open system call). When the thread returns from the call, it removes its reference, reducing the reference count to 1.

If two threads of the same task concurrently close and write the file, the reference count first goes from 1 to 2 (1 for the file descriptor table entry and 1 for the thread within the write system call; the file table's reference count is incremented by 1 at the beginning of each I/O system call, except for the close system call). If the close system call completes first, the reference count will be reduced by 1, to eliminate the file descriptor table entry's reference. But there still is a reference corresponding to the thread performing the write system call, so the file table entry remains allocated and the file remains open until this thread returns from the call. Thus the reference count enables the kernel to ensure that the file table entry and file exist as long as a thread is using them.

Another race condition that must be dealt with concerns the individual file descriptor table entries: when a file is being opened, we need to ensure that the file is not accessed by any other thread until the *open* has completed. To accomplish this, the file descriptor table entry is not made to point to the allocated file table entry until the *open* completes. However, we must make certain that this file descriptor table entry is not allocated by some other thread. Thus, when the file descriptor table entry is allocated it is marked as *reserved.* Only when the open completes is it set to point to the file table entry.

5-4. Representing an Open File

Representing an Open File, part 3



```
fdrw = open("x", rw)
fdrw2 = dup(fdrw)
write(fdrw, buf, 20)
```

H2596 5-4. 152 © 1991 Hewlett-Packard

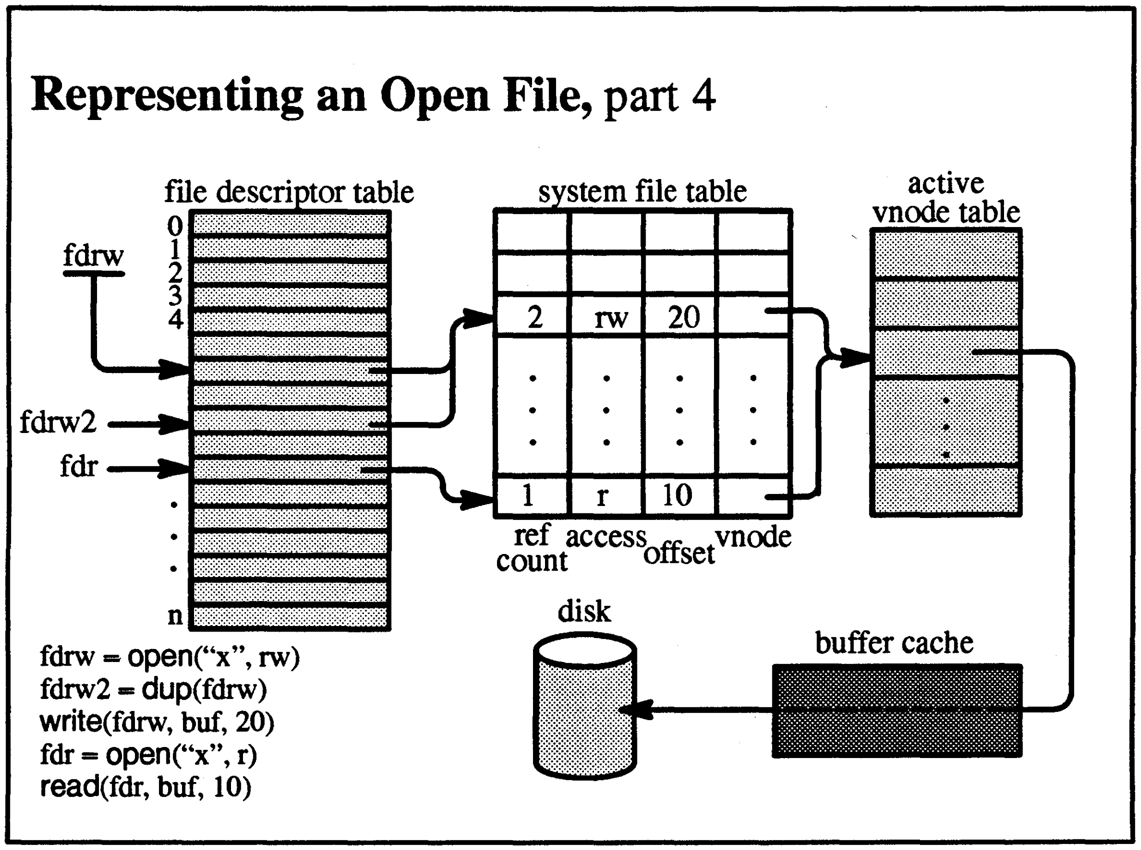
Student Notes: Representing an Open File, part 3

Dup was invented to deal with the following problem. By convention, file descriptors 1 and 2 are used for processes' normal and diagnostic output. Normally they both refer to the display, and thus diagnostic output is intermingled with normal output. Suppose, however, one wanted to redirect both file descriptors so that all output, normal and diagnostic, was sent to a file. One might open this file twice, once as file descriptor 1 and again as file descriptor 2, thereby creating two system file table entries. As file descriptor 1 receives output, the offset field of its file table entry advances with each write. After 1000 bytes have been written (sequentially), the offset field is set to 1000, representing the current end-of-file.

If at this point a diagnostic message is written to file descriptor 2, it will start at the beginning of the file, overwriting the data already there, since file descriptor 2's file table entry's offset is still at 0. This outcome is certainly not desirable.

To solve this problem, the dup system call makes file descriptors 1 and 2 both refer to the same file table entry and hence share the offset.

5–5. Representing an Open File



Representing an Open File, part 4

```
fdrw = open("x", rw)
fdrw2 = dup(fdrw)
write(fdrw, buf, 20)
fdr = open("x", r)
read(fdr, buf, 10)
```

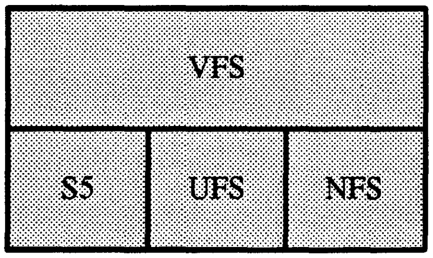
H2596 5-5. 153 © 1991 Hewlett-Packard

Representing an Open File, part 4

In this slide we see the effect of two opens of the same file within the same task.

5–6. Virtual File Systems

Generalizing the File System Concept



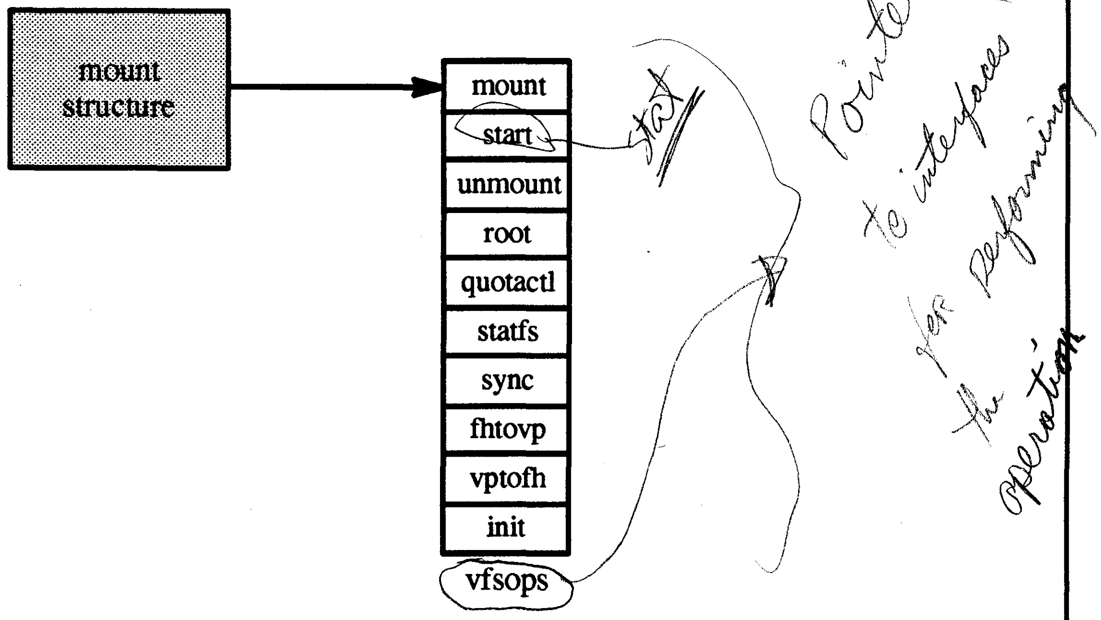
.H2596 5-6. 154 © 1991 Hewlett-Packard

Generalizing the File System Concept

In the beginning, UNIX supported only one type of file system. Modern UNIX systems now support multiple file system types. To represent different file system types, generalizations of the standard file system data structures are used. The scheme adopted in OSF/1 is based on Sun's *virtual file system* (VFS) technology (though the code has been entirely rewritten—it is adapted from 4.4BSD).

5–7. Virtual File Systems

Virtual File Systems (VFS)



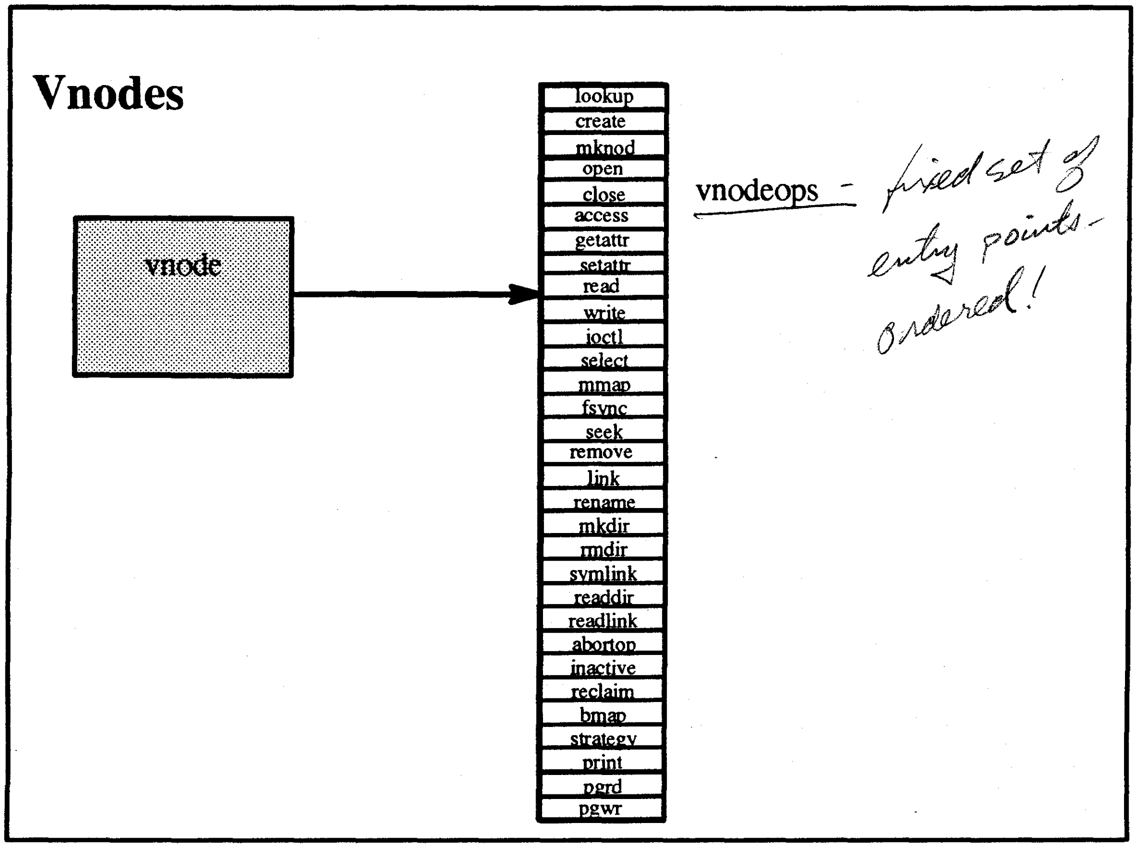
H2596 5-7. 155 © 1991 Hewlett-Packard

Virtual File Systems (VFS)

VFS is the abstraction of a file system that provides a common interface to many different file systems. OSF/1 currently supports the local UNIX file systems (S5 and UFS) and a reimplementation of Sun's NFS.

Each instance of a file system is represented by a *mount structure.* The interface to the file system is represented by an array of entry points, the *vfsops* array, that is attached to the mount structure and defines operations on the file system as a whole.

5–8. Virtual File Systems



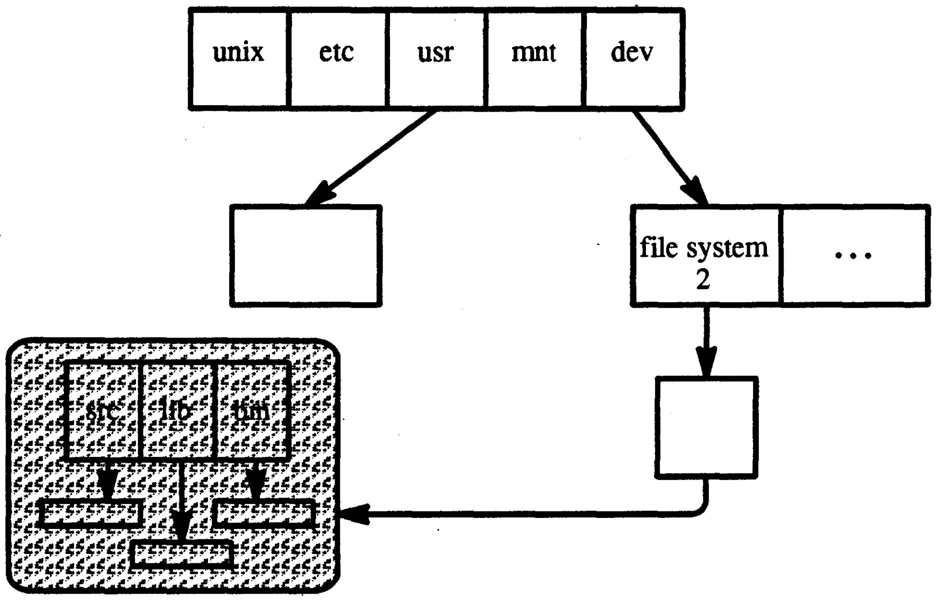
H2596 5-8. 156 © 1991 Hewlett-Packard

Vnodes

Vnodes are the abstractions of files. They represent individual files; they contain generic information about files and refer to the file-system-specific information on files (to *inodes* for UNIX files and to *nfsnodes* for NFS files). They also provide access to the various operations on the files—each vnode refers to an array of entry points called *vnodeops*.

5–9. Virtual File Systems

Mounting File Systems, part 1



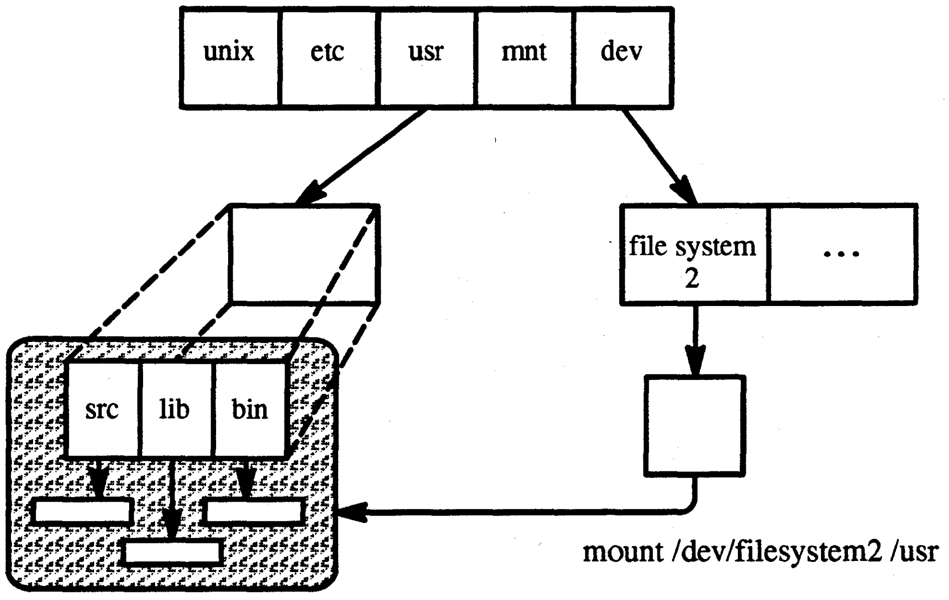
H2596 5-9. 157

Mounting File Systems, part 1

To place a file system in the tree structure directory hierarchy, one must mount it. A file system as a whole is a device that is named as a special file in the /dev directory. In order that the contents of this device be treated as files, they must be made to appear in the directory hierarchy.

5–10. Virtual File Systems

Mounting File Systems, part 2

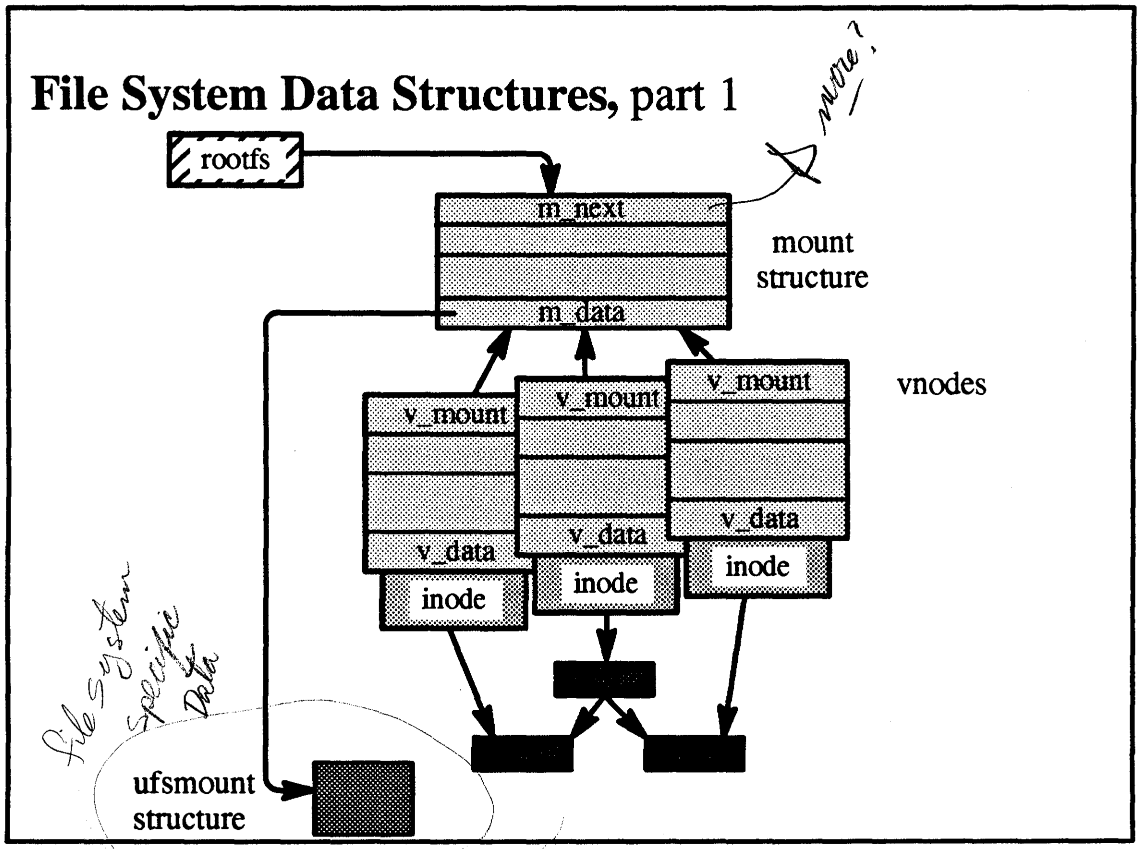


.H2596 5-10. 158 © 1991 Hewlett-Packard

Mounting File Systems, part 2

The contents of the file system are placed in the directory hierarchy when one issues the *mount* command. The *mount* command superimposes the root directory of the file system on top of the directory given in the *mount* command. Any attempt to follow a path to this directory leads one instead to the root directory of the file system. Thus the prior contents of the mounted-upon directory become invisible.

5–11. Virtual File Systems



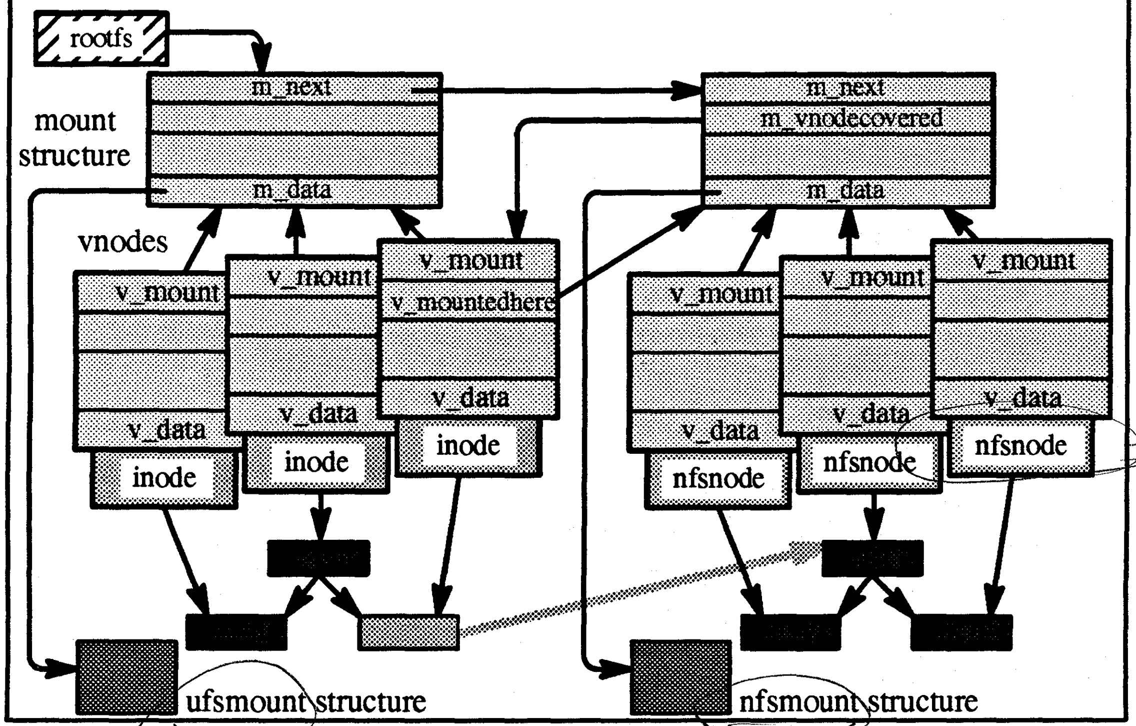
H2596 5-11. 159 © 1991 Hewlett-Packard

File System Data Structures, part 1

The data structures in this picture show a single mounted file system, the *root file system*, which happens to be a UFS file system. The field *rootfs* points to the *mount structure* of the root file system. The mount structure points to a file-system-specific mount structure, in this case the UFS mount structure. Each active file within this file system is represented by a vnode that in turn points to the mount structure. Attached to the vnode is a file-system-specific per-file data structure, in this case the *inode*. The inode represents the file within the UFS file system and is stored permanently on disk.

5–12. Virtual File Systems

File System Data Structures, part 2



Parts of NFS call this an "rnode"

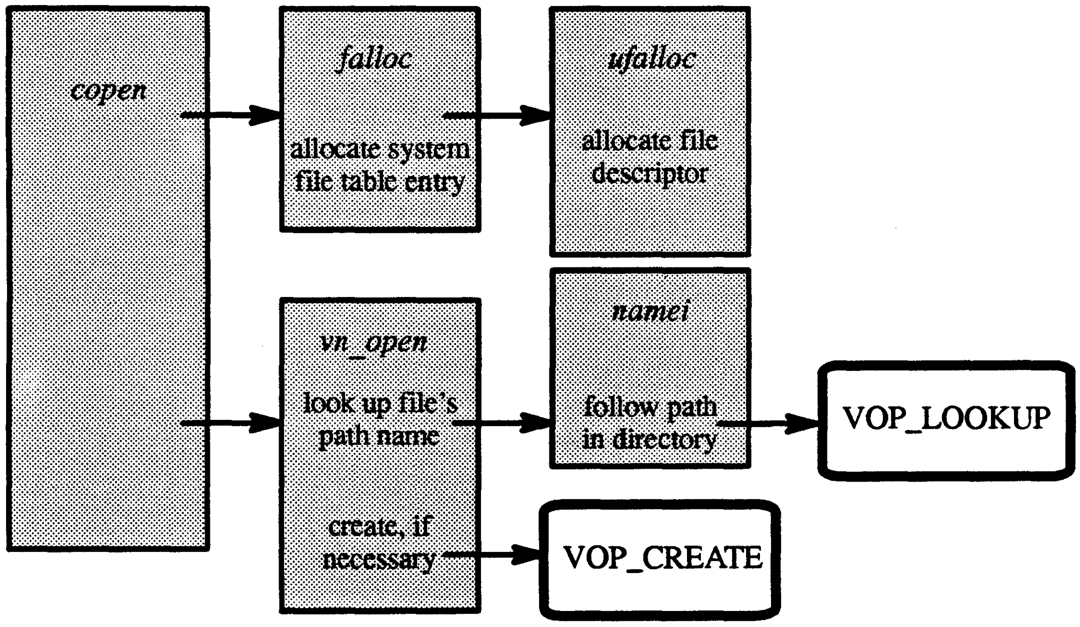
H2596 5-12. 160 © 1991 Hewlett-Packard

File System Data Structures, part 2

Here we see the effect of mounting an NFS file system in the root file system of the previous picture. The mount structure for this file system is linked to the mount structure of the root file system. The mounted file system's mount structure also points to the file-system-specific mount structure, in this case the NFS mount structure. The vnode of the mounted-upon directory is set to point to the mounted file system's mount structure to represent where the file system has been mounted. This mount structure in turn points back to the vnode. Attached to the vnodes of the active files of the mounted file system are *nfsnode* data structures, which represent the remote files.

5–13. Virtual File Systems

Open and Create: Flow of Control



.H2596 5-13. 161 © 1991 Hewlett-Packard

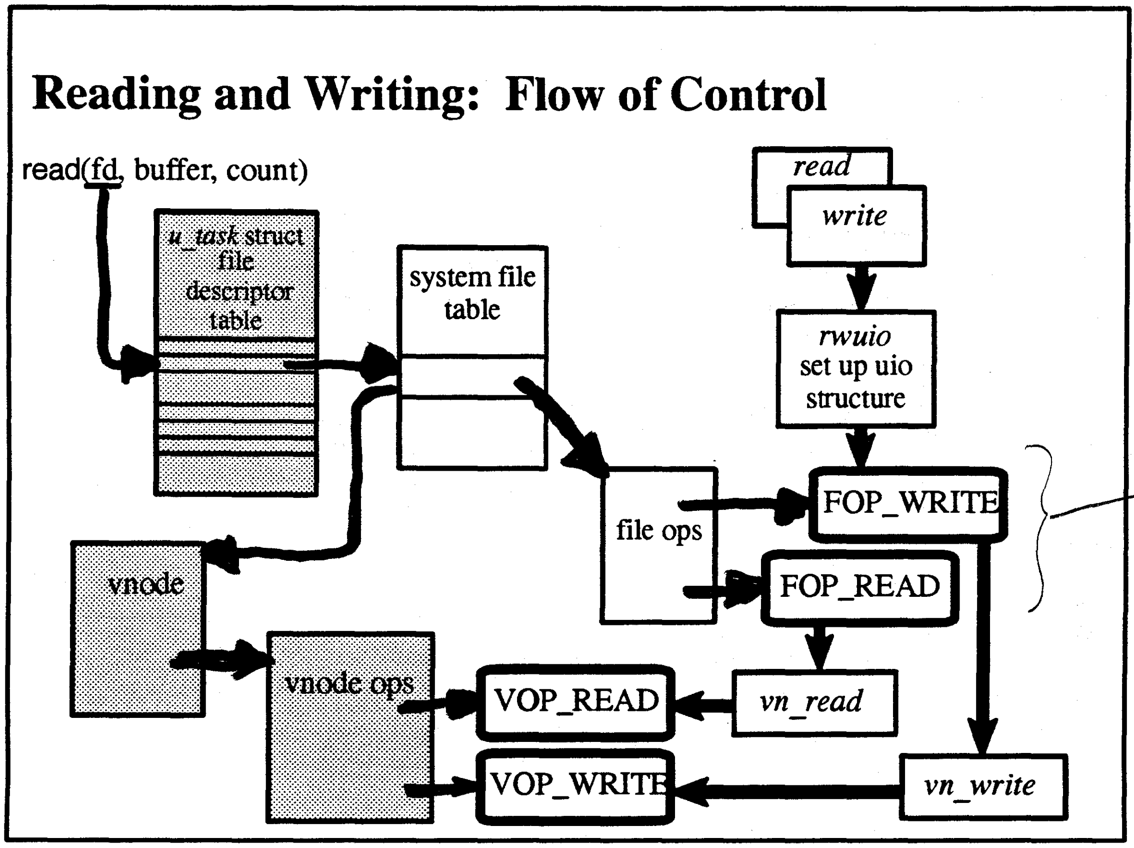
Open and Create: Flow of Control

Copen is called in the kernel in response to create and open system calls. As shown in the picture, *copen* calls *falloc*, which then calls *ufalloc*. On return from these, i.e., after the open file data structures have been set up, *copen* calls *vm_open* to initiate locating the file in the directory hierarchy. *vn_open* calls *namei*, which for each directory in the path calls the *lookup* routine associated with the directory's file system. The boxes with heavy outlines represent indirect references to a routine via a vector such as *vnodeops*. In particular, VOP_LOOKUP means to call the lookup routine listed in the *vnodeops* array attached to the vnode.

→ thin lines - Data structures

⇒ thick lines, flow of Control

5–14. Virtual File Systems



fops array either for a file or for a socket

.H2596 5-14. 162 © 1991 Hewlett-Packard

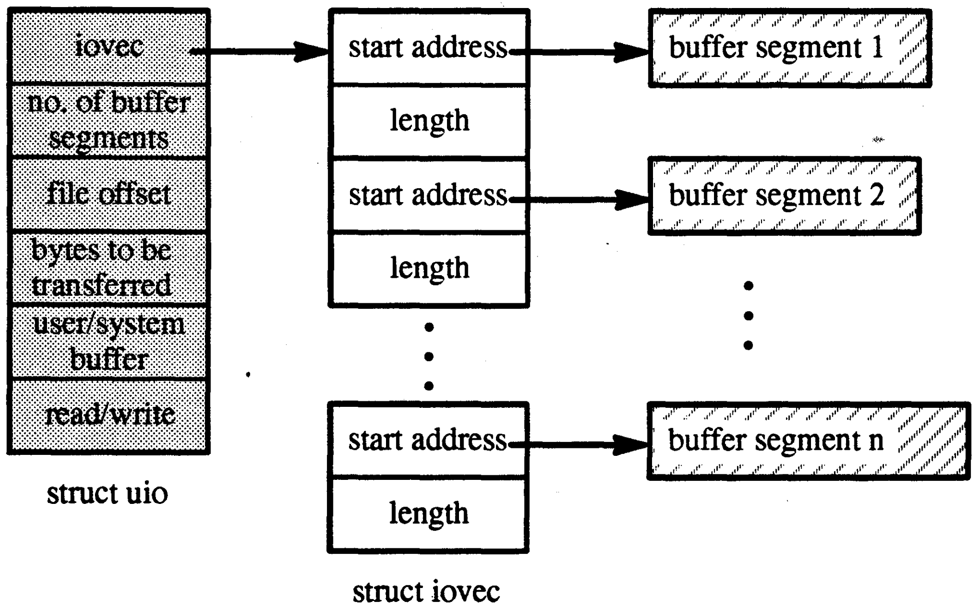
Reading and Writing: Flow of Control

Most of the work performed in reading and writing a file occurs within the file system. However, some of the work occurs at the file-system-independent level. The first step in any I/O request is to copy the parameters of the I/O request into the *uio structure* (see the next slide). The next step is to find the file table entry and the vnode and to verify that the user has permission to perform the desired operation.

An important next step for regular files and directories is to lock the file offset in the file table entry (using a blocking read-write lock) so as to make the operation *atomic*. This is done to avoid a race condition in which two threads that share the same file table entry concurrently access the file. (Locking has been done incorrectly in some versions of UNIX.)

5–15. Virtual File Systems

The Uio Structure



H2596 5-15. 163 © 1991 Hewlett-Packard

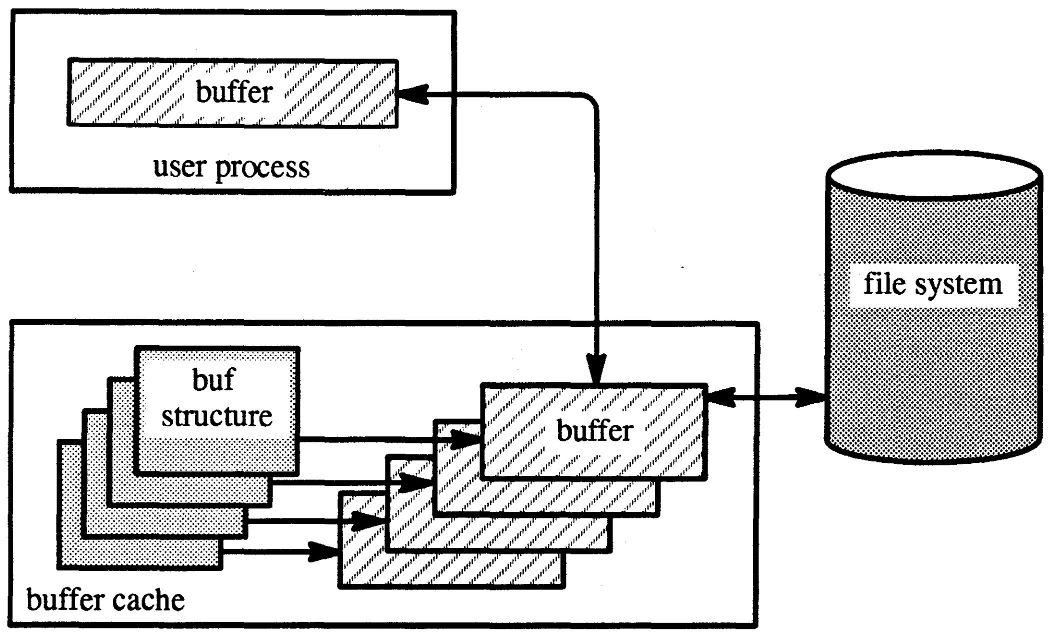
The Uio Structure

The *uio* structure represents a logical I/O request. Its contents represent what needs to be done to complete an I/O request; these contents are updated as the I/O request progresses through the system. The buffer may, in general, be composed of multiple segments, and hence an array of *iovec* structures is needed to refer to each of the pieces of the buffer. This organization is made necessary by the readv and writev system calls, which use such multicomponent buffers.

mmap → memory Map.

5–16. The Buffer Cache

The Buffer Cache



.H2596 5-16. 164 © 1991 Hewlett-Packard

The Buffer Cache

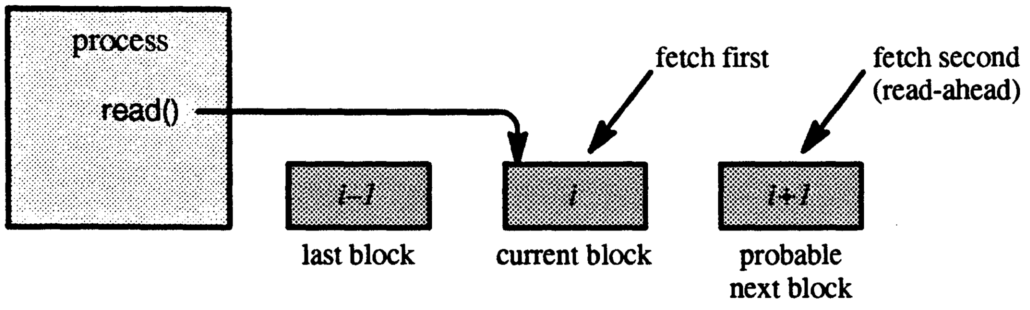
The buffer cache has two primary functions. The first, and most important, is to make possible concurrent I/O and computation within a UNIX process. The second is to insulate the user from physical block boundaries.

From a user thread's point of view, I/O is *synchronous*. By this we mean that when the I/O system call returns, the system no longer needs the user-supplied buffer. For example, after a write system call, the data in the user buffer has either been transmitted to the device or copied to a kernel buffer—the user can now scribble over the buffer without affecting the data transfer. Because of this synchronization, from a user thread's point of view, no more than one I/O operation can be in progress at a time. Thus user-implemented multibuffered I/O is not possible (in a single-threaded process). In OSF/1, however, the user can utilize multiple threads within a task to program concurrent I/O and computation.

The buffer cache provides a kernel implementation of multibuffering I/O, and thus concurrent I/O and computation are possible even for single-threaded processes.

5–17. The Buffer Cache

Multi-Buffered I/O

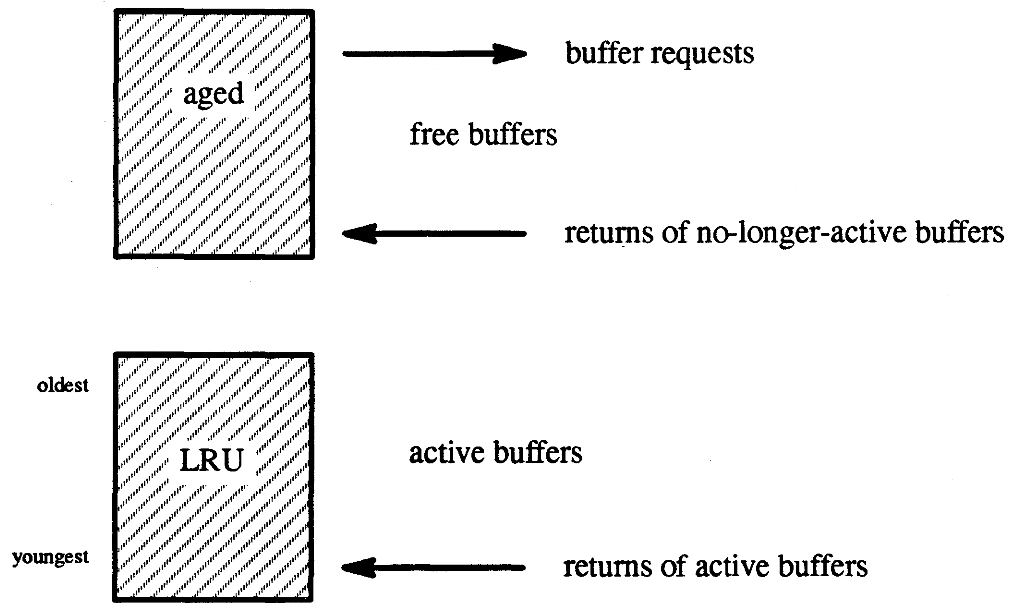


Multi-Buffered I/O

The use of *read-aheads* and *write-behinds* makes concurrent I/O and computation possible: if the block currently being fetched is block *i* and the previous block fetched was block *i–1*, then block *i+1* is also fetched. Modified blocks are normally not written out synchronously but are instead written out sometime after they were modified, asynchronously.

5–18. The Buffer Cache

Maintaining the Cache



H2596 5-18. 166

Maintaining the Cache

Active buffers are maintained in least-recently-used (LRU) order in the system-wide LRU list. Thus after a buffer has been used (as part of a read or write system call), it is returned to the end of the LRU list. The system also maintains a separate list of "free" buffers called the *aged* list. Included in this list are buffers holding no-longer-needed blocks, such as blocks from truncated files.

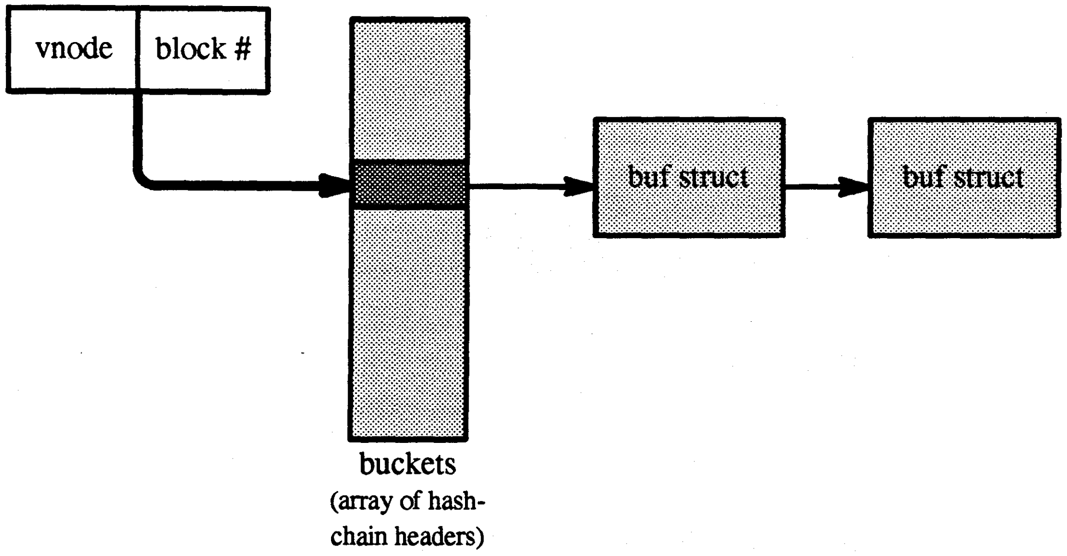
Fresh buffers are taken from the aged list. If this list is empty, then a buffer is obtained from the LRU list, as follows. If the first buffer (least recently used) in this list is clean (i.e., contains a block that is identical to its copy on disk), then this buffer is taken. Otherwise (i.e., if the buffer is dirty), it is written out to disk asynchronously and, when written, is placed at the end of the aged list. The search for a fresh buffer continues on to the next buffer in the LRU list, etc.

When a file is deleted, any buffers containing its blocks are placed at the head of the aged list. Also, when I/O into a buffer results in an I/O error, the buffer is placed at the head of the aged list.

In BSD, buffers that have been read (or written) in their entirety are placed at the end of the aged list. The assumption is that, since files are normally accessed sequentially, these buffers won't be needed for a while. This technique has not been found to improve performance and thus is not used in OSF/1.

5–19. The Buffer Cache

Accessing the Cache



.H2596 5-19. 167 © 1991 Hewlett-Packard

Accessing the Cache

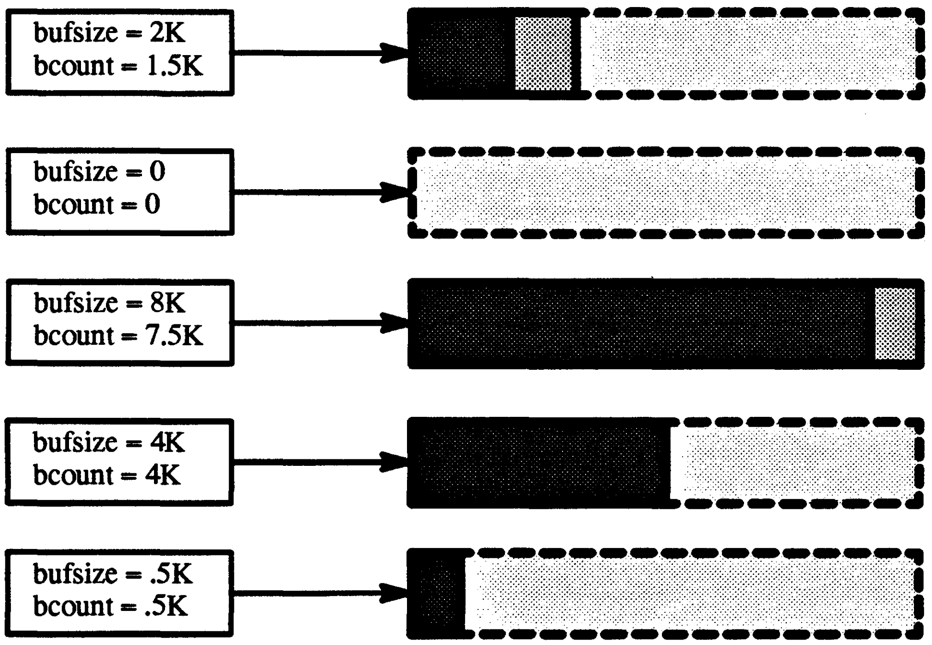
Buffers in the cache are accessed via a hash table. In older versions of UNIX, buffers in the cache were identified by file-system number and block number (within the file system). With remote file systems such as NFS, the client does not know the block number within the file system, but only knows the block number relative to the beginning of the file. OSF/1 thus uses the address of the vnode and the block number relative to the beginning of the file to identify blocks of files of not only remote but also local file systems.

This approach does not work for the *indirect blocks* and other *metadata structures* of UNIX file systems (both S5 and UFS). These are identified by the address of the vnode of the underlying file system (i.e. block special file) and the block number relative to the beginning of the file system. (A possible consistency problem that would arise when blocks of open files in a mounted file system are accessed via the block special interface is prevented by not allowing the block special interface to a mounted file system to be accessed.)

In order to improve the performance of operations such as fsync that affect the cached blocks of a particular file, each vnode heads a list of incore clean buffers and incore dirty buffers.

5–20. The Buffer Cache

Virtual Buffers



Virtual Buffers

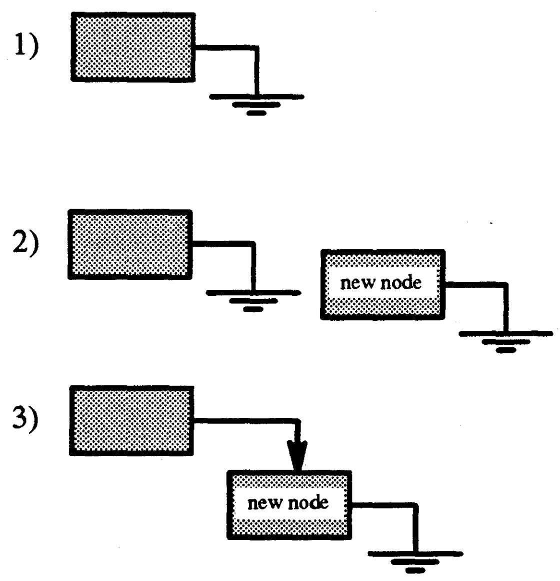
If buffers were all of the same size and files were allocated in fixed-size blocks, then allocating a buffer would be trivial. However, the UFS file system allows different file systems to have different block sizes and, within a file system, it allows the last block of a file to be smaller than the others.

Each *buf* structure is assigned a maximum-block-size amount of virtual memory (MAXBSIZE = 8K) for its buffer.

The total amount of real memory allocated for buffers is divided up among the *buf* structures; a possible result is that not all buffers will have the maximum 8K of real memory backing them up. If such an underendowed buffer is allocated when a full allotment of real memory is needed, space is "stolen" from another *buf* structure's buffer (by remapping the memory). *Buf* structures without real memory for their buffers are placed on an *empty list.* If a *buf* structure is allocated whose buffer is larger than is needed, its extra space is given to a *buf* structure on the empty list.

5–21. The Buffer Cache

File System Consistency, part 1

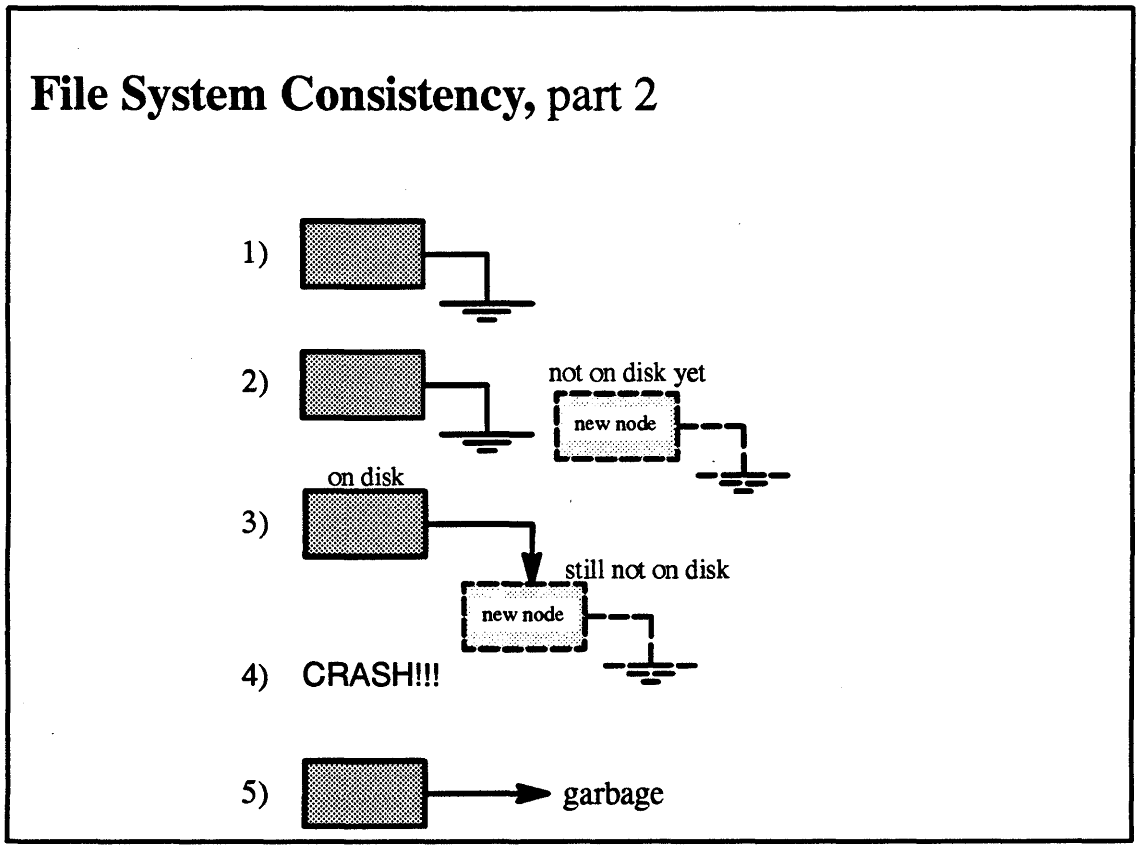


.H2596 5-21. 169 © 1991 Hewlett-Packard

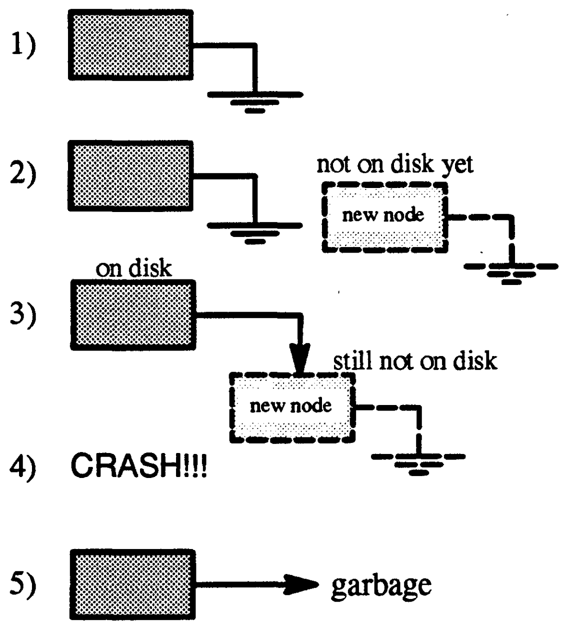
File System Consistency, part 1

In the event of a crash, the contents of the file system may well be inconsistent with any view of it the user might have. For example, a programmer may have carefully added a node to the end of the list, so that at all times the list structure is well-formed.

5–22. The Buffer Cache



File System Consistency, part 2



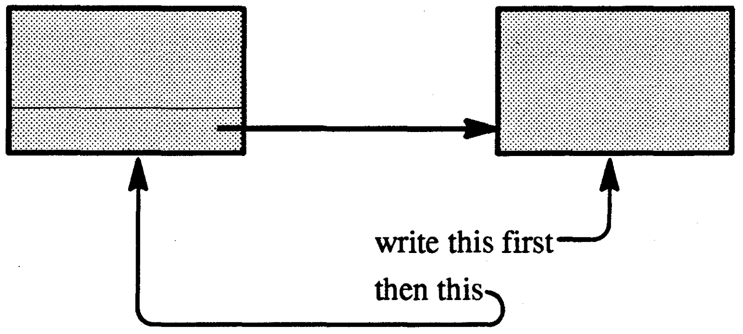
H2596 5-22. 170 © 1991 Hewlett-Packard

File System Consistency, part 2

But, if the new node and the old node are stored on separate disk blocks, the modifications to the block containing the old node might be written out first; the system might well crash before the second block is written out.

5–23. The Buffer Cache

Keeping It Consistent

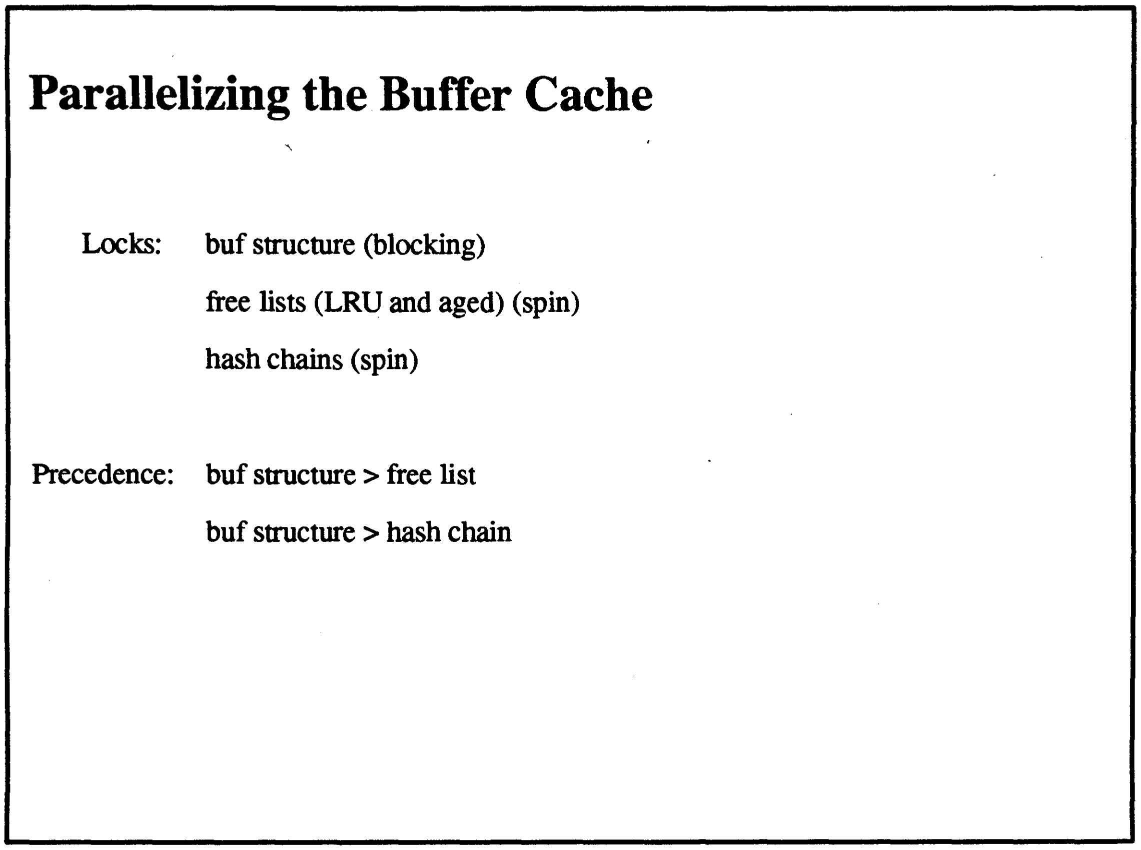


Keeping It Consistent

To deal with this problem, system data structures are written out synchronously and in the correct order (i.e., the block containing the target of a pointer is updated before that containing the pointer). This is done for directory entries, inodes, indirect blocks, etc.

No such synchronization is done for user data structures: not enough is known about the semantics of user operations to make this possible. However, a user process called *update* executes a sync system call every 30 seconds, which initiates the writing out to disk of all dirty buffers. Alternatively, the user can open a file with the *synchronous* option so that all writes are waited for; i.e, the buffer cache acts as a *write-through* cache (N.B. that this is expensive!).

5–24. The Buffer Cache



Parallelizing the Buffer Cache

Locks: buf structure (blocking)
free lists (LRU and aged) (spin)
hash chains (spin)

Precedence: buf structure > free list
buf structure > hash chain

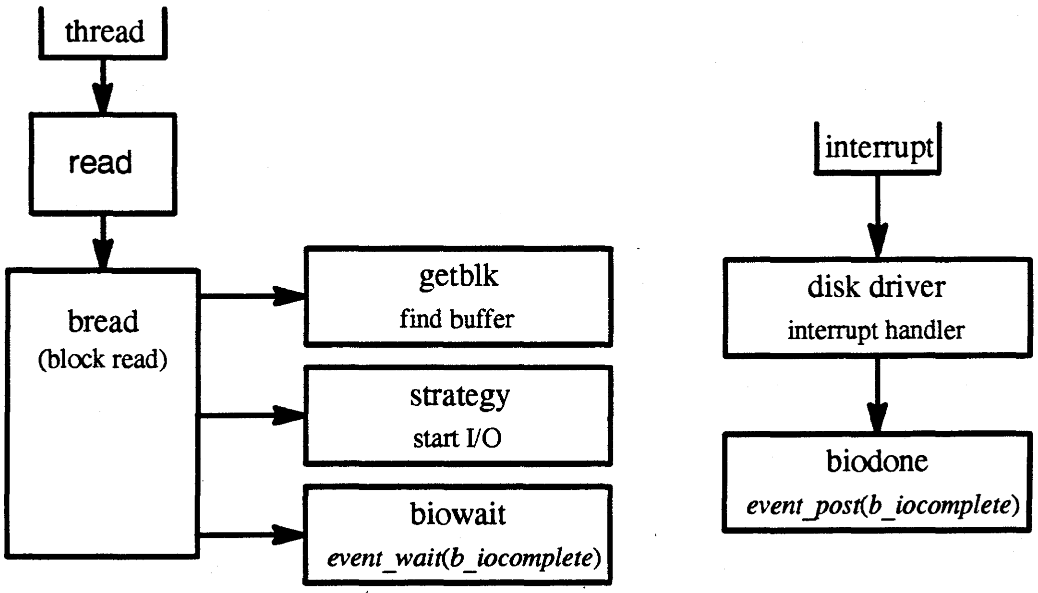
H2596 5-24. 172 © 1991 Hewlett-Packard

Parallelizing the Buffer Cache

The buffer cache is parallelized by using blocking locks on the buffers. Thus many operations may proceed simultaneously, as long as they involve different buffers. To avoid race conditions when updating the free lists and hash table, spin locks are employed. A partial precedence order on these locks is used, as shown on the slide.

5–25. The Buffer Cache

Block I/O Read

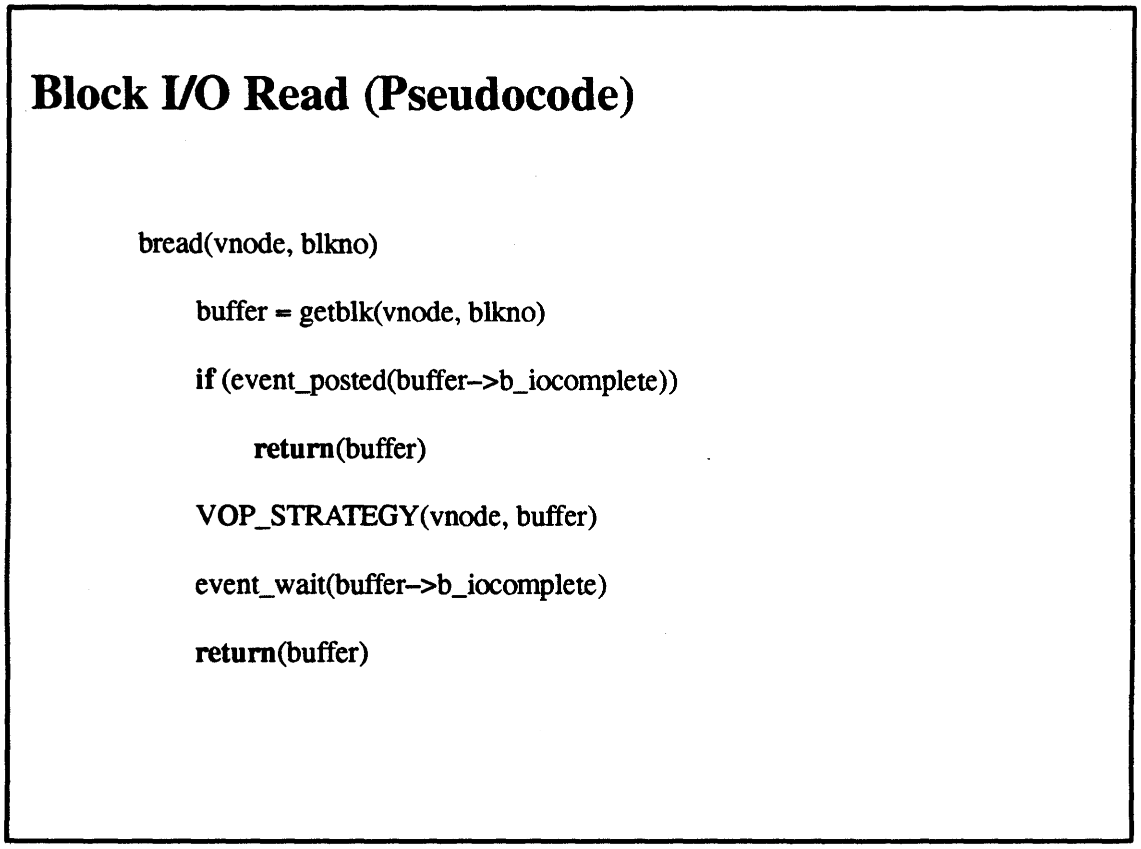


.H2596 5-25. 173 © 1991 Hewlett-Packard

Block I/O Read

Note that the use of events avoids the race condition between the *biodone* and the *biowait*: the interrupt could be handled on a different processor from the one on which the thread calling *biowait* is running.

5–26. The Buffer Cache



Block I/O Read (Pseudocode)

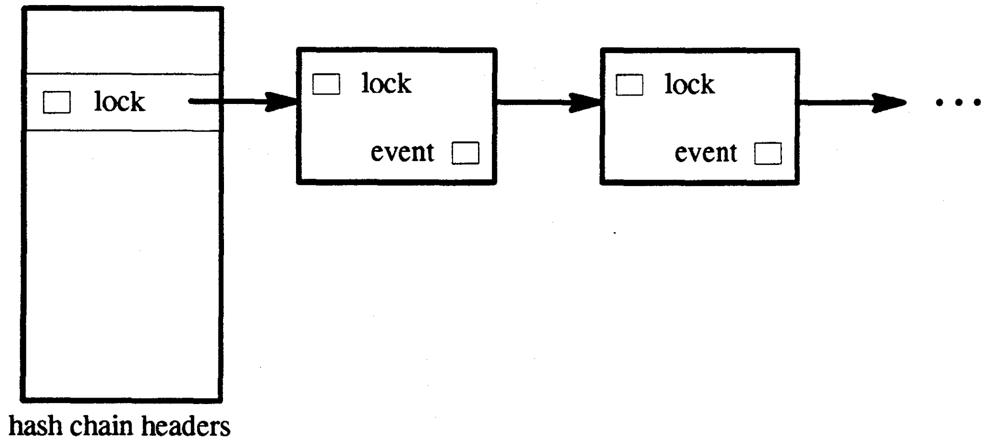
```
bread(vnode, blkno)
    buffer = getblk(vnode, blkno)
    if (event_posted(buffer->b_iocomplete))
        return(buffer)
    VOP_STRATEGY(vnode, buffer)
    event_wait(buffer->b_iocomplete)
    return(buffer)
```

.H2596 5-26. 174 © 1991 Hewlett-Packard

Block I/O Read (Pseudocode)

5–27. The Buffer Cache

Finding a Block in the Cache

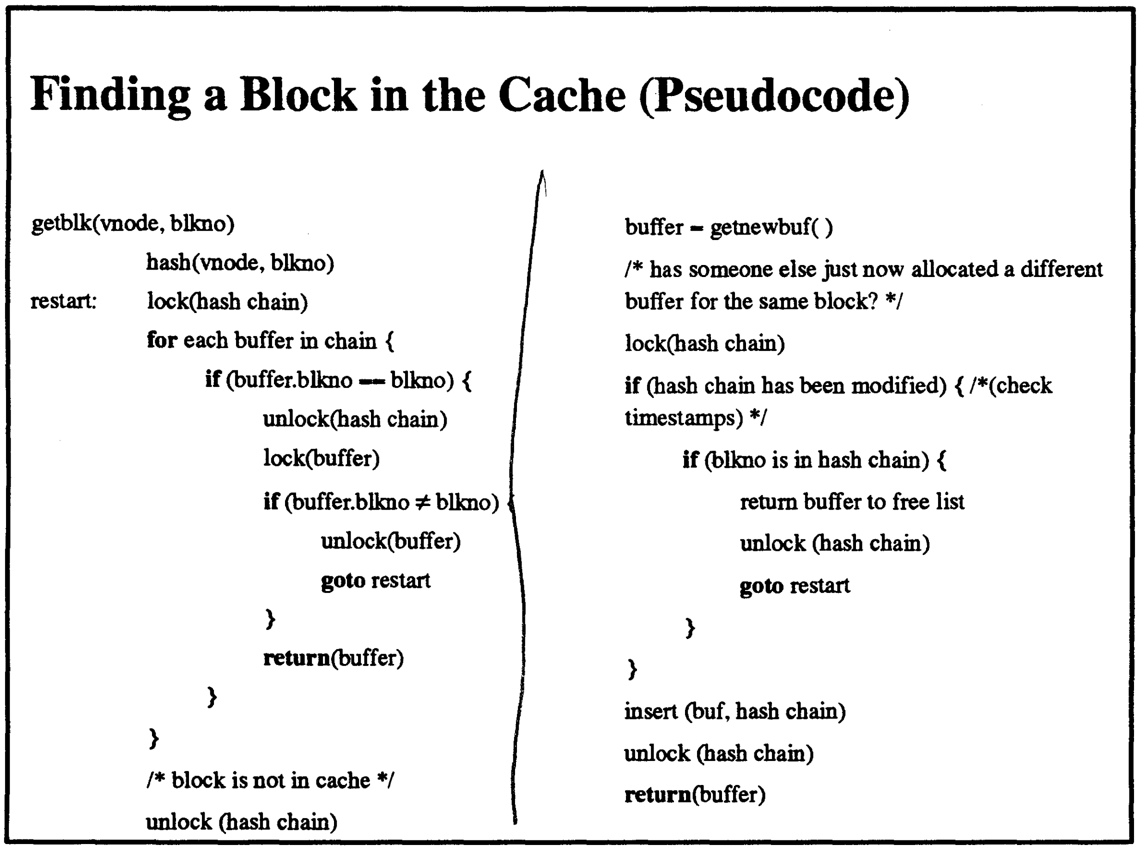


.H2596 5-27. 175 © 1991 Hewlett-Packard

Finding a Block in the Cache

First the thread takes the lock on the hash chain header (simple lock). If it finds the desired buffer, then it unlocks the header and takes the lock on the buffer (blocking lock). If, after the thread waits for the lock, it finds that the buffer no longer contains the desired block, then the thread repeats the procedure from the beginning.

5–28. The Buffer Cache



Finding a Block in the Cache (Pseudocode)

```
getblk(vnode, blkno)
            hash(vnode, blkno)
restart:    lock(hash chain)
            for each buffer in chain {
                  if (buffer.blkno == blkno) {
                        unlock(hash chain)
                        lock(buffer)
                        if (buffer.blkno ≠ blkno)
                              unlock(buffer)
                              goto restart
                        }
                        return(buffer)
                  }
            }
            /* block is not in cache */
            unlock (hash chain)
            buffer = getnewbuf( )
            /* has someone else just now allocated a different
            buffer for the same block? */
            lock(hash chain)
            if (hash chain has been modified) { /*(check
            timestamps) */
                  if (blkno is in hash chain) {
                        return buffer to free list
                        unlock (hash chain)
                        goto restart
                  }
            }
            insert (buf, hash chain)
            unlock (hash chain)
            return(buffer)
```

.H2596 5-28. 176 © 1991 Hewlett-Packard

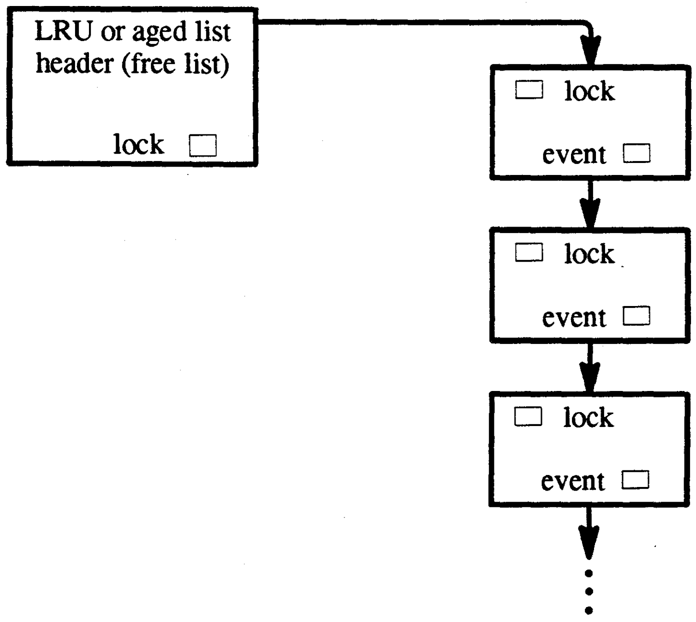
Finding a Block in the Cache (Pseudocode)

Associated with each hash chain is a timestamp that is incremented by one when the hash chain is modified (a buffer is either inserted or removed).

There is a potential race condition when *getnewbuf* is called: two threads may simultaneously discover that a particular block is not in the cache, and both call *getnewbuf* to allocate a buffer for it (and two buffers are indeed allocated). Due to the lock on the hash chain, one buffer will be inserted in the hash table first. To prevent both buffers (representing the same block) from being inserted, a check has to be made to insure that the buffer being inserted is not a duplicate. This check would involve searching the hash chain (again). To minimize the number of times this must be done, the current value of the timestamp on the hash chain is compared with its value when it was originally ascertained that the block was not present. Only if the timestamps are now different is the hash chain searched.

5–29. The Buffer Cache

Getting a New Buffer

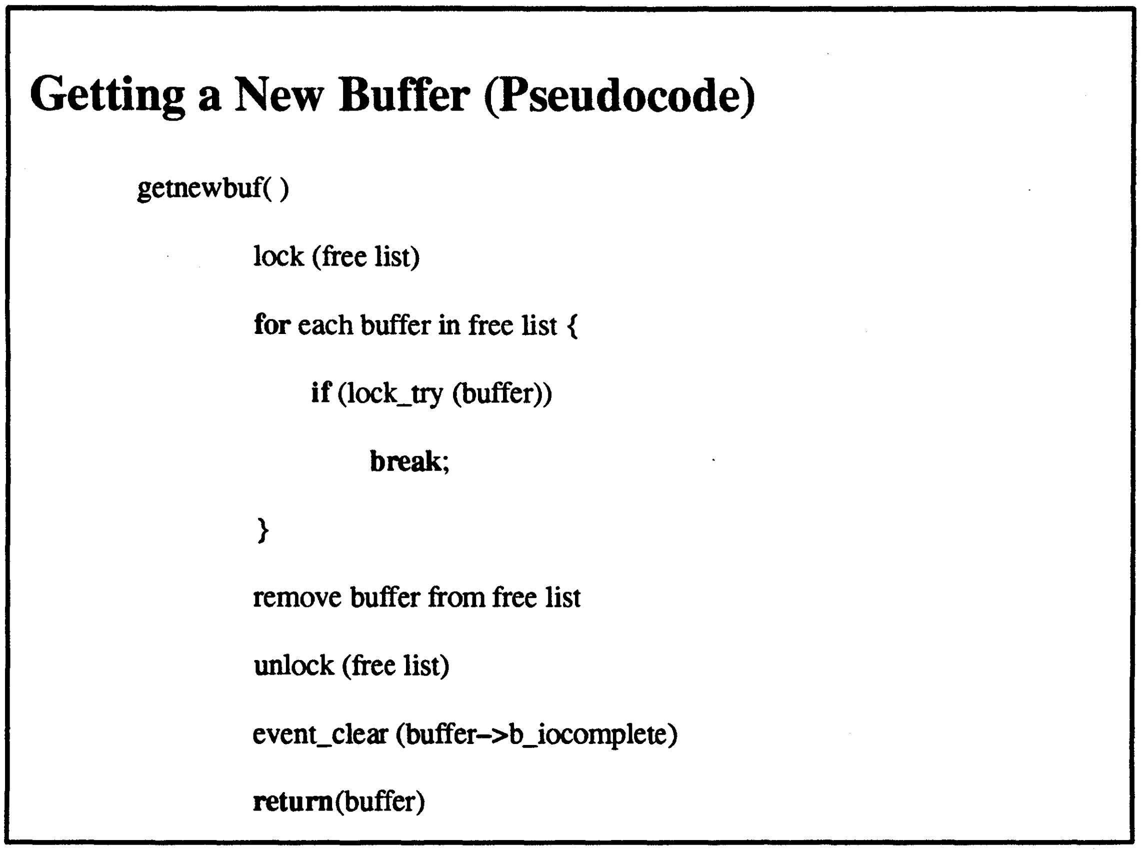


H2596 5-29. 177 © 1991 Hewlett-Packard

Getting a New Buffer

First the thread takes the lock on the header (simple lock). Then it conditionally takes the lock on the buffer. If the buffer is already locked, then the thread skips it and tries the next one. If no buffers are available, then the thread sleeps until one is.

5–30. The Buffer Cache



Getting a New Buffer (Pseudocode)

```
getnewbuf( )
        lock (free list)
        for each buffer in free list {
            if (lock_try (buffer))
                break;
        }
        remove buffer from free list
        unlock (free list)
        event_clear (buffer->b_iocomplete)
        return(buffer)
```

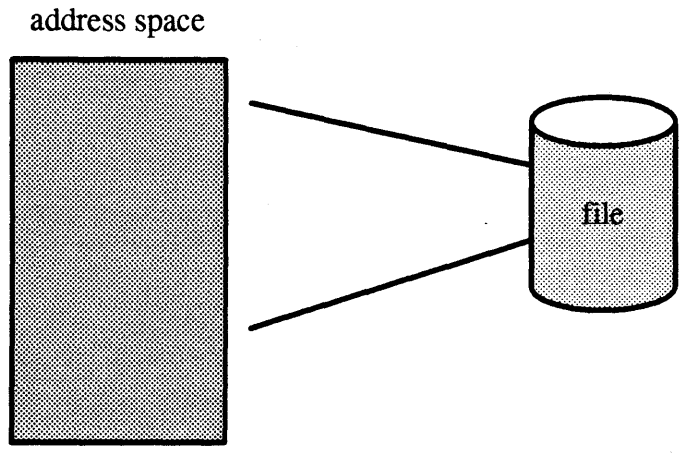
.H2596 5-30. 178 © 1991 Hewlett-Packard

Getting a New Buffer (Pseudocode)

SYS5.4 has different semantics to the file.

5–31. The Buffer Cache

Mmap



.H2596 5-31. 179 © 1991 Hewlett-Packard

Mmap

The mmap system call is used either to map a file into a process's address space or to create an anonymous memory region. Anonymous memory is shared with all of the process's descendants.

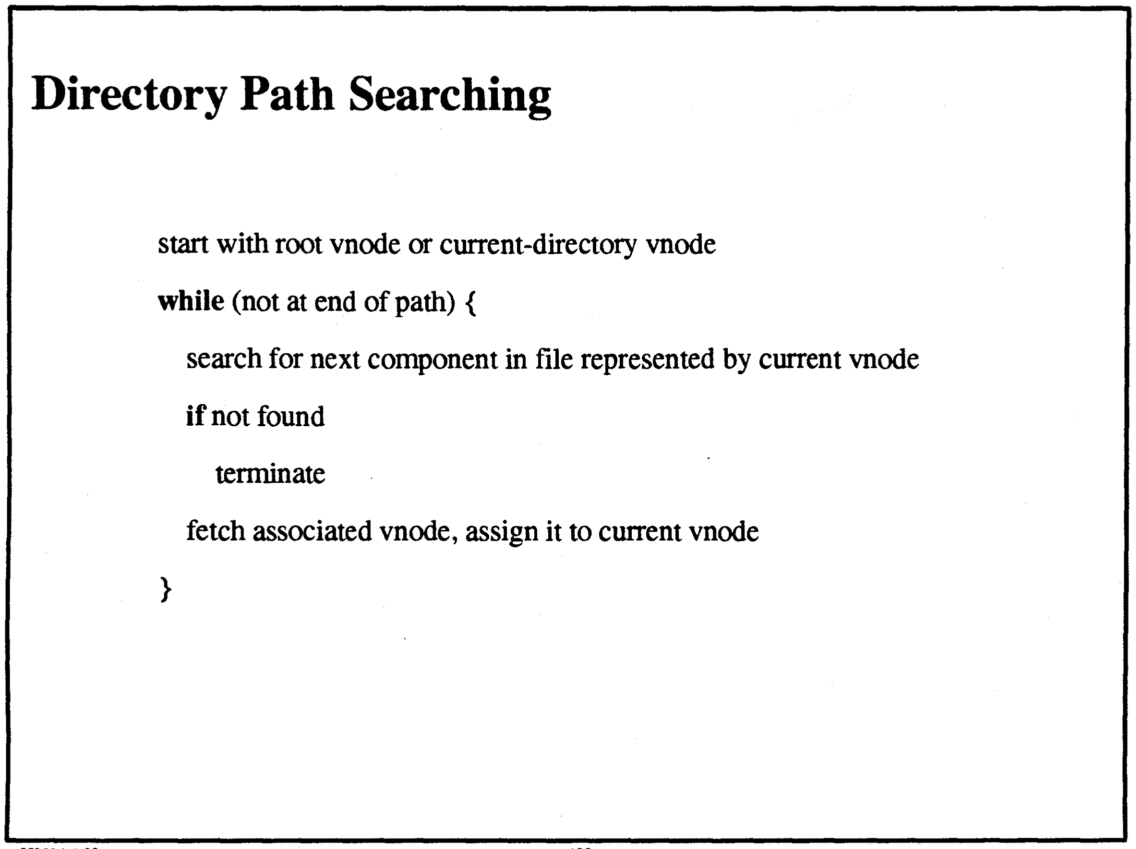
A mapped file may be *private*, meaning that changes to the mapped memory are not shared with other processes and are not reflected back to the file.

A mapped file may be *shared*, meaning that changes to the mapped memory are shared with other processes that have a shared mapping of the file, and these changes are reflected back to the file.

Two important issues arise with mmap. First, does a process that has a private mapping of a file "see" the changes made by processes with shared mappings? In OSF/1, the answer is no.

The other issue involves the simultaneous access of a file via mmap and read/write system calls. In the current implementation there is a consistency problem, since two copies of blocks of the file may exist in primary memory: one in the buffer cache and one in a page frame to which a virtual page has been mapped.

5–32. Directory Path Searching



Directory Path Searching

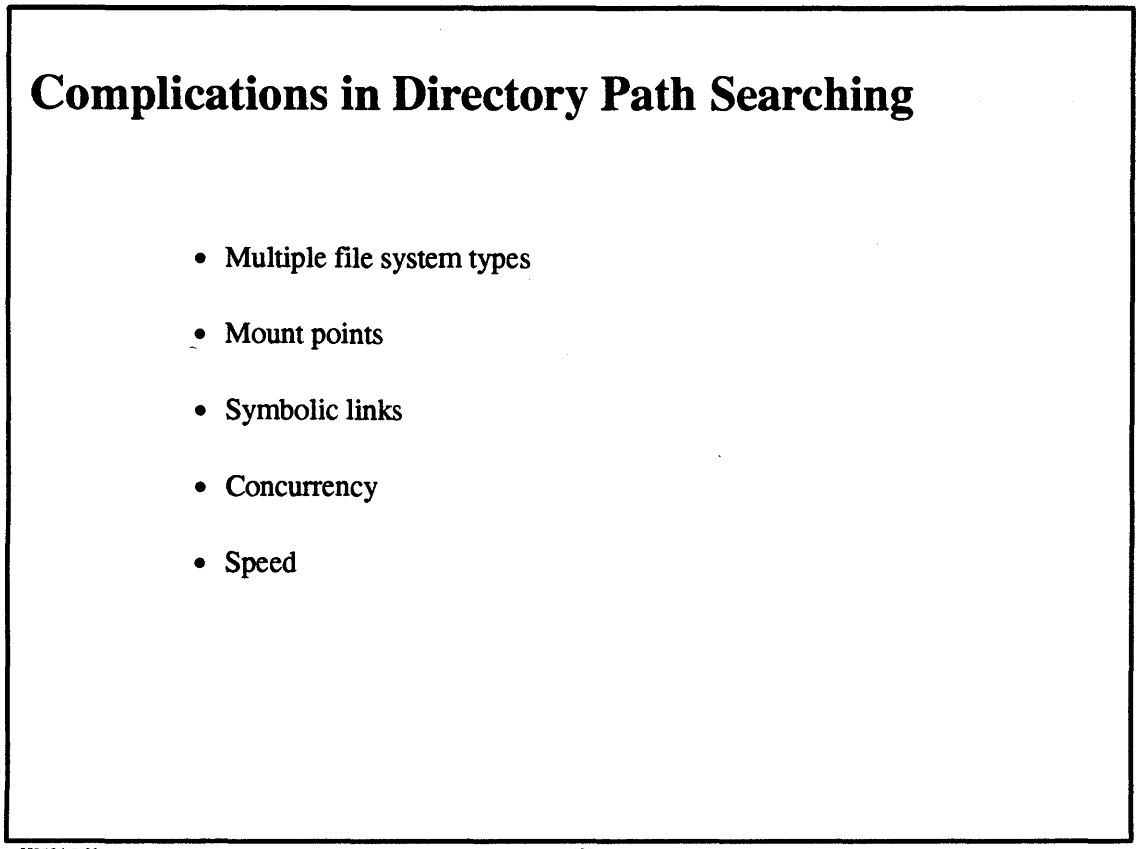
```
start with root vnode or current-directory vnode
while (not at end of path) {
   search for next component in file represented by current vnode
   if not found
      terminate
   fetch associated vnode, assign it to current vnode
}
```

.H2596 5-32. 180

Directory Path Searching

Following directory paths would seem to be quite trivial. The basic algorithm is shown in the picture. However, as will be discussed, the actual procedure is fairly complex, and this subsystem is a very important part of the operating system.

5–33. Directory Path Searching



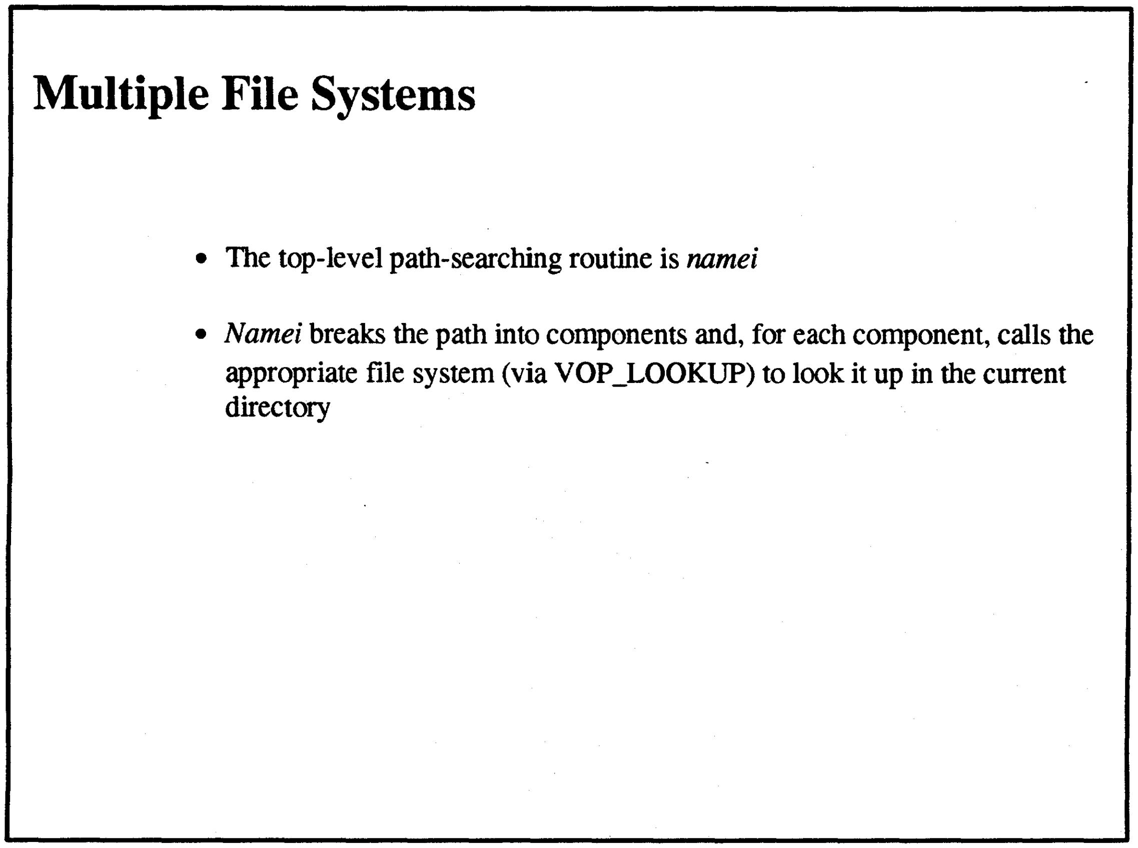
Complications in Directory Path Searching

- Multiple file system types
- Mount points
- Symbolic links
- Concurrency
- Speed

H2596 5-33. 181 © 1991 Hewlett-Packard

Complications in Directory Path Searching

5–34. Directory Path Searching



Multiple File Systems

- The top-level path-searching routine is *namei*
- *Namei* breaks the path into components and, for each component, calls the appropriate file system (via VOP_LOOKUP) to look it up in the current directory

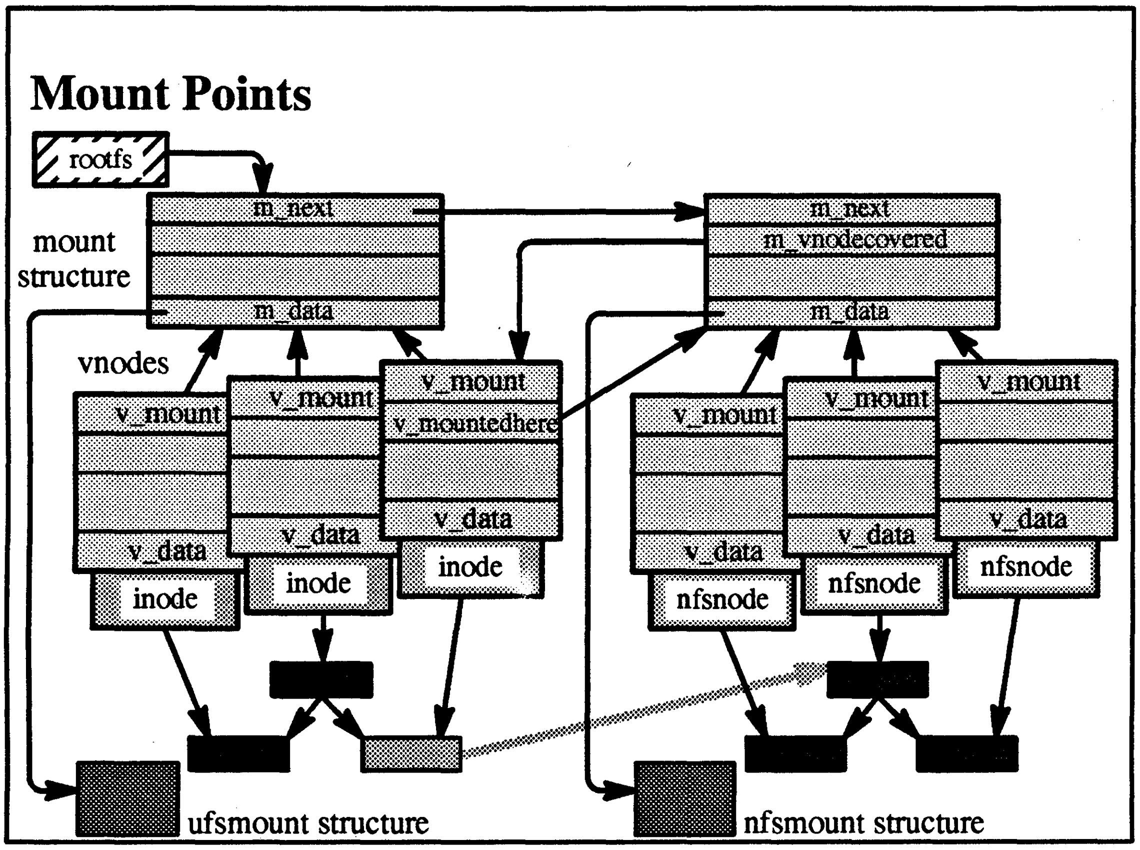
H2596 5-34. 182 © 1991 Hewlett-Packard

Multiple File Systems

One might think that a more efficient technique for following a path would be to give the file system lookup routine all of the remaining portion of the path so that it can follow it as far as possible. This technique is not easy, however, for a number of reasons.

In NFS, it is up to the client to determine which character separates components; the server is not involved. For example, UNIX clients use "/" as the component separator, whereas MS-DOS uses "\". Only the client can break a pathname into its components (though one might argue that the client could pass the component-separator as an argument to the server). But, furthermore, mount points are interpreted strictly by the client, and server mount points mean nothing to the client.

5–35. Directory Path Searching



H2596 5-35. 183 © 1991 Hewlett-Packard

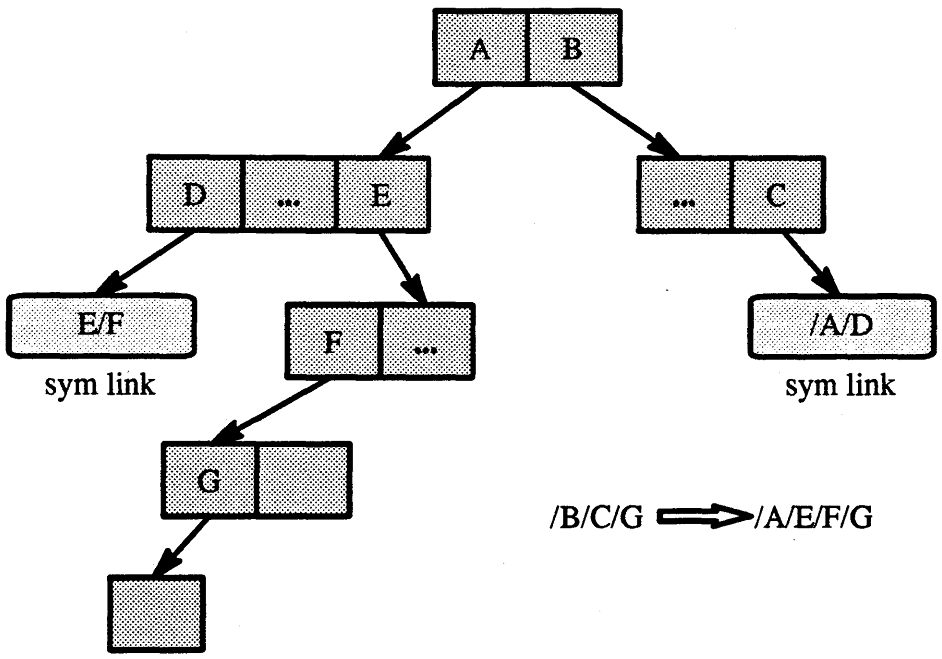
Mount Points

Mount points are encoded in the *vnode* and *mount* structures. This system makes it not only possible but obligatory that clients view a file system independent of mounts done by the server. *Namei* tests each directory it encounters to determine if it is a mount point; if it is, *namei* calls the mounted file system's VFS_ROOT routine to obtain its root vnode.

A related scenario is following ".." links out of a mounted file system. In this case, *namei* consults the mounted file system's mount structure to find the address of the vnode that it covers, and then it follows that directory's ".." link of the directory represented by that vnode.

5–36. Directory Path Searching

Symbolic Links



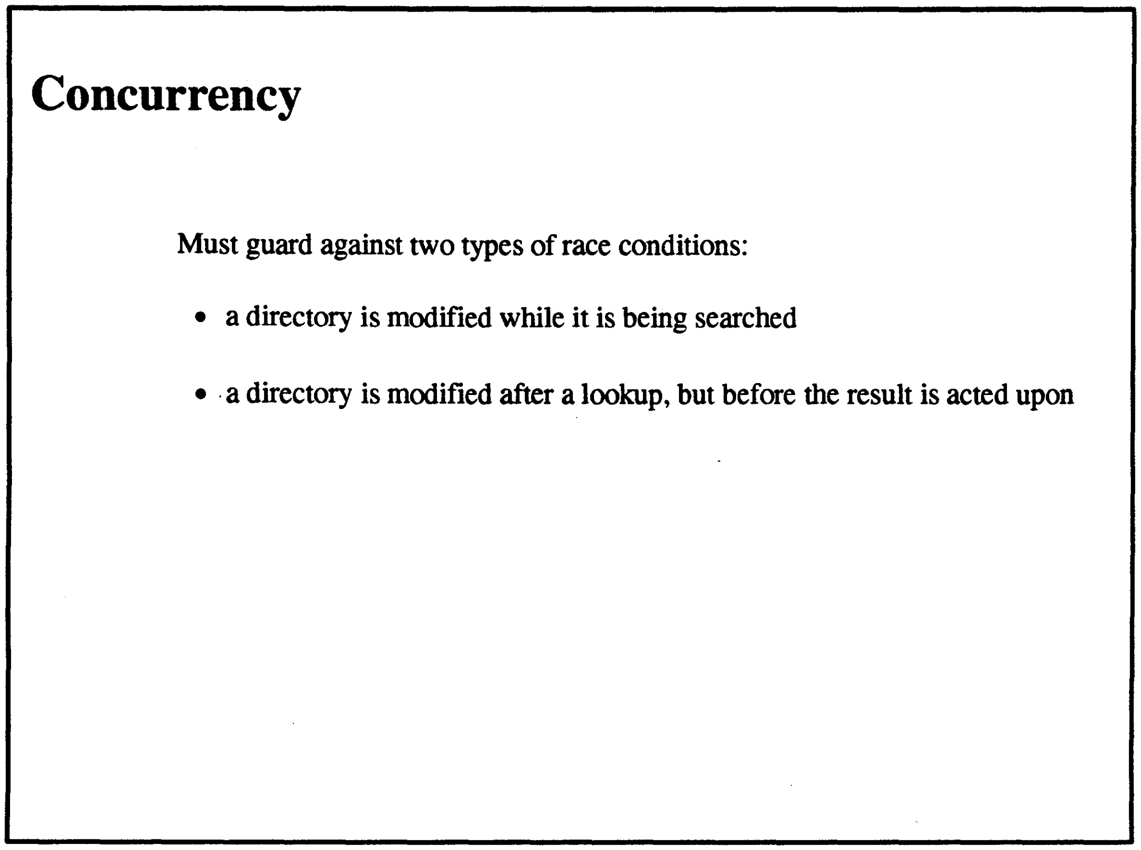
.H2596 5-36. 184 © 1991 Hewlett-Packard

Symbolic Links

If a vnode marked as a symbolic link is encountered, then the file system's VOP_READLINK routine is called to get the link. The routine replaces that portion of the path that has already been followed with the value of the symbolic link, and then restarts the search from the beginning of the newly modified path name.

To avoid loops caused by careless placement of symbolic links, no one path may be composed of more than MAXSYMLINKS (32) symbolic links.

5–37. Directory Path Searching



Concurrency

Must guard against two types of race conditions:

- a directory is modified while it is being searched
- a directory is modified after a lookup, but before the result is acted upon

H2596 5-37. 185 © 1991 Hewlett-Packard

Concurrency

OSF/1 deals with the first case by requiring that a thread hold a *read lock* on a directory while searching it and hold a *write lock* on the directory while modifying it.

Timestamps are used to deal with the second case. Each time a thread modifies a directory, it increments the directory's timestamp by one. When a thread searches a directory it records the directory's current timestamp. Before the thread modifies a directory, it compares the timestamp it obtained in the lookup request to the directory's current timestamp. If the timestamps are different, then the directory must have been modified since the lookup, and the thread repeats the lookup. If the lookup fails a second time, then the operation fails.

For example, suppose two threads issue concurrent delete requests for the same directory entry. The net result should be that one succeeds and the other fails. Both threads do a successful lookup of the entry and one thread succeeds in deleting the entry. The other thread will note before it attempts to remove the entry that the timestamp has changed. It will thus repeat the lookup, the lookup will fail, and so the delete system call will fail.

An example of the destructive effects of the second race condition is the following set of operations, each performed by a separate thread:

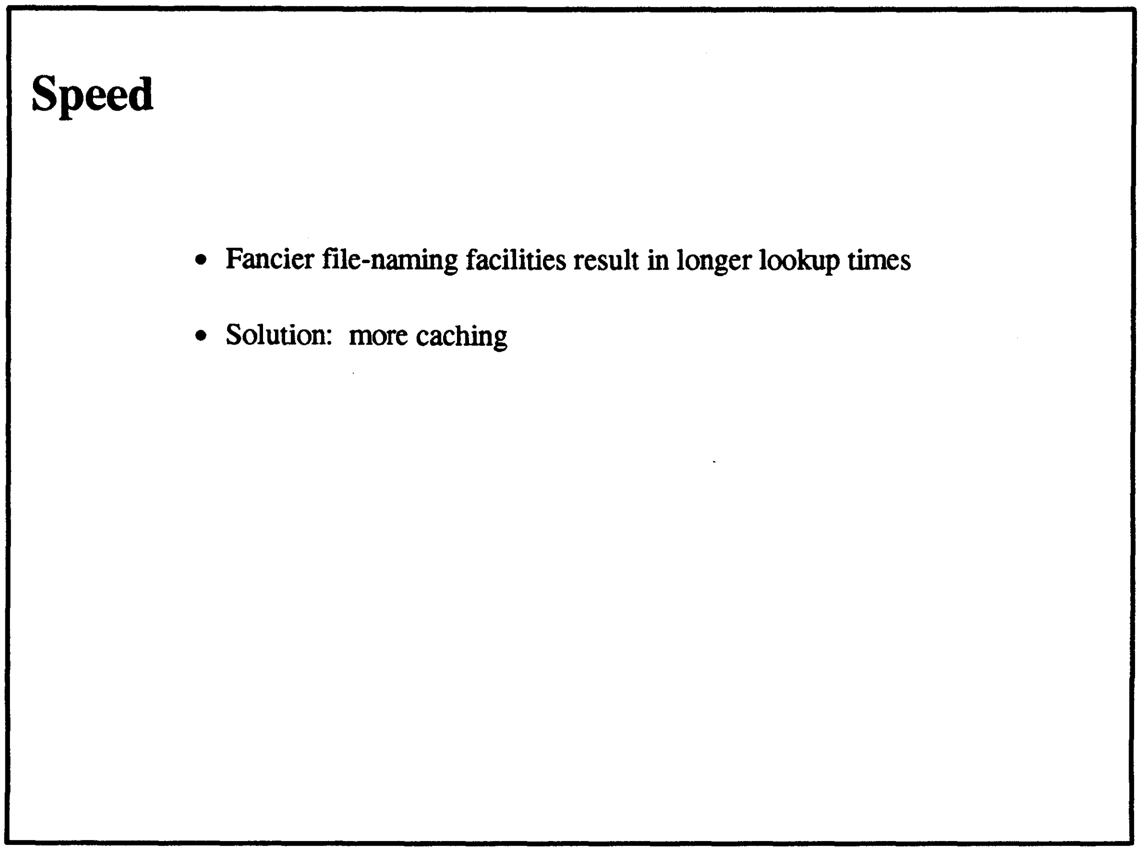
```
thread 1: rm  /A

thread 2: rm  /A

thread 3: cp /C  /B
```

The two *rm*s are executed concurrently: both threads do a successful lookup to determine that /A exists; as a side effect the lookup returns the position of the component A within the / directory. This lookup is performed while holding a read lock; thus both threads can do it in parallel. Modifying the directory to delete the entry A, however, requires an exclusive write lock. Thus one thread blocks while the other thread removes the entry A. However, it happens that immediately after thread 1 removes entry A, thread 3 creates the entry B in the directory slot just vacated by A. When thread 2 wakes up and completes its operation, it removes what it thinks is entry A but is in fact entry B.

5–38. Directory Path Searching



Speed

- Fancier file-naming facilities result in longer lookup times
- Solution: more caching

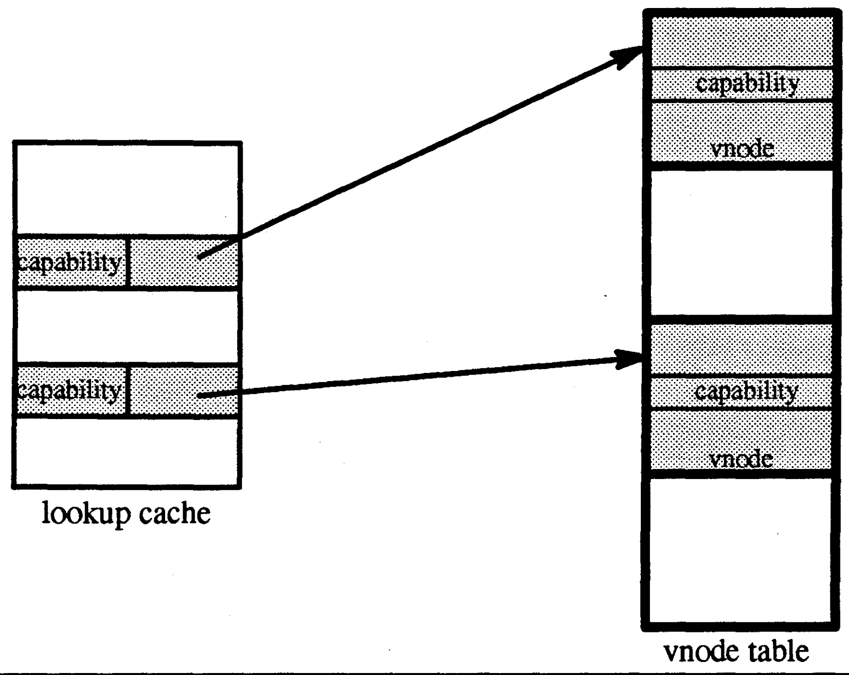
.H2596 5-38. 186 © 1991 Hewlett-Packard

Speed

4.2BSD added many new facilities not present in earlier versions of UNIX. One result of these additions was that 4.2BSD was considerably slower than 4.1BSD. Kernel profiling showed that approximately 25% of system time spent in the kernel was spent in routines translating directory paths. This was much too much time for such chores, so to speed things up, two forms of caching were introduced. Both forms were designed for use with the UFS file system, but may be used with any file system. With the addition of the two types of caching, the system time devoted to name translation dropped from 25% to less than 10%.

5–39. Directory Path Searching

The Lookup Cache



.H2596 5-39. 187

Lookup Cache

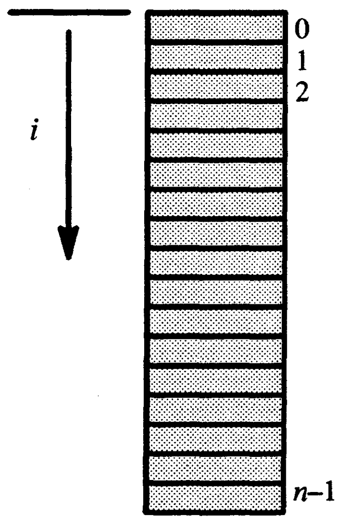
The *lookup cache* is a cache of the most recent component-name-to-vnode translations. Searching a directory for a component name can be expensive, so the most recent lookups are kept in a cache. (Note that this is not a cache of path names, but merely of component names.)

A vnode reference presents a problem in representing the result of a translation. If the cache contains actual "reference-counted" references to the vnode, then the reference count on vnodes themselves remains positive, and incore vnodes are not freed. (SVR4 actually does employ this technique: when the system is low on available vnodes, it makes a pass through the cache and frees those vnodes whose only reference is due to the cache.)

The OSF/1 cache, derived from 4.4BSD, contains "soft" references to vnodes, i.e. references that do not show in the reference count. The problem here is that if a file is deleted and its vnode reused for another file, the cache continues to contain a reference to the vnode in its previous incarnation, since there is no indication in the vnode that a cache entry refers to it. To deal with this, vnodes and the cache contain *capabilities* (version numbers)—32-bit integers. Each vnode has an assigned capability. When the vnode is invalidated, the version number is incremented by one. Each cache translation also contains a version number, which is set equal to that of the vnode. If the version numbers do not match when the translation is accessed, then the cache entry is considered invalid and is flushed. Berkeley's figures indicate that this cache has a "hit rate" of 70–80%.

5–40. Directory Path Searching

The Search Cache



.H2596 5-40. 188 © 1991 Hewlett-Packard

The Search Cache

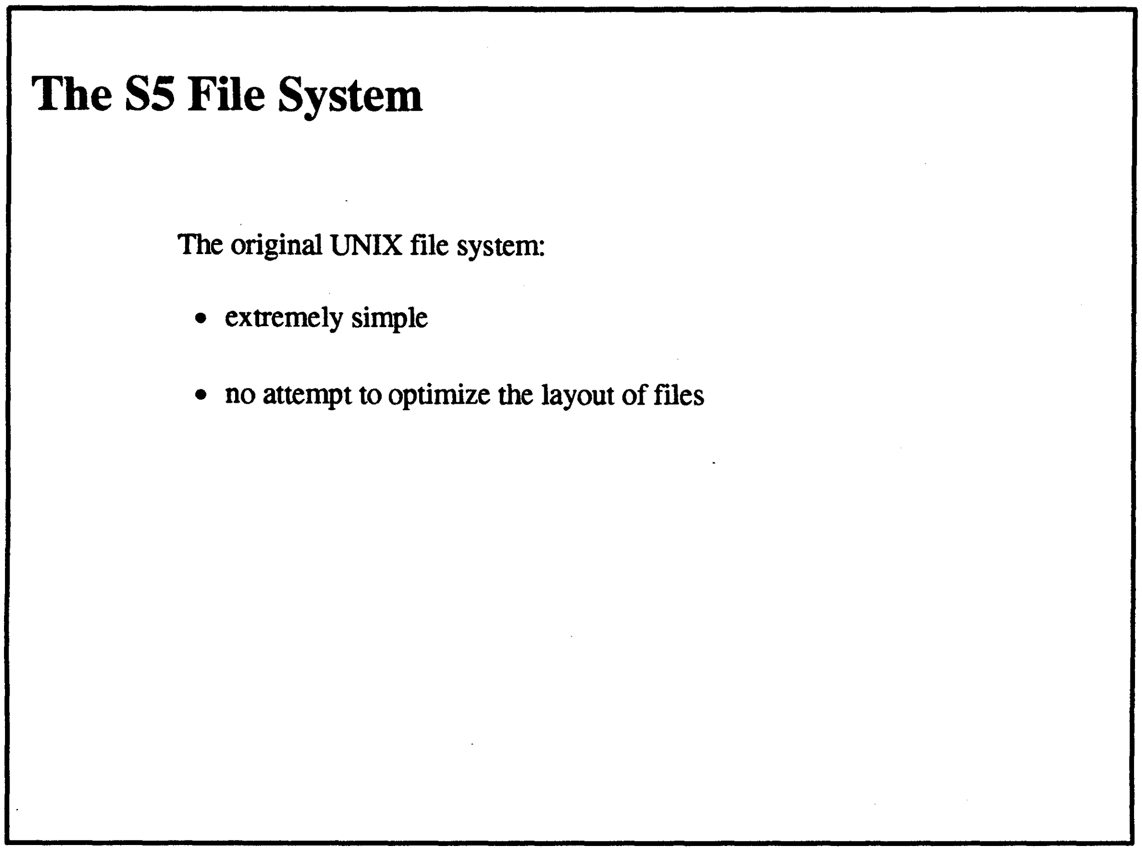
The second form of caching deals with repeated lookups of one directory. Consider a command such as *ls –l*: its implementation involves reading the contents of the directory, then performing a stat system call on each entry. It takes time proportional to i to search for the ith entry, since the search always starts at the beginning of the directory. Thus, for n entries, time proportional to n^2 is needed to find each entry in the directory. For a large directory this could be rather significant.

By storing in the inode the offset of where the last search terminated, a linear algorithm for *ls –l* (and others) can be devised, since the search for the next item in the directory will start where the previous item was found.

Note that 4.2BSD and 4.3BSD stored the offset in the *user* structure. This approach seems better, especially if multiple threads are each doing the equivalent of *ls –l* on the same directory. However, storing the offset in the inode makes this technique work when *ls –l* is applied to directories other than the current directory (i.e., when each directory lookup involves searching a path).

The search cache has a "hit rate" of 5–15%.

5–41. S5 File System



The S5 File System

The original UNIX file system:

- extremely simple
- no attempt to optimize the layout of files

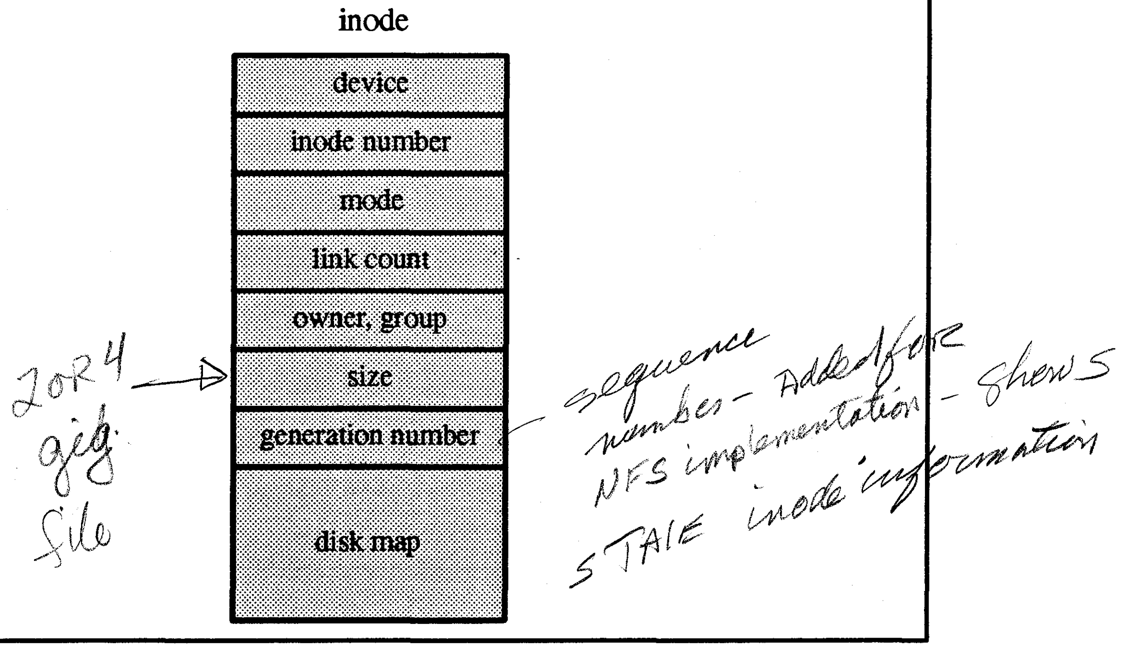
.H2596 5-41. 189 © 1991 Hewlett-Packard

The S5 File System

The S5 file system, provided primarily for compatibility reasons, is generally always slower than the UFS file system. However, it has a few things in common with the UFS file system, in particular the notion of inodes (including the disk map).

5–42. S5 File System

Inodes

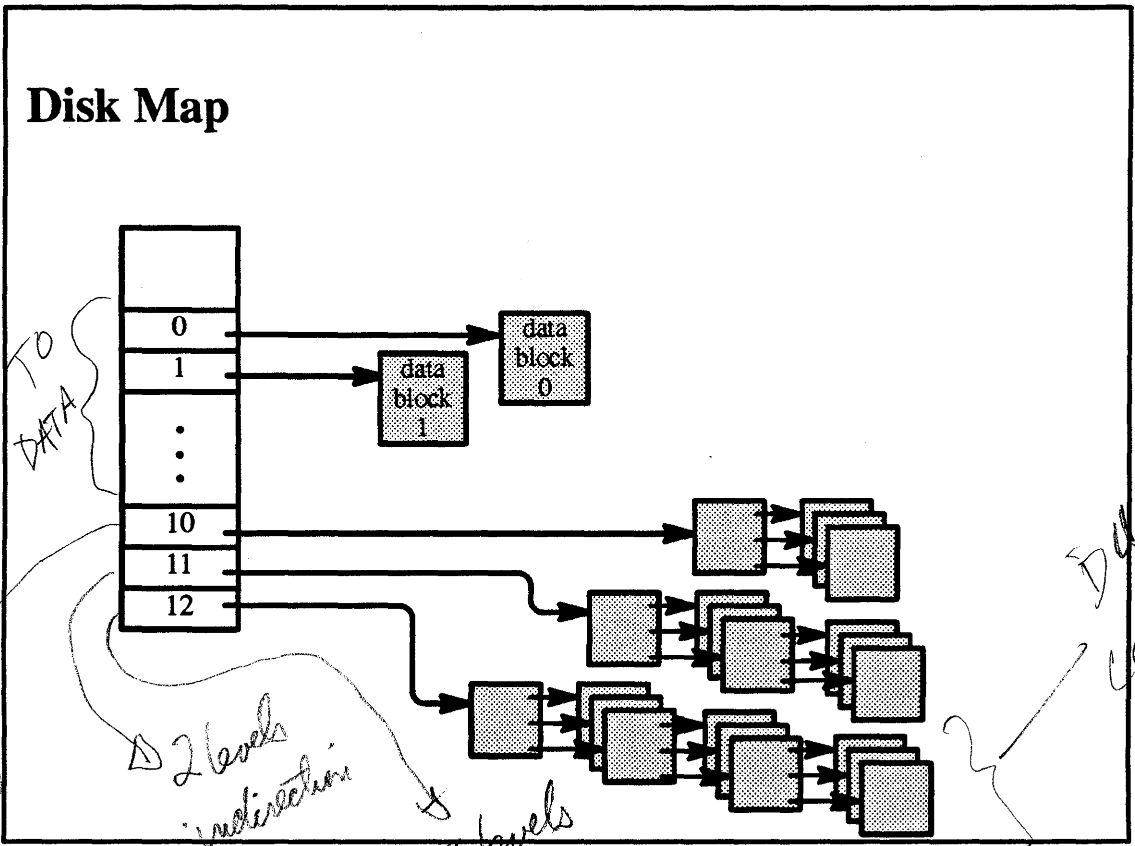


.H2596 5-42. 190 © 1991 Hewlett-Packard

Inodes

Inodes are the focus of all file activity, i.e., every access to a file must go through the inode. Every file has a inode on permanent storage; this on-disk inode is of type *struct dinode* in the S5 file system. All open files, current directories, mounted-on directories, and the root have incore inodes of type *struct S5inode.* Once brought into primary storage, an inode stays there until its associated file is deleted or its storage is needed for some other purpose.

5–43. S5 File System



Due to constraints on size of inode - can't fully use

.H2596 5-43. 191 © 1991 Hewlett-Packard

Disk Map

The purpose of the disk-map portion of the inode is to map block numbers relative to the beginning of a file into block numbers relative to the beginning of the file system. An S5 file system may be configured with a 512-byte, 1K-byte, or 2K-byte block size. We assume a 1K block size from here on.

The disk map consists of 13 pointers to disk blocks, the first 10 of which point to the first 10 blocks of the file. Thus the first 10Kb of a file are accessed directly. If the file is larger than 10Kb, then pointer number 10 points to a disk block called the *indirect block.* This block contains up to 256 (4-byte) pointers to data blocks (i.e., 256Kb of data). If the file is bigger than this (256K +10K = 266K), then pointer number 11 points to a double indirect block containing 256 pointers to indirect blocks, each of which contains 256 pointers to data blocks (64Mb of data). If the file is bigger than this (64Mb + 256Kb + 10Kb), then pointer number 12 points to a triple indirect block containing up to 256 pointers to double indirect blocks, each of which contains up to 256 pointers pointing to single indirect blocks, each of which contains up to 256 pointers pointing to data blocks (potentially 16Gb, although, as will be discussed, the real limit is either 2Gb or 4Gb).

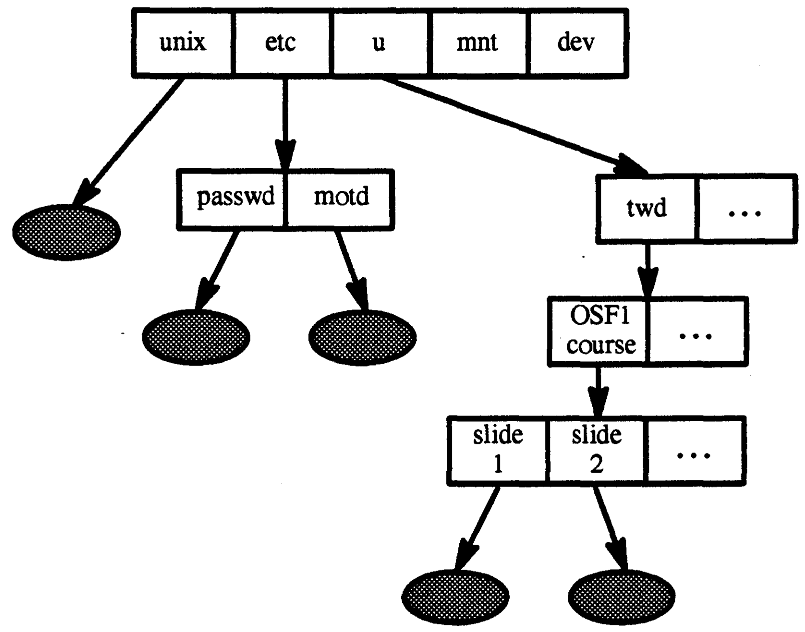
The structure of the UFS file system is similar, except that the block size is either 4K or 8K and the disk map consists of 15 pointers, the first 12 of which point to the first 12 data blocks. Because of the larger block size, the triple indirect block is unusable, since the double indirect block can represent a file size larger than 4Gb. A hard limit on file size for 32-bit architectures is 4Gb (or perhaps 2Gb, depending on one's feelings about sign bits), since the offset into a file must fit in a word!

This data structure allows the efficient representation of *sparse* files, i.e., files whose content is mainly zeros. Consider, for example, the effect of creating an empty file and then writing one byte at location 2,000,000,000. Only four disk blocks are allocated to represent this file: a triple indirect block, a double indirect block, a single indirect block, and a data block. All pointers in the disk map, except for the last one, will be zero. If the file is read, all bytes up to the last one will read as zero. This is because a zero pointer is treated as if it points to a block containing all zeros: a zero pointer to an indirect block is treated as if it points to an indirect block filled with zero pointers, each of which is treated as if it points to a data block filled with zeros. However, one must be careful about copying such a file, since commands such as *cp* and *tar* actually attempt to write all the zero blocks! (The *dump* command, on the other hand, copes with sparse files properly.)

The units of the pointers in the disk map in the S5 file system are in blocks (1K). For the UFS file system, the units are in fragments that can be any multiple of 512 bytes, from 512 bytes to 8K bytes (this value is fixed for each instance of the file system).

5–44. S5 File System

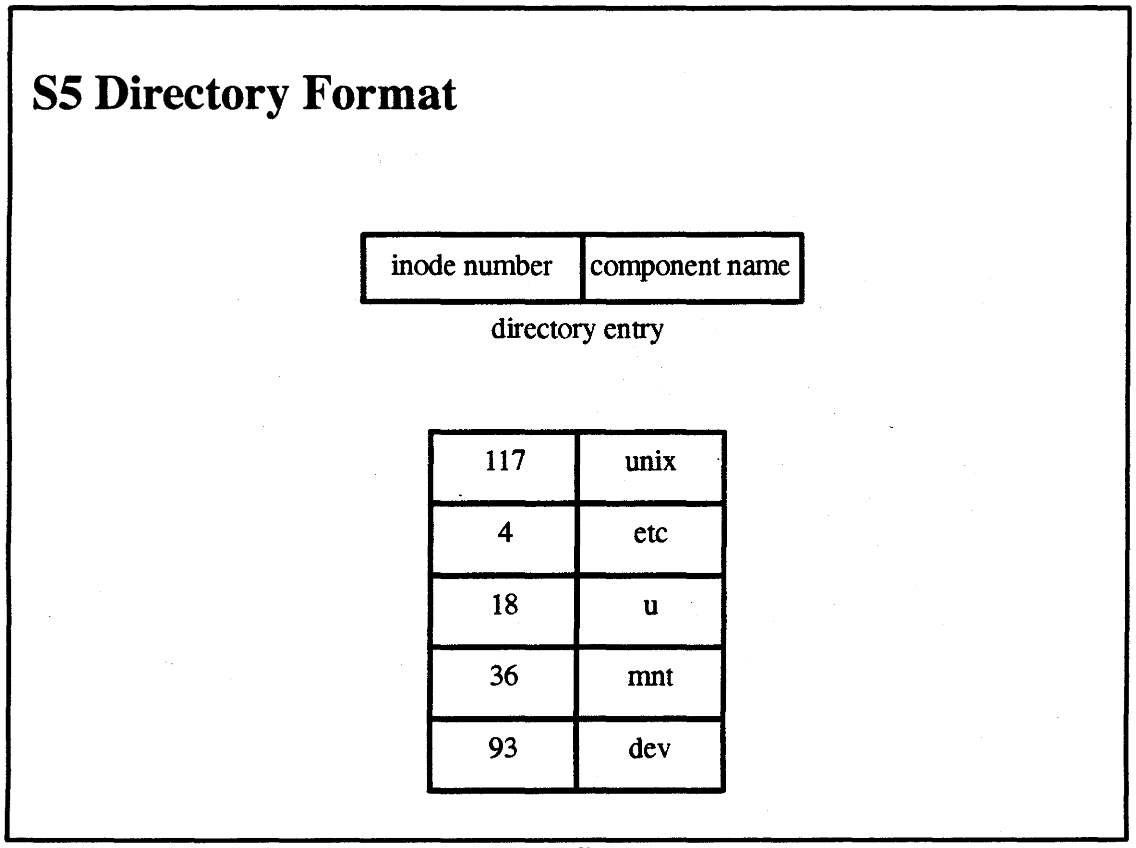
Directory Structure



H2596 5-44. 192

Directory Structure

5–45. S5 File System



S5 Directory Format

inode number	component name

directory entry

117	unix
4	etc
18	u
36	mnt
93	dev

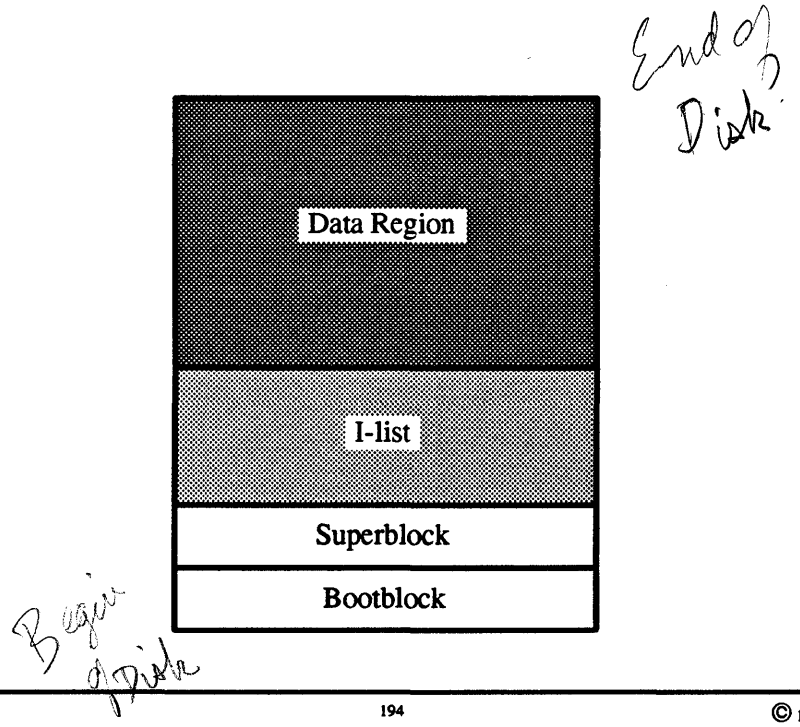
.H2596 5-45. 193 © 1991 Hewlett-Packard

S5 Directory Format

The S5 directory consists of an array of pairs of inode number and component number. An important restriction is that the component name may be no longer than 14 bytes, thereby making a fixed length format possible. Note that identifying a file requires a reference to the file system as well as the inode number, but only the latter is supplied in each directory. The file system is assumed to be the one that contains the directory. Thus the only way a path can cross a file system boundary is via mount points.

5–46. S5 File System

File System Layout



.H2596 5-46. 194

File System Layout

- Bootblock
  — used on some systems to contain a bootstrap program
- Superblock
  — describes the file system:
    - ♦ total size
    - ♦ size of inode list (I-list)
    - ♦ header of free-block list
    - ♦ list of free inodes

- ♦ modified flag
- ♦ read-only flag
- ♦ number of free blocks and free inodes
- ♦ resides in a buffer borrowed from the buffer cache while the file system is mounted

- I-list

— area for allocating inodes

- Data region

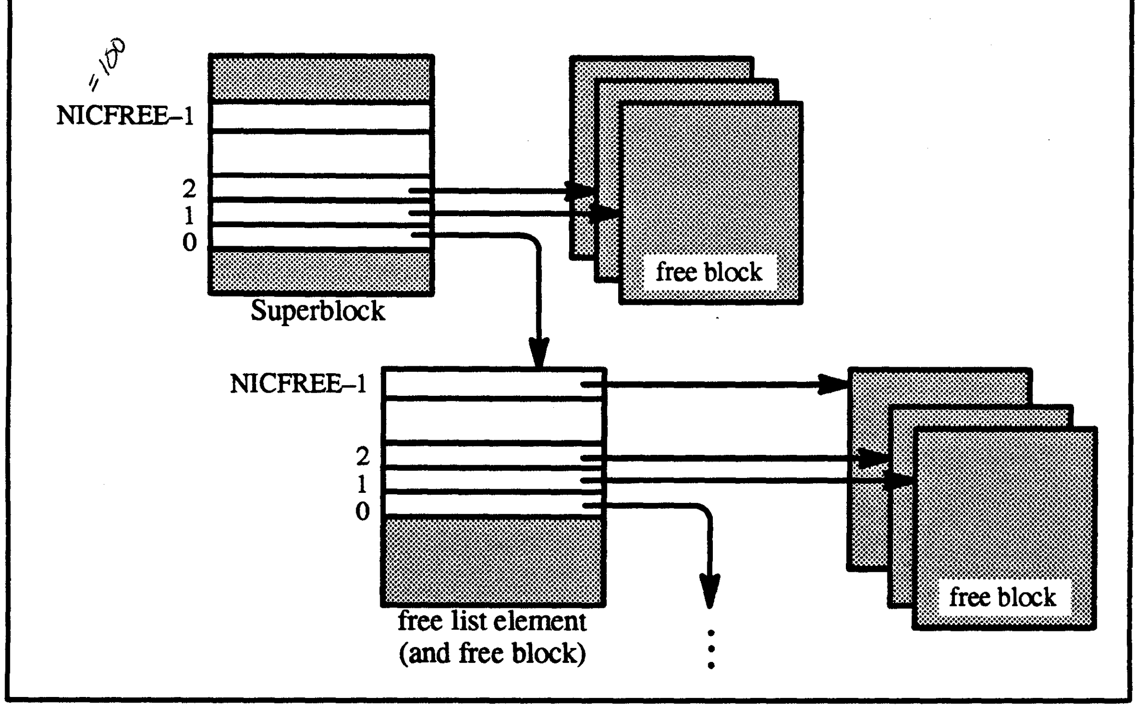
— remainder of file system is for data blocks and indirect blocks

A problem with this organization is that the I-list and the data region are separated from each other. Since one must always fetch the inode before reading or writing the blocks of a file, the disk head is constantly moving back and forth between the I-list and the data region.

in Superblock

5–47. S5 File System

Free Block List



H2596 5-47. 195 © 1991 Hewlett-Packard

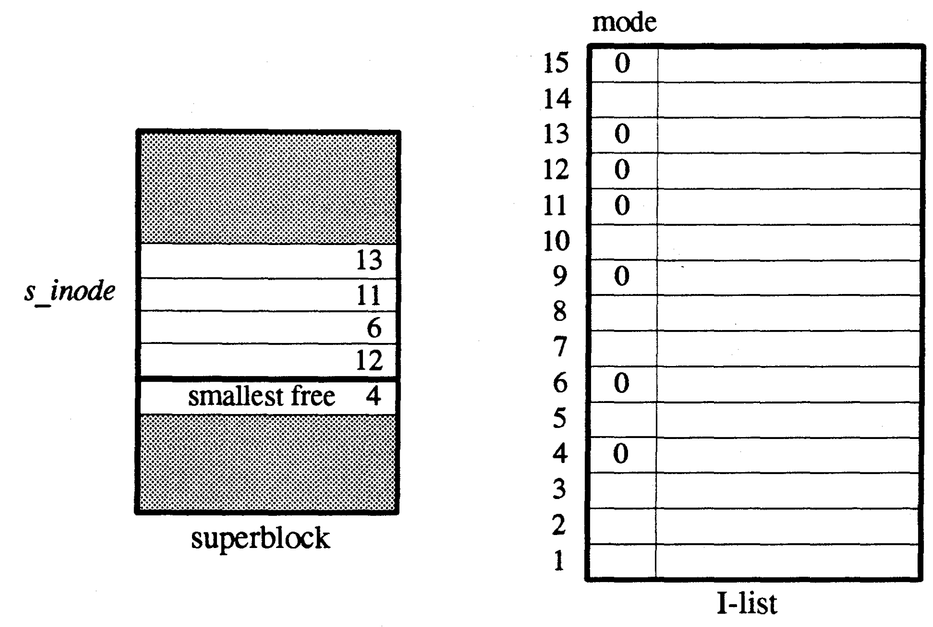
Free Block List

Free disk blocks are organized as shown in the picture. The superblock contains the address of up to NICFREE (= 100) free disk blocks. The last of these disk blocks contains NICFREE pointers to additional free disk blocks. The last of these pointers points to another block containing up to NICFREE free disk blocks, etc., until all free disk blocks are represented. Thus most requests for a free block can be satisfied by merely getting an address from the superblock. When the last block reference by the superblock is consumed, however, a disk read must be called to fetch the addresses of up to 100 more free disk blocks. Freeing a disk block results in reconstructing the list structure.

This organization, though very simple, scatters the blocks of files all over the surface of the disk. When allocating a block for a file, one must always use the next block from the free list; there is no way to request a block at a specific location. No matter how carefully the free list is ordered when the file system is initialized, it becomes fairly well randomized after the file system has been used for a while.

5–48. S5 File System

Managing Inodes



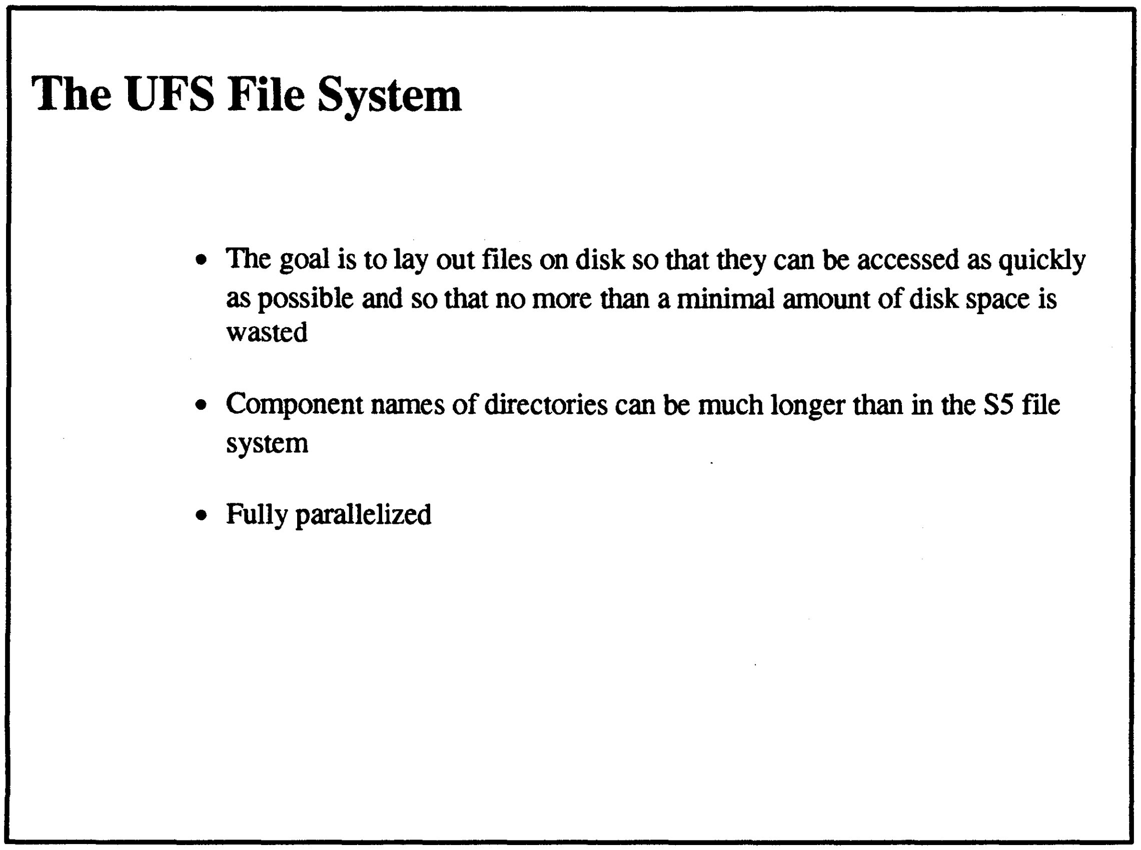
H2596 5-48. 196 © 1991 Hewlett-Packard

Managing Inodes

Inodes are allocated from the I-list. Free inodes are represented simply by zeroing their mode bits. The superblock contains a cache of indices of free inodes in an array called *s_inode* (of size NICINOD). When a free inode is needed (i.e., to represent a new file), its index is taken from this cache. If the cache is empty, then the I-list is scanned sequentially until enough free inodes are found to refill the cache.

To speed this search somewhat, the cache contains a reference to the inode with the smallest index that is known to be free. When an inode is free, it is added to the cache if there is room, and its mode bits are zeroed on disk.

5–49. UFS File System



The UFS File System

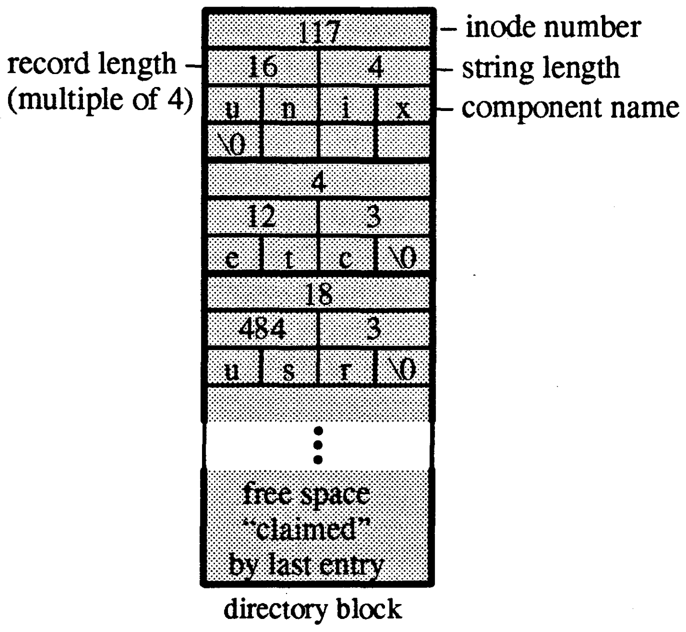
- The goal is to lay out files on disk so that they can be accessed as quickly as possible and so that no more than a minimal amount of disk space is wasted
- Component names of directories can be much longer than in the S5 file system
- Fully parallelized

.H2596 5-49. 197 © 1991 Hewlett-Packard

The UFS File System

5–50. UFS File System

UFS Directory Format



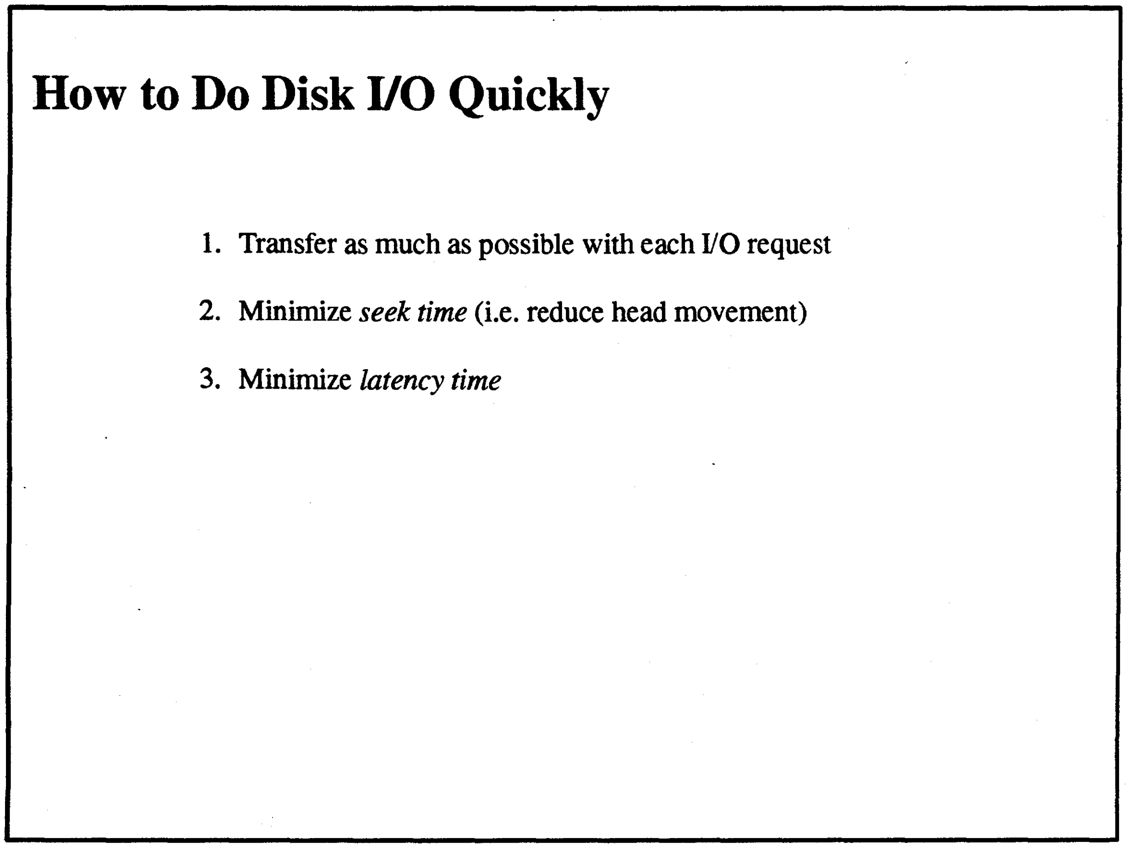
UFS Directory Format

UFS allows component names to be up to 255 characters long, thereby necessitating a variable-length field for components. Directories are composed of 512-byte blocks and entries must not cross block boundaries. This design adds a degree of atomicity to directory updates. It should take exactly one disk write to update a directory entry (512 bytes was chosen as the smallest conceivable disk sector size). If it takes two disk writes to modify a directory entry, then clearly the disk will crash between the two disk writes!

Like the S5 directory entry, the UFS directory entry contains the inode number and the component name. Since the component name is of variable length, there is also a string length field (the component name includes a null byte at the end; the string length does not include the null byte). In addition to the string length, there is also a record length, which is the length of the entire entry (and must be a multiple of four to ensure that each entry starts on a four-byte boundary). The purpose of the record length field is to represent free space within a directory block. Any free space is considered a part of the entry that precedes it, and thus a record length longer than necessary indicates that free space follows. If a directory entry is free, then its record length is added to that of the preceding entry. However, if the first entry in a directory block is free, then this free space is represented by setting the inode number to zero and leaving the record length as is.

Compressing directories is considered to be too difficult. Free space within a directory is made available for representing new entries, but is not returned to the file system. However, if there is free space at the end of the directory, the directory may be truncated to a directory block boundary.

5–51. UFS File System



How to Do Disk I/O Quickly

1. Transfer as much as possible with each I/O request
2. Minimize *seek time* (i.e. reduce head movement)
3. Minimize *latency time*

H2596 5-51. 199 © 1991 Hewlett-Packard

How to Do Disk I/O Quickly

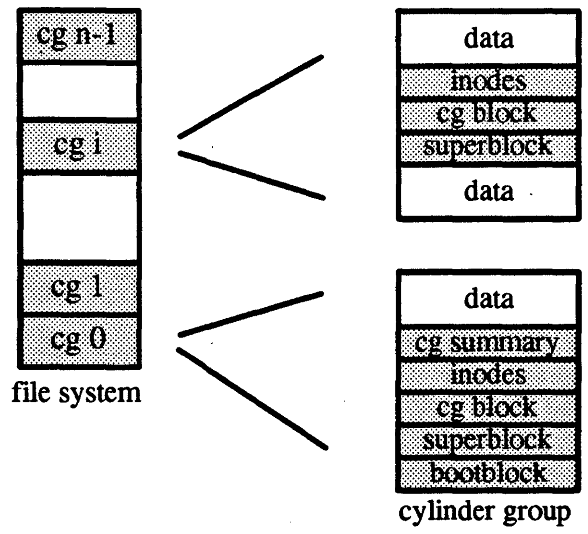
The UFS file system uses three techniques to improve I/O performance. The first technique, which has perhaps the greatest payoff, maximizes the amount of data transferred with each I/O request by using a relatively large block size. UFS block sizes may be either 4K bytes or 8K bytes (the size is fixed for each individual file system). A problem with using a large block size is the wastage due to internal fragmentation: on the average, half of a disk block is wasted for each file. To alleviate this problem, blocks under certain circumstances may be shared among files.

The second technique to improve performance is to minimize seek time by attempting to locate the blocks of a file so that they are near to one another.

Finally, UFS attempts to minimize latency time, i.e. to reduce the amount of time spent waiting for the disk to rotate to bring the desired block underneath the desired disk head (many modern disk controllers make it either impossible or unnecessary to apply this technique).

5-52. UFS File System

UFS Layout



UFS Layout

- Superblock *(struct fs)*
  - — incore while the file system is mounted
  - — contains the parameters describing the layout of the file system
  - — for paranoia's sake, one copy is kept in each cylinder group, at a rotating track position
- Cylinder group summary *(struct csum,* one for each cylinder group)
  - — incore while the file system is mounted
  - — contains a summary of the available storage in each cylinder group
  - — allocated from the data section of cylinder group 0

- Cylinder group block *(struct cg)*
  - — resides in the buffer cache "as needed"
  - — contains free block map and all other allocation information

Note: the superblock contains two sorts of information, *static* and *dynamic*. The *static* information describes the layout of the entire file system and is essential to make sense of the file system. The *dynamic* information describes the file system's current state and can be computed from redundant information in the file system. If the static portion of the superblock is lost, then the file system cannot be used. To guard against this, each cylinder group contains a copy of the superblock (just the static information needs to be copied).

A possible (though unlikely) failure condition might be that the entire contents of one surface are lost, but the remainder of the disk is usable. However, if this surface contains all copies of the superblock, then the rest of the disk would be effectively unusable. To guard against this, the copy of the superblock is placed on a different surface in each cylinder group. Of course, the system must keep track of where these copies are. This information is kept in the disk label (along with information describing how the physical disk is partitioned).

5–53. UFS File System

Minimizing Fragmentation Costs

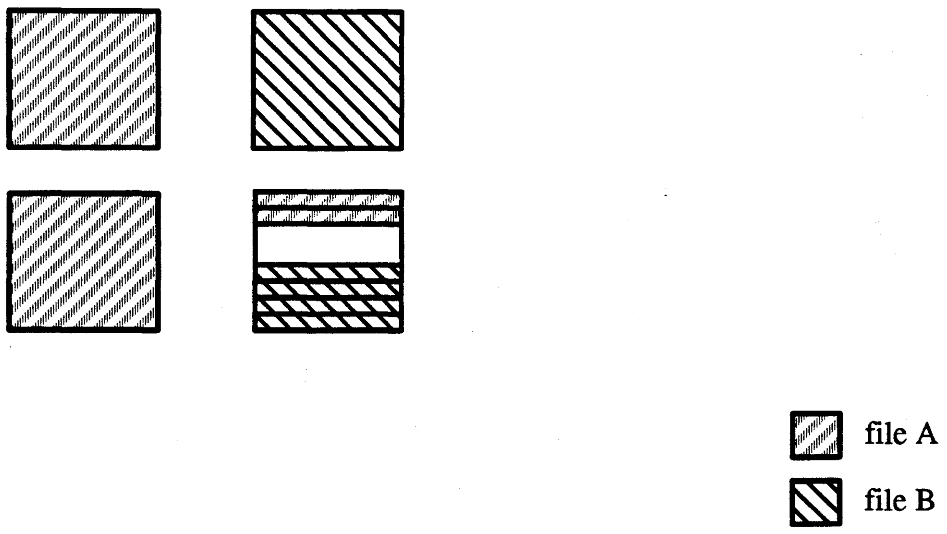
- A file system block may be split into fragments that can be independently assigned to files
  — fragments assigned to a file must be contiguous and in order
- The number of fragments per block (1, 2, 4, or 8) is fixed for each file system
- Allocation in fragments may only be done on what would be the last block of a file, and only if the file does not contain indirect blocks

H2596 5-53. 201

Minimizing Fragmentation Costs

5–54. UFS File System

The Use of Fragments, part 1



.H2596 5-54. 202

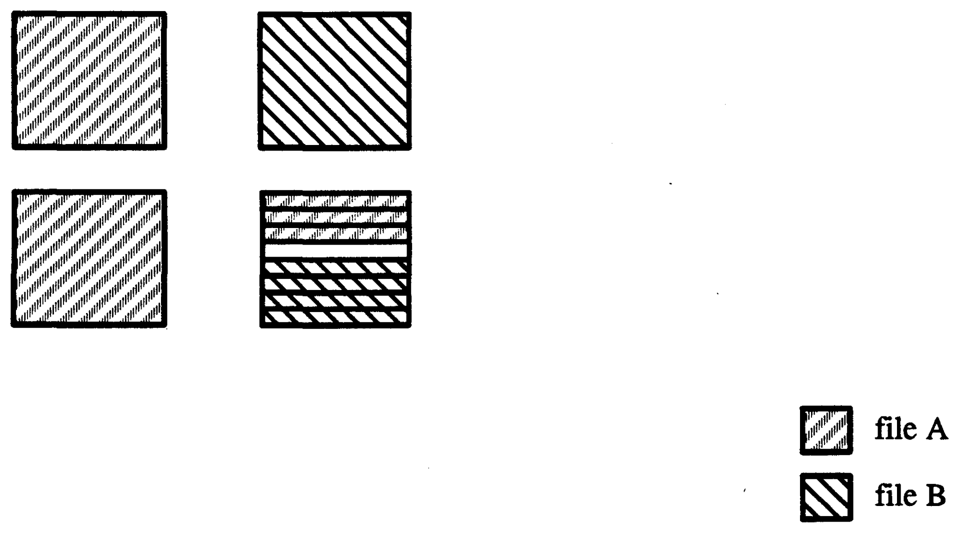
The Use of Fragments, part 1

This example illustrates a difficulty associated with the use of fragments. The file system must preserve the invariant that fragments assigned to a file must be contiguous and in order, and that allocation of fragments may be done only on what would be the last block of the file. In the picture, the direction of growth is downwards. Thus file A may easily grow by up to two fragments, but file B cannot easily grow within this block.

In the picture, file A is 18 fragments in length, file B is 12 fragments in length.

5–55. UFS File System

The Use of Fragments, part 2



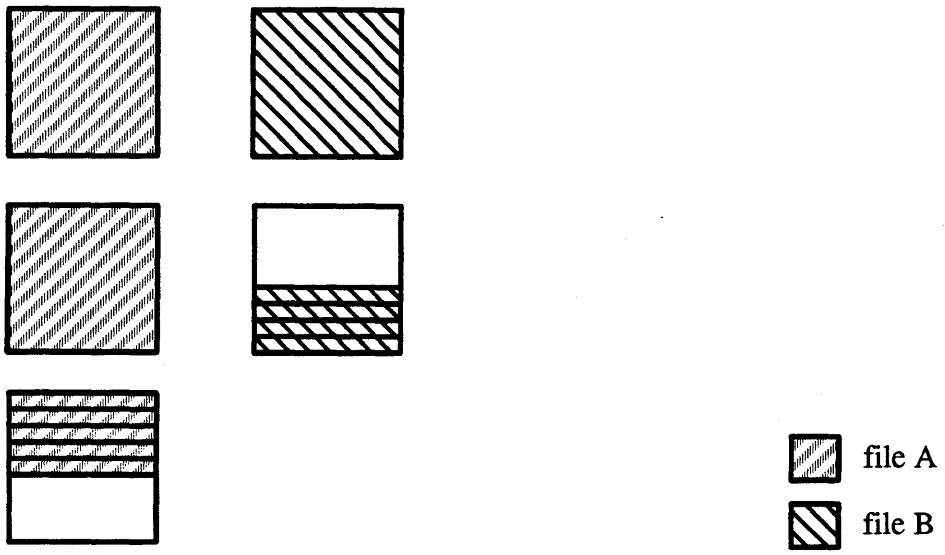
.H2596 5-55. 203 © 1991 Hewlett-Packard

The Use of Fragments, part 2

File A grows by one fragment.

5–56. UFS File System

The Use of Fragments, part 3



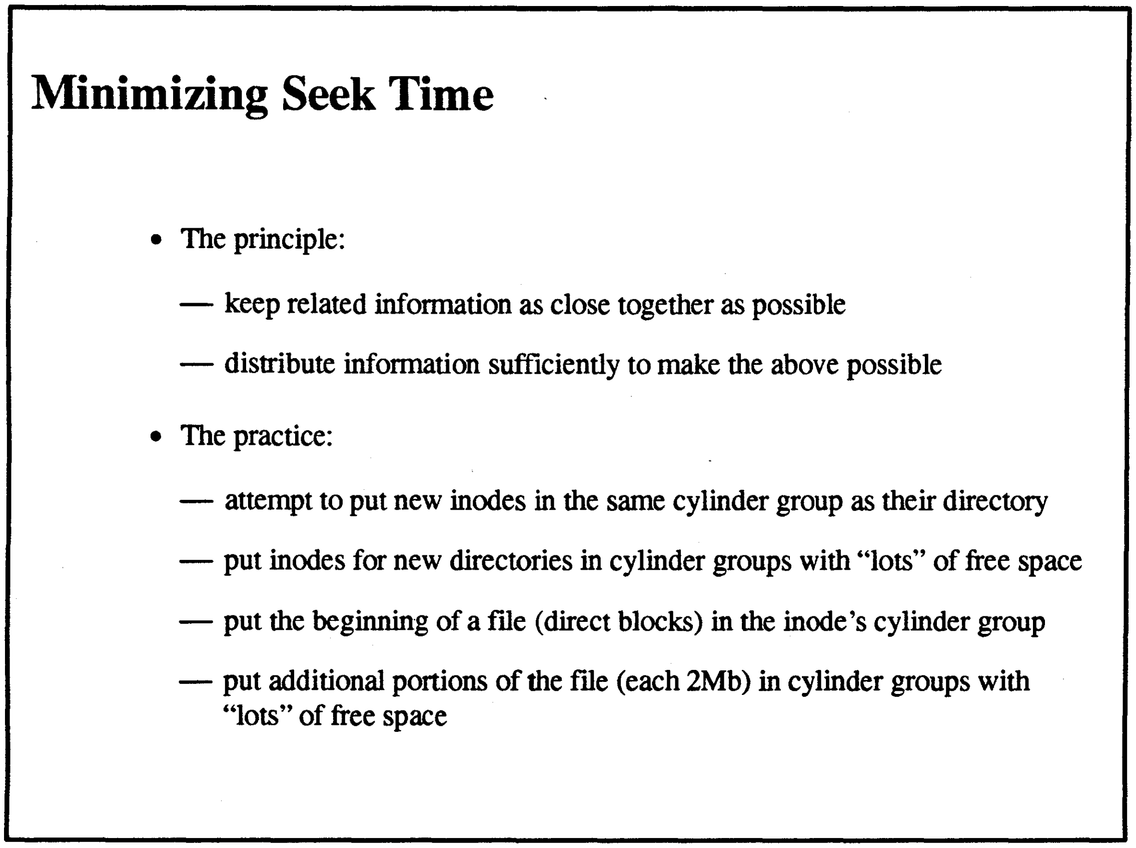
The Use of Fragments, part 3

File A grows by two more fragments, but since there is no space for it, the file system allocates another block and copies file A's fragments into it. How much space should be available in the newly allocated block? If the newly allocated block is entirely free, i.e., none of its fragments are used by other files, then further growth by file A will be very cheap. However, if the file system uses this approach all the time, then we do not get the space-saving benefits of fragmentation. An alternative approach is to use a "best-fit" policy: find a block that contains exactly the number of free fragments needed by file A, or if such a block is not available, find a block containing the smallest number of contiguous free fragments that will satisfy file A's needs.

Which approach is taken depends upon the degree to which the file system is fragmented. If disk space is relatively unfragmented, then the first approach is taken ("optimize for time"). Otherwise, i.e., when disk space is fragmented, the file system takes the second approach ("optimize for space").

The points at which the system switches between the two policies is parameterized in the superblock: a certain percentage of the disk space, by default 10%, is reserved for superuser. (Disk allocation techniques need a reasonable chance of finding free disk space in each cylinder group in order to optimize the layout of files.) If the total amount of fragmented free disk space (i.e., the total amount of free disk space not counting that portion consisting of whole blocks), increases to 8% of the size of the file system (or, more generally, increases to 2% less than the reserve), then further allocation is done using the best-fit approach. Once this approach is being used, if the total amount of fragmented free disk space drops below 5% (or half of the reserve), then further allocation is done using the whole-block technique.

5–57. UFS File System



Minimizing Seek Time

- The principle:
  - — keep related information as close together as possible
  - — distribute information sufficiently to make the above possible
- The practice:
  - — attempt to put new inodes in the same cylinder group as their directory
  - — put inodes for new directories in cylinder groups with "lots" of free space
  - — put the beginning of a file (direct blocks) in the inode's cylinder group
  - — put additional portions of the file (each 2Mb) in cylinder groups with "lots" of free space

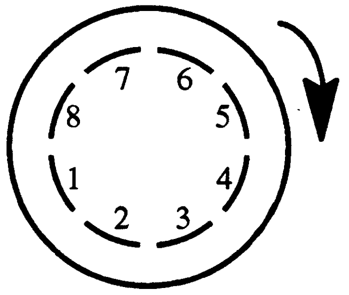
H2596 5-57. 205 © 1991 Hewlett-Packard

Minimizing Seek Time

The definition of "lots of free space" is a greater than average amount of free space, a quantity that is easily computed from the (incore) cylinder group summary.

5–58. UFS File System

Minimizing Latency, part 1



.H2596 5-58. 206 © 1991 Hewlett-Packard

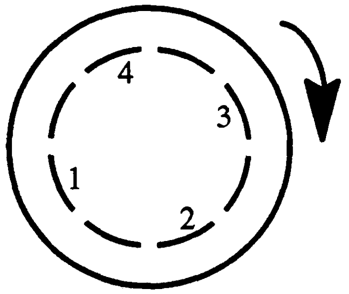
Minimizing Latency, part 1

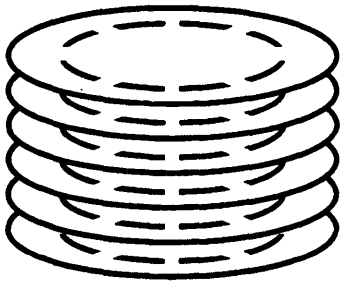
A naive way of laying out consecutive blocks of the file on a track would be to put them in consecutive locations. The problem with this is that some amount of time passes between the completion of one disk request and the start of the next. During this time, the disk rotates a certain distance, probably far enough so that the disk head is positioned after the next block. Thus it will be necessary to wait for the disk to rotate almost a complete revolution for it to bring the beginning of the next block underneath the disk head. This delay could cause a significant slowdown.

This technique is perhaps not as useful today as in the past, since many disk controllers buffer entire tracks and hide the relevant disk geometry.

5–59. UFS File System

Minimizing Latency, part 2





stacks of rotationally
equivalent blocks

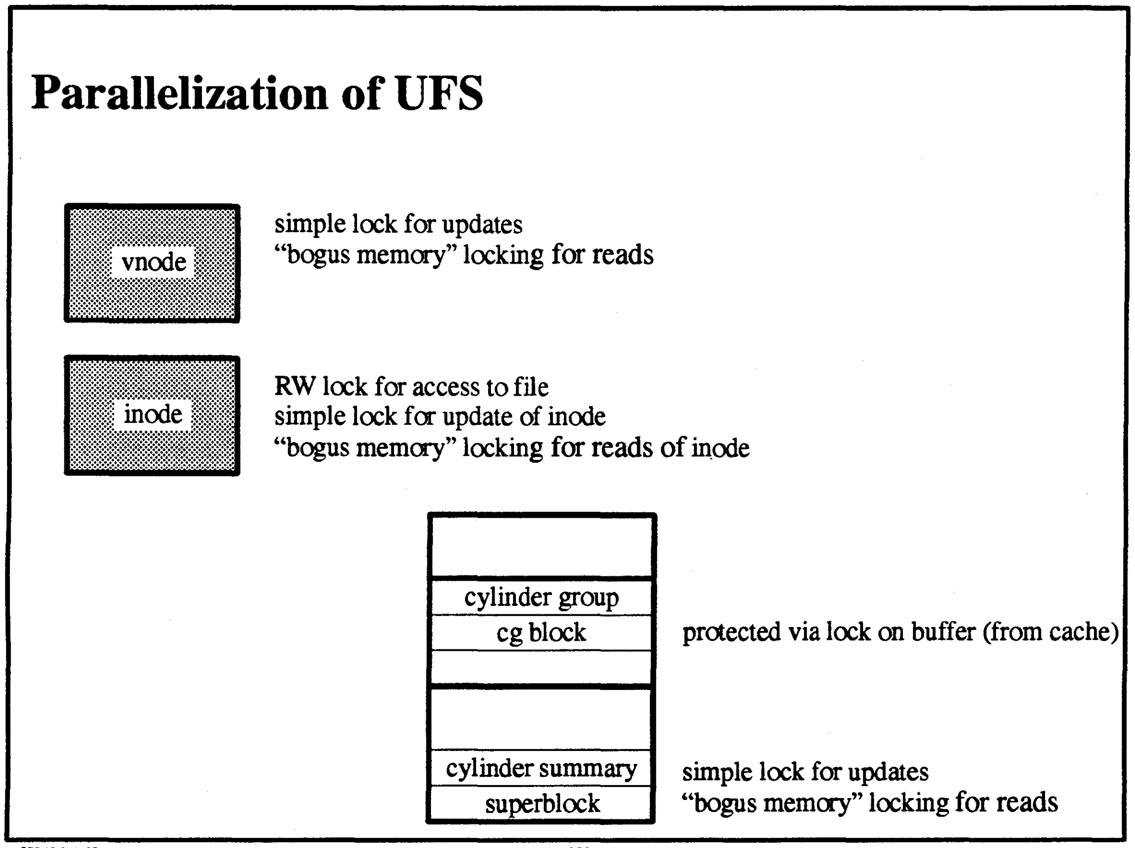
H2596 5-59. 207 © 1991 Hewlett-Packard

Minimizing Latency, part 2

A better technique is not to lay out the blocks on the track consecutively, but to leave enough space between them so that the disk will rotate no further than to the position of the next block during the time between disk requests.

It may be that when a new block is allocated for a file, the optimal position for the next block is already occupied. If so, one may be able to find a block that is just as good. If the disk has multiple surfaces (and multiple heads), then we can make the reasonable assumption that the blocks underneath each head can be accessed equally quickly. Thus the stack of blocks underneath the disk heads at one instant are said to be *rotationally equivalent.* If all of these blocks are occupied, then the next stack of rotationally equivalent blocks in the opposite direction of disk rotation is almost as good as the first. If all of these blocks are taken, then the third stack is almost as good, and so forth all the way around the cylinder. If all of these are taken, then any block within the cylinder group is chosen.

5–60. UFS File System



Parallelization of UFS

vnode — simple lock for updates; "bogus memory" locking for reads

inode — RW lock for access to file; simple lock for update of inode; "bogus memory" locking for reads of inode

cylinder group / cg block — protected via lock on buffer (from cache)

cylinder summary / superblock — simple lock for updates; "bogus memory" locking for reads

.H2596 5-60. 208 © 1991 Hewlett-Packard

Parallelization of UFS

Two sorts of locking are used with UFS, blocking RW locks and simple locks (spin locks):

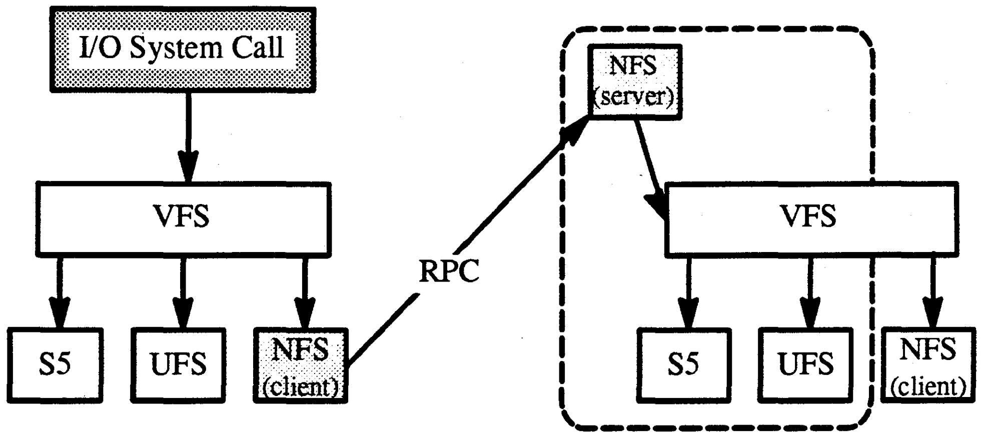
1. *blocking RW locks* (on inodes): used to protect the file across logical operations. I.e., synchronization is supplied at the granularity of the operations described by *uio* structures. As a special case, *cg blocks* reside in the buffer cache and are locked via the blocking lock on the buffer from the cache.

2. *simple locks* (spin locks): used to protect important system data structures (*inodes, vnodes,* and *superblocks*). Modifications to these data structures are always synchronized with simple locks. However, on many architectures, such synchronization is not necessary for reads: if the architecture guarantees that 32-bit, aligned items can be read atomically, then no locking is required. Thus, for example, a thread can read the *mode* bits from the inode and be guaranteed that they make sense.

   Parallel architectures that do not supply such atomicity guarantees are deemed to have *bogus memory*. These cases are dealt with in the source code with the *BM* macro: *BM(lock(x))* expands to *lock(x)* on bogus-memory machines and expands to the null string on other machines. Thus locking is compiled conditionally.

5–61. NFS File System

Network File System (NFS)



.H2596 5-61.   209   © 1991 Hewlett-Packard

Network File System (NFS)

5–62. NFS File System

NFS Highlights

Servers are stateless:

- server crash recovery is trivial
- NFS does not support full UNIX semantics
- NFS is "easily" supported on other operating systems

H2596 5-62. 210

NFS Highlights

Since servers contain no information about their clients, crash recovery is trivial in NFS: there is no information to be recovered after a crash. However, some state information is required for implementing certain UNIX I/O calls, and thus NFS cannot duplicate UNIX semantics exactly.

For example, a common technique for creating a temporary file is for a process to create a file and then to unlink the newly created file. Since the file is open, it continues to exist even though it has a zero link count (the reference count on its vnode is positive). The file is removed only when it is closed.

If this technique is practiced over NFS, the server does not know that the file is open (since this would be state information), and thus removes the file as it is unlinked. Since a number of important applications use this technique for creating a temporary file, the method must be accommodated. The client-side NFS code (executing in the kernel) converts *unlink* requests into *rename* requests, changing the name of the file to a temporary name. When the client application finally closes the file, the close is converted into an unlink and the file is removed.

Another example of the difference between UNIX and NFS semantics arises when an application changes the access permissions of an open file. Access checks for UNIX files are performed only when the file is opened. Thus, if the user successfully opens a file for read-write access and subsequently changes the permissions to read-only, write access to the already open file is still allowed. However, since the NFS server must check access permissions with each access to a file, write access would be denied in this case.

OSF/1 (and other UNIX implementations of NFS) provides only a partial solution to this problem. The NFS server allows the owner of a file read-write-execute access regardless of the permissions associated with the file; the NFS client filters requests to the NFS server on the basis of how the file was opened. Thus if the file was opened successfully for read-write access, then the client side allows read and write calls to be processed. However, if the file was opened as read-only, then the client side denies write requests.

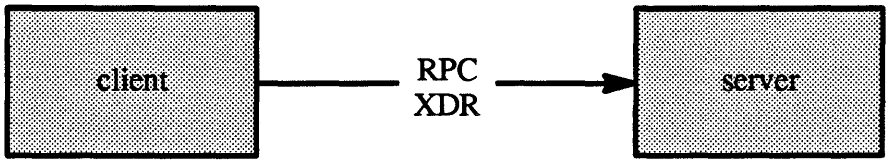
A further difference between UNIX and NFS semantics is caused by the fact that NFS clients cache blocks from files provided by NFS servers. This means that processes on different machines do not necessarily have a consistent view of shared files.

5–63. NFS File System

NFS and RPC

Issues

- reliability
- security



NFS and RPC

The client and server communicate via Sun's RPC protocol. The XDR protocol copes with the heterogeneous environment. The two major issues are reliability and security.

The transport protocol is typically UDP, an unreliable protocol. Thus NFS itself must provide reliability. NFS accomplishes this by taking advantage of the request/response semantics of the client-server interaction.

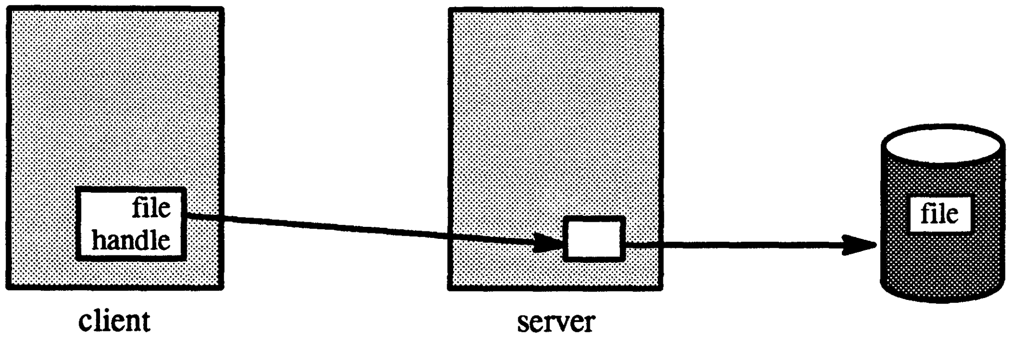
For example, suppose that a client issues a *write* request but receives no response. The client will repeat the request under the assumption that the original request was lost.

However, suppose that it was the response that was lost, and not the request. Now the server receives the *write* request twice. This usually presents no problems, because most NFS requests, such as *write*, are *idempotent*, meaning that the effect of performing the request twice is the same as performing it once. The *write* request is idempotent since it contains the location in the file to which the data is to be written. However, there are additional problems with reliability, as will be seen.

Security has always been a problem in NFS. The model for authentication is essentially "Trust me." Each NFS RPC request contains as part of its header the numeric user id of the caller. Servers refuse requests from the superuser but will trustingly honor any other requests. Sun uses an enhanced authentication technique for RPC involving a combination of DES and public-key encryption. OSF will deal with such problems through its *distributed computing environment* (DCE).

5–64. NFS File System

File Handles



File Handles

When a file is opened, it is identified by its path name. The NFS server verifies that the file exists, checks that the desired access is currently allowed, and returns a *file handle* that the client will use to identify the file on subsequent accesses. Using the file handle for subsequent accesses thus avoids expensive path traversal for each access. This file handle is of an *opaque* data type meant not to be interpreted by the client but only to be passed back to the server.

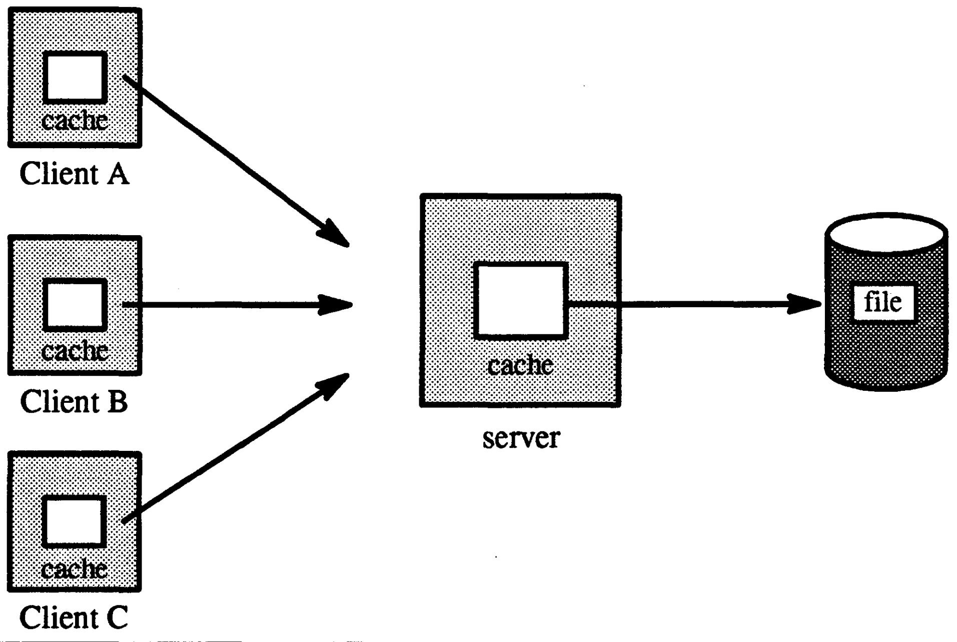
UNIX servers pass back a handle consisting of:

- file system number
- inode number
- inode generation number

The *generation number* copes with the confusion that could arise from the reuse of inodes. One client may open a file, another delete it, and a third might reuse the inode when it creates an entirely new file. When the first client attempts to access the original file, the server must be able to determine that the desired file no longer exists. So, when a client reuses an inode, the inode receives a new generation number to distinguish its current use from past uses. When a client accesses a no-longer-extant file, a "stale file handle" error message is returned to the client. The generation number is stored on disk in the inode.

5–65. NFS File System

Client-Side Caching



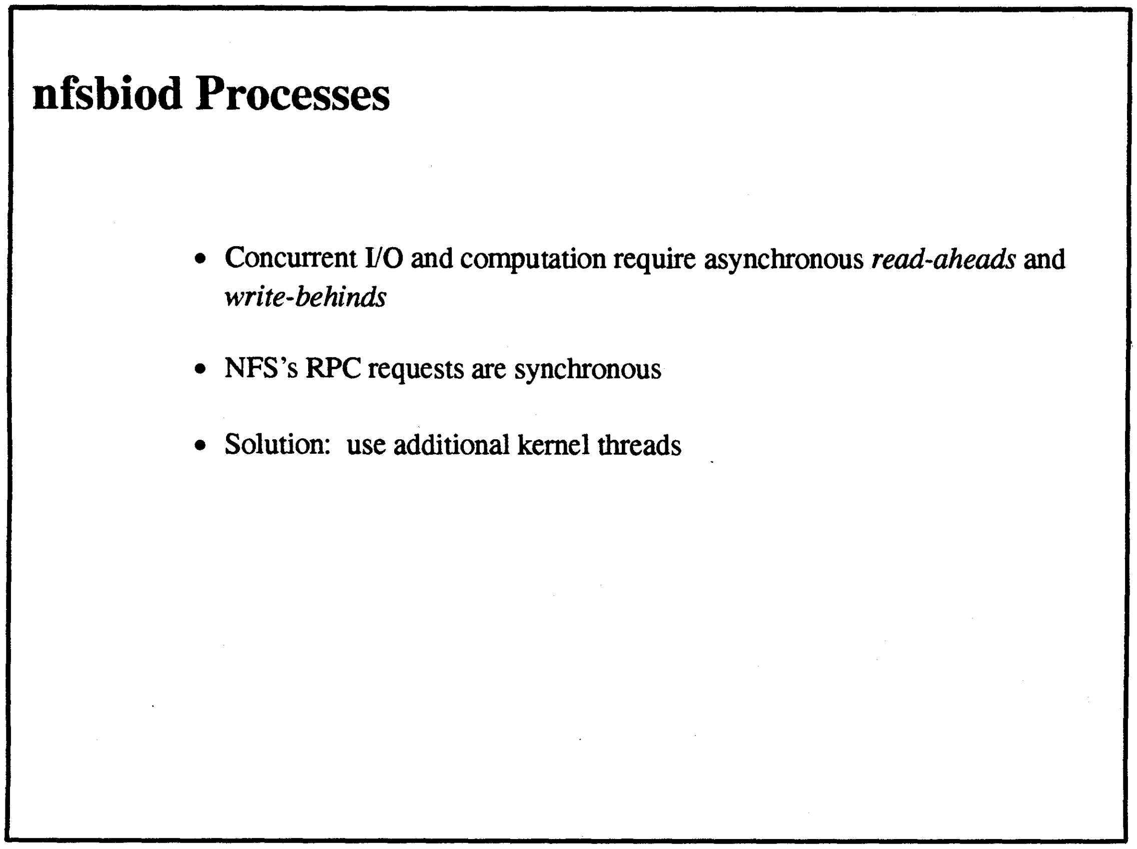
H2596 5-65. 213 © 1991 Hewlett-Packard

Client-Side Caching

Remote disk blocks are cached in the client's buffer cache. If multiple clients use the same file, there may be a consistency problem. While it is considered too expensive to keep the various caches consistent, an attempt is made to keep things from being too inconsistent. In each *nfsnode* is a copy of the associated remote file's attributes (i.e., what is obtained from a stat system call—information such as the file's modification time). Every time the attributes are fetched from the server, an *expiration time* of some number of seconds is set (five seconds in OSF/1). If the file is accessed before the attributes expire, then it is assumed that any locally cached blocks of the file are valid. If the attributes have expired, the new attributes must be obtained from the server and, if the file has been modified, then the locally cached blocks are flushed. (Modified cached blocks are written to the server.)

The cache is cleaned in response to close, sync, and fsync system calls (fsync is performed synchronously over NFS).

5–66. NFS File System



nfsbiod Processes

- Concurrent I/O and computation require asynchronous *read-aheads* and *write-behinds*
- NFS's RPC requests are synchronous
- Solution: use additional kernel threads

.H2596 5-66. 214 © 1991 Hewlett-Packard

nfsbiod Processes

When a process accesses files through the buffer cache, concurrency between I/O and computation is achieved by exploiting *read-aheads* and *write-behinds.* This is easy to do for I/O for local files, because the interface to the device driver is asynchronous. For example, when reading a file sequentially, one can start I/O *read* requests for the current block and the next block without waiting for the request to the latter. When writing a block, the user process merely modifies the buffer cache and the file itself is modified later, asynchronously.

The interface between the client and the NFS server is synchronous, since RPC requests are inherently synchronous. To achieve the desired concurrency, separate threads are used on the client to perform many NFS client RPC calls. These threads are pre-created and are known as *nfsbiod* processes (these are user processes that have executed the async_daemon system call). Whenever an asynchronous I/O request is desired, the client checks to see if an *nfsbiod* process is available. If so, then the request is given to it to perform in its own context. Otherwise, the caller performs the request in the caller's context (and blocks until the request is completed).

5–67. NFS File System

nfsd Processes

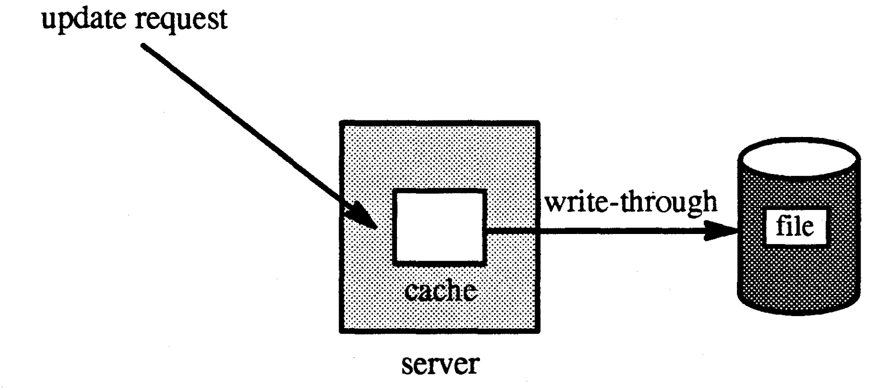
- Kernel processes that handle NFS requests on the server
- Must deal with "authentication" and access checking

nfsd Processes

Each server has a number of *nfsd* processes that handle the incoming RPC requests for NFS. (These are user processes that have executed the nfssvc system call.) Unlike the *nfsbiod* processes, the *nfsd* processes are essential. NFS requests are handled only in their context on the server. When such a process receives a request, it acts on behalf of the caller and temporarily assumes its identity. This is accomplished through the use of *credentials structures*, which contain *groupids* and a *userid* and are passed to the access-checking routines.

5–68. NFS File System

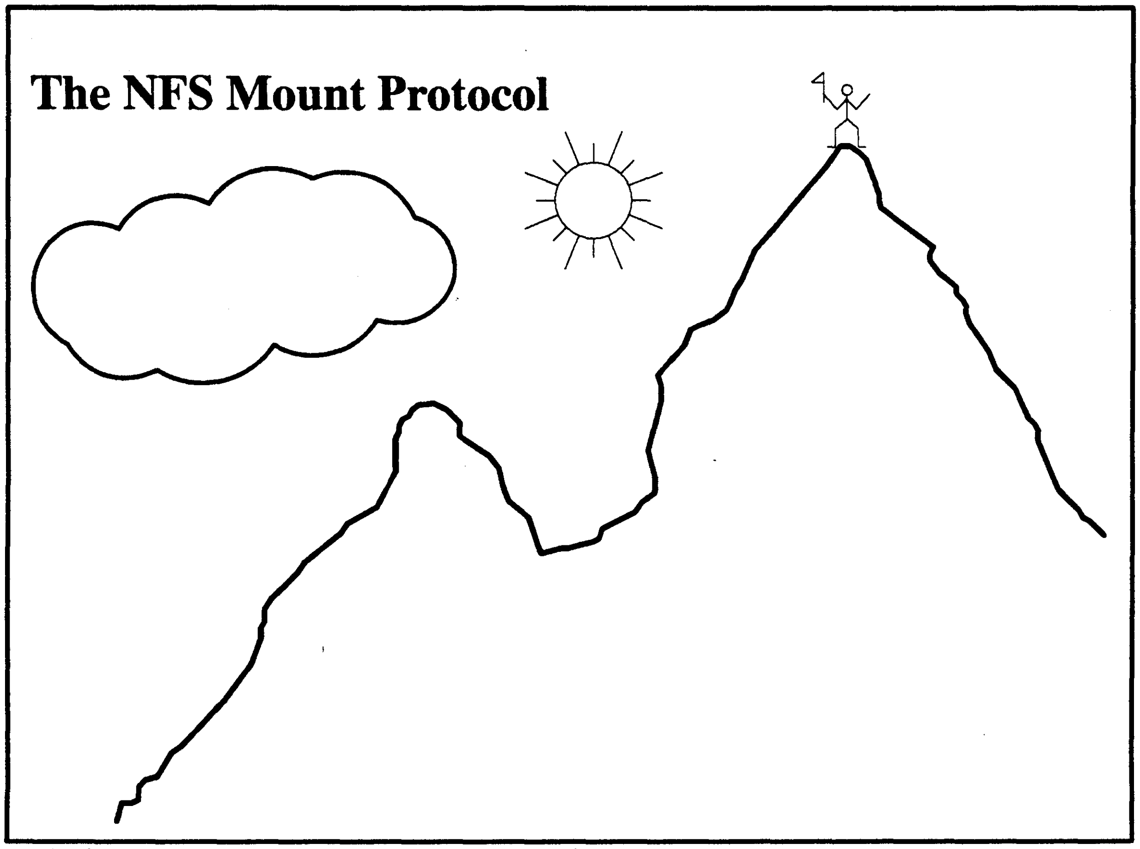
Server's Buffer Cache



Server's Buffer Cache

The server's buffer cache is used for handling client requests, but it is treated as a *write-through cache*: when an *nfsd* process handles a *write* request, not only is the cache modified but also the data is written to the disk immediately and the RPC call does not return until the disk-write completes. This technique is consistent with the idea that NFS servers are stateless: data that is in the cache but not on disk is state information that the client would not want the server to lose if the server were to crash. The client is assured that, when an NFS RPC request returns, any requested changes to a file have been reflected on disk.

5–69. NFS File System



The NFS Mount Protocol

H2596 5-69. 217 © 1991 Hewlett-Packard

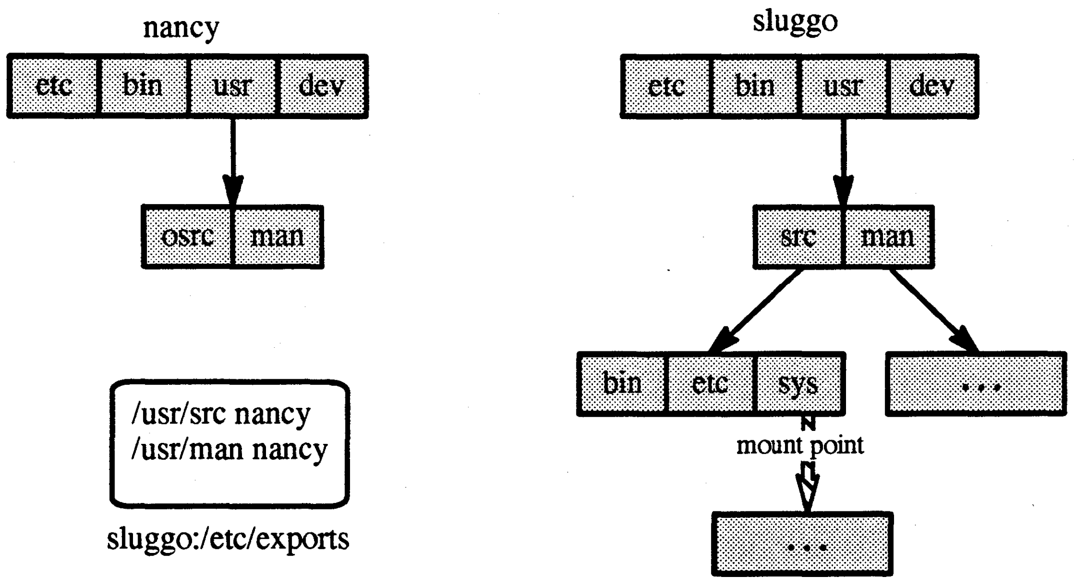
The NFS Mount Protocol

Like local file systems, in UNIX a remote file system must be *mounted* in the client's directory hierarchy in order to be used.

In OSF/1, the mount shell command makes an RPC request to the server's *mountd* process to obtain a file handle for the mount point. The *mountd* process is a user process that implements the server side of the mount protocol (the mount shell command implements the client side). Each server maintains in the */etc/exports* file a list of exported file systems and the clients to which they are exported. The *mountd* process first makes certain that the client is allowed to mount the requested file system, then returns to it the file handle for the root of the file system. The mount shell command then issues a mount system call, passing to the kernel the file handle and the path name of the mount point.

5–70. NFS File System

Remote Mounting, part 1



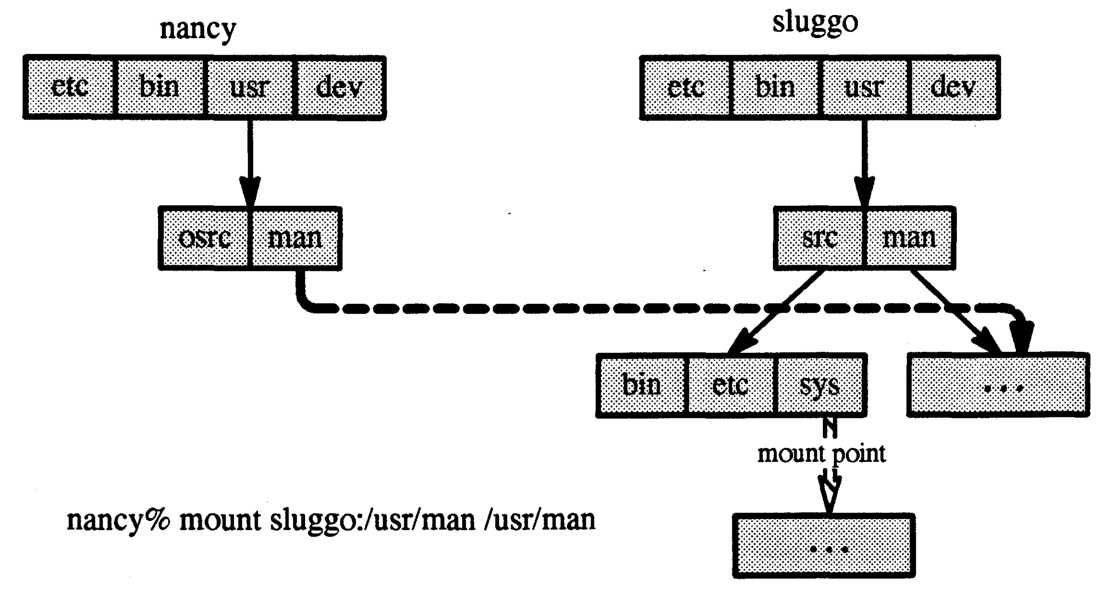
.H2596 5-70. 218 © 1991 Hewlett-Packard

Remote Mounting, part 1

In this picture we have two machines, nancy and sluggo. Sluggo exports two file systems to nancy, identified as */usr/src* and */usr/man.*

5–71. NFS File System

Remote Mounting, part 2



```
nancy% mount sluggo:/usr/man /usr/man
```

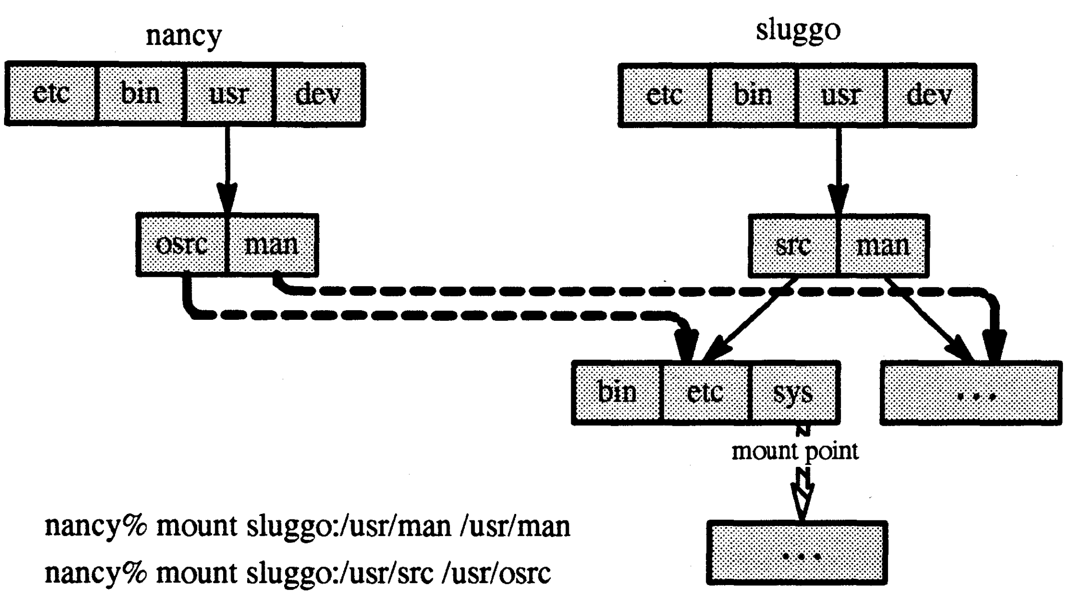
.H2596 5-71. 219 © 1991 Hewlett-Packard

Remote Mounting, part 2

Nancy mounts sluggo's */usr/man* on its own */usr/man* directory.

5–72. NFS File System

Remote Mounting, part 3



```
nancy% mount sluggo:/usr/man /usr/man
nancy% mount sluggo:/usr/src /usr/osrc
```

H2596 5-72. 220 © 1991 Hewlett-Packard

Remote Mounting, part 3

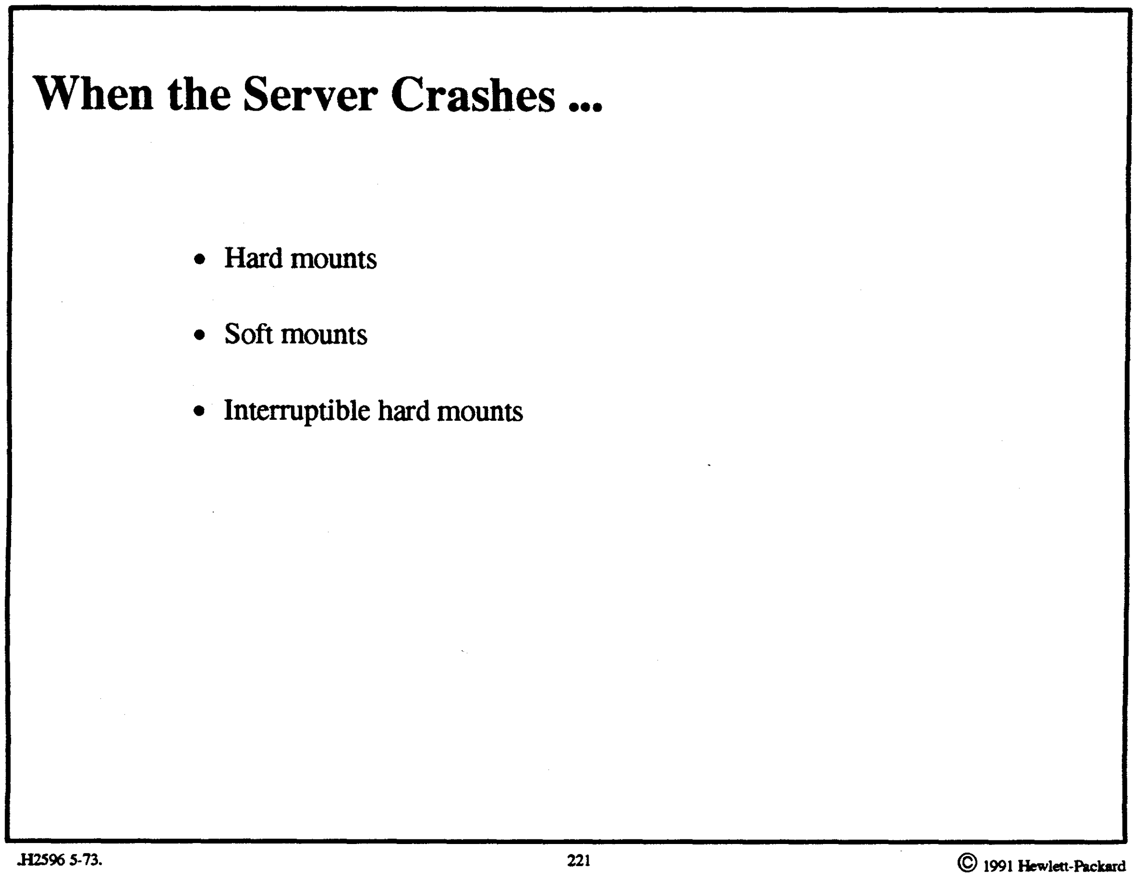
Nancy now mounts sluggo's */usr/src* on nancy's */usr/osrc.* On sluggo, */usr/src/sys* is a mount point: another file system, X, has been mounted here and, from sluggo's point of view, the root directory of this file system is superimposed on top of the directory */usr/src/sys* (thus the original contents of this directory are invisible to sluggo). However, nancy does not see this mount point.

The directory (on nancy) */usr/osrc/sys* is not mounted upon. Unlike sluggo, nancy sees the actual contents of this directory. If it is desired that this mount point exist in nancy's view as it does in sluggo's, then nancy could explicitly mount file system X on top of the directory */usr/osrc/sys.*

The reasons for not having a client use the server's mount points are partly for security, but mainly for simplicity. Suppose on server B, file system Y (from server C) is mounted on a directory within file system X and file system Z (on the same server as X) is also mounted within file system X. If client A mounts X (and thus appears in B's /etc/exports list), what would be required for it to be able to follow the mount point to Y on server C? A must appear in C's /etc/exports list. But C has only verified that B is there. If B were to pass on A's requests to C, it would have to ensure that C approves of A. Rather than do this complicated checking, the convention is that A must mount Y itself.

Note that it wouldn't be very difficult for B to allow A to follow the mount point to Z, but, again, for simplicity, this is not done.

5–73. NFS File System



When the Server Crashes ...

- Hard mounts
- Soft mounts
- Interruptible hard mounts

H2596 5-73. 221 © 1991 Hewlett-Packard

When the Server Crashes ...

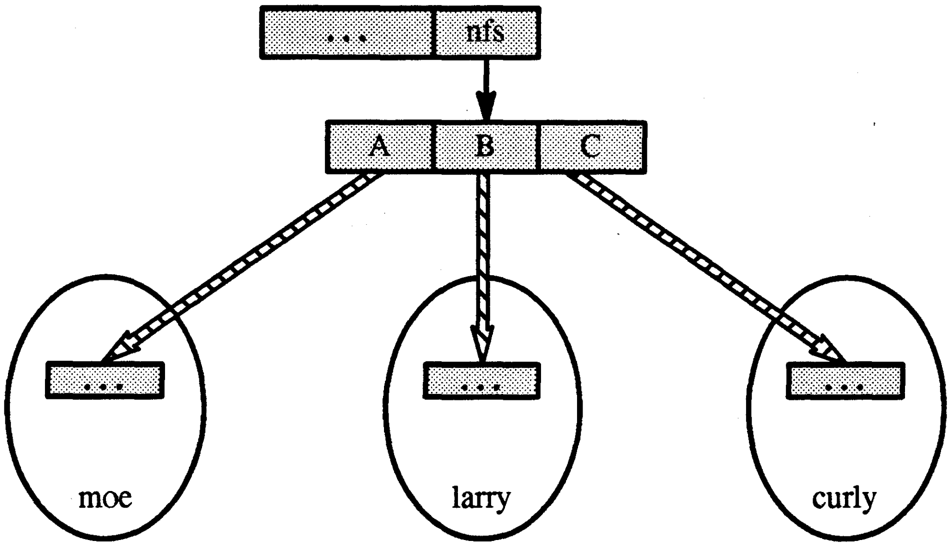
The client's response to server crashes depends upon an option specified when the remote file system was mounted. If the client specified a *hard mount*, then any system call involving a file on the remote machine blocks until the machine comes back up (whether this takes seconds or weeks). Such system calls block *uninterruptibly*, so there is no way to abort the process making the system call. This can be very annoying.

Another option is the *soft mount*. Any system calls involving files on the dead remote machine will return (eventually) with the error code ETIMEDOUT. This option might seem a good idea, but there are difficulties. A number of UNIX applications pay no attention to error returns on I/O system calls (if the open succeeded, there could not possibly be any problems with reads and writes...). Thus damage may be done because the client is unaware of the crash.

A more reasonable way of mounting the remote file system is the *interruptible hard mount*. With this option, as before, system calls involving a file on the remote machine block until the machine comes back up, but the wait is *interruptible* (i.e., by signals). However, the interrupt is not immediate: the underlying RPC layer performs many retries before checking to see if a signal is pending.

5–74. NFS File System

More on Server Crashes, part 1



More on Server Crashes, part 1

Here moe, larry, and curly are the names of NFS servers. Each contains a file system that has been mounted, respectively, in */nfs/A, /nfs/B,* and */nfs/C* (i.e., the client has set up its directory hierarchy so that all NFS mounts are in one directory). Suppose one's current directory is in the root directory of curly's file system and one executes the *pwd* command. The result should be */nfs/C.* How does the *pwd* command work? It determines the inode of the current directory ("."), and then searches the parent directory ("..") until it finds the component name associated with the matching inode number. It then repeats this procedure backwards along the path until it reaches the root directory.

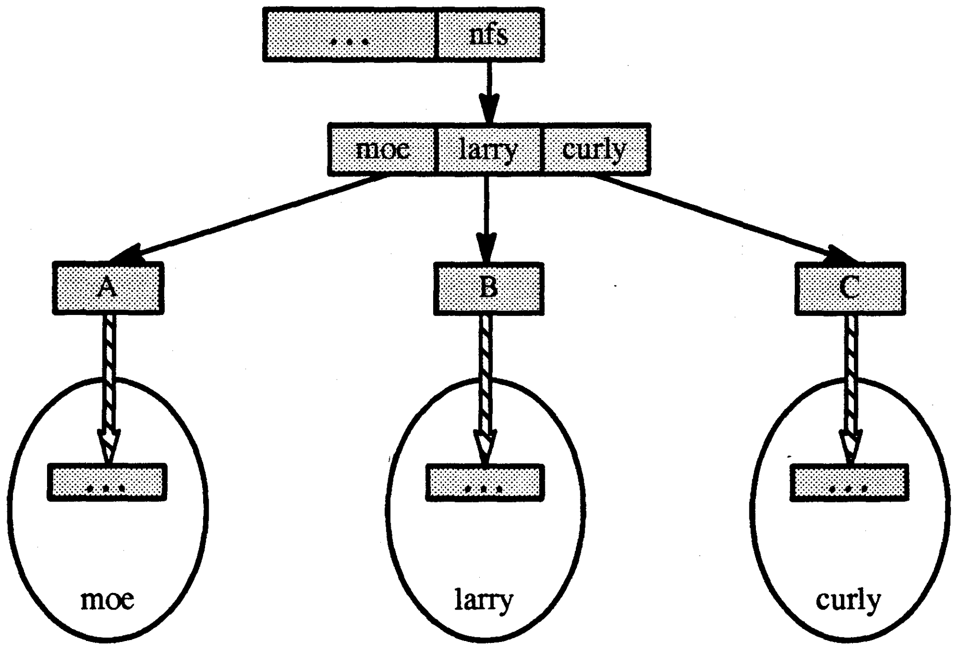
However, when a mount point is encountered, the parent directory of the mount point does not contain the inode number of the root directory of the mounted file system. Instead, the *pwd* command must issue the stat system call for each entry of the parent directory until it finds the entry that refers to the mounted file system.

Back to our example. Suppose that NFS server moe is down. When the *pwd* command is executed starting with curly's root directory, it will be necessary to stat each of the entries in the */nfs* directory to determine which of them refers to curly. But, since moe is down, the stat call will hang when it is applied to */nfs/A*. Thus it will be impossible to complete the *pwd* command until machine moe comes back up, even though there is no logical connection between the path */nfs/C* and the machine moe.

This is especially annoying because both *csh* and *ksh* perform a *pwd* when starting up.

5–75. NFS File System

More on Server Crashes, part 2



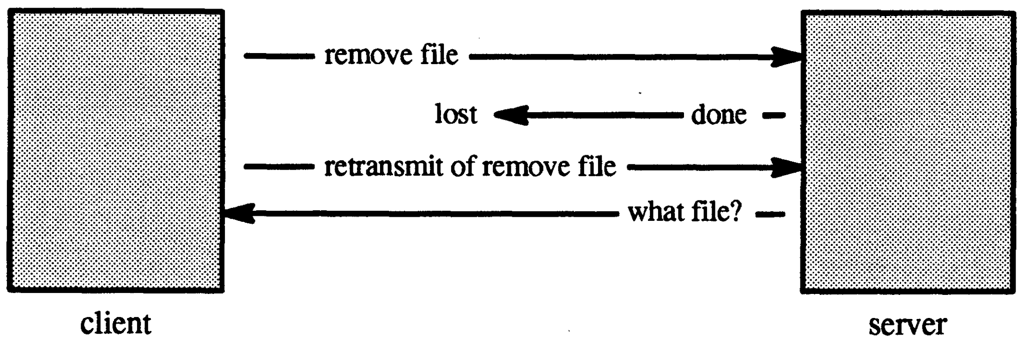
.H2596 5-75. 223 © 1991 Hewlett-Packard

More on Server Crashes, part 2

This picture illustrates a safer NFS mount technique. An extra level of directories has been added so as to avoid the problems with *pwd.*

5–76. NFS File System

The Problem of (Non)Idempotency



.H2596 5-76. 224 © 1991 Hewlett-Packard

The Problem of (Non)Idempotency

As previously mentioned, NFS is typically implemented on top of an unreliable protocol and thus must implement reliability guarantees itself. To accomplish this, it exploits the request/response nature of its interaction: if a client receives no response to its request, it assumes that the request was lost and repeats it. However, difficulties can occur if it was the response that was lost, not the request.

This situation should be no problem as long as the requests are *idempotent*, as was discussed on page 5-78. Certain requests, however, are known to be *nonidempotent*. For example, suppose that a *remove file* request is repeated because the first response was lost. The response to the second request indicates an error because the file no longer exists. But, other than the error, the desired effect has been achieved—the file has been removed, though the programmer may end up somewhat confused.

With some cooperation by the server, this sort of nonidempotency, known as *nondestructive nonidempotency*, can be made transparent. In the original reference port for NFS, the server maintains a queue of completed nonidempotent requests and their responses. If a nonidempotent request fails, the server checks this queue to see if this is a repeat of an earlier request (the RPC headers contain a *transmission id (xid)* to facilitate this duplicate detection). If it is, then the server repeats the previous response.

However, as the next slide shows, there are other, more subtle cases that are not dealt with.

5–77. NFS File System

A Problem Case*

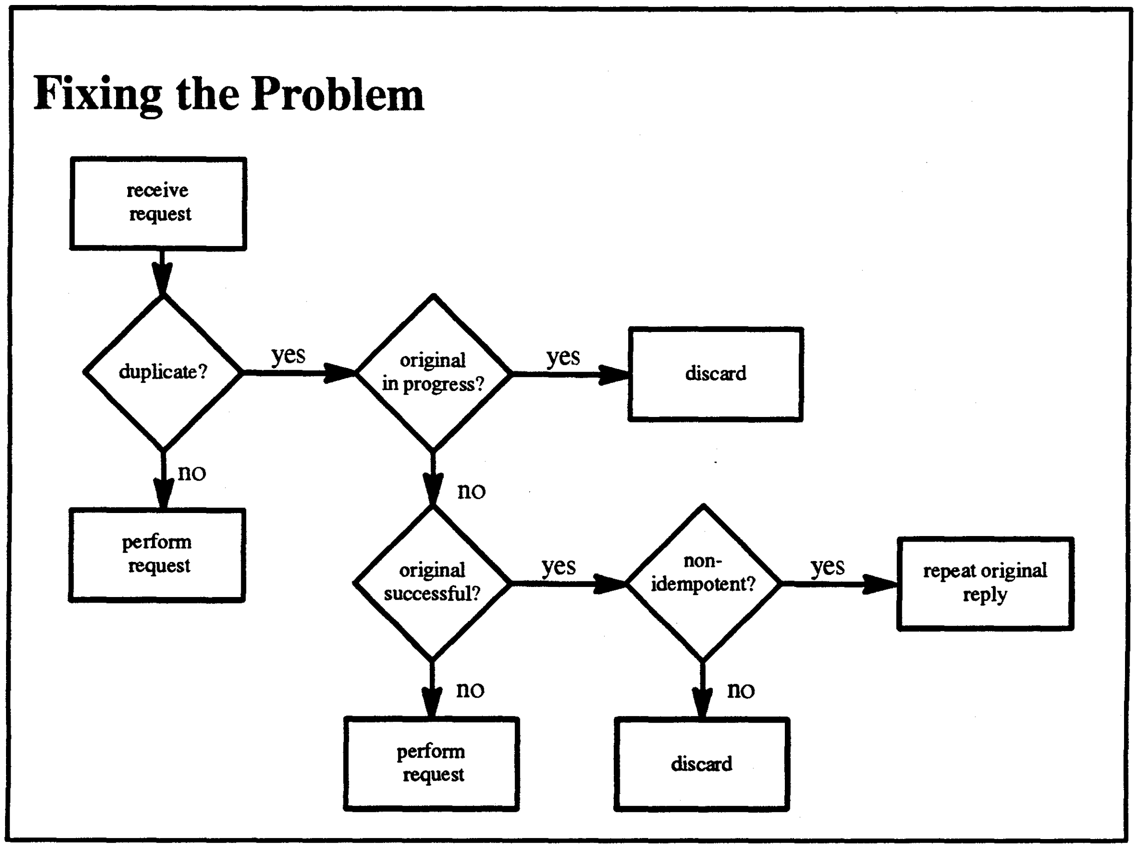
Time	Client Activity	Server Activity
t0	process starts	idle
t1	transmit *creat* request (C0)	idle
t2	wait for *creat* response	receive C0; schedule *nfsd1*
t3	retransmit *creat* request (C1)	*nfsd1*: complete C0, truncate file, send *creat* response
t4	receive *creat* response; process resumes	receive C1; schedule *nfsd1*
t5	transmit *write* request (W0)	*nfsd1*: starts but blocks on a system resource
t6	wait for *write* response	receives W0, schedules *nfsd2*
t7	wait for *write* response	*nfsd2*: complete W0, send *write* response
t8	receive *write* response; process completes	*nfsd1*: complete C1, truncate file, send *creat* response
t9	receive *creat* response—discard it	idle

* from "Improving the Performance and Correctness of an NFS Server," by Chet Juszczak, *Conference Proceedings of 1989 Winter USENIX Technical Conference.* Used with permission.

A Problem Case

A side effect of a *creat* request is to truncate the file to zero length if it already exists. In this example, the intention was to truncate the file and then write to it, but the result was the opposite: the file was written to, then truncated. The problem is that, though the *write* request and the *creat* request are by themselves idempotent, more complicated interactions have occurred. That is, idempotency itself is not sufficient.

5–78. NFS File System



Fixing the Problem

.H2596 5-78. 226 © 1991 Hewlett-Packard

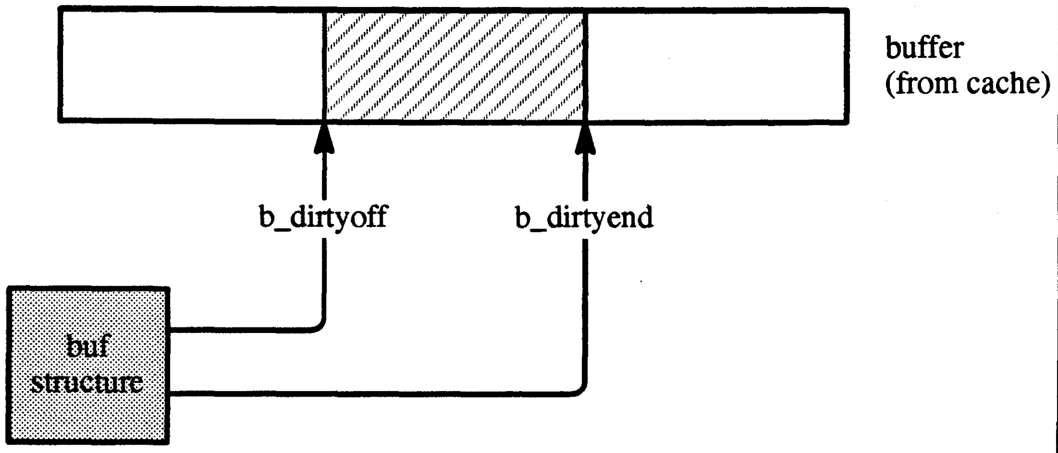
Fixing the Problem

OSF/1 solves this problem by using the technique described in the paper referenced on the previous slide. The NFS server maintains a cache of active and completed requests. Items stay in this cache for a finite period (2 seconds). When the server receives a request, it immediately checks if it is a duplicate of a request still in the cache. If it is, and if the original is still in progress, then the duplicate is discarded, i.e. the client timed out prematurely. If the original completed successfully, the duplicate is again discarded. (Here we are assuming that the response was not lost but that the client again timed out prematurely—from observation, this is the usual case.)

If the response was indeed lost, the client will continue to retry the request; eventually the original request will have been removed from the cache, so that a retry will not be recognized as such and will actually be retried. The problem outlined in the previous slide will not occur, since the client does not move on to its next request until it finally gets a response from its current request. If the original failed, the server retries the duplicate (there is no particular rationale for retrying the duplicate other than that this is the behavior of the original implementation of NFS).

5–79. NFS File System

Optimizing NFS Writes in OSF/1



.H2596 5-79. 227 © 1991 Hewlett-Packard

Optimizing NFS Writes in OSF/1

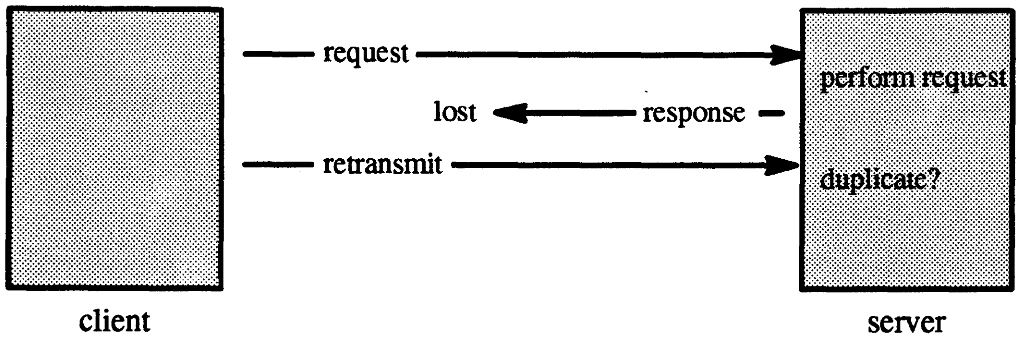
Normally (i.e., when using a local file system), when one writes to a file a span of data that does not fill an entire buffer from the cache, the block I/O subsystem first reads a whole block, then modifies the desired portion of the block. To eliminate the need for these (expensive) reads when writing to an NFS file, the block I/O subsystem keeps track of what portion of a buffer has been modified (using two new fields in the buf structure: *b_dirtyoff* and *b_dirtyend*). Thus when the buffer is "cleaned," just the modified portion is written to the server.

This presents a problem if the entire buffer is read by an application before the modified portion is sent to the server: consider a situation in which bytes 2048 through 8191 of a file are modified on the client, and no blocks of the file currently reside in the client's cache. An 8K buffer is allocated on the client, but only locations 2048 through 8191 are written. At this point, a thread on the client attempts to read the entire 8K portion of the file. Rather than complicate the client-side code so that it will recognize that it must first fetch bytes 0 through 2047 from the server, the client, whenever it reads from an NFS file that it has recently written to, first cleans its buffer cache of blocks from this file (by sending dirty blocks to the server). Then it checks the attributes of the file with the server and fetches the block from the server if necessary.

For further discussion about this implementation of NFS, see Macklem, 1991.

5–80. NFS File System

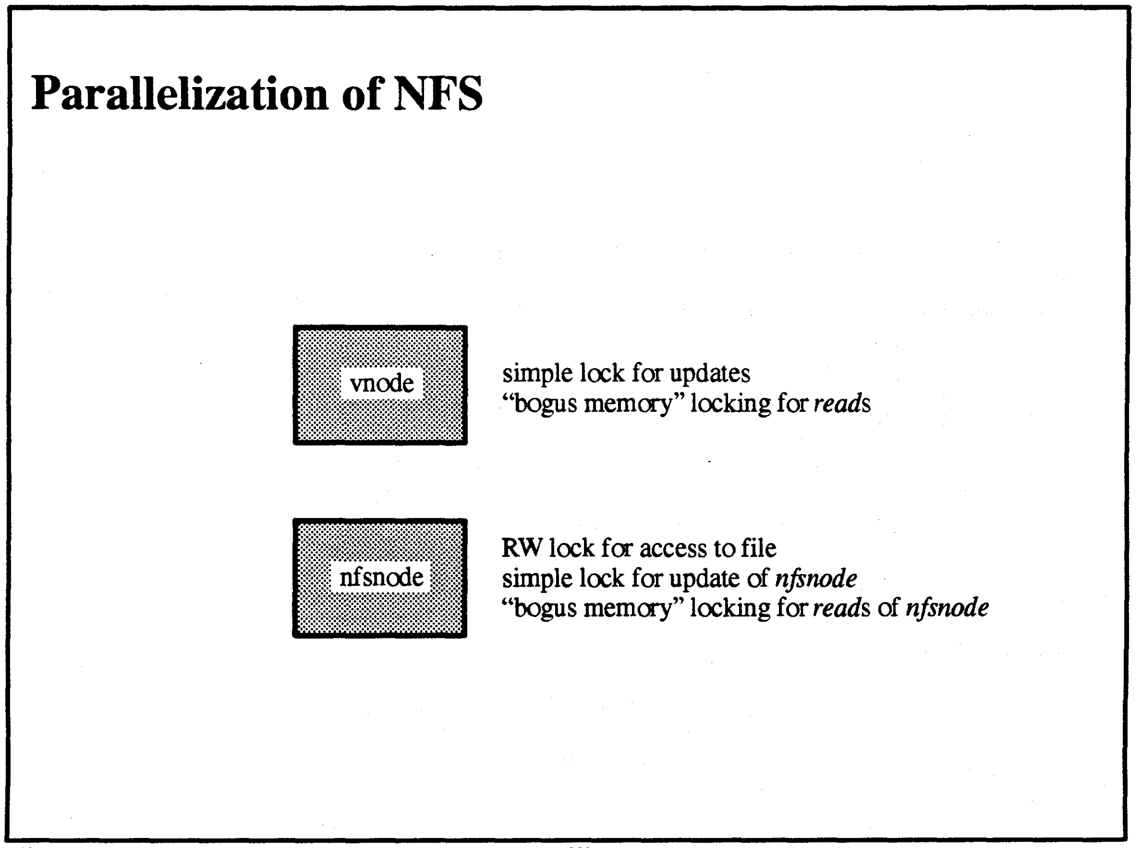
Duplicate Detection



Duplicate Detection

As mentioned previously, duplicate detection relies on an *xid* supplied by the RPC level. However, retransmissions are performed by a pair of nested loops. In the inner loop, retransmissions are done by the RPC layer, which does not modify the value of the *xid*. However, after this loop is performed a finite number of times, control passes to the outer loop, which is performed by the NFS layer (client side). In the original reference port, at each iteration of this loop the *xid* would change. OSF/1 (which does not have separate NFS and RPC layers, but combines them into a single layer) fixes this by ensuring that the *xid* never changes during retransmissions.

5–81. NFS File System



Parallelization of NFS

vnode

simple lock for updates
"bogus memory" locking for *reads*

nfsnode

RW lock for access to file
simple lock for update of *nfsnode*
"bogus memory" locking for *reads* of *nfsnode*

.H2596 5-81. 229

Parallelization of NFS

At the higher levels, the parallelization is very similar to that of UFS: there are blocking RW locks on *nfsnodes* to protect access to files at the level of operation described by *uio* structures. Simple locks are used to synchronize updates to *vnodes* and *nfsnodes*. *Reads* of these data structures need only be protected on architectures with "bogus memory."

Simple locks are used at lower levels for synchronization of NFS data structures. For example, the client side maintains a queue of NFS RPC requests (which are waiting for responses). The server side maintains a table of active and completed requests, accessed via a hash table.

Module 5 — File Systems

Exercises:

1. Explain the use of the reference count in the system file table entries.

2. a. What are the roles of the *vfsops* and *vnodeops* data structures in the abstraction of the file system concept?

   b. Some versions of UNIX maintain a "mount table" in the kernel, representing in tabular form which file systems are mounted where. How is this sort of information represented in OSF/1?

   c. Why is it necessary for a thread to hold a lock over the entire period during which it is using the offset field of the system file table entry?

3. a. How does the buffer cache facilitate concurrent I/O and computation?

   b. Why are blocks in the buffer cache identified by vnode and block number?

   c. What happens when two threads simultaneously access a file block that is not currently in the cache?

4. a. How are directories protected from concurrent conflicting updates?

   b. Explain the concept and use of *capabilities* in the directory lookup cache.

5. a. What aspects of the operating system limit the maximum possible size of the file?

   b. What are the performance problems inherent in the standard S5 file system?

6. a. How is free space represented in a UFS directory?

   b. List the techniques used in the UFS file system to improve performance.

   c. What are the two different policies for allocating fragments for a file? Under what circumstances is each policy used?

   d. Suppose that two threads are extending the size of two different files within the same file system. What data structures need to be protected from concurrent access? What types of locks are employed for this protection? Under what circumstances can the two threads proceed without one having to wait for the other?

7. a. List three differences between NFS semantics and UNIX semantics.

   b. Explain how generation numbers are used.

   c. What is the function of *nfsbiod* processes?

   d. Why is the server's buffer cache accessed in a synchronous write-through fashion?

   e. What are the differences between hard mounts, soft mounts, and interruptible hard mounts?

f. Why is it necessary for the server to maintain a queue of recent NFS requests?

Advanced Questions:

8. A thread executing an I/O system call involving an ordinary file must obtain blocking locks on the *file table entry* and on *buffers* in the *buffer cache.* On a multiprocessor, a simple lock is needed for operations on the *vnode.* Why are all of these locks necessary? Could fewer be used?

9. On page 5-42, we discuss the use of timestamps to avoid a race condition. Why isn't this race condition dealt with by combining the lookup and delete operations into a single operation?

Module 6 — Device Drivers and Terminal I/O

Module Contents

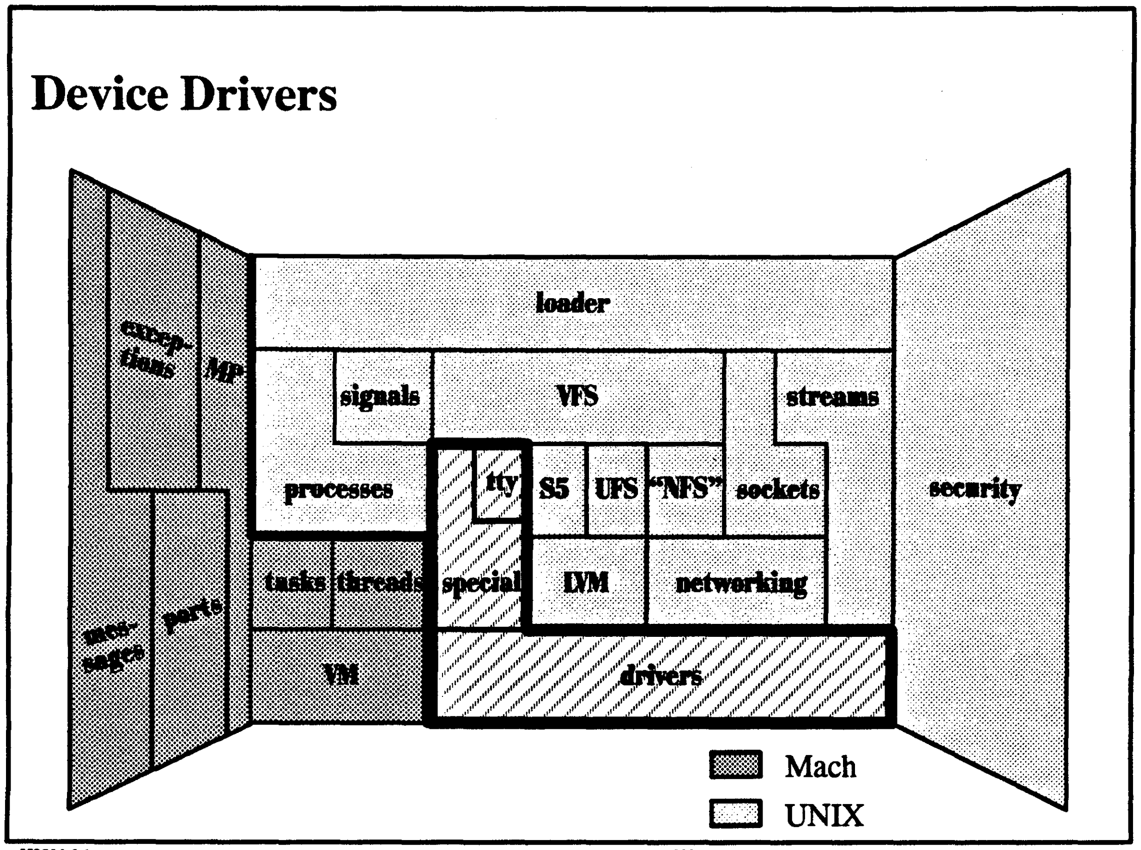
Module Objectives

In order to demonstrate an understanding of device drivers and terminal I/O, the student should be able to:

- explain the problem of aliasing in special files and how it is dealt with in OSF/1
- list the steps necessary to dynamically add a module to the operating system
- list what has been done in the OSF/1 kernel to support internationalization
- list the data structures supporting terminals and sessions

Module 6 — Device Drivers and Terminal I/O

6–1. The Big Picture



.H2596 6-1. 230 © 1991 Hewlett-Packard

Device Drivers

OSF/1 does not provide any device drivers itself, since they are necessarily extremely machine-dependent. Device drivers are, however, supplied with the reference ports, and they can be used as a guide for constructing one's own device drivers. OSF has added the dynamic configurability and loading of device drivers. In particular, one of the reference ports is for a symmetric multiprocessor, and its device drivers provide an example of how other device drivers may be parallelized.

This material is discussed in chapters 17 and 18 of Open Software Foundation, 1990a.

6–2. Special Files

Devices

- Accessed via *special files*
  - — block interface — Vectored throug buffer Cache
  - — character interface — NOT vectored through buffer cache
- Identified by *device number*
  - — *major portion* identifies driver — offset into driver array
  - — *minor portion* interpreted by driver — passed into driver

	HP-UX	OSF
major	8	8
minor	24	8

} # bits used for

.H2596 6-2. 231 © 1991 Hewlett-Packard

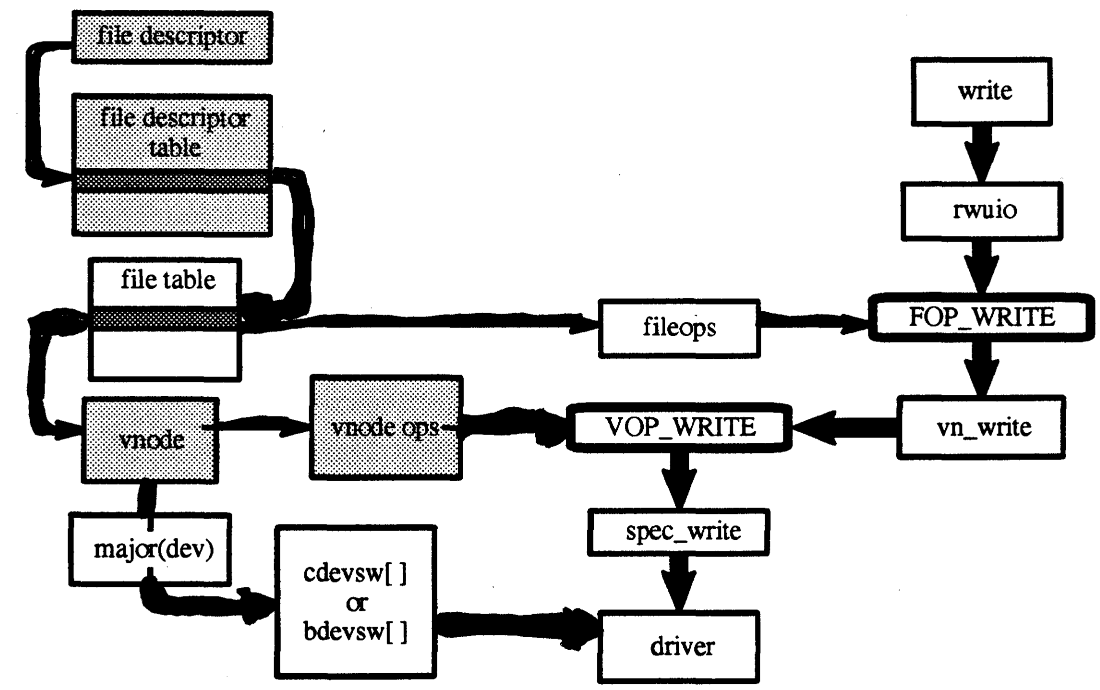
Devices

Devices are treated as a special form of a file in that they are named by paths in the directory hierarchy. A device may be accessed via two different interfaces: the *block* interface, meaning that all access is through the buffer cache, and the *character* interface, meaning that the buffer cache is not used.

A device is identified by a device number that has two parts: the major portion, identifying the driver, and a minor portion to be interpreted by the driver but usually identifying the device, among other things.

6–3. Special Files

Device I/O: Flow of Control



H2596 6-3. 232 © 1991 Hewlett-Packard

Device I/O: Flow of Control

Special files are represented by inodes in both the S5 and UFS file systems. However, the vnode set up for the inode refers to the vnode operations for special files. Thus, for example, a write system call results in a call to *spec_write*.

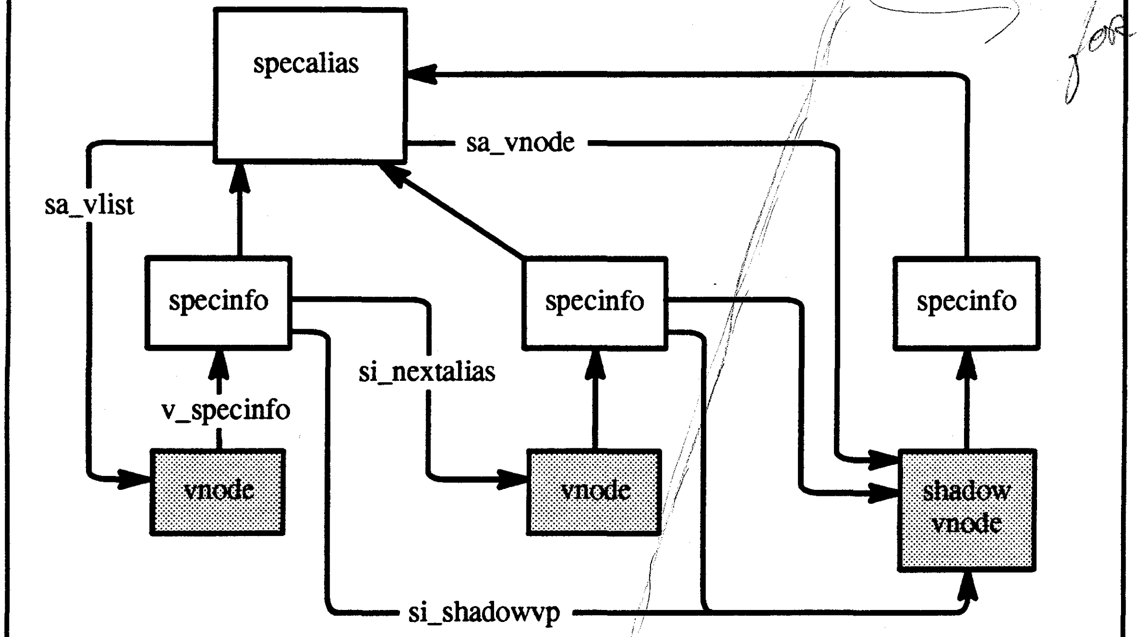
These special file vnode operations must identify the device driver that controls the device. They do so by using the major portion of the device number as an index into the *cdevsw* table for the character interface and the *bdevsw* table for the block interface. (However, for the block interface, the driver is actually called from the block I/O routines.) Each entry in the *cdevsw* and *bdevsw* tables is a structure containing entry points of the associated driver.

6–4. Special Files

For Character & Block Device files

Additional for Block Device files

Aliases and Shadows



.H2596 6-4. 233 © 1991 Hewlett-Packard

Aliases and Shadows

A difficulty with devices is that one device might have multiple names (*special files*). This arrangement could lead to problems: for example, when a device driver's *close* routine is called, the driver must be assured that this is the last *close* of the device, regardless of the name used to *open* it. For accesses to a device via its block interface, we must ensure that, no matter which name of the device is used, all accesses use the same buffers in the cache.

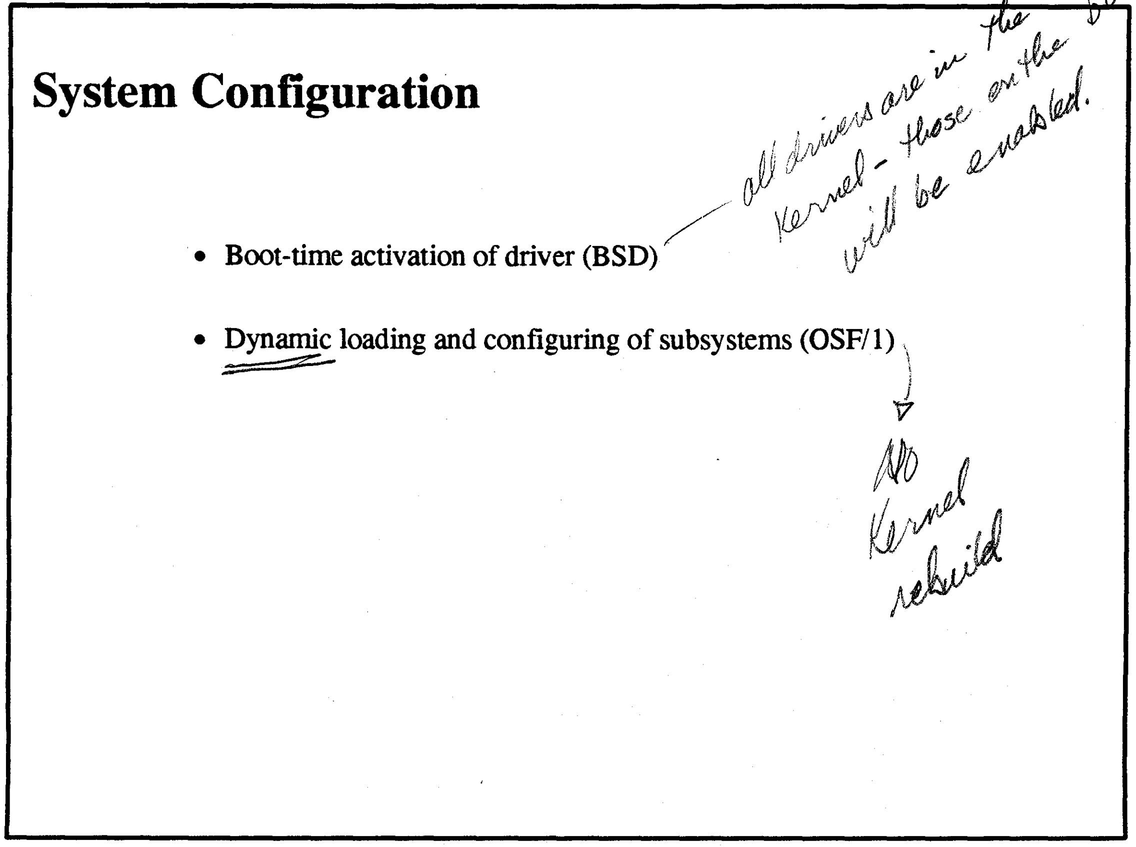
All vnodes (representing open files) of the same device are linked on a chain headed by a *specalias* structure. The actual links are contained in *specinfo* structures, which are allocated along with the vnodes when the underlying inodes are brought into primary memory.

Another problem occurs when one block device has multiple names. Since blocks in the buffer cache are identified by the pair of vnode address (of the block device) and block number, there would be multiple names for each block of the device, one for each of the device's path names. Thus, depending upon which special file is used to access a particular block, the block would be identified differently in the cache. This problem is avoided though the use of *shadow vnodes.*

If a device is opened via its block interface, then the system allocates a shadow vnode. If the same device is subsequently opened via its block interface but with a different name, then the same shadow vnode is used. This vnode is used to refer to the device in all accesses to the buffer cache, thus ensuring consistency.

A related problem might occur when a mounted file system is accessed via its block special interface. In this situation, a single block might have two identities in the cache; it is a block within an ordinary file and it is a block within the block device. This problem is dealt with by prohibiting access to a mounted file system via its block device interface.

6–5. Dynamic Configuration



System Configuration

- Boot-time activation of driver (BSD) — all drivers are in the kernel - those on the backplane will be enabled.
- Dynamic loading and configuring of subsystems (OSF/1) → No Kernel rebuild

.H2596 6-5. 234 © 1991 Hewlett-Packard

System Configuration

In BSD-style autoconfiguration, device drivers are statically linked into the kernel. At boot time, autoconfiguration code determines which devices are present and "activates" the appropriate drivers.

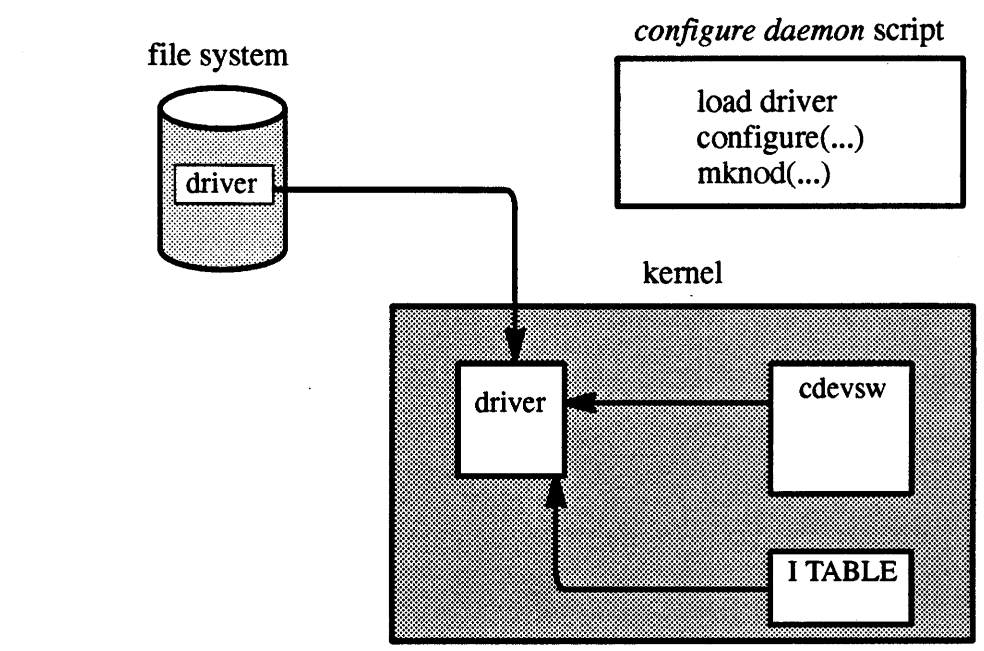
OSF/1 supports *dynamic* configuration of:

- device drivers
- file systems
- streams modules and drivers
- network protocols

Drivers and other modules may be loaded into or unloaded from a running system.

6–6. Dynamic Configuration

Dynamically Adding a Driver



H2596 6-6. 235 © 1991 Hewlett-Packard

Dynamically Adding a Driver

A driver is loaded into the operating system with the aid of the run-time loader, as will be discussed in Module 10. The run-time loader links the driver to the rest of the operating system, but the loaded driver is responsible for linking the rest of the operating system to itself.

Each dynamically configurable driver has a *configure* entry point that is called after it has been loaded. The *bottom half* of the driver, i.e. that portion of the driver that responds to interrupts, must link its interrupt handler into the rest of the kernel. It accomplishes this linkage by calling a pair of routines:

- *handler_add* registers a new interrupt handler
- *handler_enable* "turns on" a registered interrupt handler

The *top half* of the driver, i.e. that part of the driver called in the thread context in response to system calls, must make itself known to the rest of the kernel. It does this by creating entries in one or both of the *cdevsw* and the *bdevsw* by calling:

- *cdevsw_add*

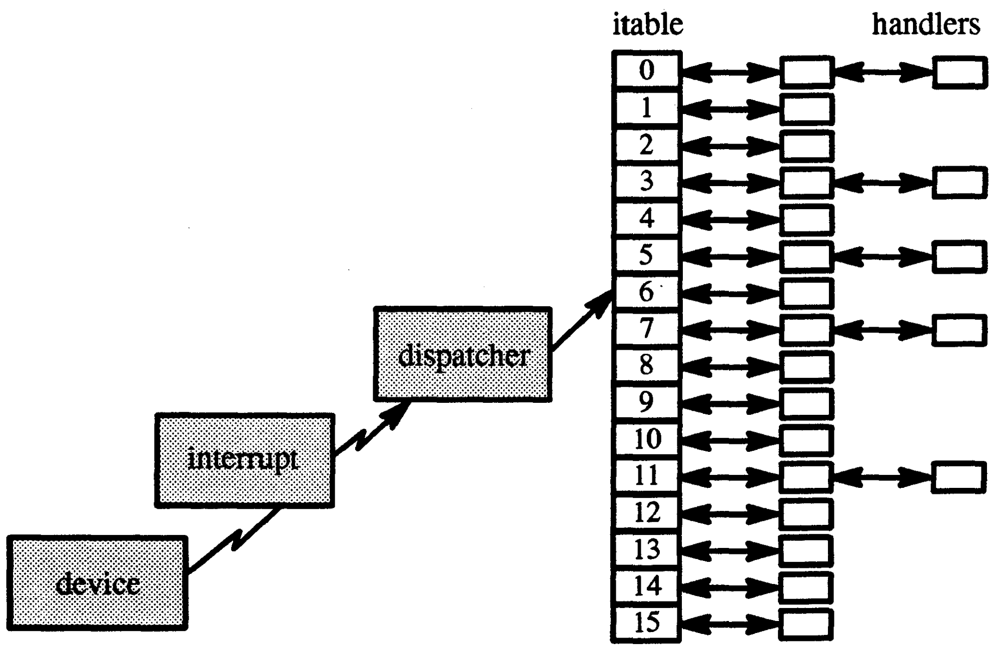
- *bdevsw_add*

In both cases, the caller either supplies a major device number or is assigned one.

For further information on dynamic configuration, see chapters 1 through 6 in Open Software Foundation, 1990a.

6–7. Dynamic Configuration

Configuring the Interrupt Handler



H2596 6-7. 236 © 1991 Hewlett-Packard

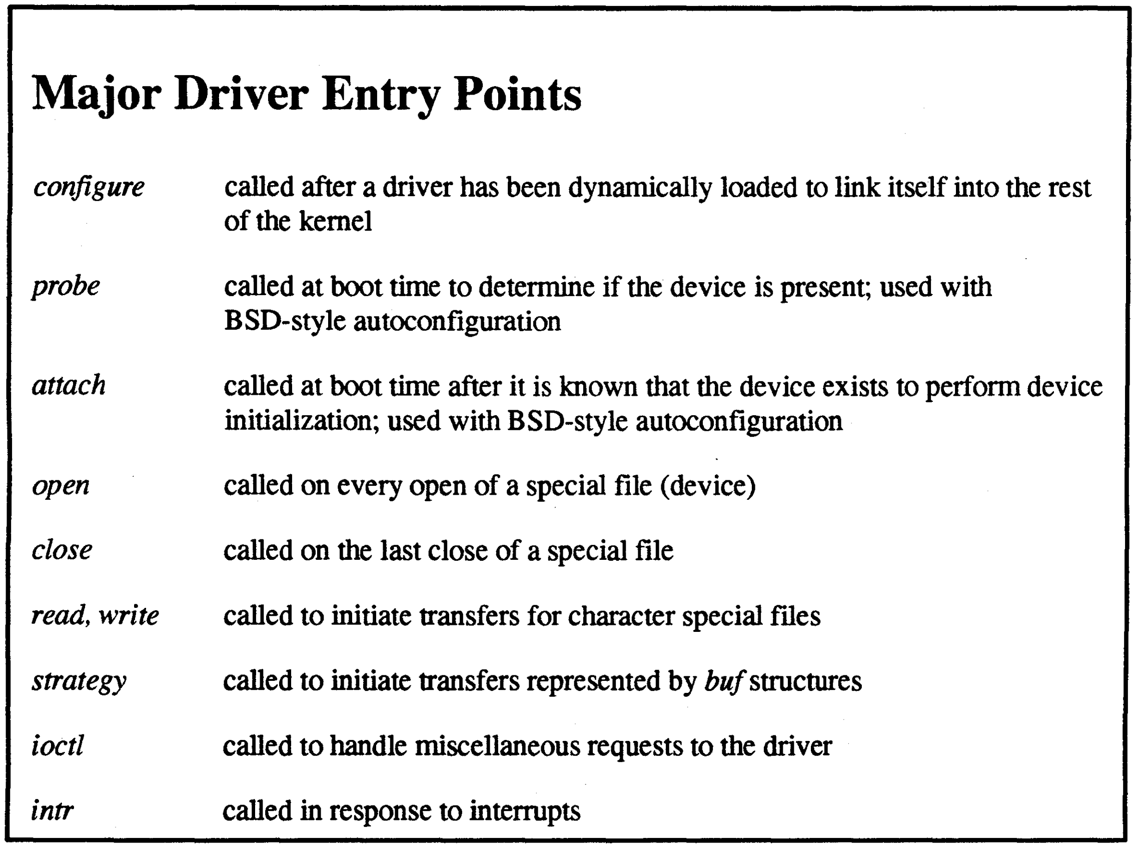
Configuring the Interrupt Handler

In the BSD kernel, the notion of interrupt vectoring is "wired into" the kernel; i.e., there is no convenient technique for adding interrupt handlers dynamically. OSF/1 provides an approach for adding and removing interrupt handlers dynamically, though the method involves some very machine-dependent facilities and must be tailored for each architecture.

When an interrupt occurs, an *interrupt dispatcher* is invoked in some machine-dependent fashion. The dispatcher, using machine-dependent techniques, consults the *itable* for the appropriate handler and forwards the interrupt to it (i.e. calls it).

A typical *itable* might be an array indexed by the interrupt level, as shown in the picture. Each element of the array would head a linked list of handlers for interrupts at that level. Chosing the correct handler would depend upon machine-dependent information, such as a vector address.

6–8. Device Drivers



Major Driver Entry Points

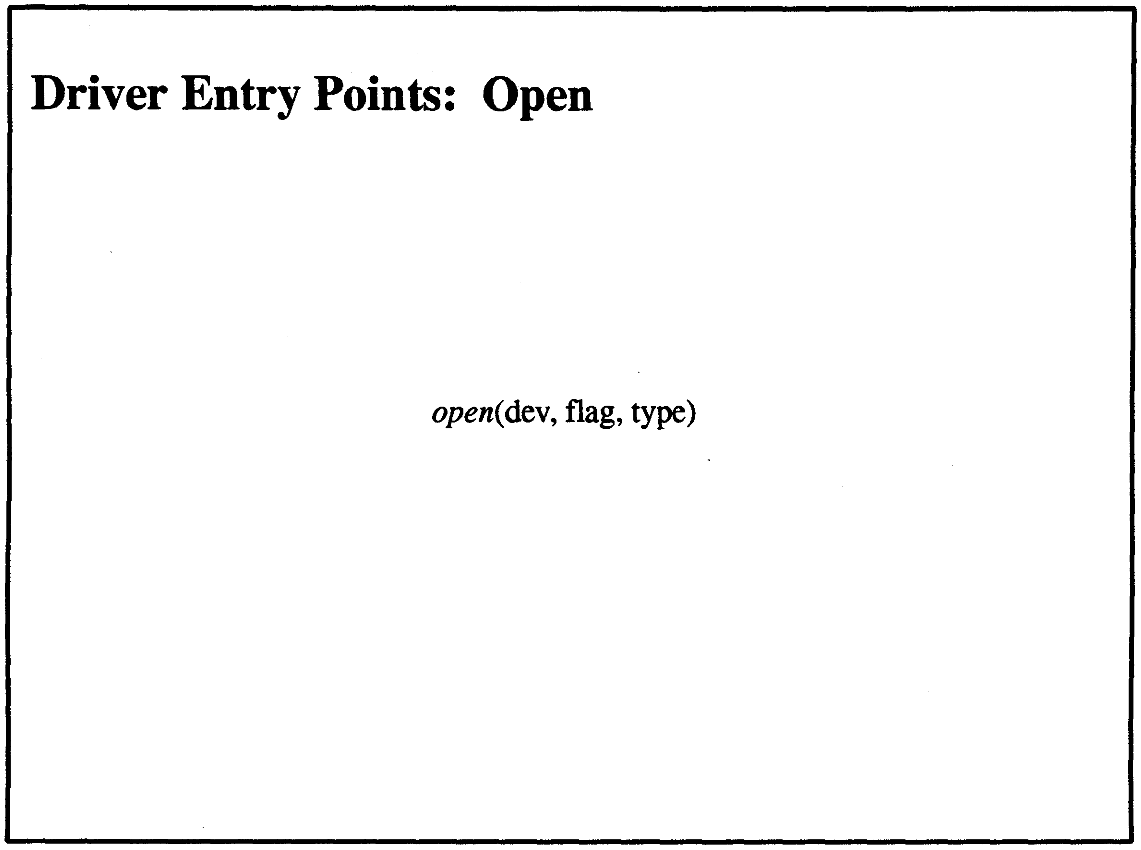
configure	called after a driver has been dynamically loaded to link itself into the rest of the kernel
probe	called at boot time to determine if the device is present; used with BSD-style autoconfiguration
attach	called at boot time after it is known that the device exists to perform device initialization; used with BSD-style autoconfiguration
open	called on every open of a special file (device)
close	called on the last close of a special file
read, write	called to initiate transfers for character special files
strategy	called to initiate transfers represented by *buf* structures
ioctl	called to handle miscellaneous requests to the driver
intr	called in response to interrupts

.H2596 6-8. 237 © 1991 Hewlett-Packard

Major Driver Entry Points

The *probe* and *attach* routines are called only at boot time and only with BSD-style autoconfiguration. If they are used, then *configure* is not needed, and vice versa.

6–9. Device Drivers



Driver Entry Points: Open

open(dev, flag, type)

Driver Entry Points: Open

- *dev*: device number (major and minor)
- *flag*: *flags* parameter from the open system call
  - O_RDONLY, O_RDWR, O_NDELAY, etc.
- *type*: indicates whether the character or block interface is being used (this argument is new and rarely used)

Possible actions:

- initialize per-device state information
- "turn on" device

- wait for device to be ready (e.g. wait for carrier-detect)
- check device status (e.g. opening for write, but no write ring on a tape drive)
- etc.

Requirements: return 0 if everything is ok, error code otherwise (e.g. ENXIO if device does not exist, EBUSY if device must be used exclusively but is busy)

6–10. Device Drivers

Driver Entry Points: Close

close(dev, flag, type)

Driver Entry Points: Close

- *dev*: device number (major and minor)
- *flag*: *flags* parameter from the open system call
- *type*: indicates whether the character or block interface is being used (this argument is new and rarely used)

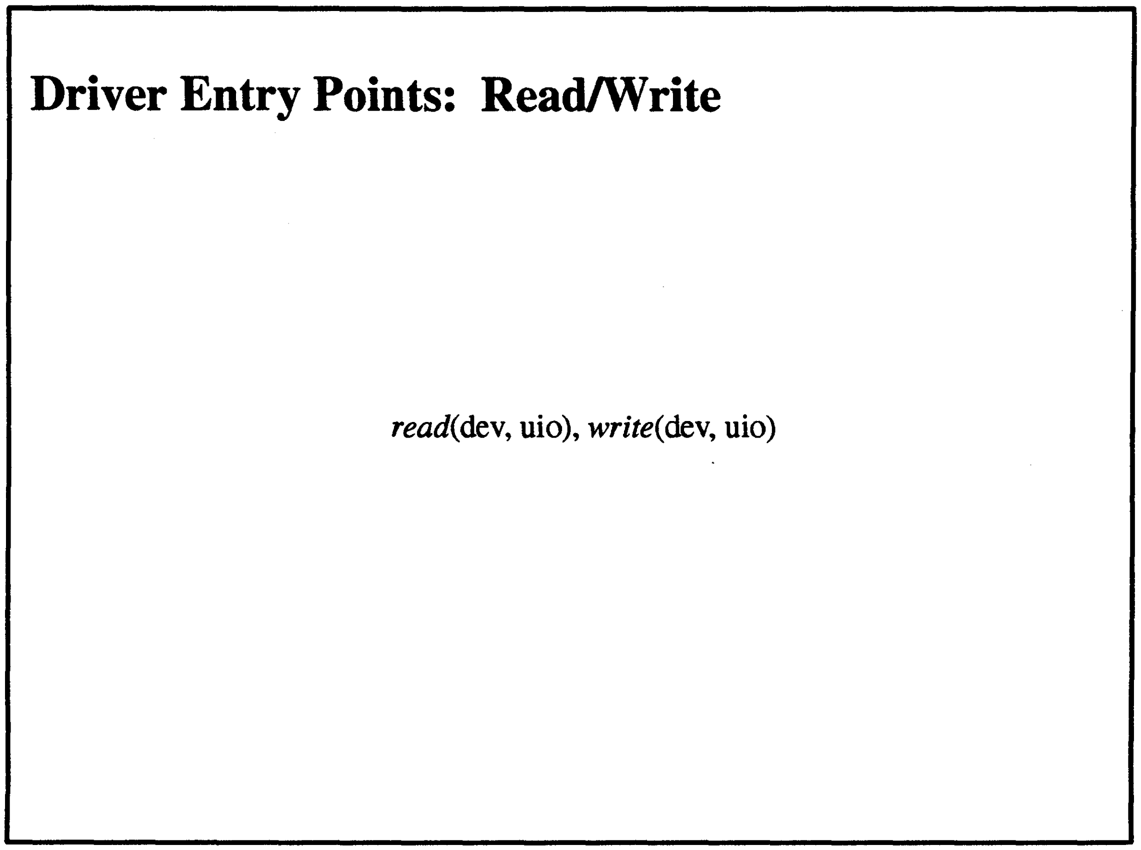
Possible actions:

- turn device "off"
- hang up phone line

- etc.

Requirements: none

6–11. Device Drivers



Driver Entry Points: Read/Write

read(dev, uio), *write*(dev, uio)

H2596 6-11. 240 © 1991 Hewlett-Packard

Driver Entry Points: Read/Write

- *dev*: device number (major and minor)
- *uio*: pointer to the *uio structure* describing the request

Actions:

- does I/O directly in simplest drivers
- for terminal I/O drivers: calls *line discipline* transfer routine
- for drivers which transfer directly into or out of the buffer provided by the user: calls *physio* (which fetches and wires the user's buffer into primary memory), which then calls *strategy*

Requirements:

- if this routine is doing the transfer, it should set *uio–>uio_resid* to the number of bytes not transferred
- returns 0 if no errors, or returns an error code (e.g. EIO for an I/O error, EFAULT if an invalid address was given for a buffer)

6–12. Device Drivers

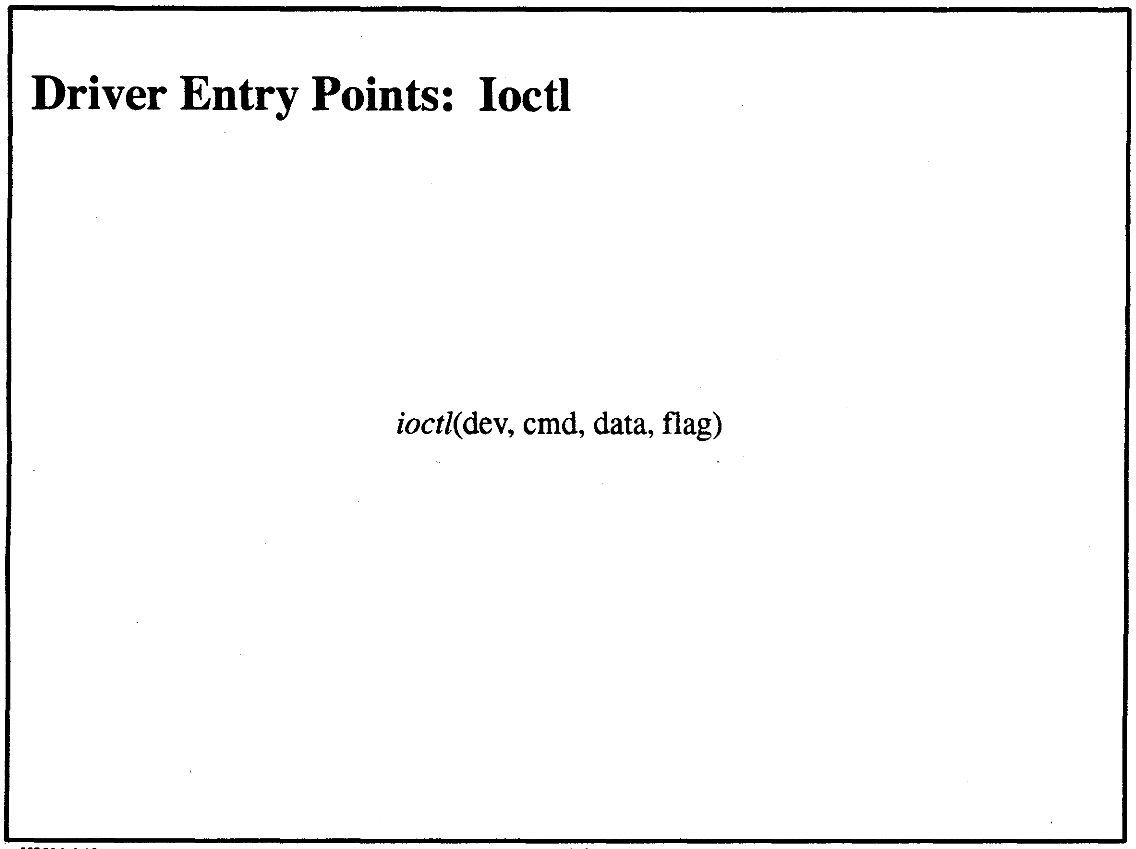
Driver Entry Points: Strategy

strategy(bp)

Driver Entry Points: Strategy

- *bp*: pointer to the *buf* structure describing the request
  - — either initiates the I/O request or queues it for eventual action

6–13. Device Drivers



Driver Entry Points: Ioctl

ioctl(dev, cmd, data, flag)

H2596 6-13. 242 © 1991 Hewlett-Packard

Driver Entry Points: Ioctl

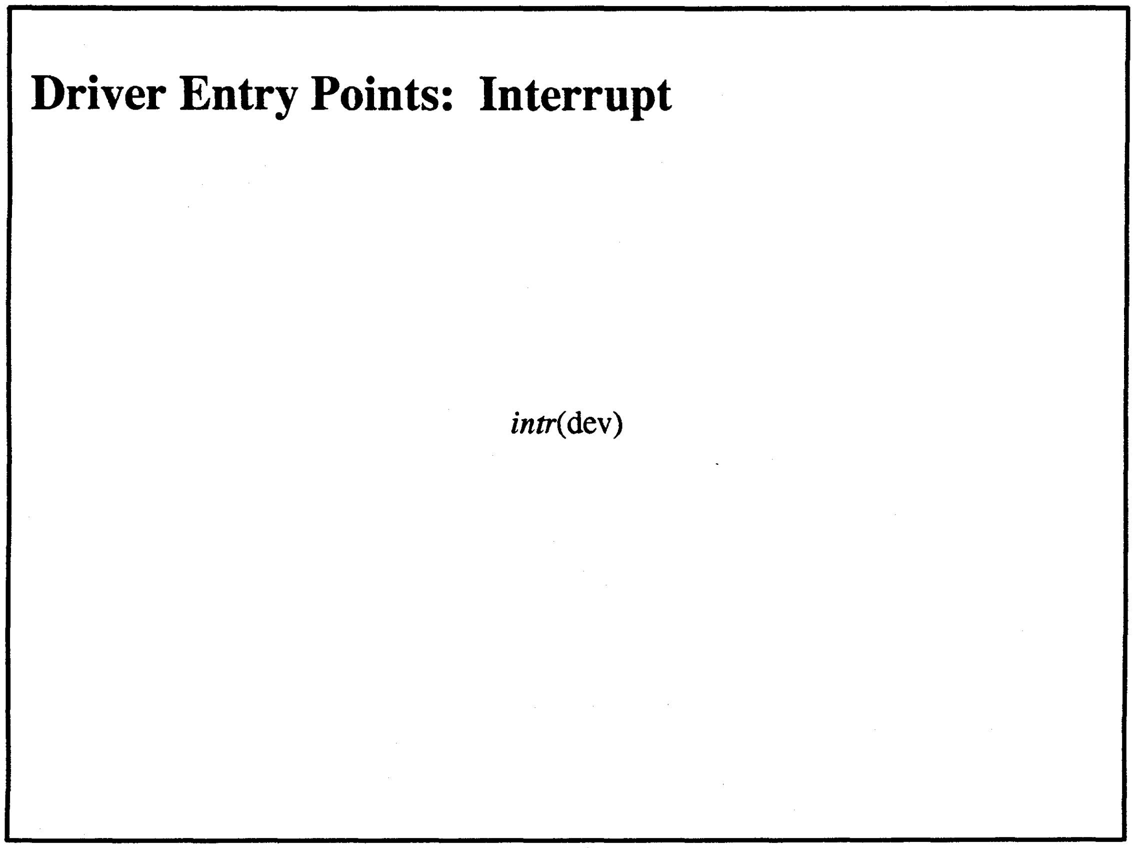
- *dev*: device number (major and minor)
- *cmd*: command code (second argument of the ioctl system call)
  - — encoded in this command code is a description of the *data* that needs to be transferred from user to system or from system to user; this allows the higher-level code to perform this transfer for all drivers
- *data*: pointer to the argument (either *in* or *out)* of the ioctl; these are interpreted differently by each driver
- *flag*: *flags* parameter from the open call

Possible actions:

- turn terminal modes
- rewind a tape drive
- etc.

Requirements: return an appropriate error code (usually ENOTTY if the command makes no sense)

6–14. Device Drivers



Driver Entry Points: Interrupt

intr(dev)

H2596 6-14. 243

Driver Entry Points: Interrupt

- *dev*: the *hardware device number* of the interrupting device, not its major and minor device numbers

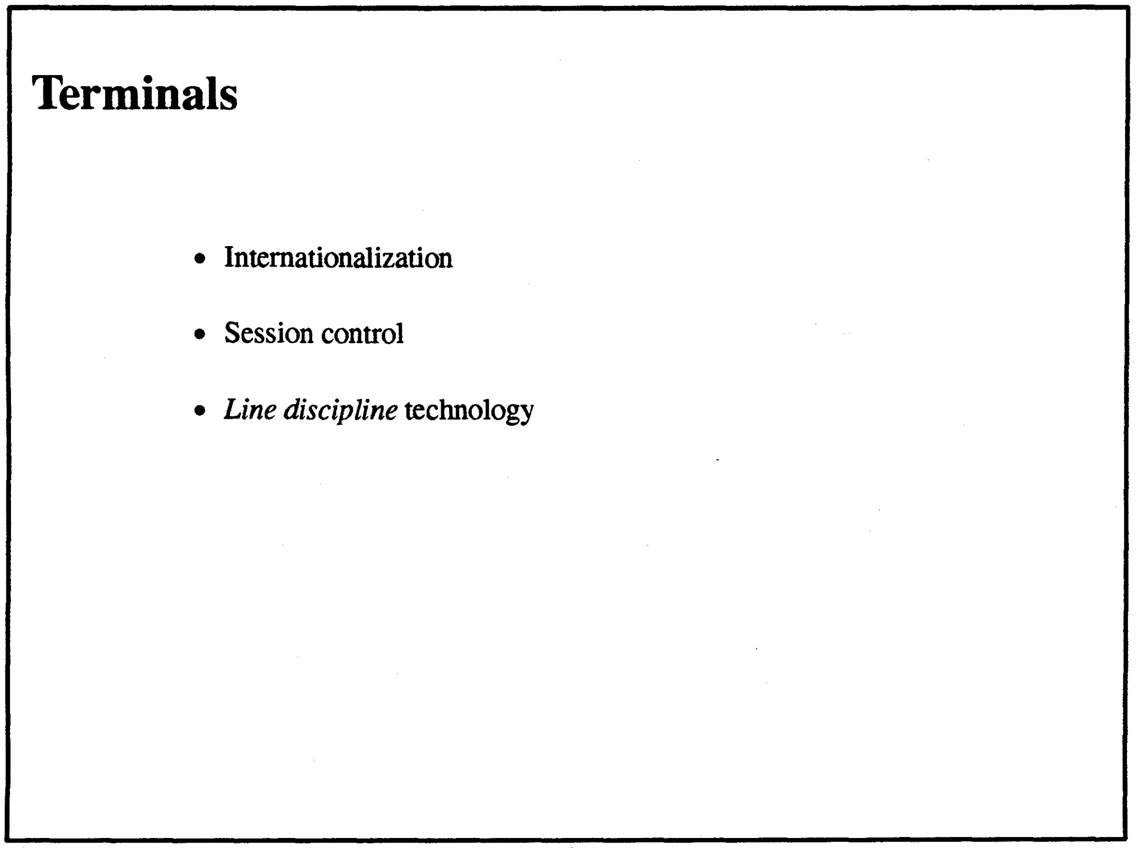
Possible actions:

- check for and react to errors
- *wake up* threads waiting for the I/O completion
- acknowledge the interrupt in the controller registers

Requirements:

- be quick!
- don't sleep; execution is in the interrupt context and not in the context of any thread

6–15. Terminal I/O



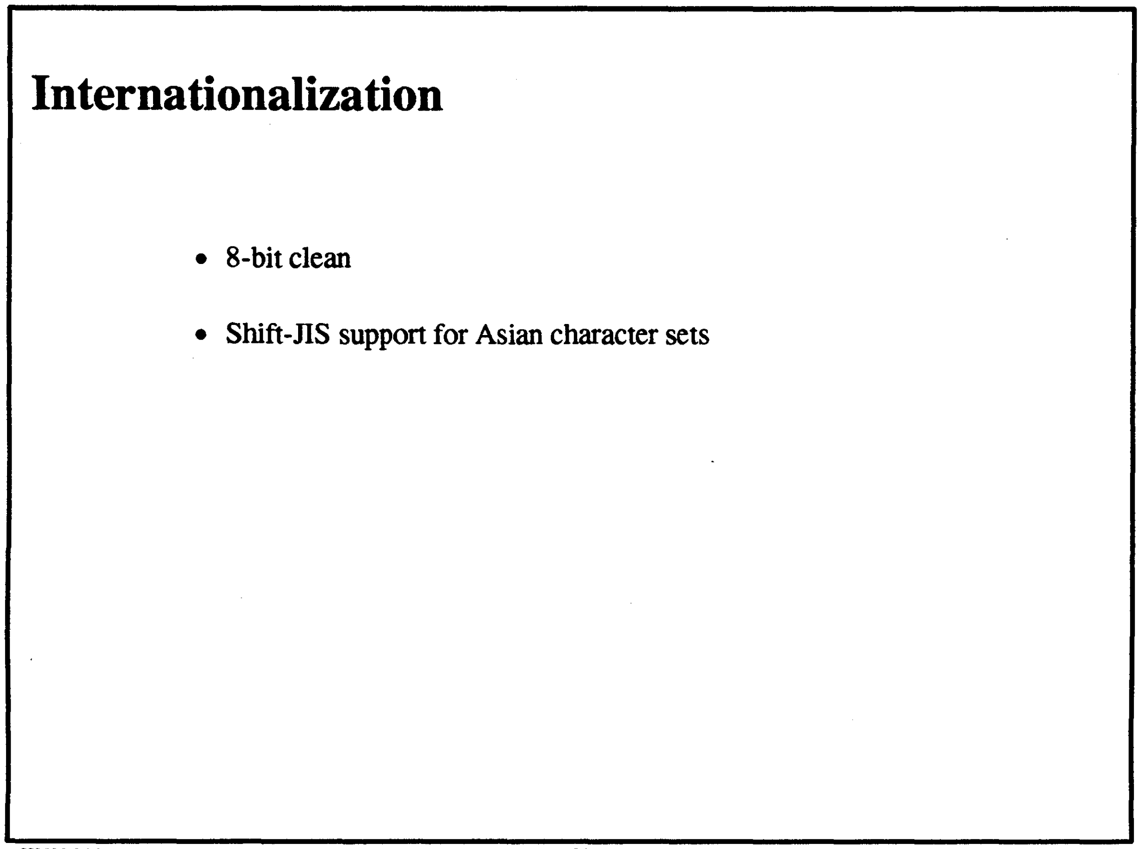
Terminals

- Internationalization
- Session control
- *Line discipline* technology

.H2596 6-15. 244 © 1991 Hewlett-Packard

Terminals

6–16. Terminal I/O



Internationalization

- 8-bit clean
- Shift-JIS support for Asian character sets

.H2596 6-16. 245 © 1991 Hewlett-Packard

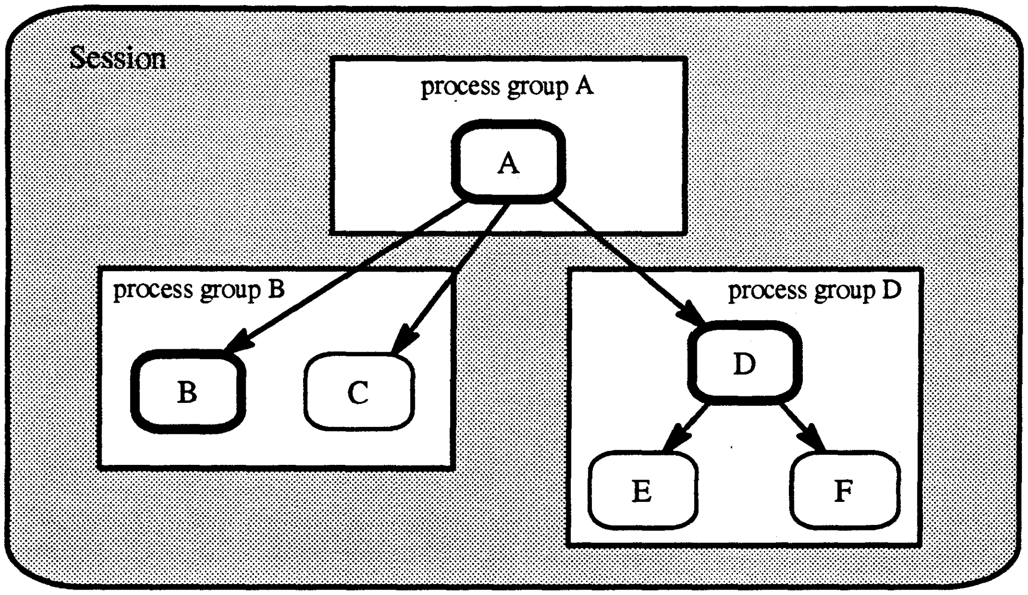
Internationalization

Earlier versions of UNIX were designed strictly for use with the ASCII character set. This is a seven-bit character set and, programmers being programmers, the extra eighth bit was used for a variety of purposes. The seven-bit/eight-bit problem has been cleaned up in OSF/1, and the entire kernel and all of the libraries are eight-bit clean: no special use is made of the eighth bit in characters, and thus eight-bit character sets can be supported.

A much more difficult problem is dealing with character sets in which characters are larger than bytes. OSF/1 includes support for Shift-JIS, but this is not considered the final word on the subject and more will be available in Release 1.1.

6–17. Terminal I/O

Sessions



Sessions

POSIX introduced the concept of sessions to clean up the BSD notion of job control. A session is a terminal session, and hence a collection of processes sharing a terminal. Traditionally there has been the notion of foreground processes and background processes: foreground processes are affected by signals generated by key strokes, background processes are not. Job control is a means for moving processes back and forth between the foreground and the background.

The picture shows three process groups whose names are the process id of their first and founding members.

- Process A formed the session and spawned processes B, C, and D
  - — process A is in its own process group
  - — process B formed a new process group with itself as leader (giving its name to the group) and C as another member

— process D formed another new process group and spawned two children that stayed in the group

- Each process group forms a "job" that can be suspended or placed in the foreground or background

- A *stop signal* suspends a process group

— sent by a thread in another process (usually an ancestor)

— sent by the kernel due to actions on the terminal (e.g., a background process reading from a terminal)

— all threads within each process of the process group are suspended (using *task_suspend*)

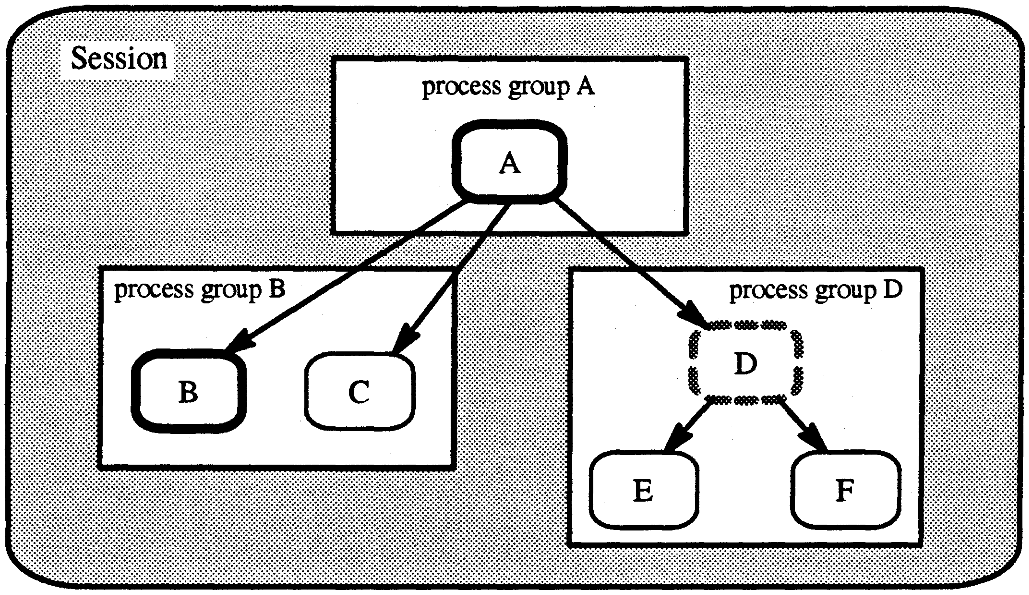
- A *continue signal* (usually from an ancestor) resumes a stopped process

— if an orphaned process receives a *stop* signal because of actions on the terminal (if, say, it is a background process and is reading from the terminal, and thus receives the SIGTTIN signal), it is unlikely that any thread will send it a *continue* signal; such a *stop* signal should be ignored (the I/O system call will return the error EIO). Note that this differs from BSD's solution—in BSD the process would have been sent the SIGKILL signal (and hence would have been terminated)

— the generalization of the orphaned process is the orphaned process group: a process group whose members have no parents within the session

6–18. Terminal I/O

Orphaned Process Groups



.H2596 6-18. 247 © 1991 Hewlett-Packard

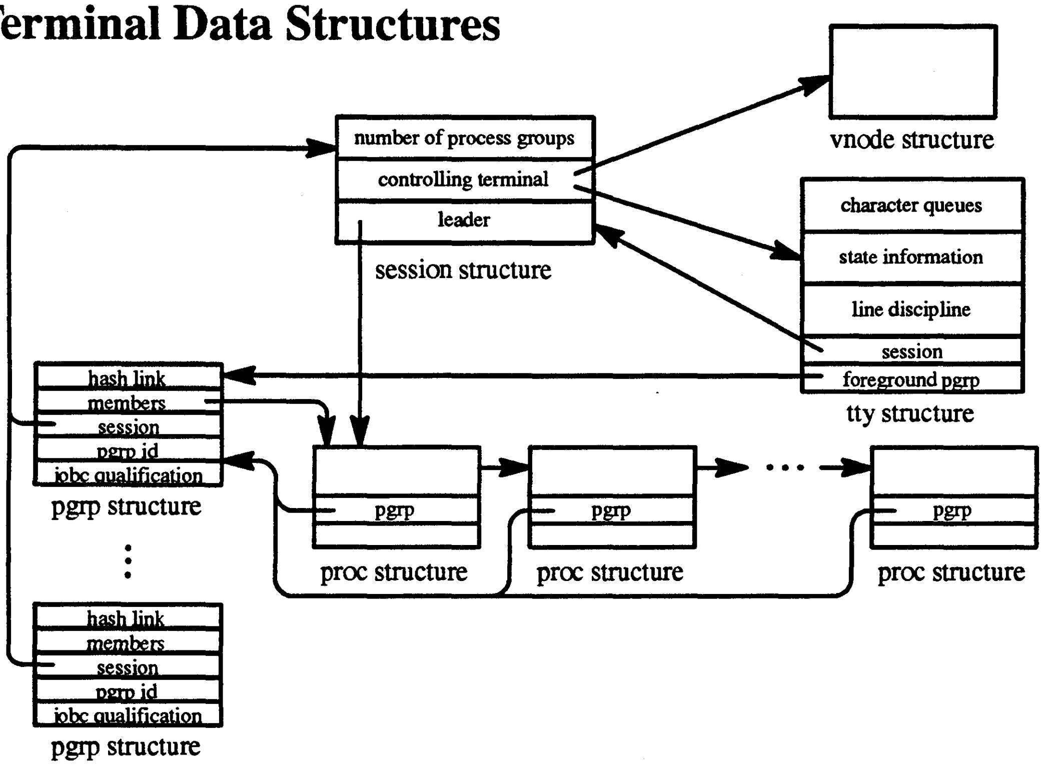
Orphaned Process Groups

- If process D terminates, then the *init* process adopts processes E and F
  - — process group D is thus an orphan, since it has no processes with a parent in the session
  - — processes E and F receive the EIO error if they attempt to read from their terminal

A process group is considered an orphan if none of its members have an ancestor that is the session leader. Thus in the picture, if process C is moved to process group D, then process group D is no longer an orphan.

6–19. Terminal I/O

Terminal Data Structures



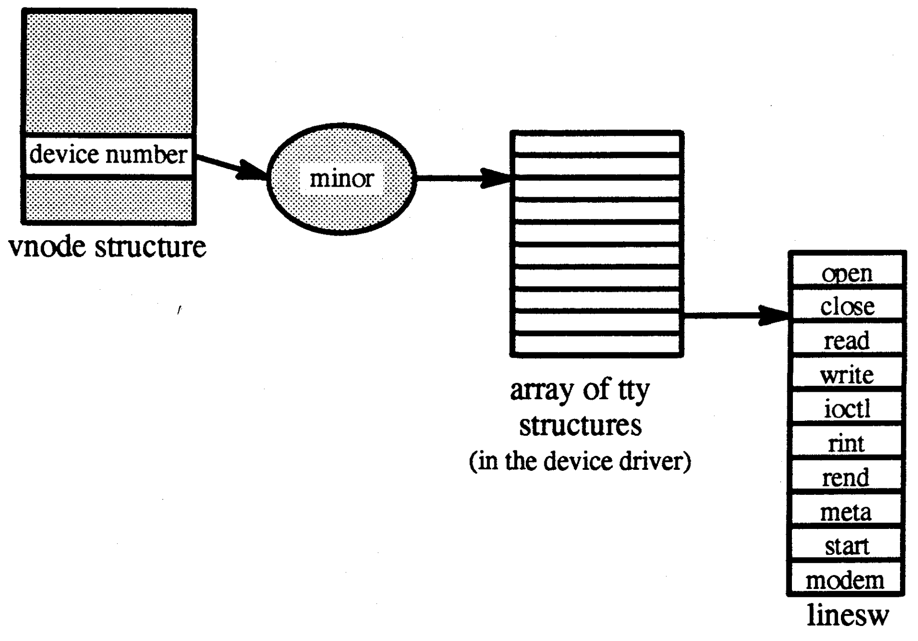
.H2596 6-19. 248

Terminal Data Structures

The *jobc qualification* field of the *pgrp* structure is used to indicate whether or not the process group is an orphan (and hence not qualified to receive job control). The field contains a count of the number of processes in the process group whose parents are both outside of the process group and qualified for job control.

6–20. Terminal I/O

Line Discipline



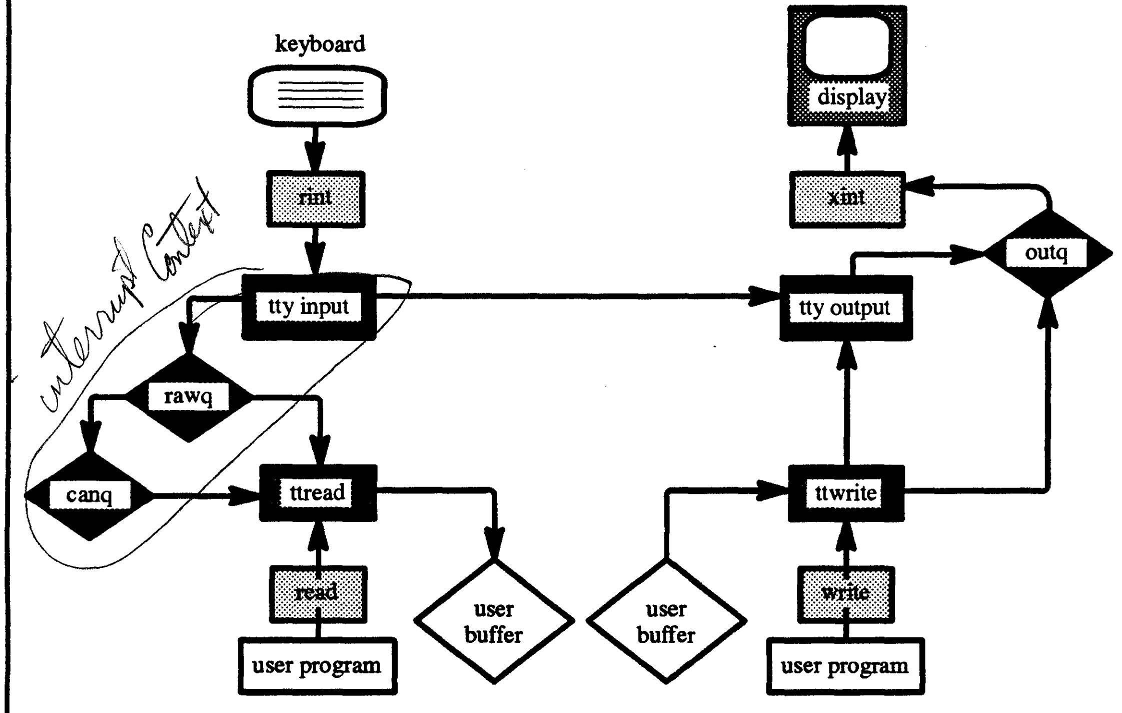
.H2596 6-20. 249 © 1991 Hewlett-Packard

Line Discipline

This slide illustrates how the device number (from the vnode structure) is used to identify the *tty* structure and hence to identify the entry points into the terminal's line discipline. Each terminal device driver maintains a table mapping minor device numbers into pointers to *tty* structures.

6–21. Terminal I/O

Terminal I/O Flow



H2596 6-21. 250 © 1991 Hewlett-Packard

Terminal I/O Flow

When a character is typed on the keyboard, the driver's read interrupt routine (*rint*) is called (in the interrupt context). It determines which *line discipline* is being used by consulting the *tty* data structure associated with the terminal. In this example we assume that the standard *tty* line discipline is being used. Thus control is passed to the *ttyinput* routine. This routine looks at the tty structure to determine the terminal's *mode*. If it is in *cooked* mode, then the incoming characters are interpreted as parts of lines of text which may be edited. Characters are first placed on the *raw* queue, where they may be edited in response to edit characters. Once a line delimiter (e.g., carriage return) is received, the line of text is copied to the *can* (canonical) queue. Otherwise, if the terminal is in either *raw* mode or *cbreak* mode, incoming characters cannot be edited, and are left on the *raw* queue. If echoing is enabled, then the *ttyinput* routine calls the *ttyoutput* routine to echo the characters. Since this processing occurs in the interrupt context, the interrupted context is resumed after *ttyinput* returns.

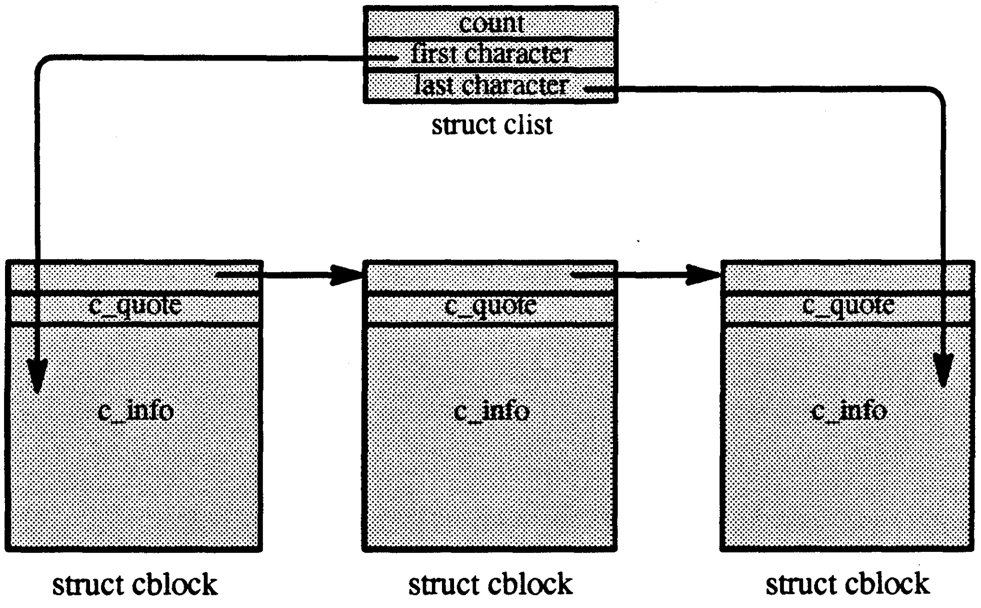
In the context of a thread performing a read system call, control enters the device driver's *read* routine which consults the *tty* structure and, in our case, calls the *tty* line discipline's *ttread* routine. This routine determines the mode of the terminal. If the terminal is in *cooked* mode, *ttread* looks in the *can* queue for characters. Otherwise it looks in the *raw* queue. If there are no characters in the appropriate queue and if blocking is permitted, the calling thread blocks until characters are received. If characters are available then they are copied to the user's buffer.

In the context of a thread performing a write system call, control enters the device driver's *write* routine, which, in our case, calls the line discipline's *ttywrite* routine. This routine copies characters into the kernel from the user's buffer. Those characters requiring no special processing are appended to the *out* queue. Characters requiring special processing are processed by the *ttyoutput* routine (the primary purpose of this routine is to deal with characters not present in the terminal's character set; this routine is rarely needed in modern systems).

The device driver, usually executing in the interrupt context, transfers characters from the *out* queue to the device.

6–22. Terminal I/O

Terminal I/O Data Structures



H2596 6-22. 251 © 1991 Hewlett-Packard

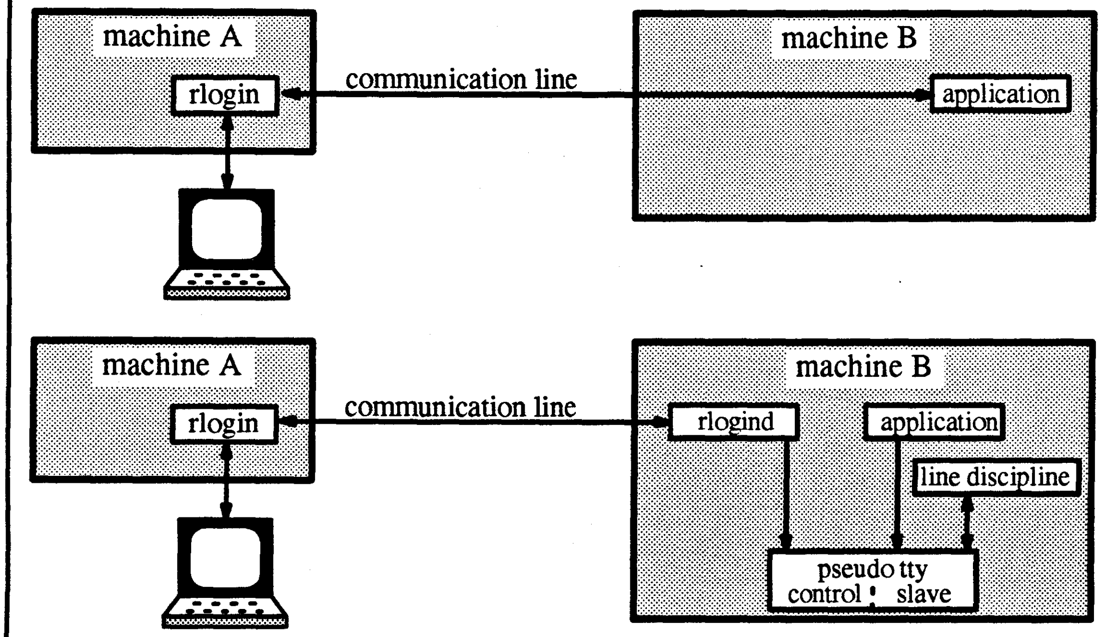
Terminal I/O Data Structures

The *raw* queue, *can* queue, and *out* queue of the previous slide are instances of data structures known generically as *clists*, which represent queues of 8-bit characters. The format of the list, shown in the accompanying picture, consists of a header followed by a number of fixed-size *cblocks*. The *cblocks* are typically 64 bytes in length, with 12 bytes of overhead and 52 bytes of data (*c_info*).

Of particular interest is the *c_quote*, which is used to indicate which characters within *c_info* have been "quoted," meaning that they are not to be interpreted as providing any sort of special function such as erasing a character or suspending the process group. In earlier versions of UNIX, such quoting was done by setting the eighth bit of a character. However, this technique only works with English character sets.

6–23. Terminal I/O

Pseudo Terminals



.H2596 6-23. 252 © 1991 Hewlett-Packard

Pseudo Terminals

Pseudo terminals are used to present a terminal interface to an application that is not connected to a real terminal. As an example, consider a remote-login-type application. In the top portion of the picture, the real terminal is connected to machine A, and the user is running *rlogin* so as to run an application on machine B. This application might be an editor, which might attempt to make changes to the terminal's mode, e.g., switch from *cooked* mode to *cbreak* mode. However, such attempts would fail because it is actually connected to a communication line that is different from a terminal and does not respond to requests to change modes.

One can think of two approaches to this problem. The first would be to forward the terminal-oriented system calls to the machine to which the terminal is connected. The other would be to run the line-discipline code on the application's machine (machine B in this example). Berkeley UNIX, and hence OSF/1, use the latter approach.

A *pseudo device driver* or a *pseudo terminal* is set up in machine B's kernel. It appears to the rest of the operating system to be a pair of ordinary device drivers, but the pair of drivers communicate with each other instead of with devices. A user process, in this case the *rlogind*, communicates with *rlogin* via the communication line. Incoming bytes are written to one of the pseudo devices—the control device. Output through this device is made to reappear as input from the slave device, i.e. as if a character had just been received from a terminal. This character is then processed by the line discipline code discussed on page 6-30: characters are queued on either the *raw* queue or the *can* queue. When the application issues a read system call, it receives these characters after they have been processed by the line discipline. Output from the application is processed by the line discipline as if the characters were being sent to a real terminal, but instead they are made to appear as incoming characters in the control pseudo device. These characters are read by the *rlogind*, which sends them across the communication line to *rlogin*.

Module 6 — Device Drivers and Terminal I/O

Exercises:

1. a. Is there any difference between a special file whose inode is in an S5 file system and one whose inode is in a UFS file system?

   b. Why are the *specalias* and *specinfo* structures necessary?

2. a. List the steps necessary to add a module to the operating system dynamically.

   b. Why is it necessary that dynamically loadable modules have *configure* routines (for example, why couldn't the work performed by the *configure* routine be performed by user-level code?)?

3. What are the differences between BSD device drivers and OSF/1 device drivers?

4. a. What is an "orphaned" process group?

   b. How is it determined what a terminal's line discipline is?

   c. Explain the relationship between pseudo terminals and device drivers.

Module 7 — Streams

Module Contents

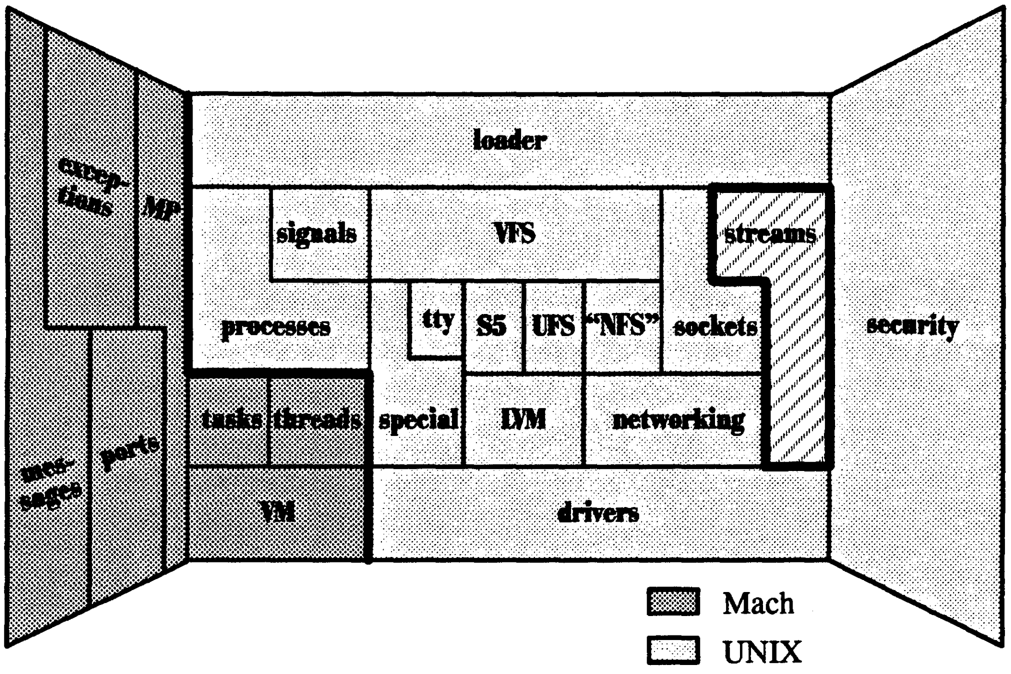
Module Objectives

In order to demonstrate an understanding of the concept of streams and their implementation in OSF/1, the student should be able to:

- list the three types of streams components
- list and show the interconnections between the data structures used to represent a streams message
- explain why it is necessary for streams-oriented service calls to be serialized
- list the data structures used to represent and access an open stream
- explain the purpose of cloning
- list the synchronization options available in the OSF/1 implementation of streams

7–1. The Big Picture

Streams

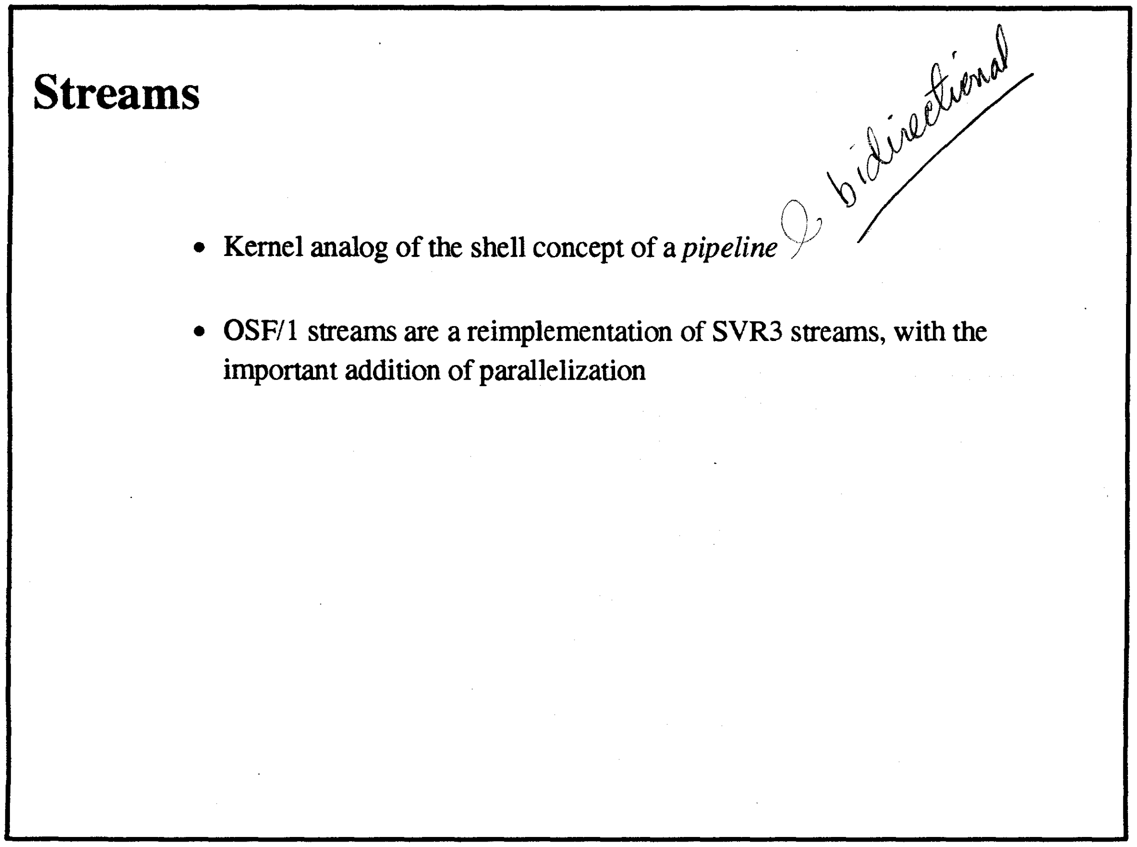


H2596 7-1. 253 © 1991 Hewlett-Packard

Streams

The best general introduction to streams is AT&T, 1989. The OSF/1 implementation of streams is discussed in chapter 13 of Open Software Foundation, 1990a.

7–2. Streams Concepts



Streams

- Kernel analog of the shell concept of a *pipeline* ? & bidirectional
- OSF/1 streams are a reimplementation of SVR3 streams, with the important addition of parallelization

H2596 7-2. 254 © 1991 Hewlett-Packard

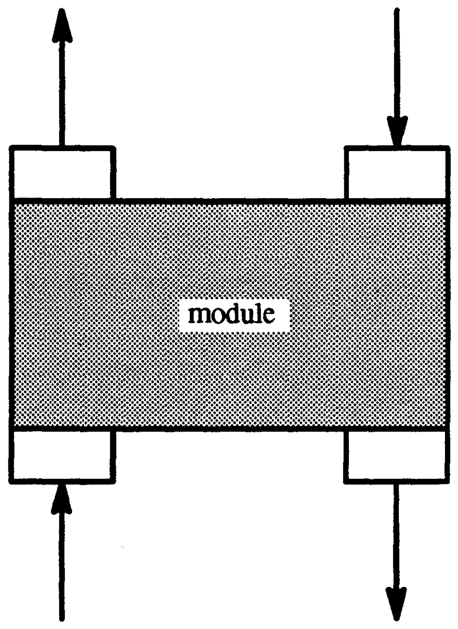
Streams

A *shell pipeline* is a unidirectional stream of bytes processed by one or more filters. A *kernel stream* is a bidirectional stream of messages being processed in one or more modules. The endpoints of a kernel stream can be in two user processes, or, most commonly, one endpoint may be in a user process and the other in a device driver.

In shell pipelines, each filter is implemented as a separate process. This technique could be extended to kernel streams through the use of kernel threads. However, the original designer of streams, Dennis Ritchie, felt that so many streams modules would be active at once that, even with very lightweight kernel threads, a thread per module would be too expensive. Thus each module is a collection of procedures that can be called in a variety of contexts. Associated with each module are data structures to contain its state, so that the module's execution can be started in the context of one caller and continued in the context of another.

7–3. Streams Concepts

Stream Components, part 1



.H2596 7-3. 255 © 1991 Hewlett-Packard

Stream Components, part 1

Each *module* consists of a set of routines to process data and (possibly) a pair of queues, one for each direction.

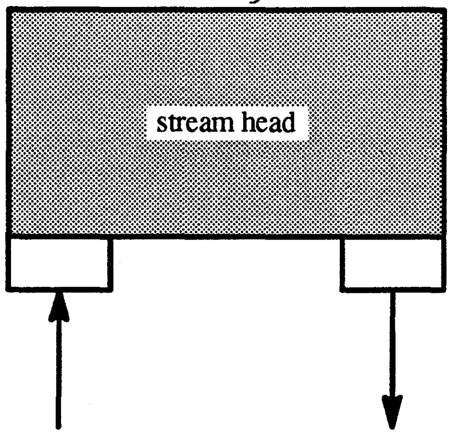
A simple example of a module is one that capitalizes all characters going in either direction. A more complicated module might be one that encrypts data going downstream and decrypts data going upstream.

7–4. Streams Concepts

Read(2)
write(2)

Stream Components, part 2

Device file
Interface



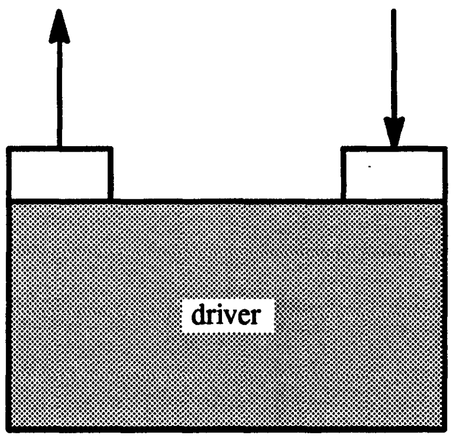
.H2596 7-4. 256 © 1991 Hewlett-Packard

Stream Components, part 2

Stream head is a special case of a stream module: it supplies the interface to the system-call layer, converts system calls into messages sent down a stream, and converts messages arriving from the stream into responses to system calls.

7–5. Streams Concepts

Stream Components, part 3



H2596 7-5. 257 © 1991 Hewlett-Packard

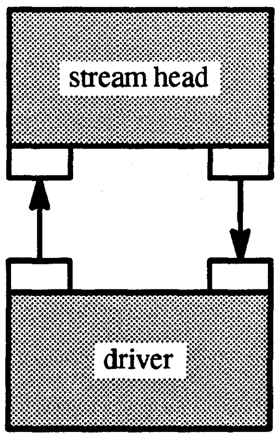
Stream Components, part 3

A streams driver can be an interface to a real device or can be an interface to other streams, as will be seen.

7–6. Streams Concepts

Stream Setup

```
fd = open("/dev/stream_device", O_RDWR);
```



.H2596 7-6. 258 © 1991 Hewlett-Packard

Stream Setup

A stream may be created by opening a streams device.

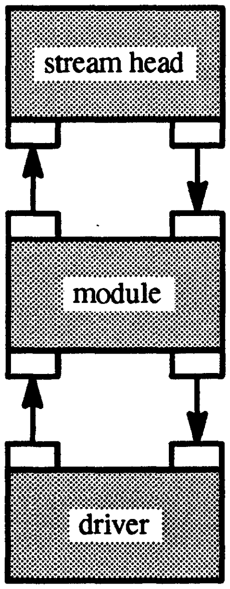
Streams devices appear as character special devices, i.e., they are identified as special files and have entries in the *cdevsw*.

A streams device may be read from and written to immediately after being opened, but, at this point, it doesn't supply much additional functionality over non-streams devices.

7–7. Streams Concepts

Stream Push

```
ioctl(fd, I_PUSH, "module name");
```

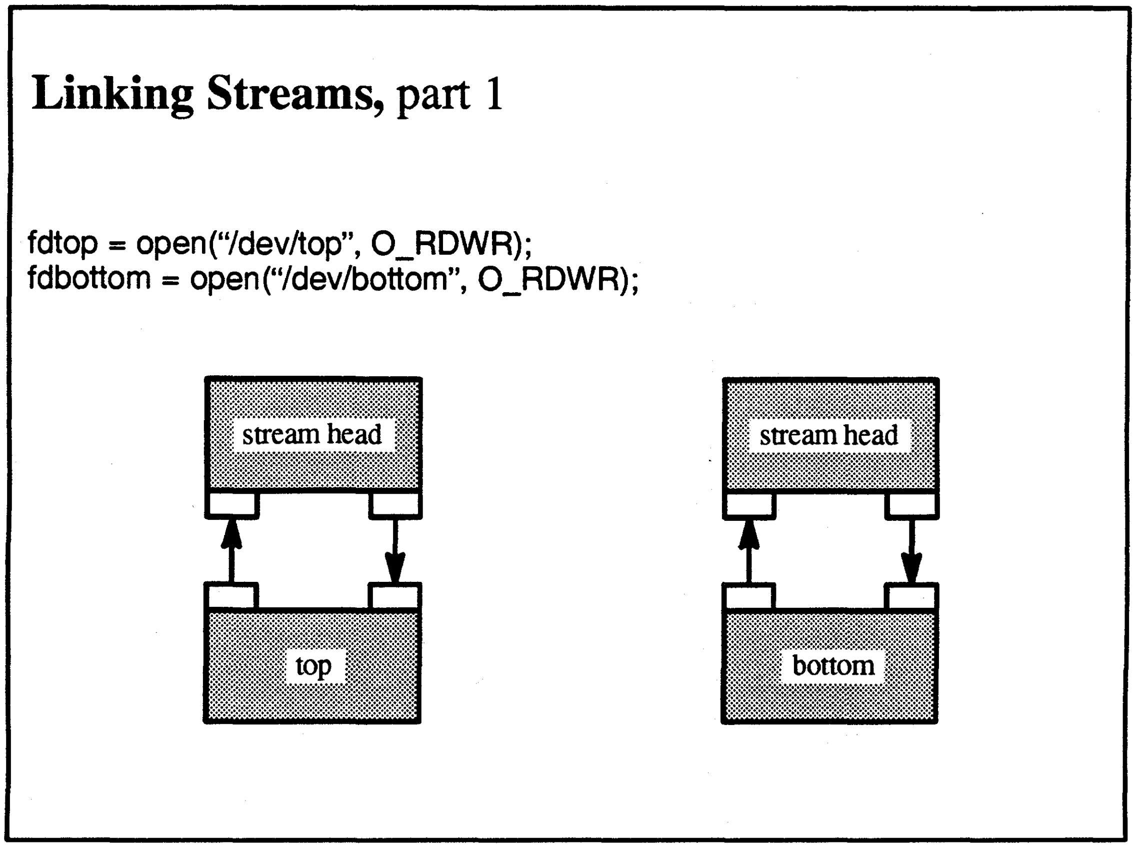


H2596 7-7. 259 © 1991 Hewlett-Packard

Stream Push

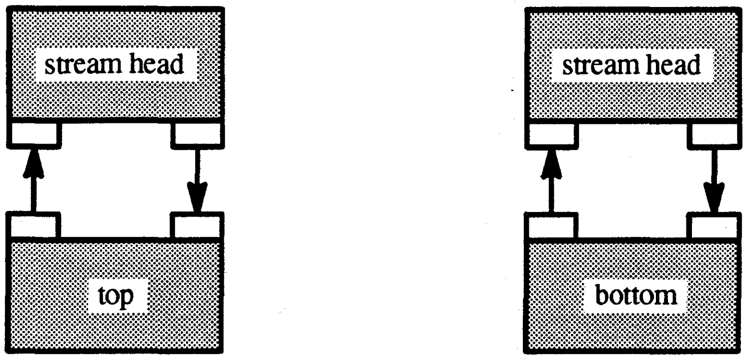
A module may be inserted at the top *(pushed)* by executing an ioctl with the I_PUSH command. Data being sent in either direction through the stream will now be processed by the module.

7–8. Streams Concepts



Linking Streams, part 1

```
fdtop = open("/dev/top", O_RDWR);
fdbottom = open("/dev/bottom", O_RDWR);
```

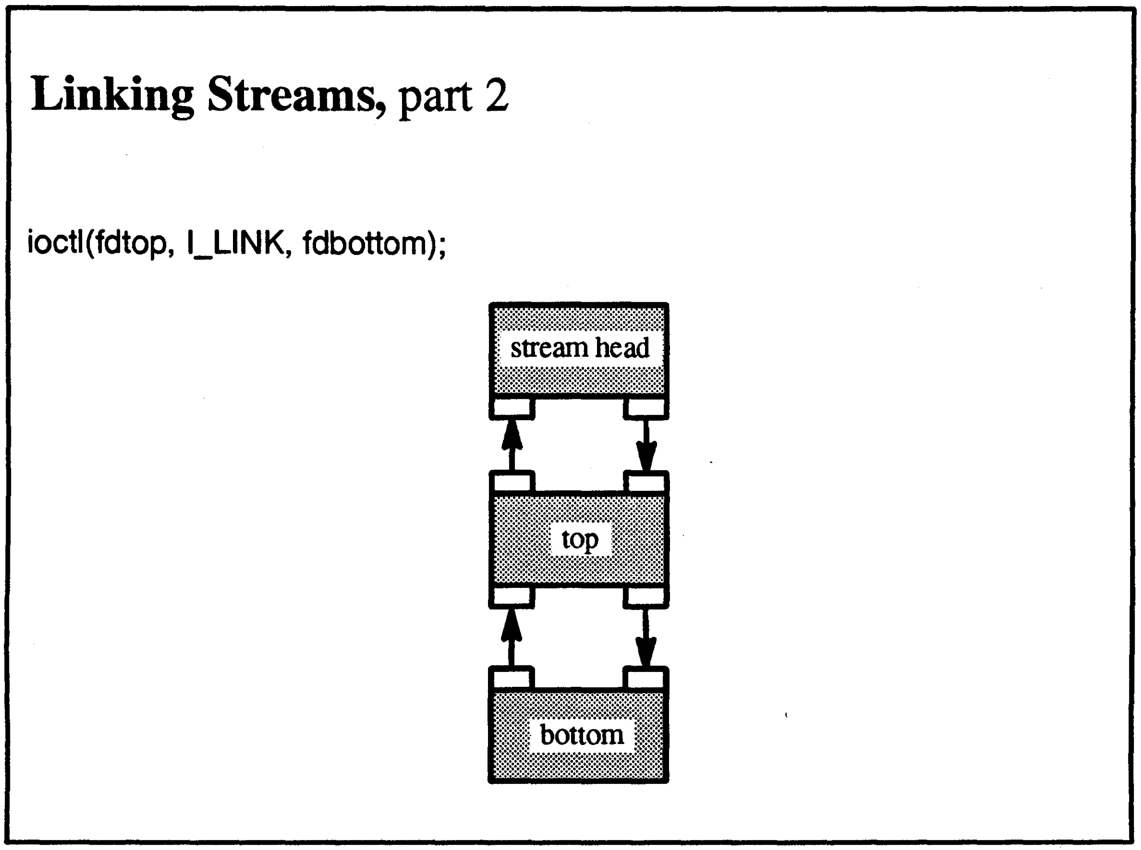


H2596 7-8. 260

Linking Streams, part 1

In this picture the streams driver module *top* is a multiplexer; i.e., it is an interface between a stream and other streams, not an interface between a stream and a device.

7–9. Streams Concepts



Linking Streams, part 2

ioctl(fdtop, I_LINK, fdbottom);

stream head
top
bottom

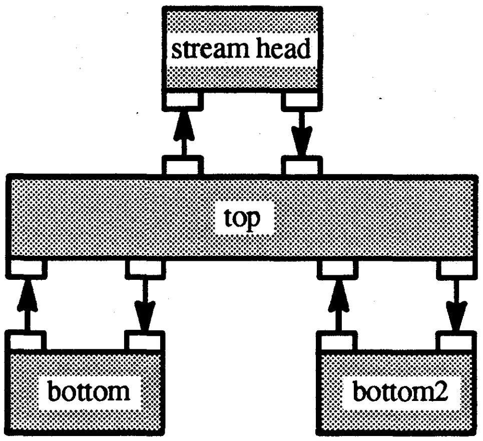
Linking Streams, part 2

The *bottom* pipeline of the previous picture has been linked beneath the streams driver *top*. The streams driver *top* now has an upper and a lower part. Part of *top*'s function will be to transfer data from the upper part to the lower part and vice versa.

7–10. Streams Concepts

Multiplexing Streams, part 1

```
fdbottom2 = open("/dev/bottom2", O_RDWR);
ioctl(fdtop, I_LINK, fdbottom2);
```



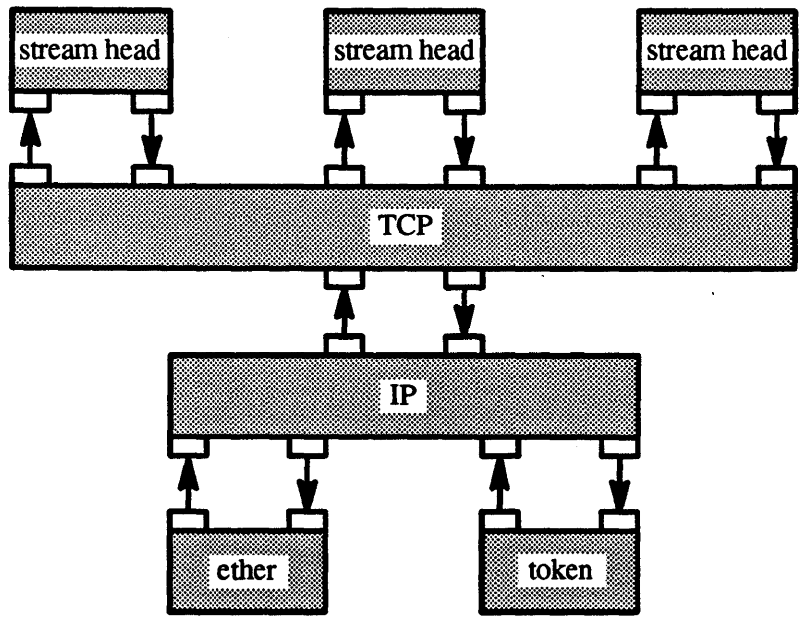
Multiplexing Streams, part 1

In this picture the driver for *top* is responsible for the multiplexing/demultiplexing of messages coming up from below or going down from above. For messages going downwards, code supplied in the *top* module must decide through which lower stream the data should be sent. It uses its own criteria for doing so, but is likely to base its decision on the contents of the message itself.

Here *top* is a *one-to-many* multiplexer.

7–11. Streams Concepts

Multiplexing Streams, part 2



H2596 7-11. 263 © 1991 Hewlett-Packard

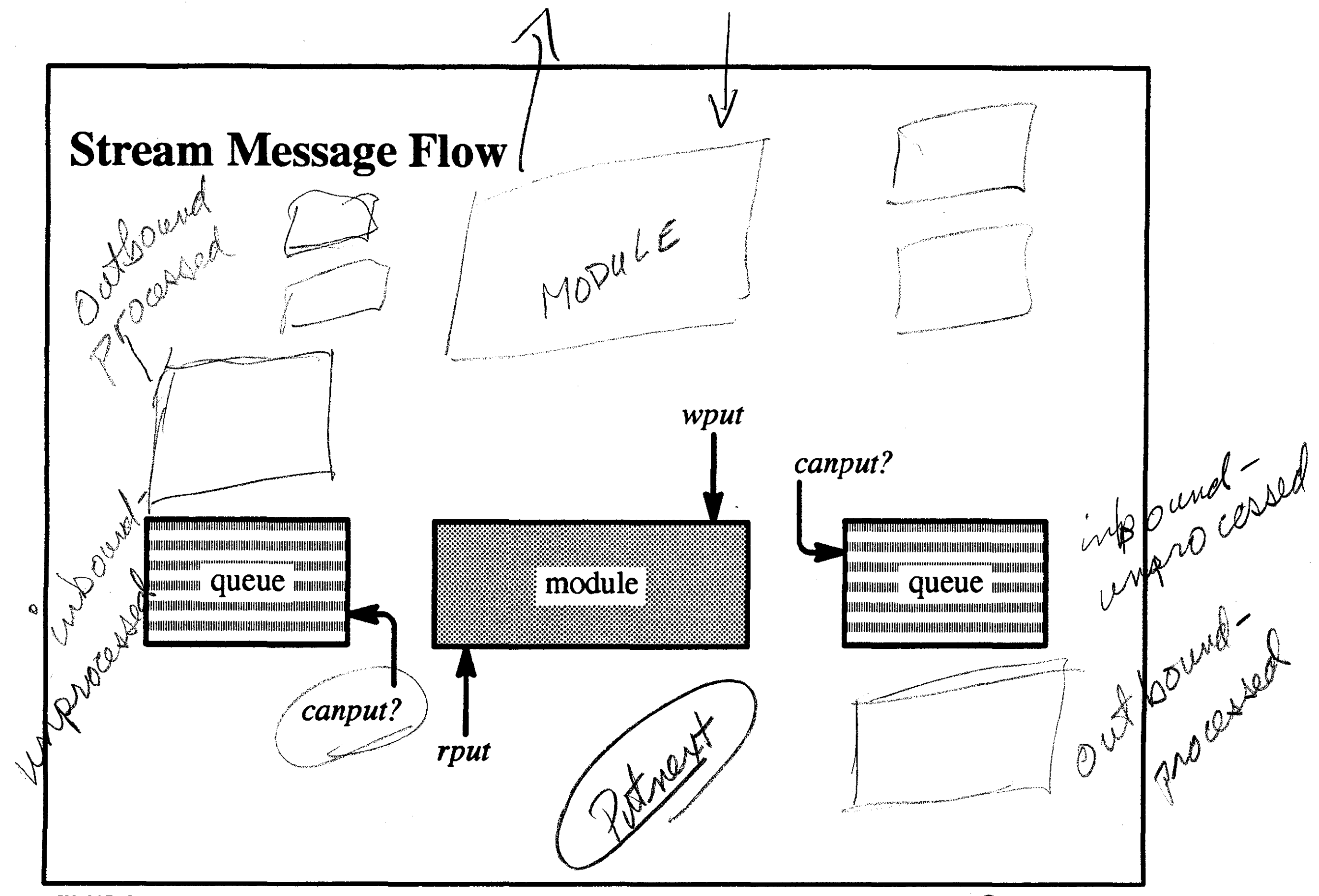
Multiplexing Streams, part 2

In this example we show a more complicated arrangement of multiplexers. TCP is a *many-to-one* multiplexer. IP in the picture is a *one-to-many* multiplexer, though in practice it is a *many-to-many* multiplexer.

To create this arrangement, first the IP, ether, and token streams are opened, and then the latter two streams are linked to IP. Then the TCP stream is opened and the IP stream linked beneath it. Each stream connected to the top of the TCP module represents a separate TCP connection.

(N.B.: in OSF/1, the TCP/IP protocols are implemented not in the streams framework, but in the socket framework. TCP/IP is used here merely as an example.)

7–12. Message Flow



H2596 7-12. 264 © 1991 Hewlett-Packard

Stream Message Flow

Each module defines (usually) a *wput* routine and an *rput* routine to be called when a message arrives from above (*wput*) or below (*rput*). From within a module, the appropriate *put* procedure of the next module is called by calling the routine *putnext*.

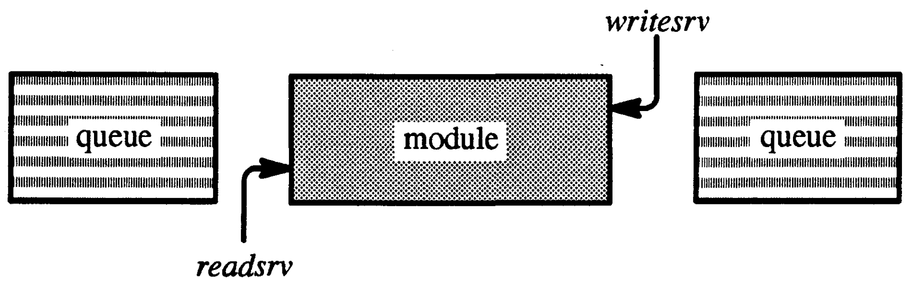
Simple modules modify incoming messages in-place and pass them on to the next module (by calling *putnext*). Other modules may need additional resources to process a message, or may need to defer processing. These modules have the ability to queue messages. Associated with these queues (one for each direction) is a size limit known as the *high-water mark*.

Because of the finite capacities of these queues, *flow control* must be established. A module that is itself capable of queueing must check for queue space ahead of it (by calling *canput*) before sending the message. The *canput* routine looks ahead for the first module that is capable of queueing and returns *true* if and only if there is room for another message in that queue. If there is no room, then the calling module must put the message in its own queue (the previous module's *canput* routine checked that there was room on the queue before it sent the message). A module may also defer the processing of a message by placing the message at the end of its queue (in either case, enqueuing is accomplished by calling *putq*).

Note that if flow control propagates upwards to the stream head, threads issuing write system calls will eventually block waiting for space. However, if flow control propagates downwards to a streams driver serving as an interface to a real device, then incoming data will be lost since there is no more buffer space and no notion of blocking the source of the data.

7–13. Message Flow

Stream Service Procedures



.H2596 7-13. 265 © 1991 Hewlett-Packard

Stream Service Procedures

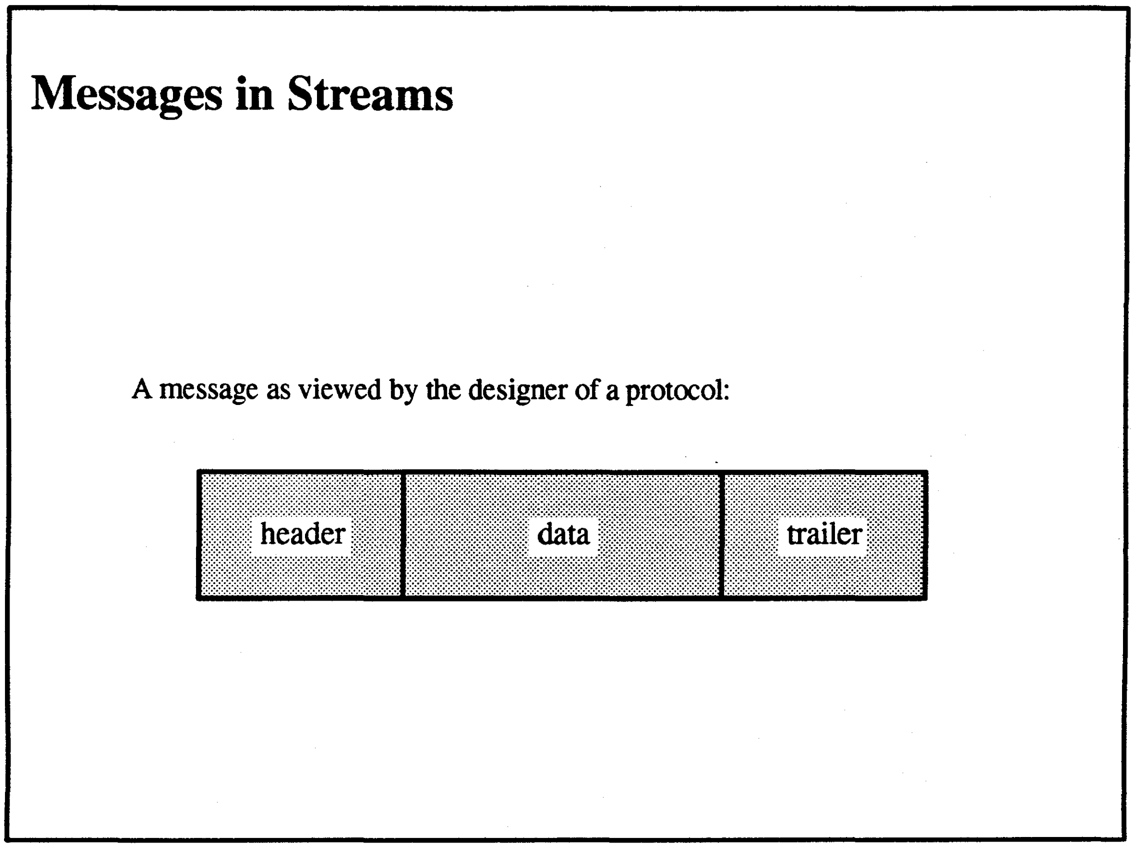
A module that may defer the processing of a message must have a *service procedure* that can be called when further processing is possible. Processing may be deferred because of:

- flow control
- yielding to more important processing
- shortage of buffer space

To cause a service procedure to be called sometime in the future, it is *enabled*, which puts it on a list of enabled service procedures. Enabled service procedures are called by members of a special streams thread pool.

A service procedure is automatically enabled whenever *putq* is called (and the queue was previously empty) and whenever a *getq* is called that removes sufficient messages from a forward queue that the queue size falls below the queue's *low-water mark.*

7–14. Message Flow



Messages in Streams

A message as viewed by the designer of a protocol:

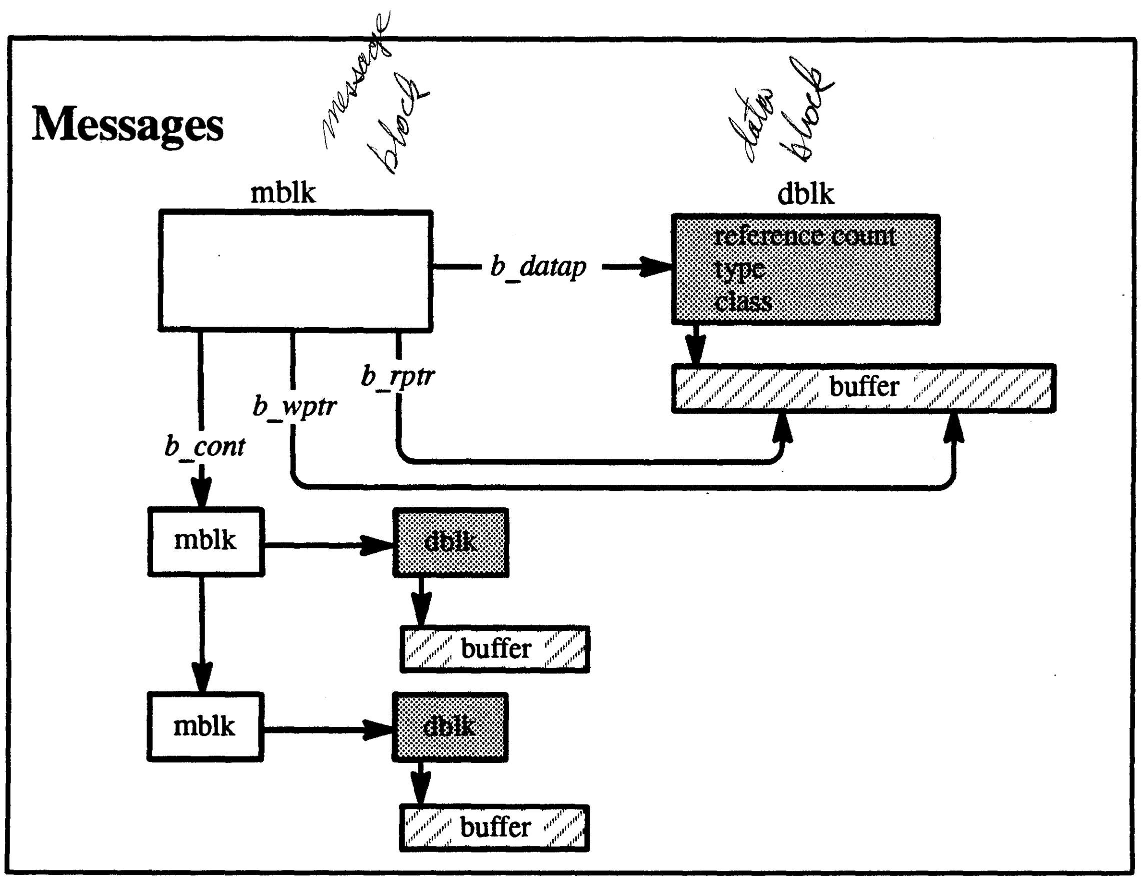
header
data
trailer

Messages in Streams

The simplest stream module merely passes its input to the next module in the stream. This transfer is very efficient, since the data is represented by a linked list of message blocks and is passed by reference. A message consists of a linked list of message blocks, each of which points to a data block, each of which points into a variable-length buffer. The justification for the data blocks is that they allow multiple message blocks to refer to the same data block and avoid the overhead of copying (the data block contains a reference count). Representing a message as a sequence of message blocks makes it easy, for example, to add headers or trailers to messages (and to strip them off).

The message shown in the picture would be implemented as a linked list of three submessages, one for the header, one for the data, and one for the trailer. Stripping off the header and trailer or appending additional information is then very easy.

7–15. Message Flow



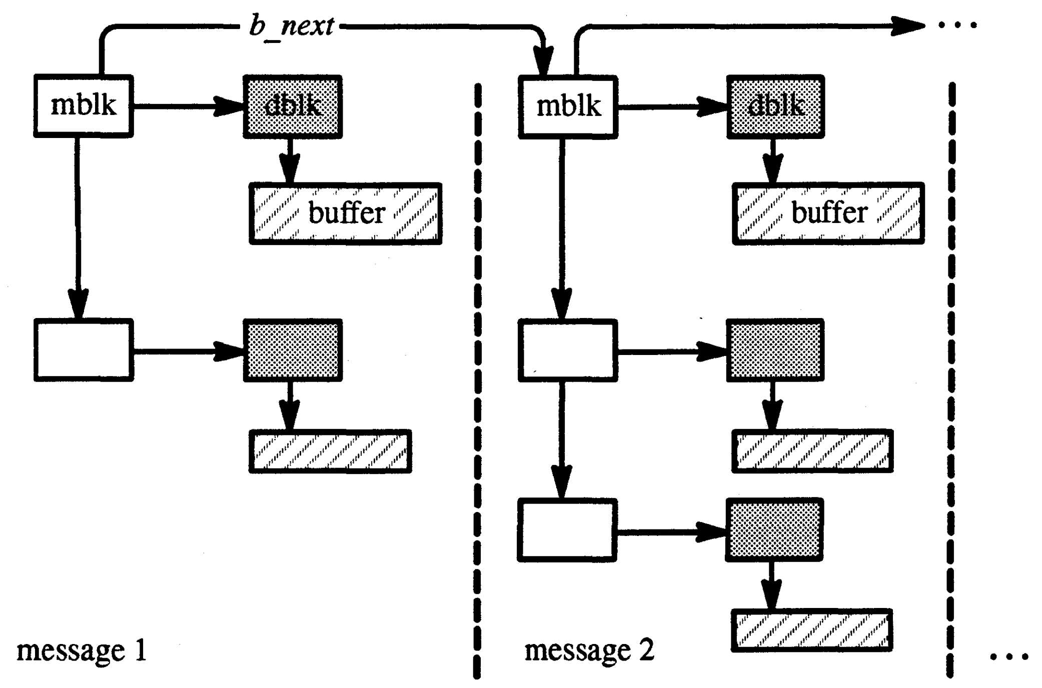
H2596 7-15. 267 © 1991 Hewlett-Packard

Messages

A message is represented as a linked list of *mblks*. Each *mblk* indirectly refers to a buffer via a *dblk*. The *dblk*, as is discussed further on page 7-20, contains the reference count and other information about the buffer. The *mblk* contains a pair of pointers pointing directly into the buffer. These pointers allow the easy representation of consuming data from the buffer and putting data into the buffer.

7–16. Message Flow

Message Queue



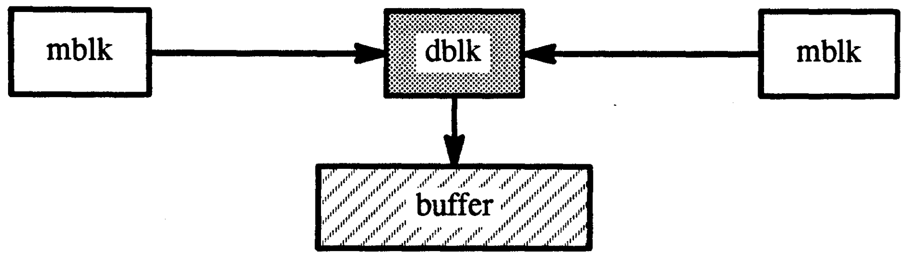
.H2596 7-16. 268 © 1991 Hewlett-Packard

Message Queue

A queue of messages is represented by linking together the first *mblks* of each message.

7–17. Message Flow

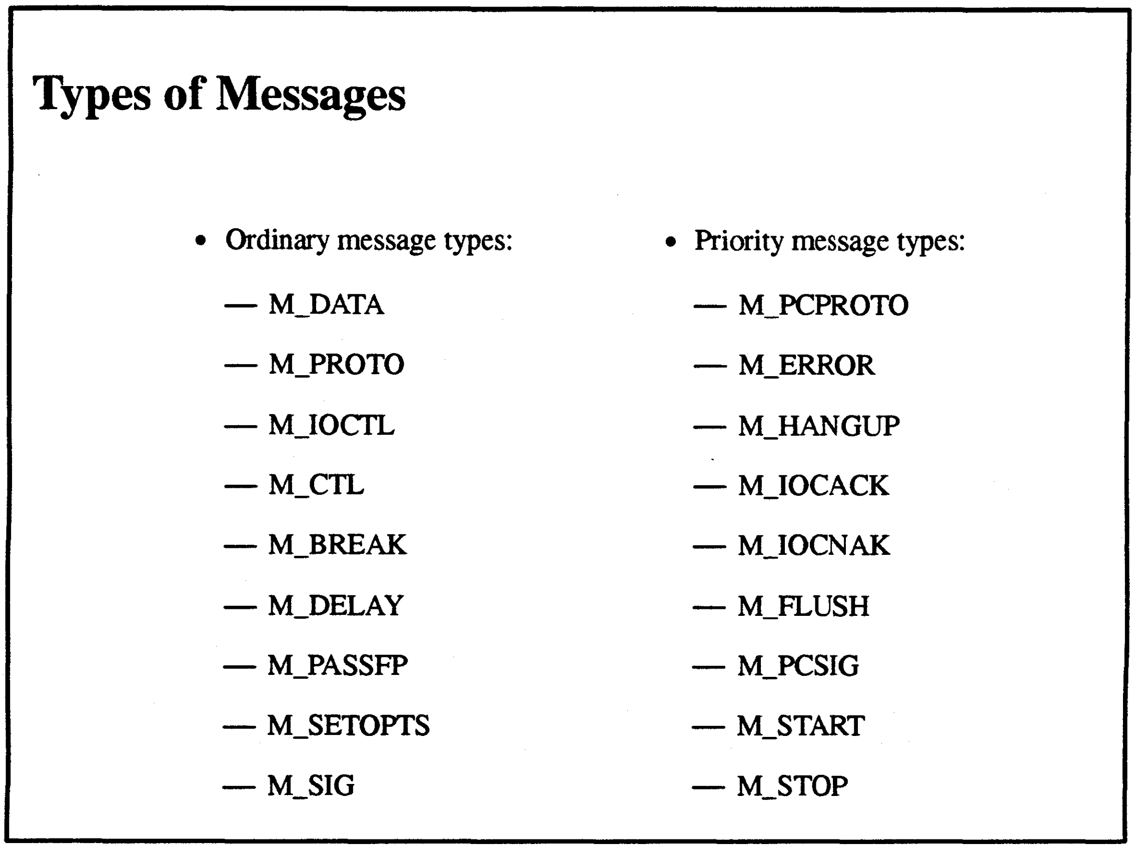
Virtual Copy (Streams Style)



Virtual Copy (Streams Style)

In many situations, it is necessary for a streams module both to pass a reference to data and to retain a reference to the same data. For example, the TCP protocol would send data to be transmitted to the IP protocol, but would retain a copy of the data just in case it is never acknowledged and thus must be retransmitted. This "virtual copy" is implemented with reference counts: each buffer's *dblk* contains the buffer's reference count. Creating a virtual copy of the buffer merely involves incrementing the reference count. Freeing a buffer causes its reference count to be decremented; if the reference count is reduced to 0, then the storage is actually liberated.

7–18. Message Flow



Types of Messages

- Ordinary message types:
  - — M_DATA
  - — M_PROTO
  - — M_IOCTL
  - — M_CTL
  - — M_BREAK
  - — M_DELAY
  - — M_PASSFP
  - — M_SETOPTS
  - — M_SIG
- Priority message types:
  - — M_PCPROTO
  - — M_ERROR
  - — M_HANGUP
  - — M_IOCACK
  - — M_IOCNAK
  - — M_FLUSH
  - — M_PCSIG
  - — M_START
  - — M_STOP

.H2596 7-18. 270 © 1991 Hewlett-Packard

Types of Messages

Each message block is assigned a type that streams modules use to determine what sort of processing is required. There are two classes of messages: those that are subject to flow control (*ordinary messages*) and those that are not (*priority messages*).

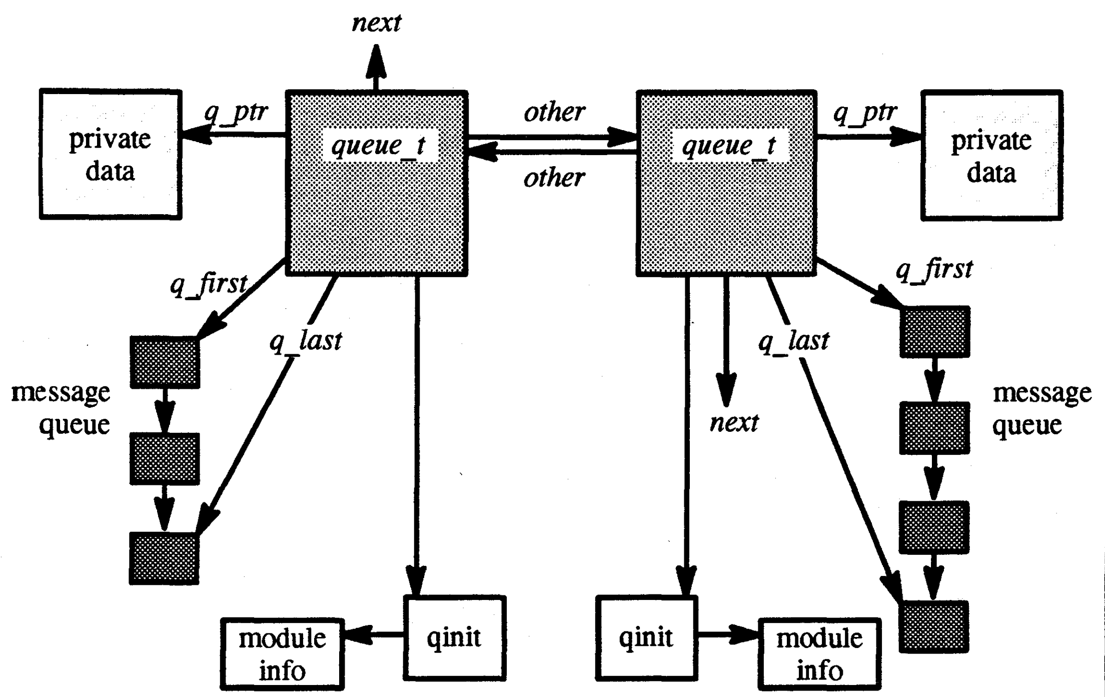
Since priority messages are not subject to flow control, they are forwarded even if subsequent queues are full.

For example, an ioctl system call might be implemented within a streams component. In this case, a message is created whose first *mblk* points to a *dblk* of type M_IOCTL, which points to a buffer containing the command. Subsequent *mblk*s of this message point to *dblk*s of type M_DATA, which point to buffers containing the associated data. This message is passed down the pipeline until it reaches a component that recognizes the ioctl request. This component performs the desired action and sends back a priority message whose first *mblk* points to a *dblk* of type M_IOCACK. Being a priority message, it is passed up to the stream head immediately without being queued, and thus it is dealt with at the stream head before any ordinary messages that might be in the queues. This prioritized delivery is necessary because the thread performing the ioctl system call does not move on to make, for example, read system calls until the ioctl system call completes.

If no component recognizes the ioctl request, then the last component, a streams driver, sends back a priority message of type M_IOCNAK.

7–19. Implementation of Streams

Queue Structure



.H2596 7-19. 271 © 1991 Hewlett-Packard

Queue Structure

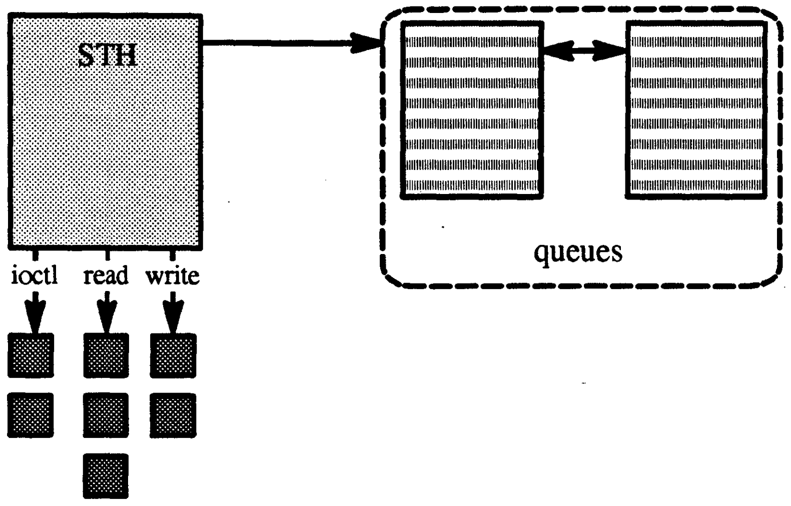
Each instance of a streams component, whether stream head, module, or driver, is represented by a pair of *queue_t* structures. Each such structure links the component to the next component in the pipeline, points to the associated queue of messages, and points to two types of data structures.

The first type of data structure, the *qinit* structure, contains information shared by all instances of the component. It contains the addresses of the *put* and *service* procedures for either the read or write half of the component and points to the *module info structure*, which gives the default values for various parameters.

The second type of data structure, pointed to by *q_ptr*, contains information that is private to each instance of a component. The contents of this data structure depend upon the component.

7–20. Implementation of Streams

Stream Head



Stream Head

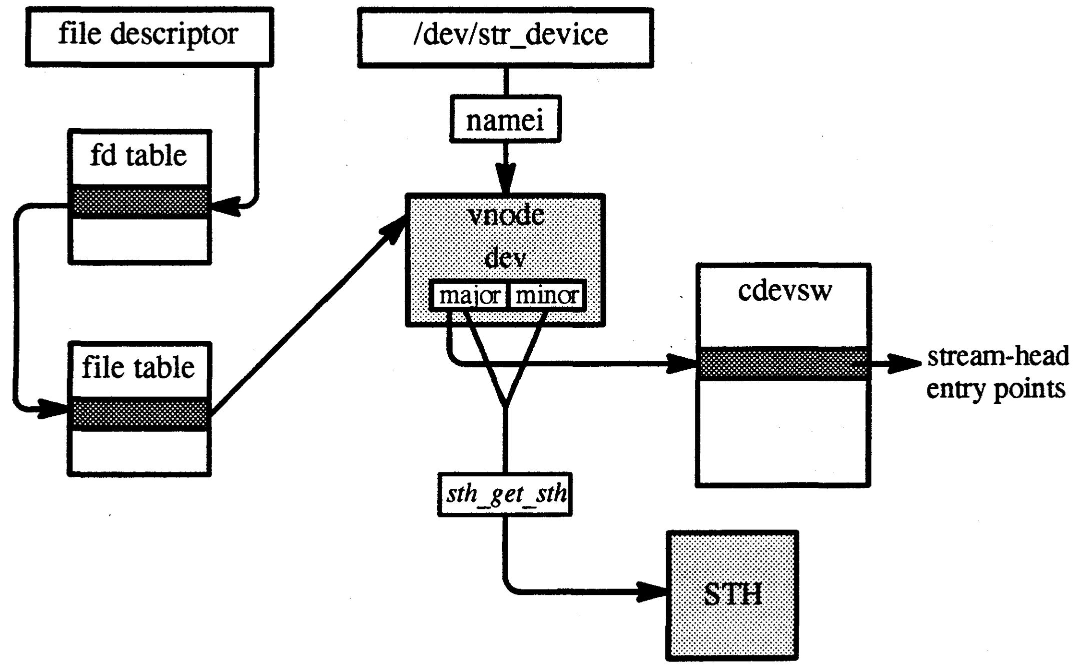
The *stream-head module* provides the interface between system calls and the stream. It is the only portion of streams in which threads may block: a thread making a system call must necessarily block until it can return.

The module is represented by a STH (stream head) structure. It contains various pieces of information about the stream and refers to the stream-head queue data structures.

Multiple threads may access the stream concurrently. Three types of system calls involve messages and streams: ioctls, reads and getmsgs, and writes and putmsgs. A user thread (executing a streams system call) creates an operating system request (OSR) structure to represent the system call. If the request cannot be satisfied immediately, then the OSR is queued on one of the STH's OSRQs (one queue for each system call type).

7–21. Implementation of Streams

Representing an Open Stream



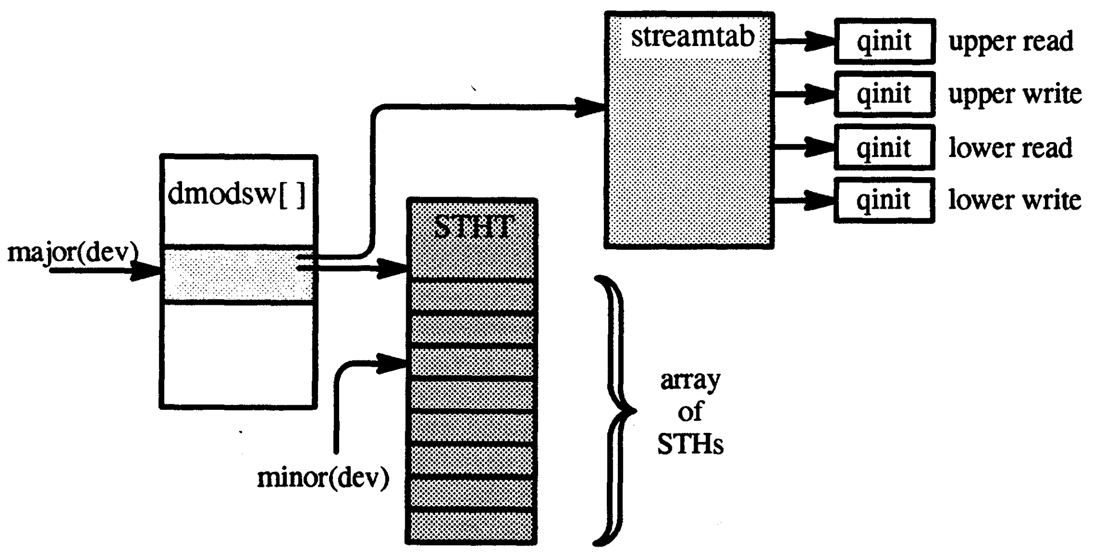
H2596 7-21. 273 © 1991 Hewlett-Packard

Representing an Open Stream

An open stream appears to the rest of the operating system as if it were a character special file. The *cdevsw* entry contains not the addresses of the streams driver's entry points but the entry points of the more general stream-head routines. This is important since the interface to the system-call layer is at the stream head and not at the streams driver. The stream-head code itself finds the desired stream by accessing the STH structure.

7-22. Implementation of Streams

Device Module Switch Table



H2596 7-22. 274 © 1991 Hewlett-Packard

Device Module Switch Table

The device module switch table is an array of *dmodsw* structures, one for each type of streams driver. It is accessed by the major portion of the device number and refers to the device's *streamtab* structure. This structure contains the addresses of the *qinit* structures for each portion of the device (the device may be a multiplexer, so the *streamtab* structure refers to the *qinit* structures for the lower-side queues as well).

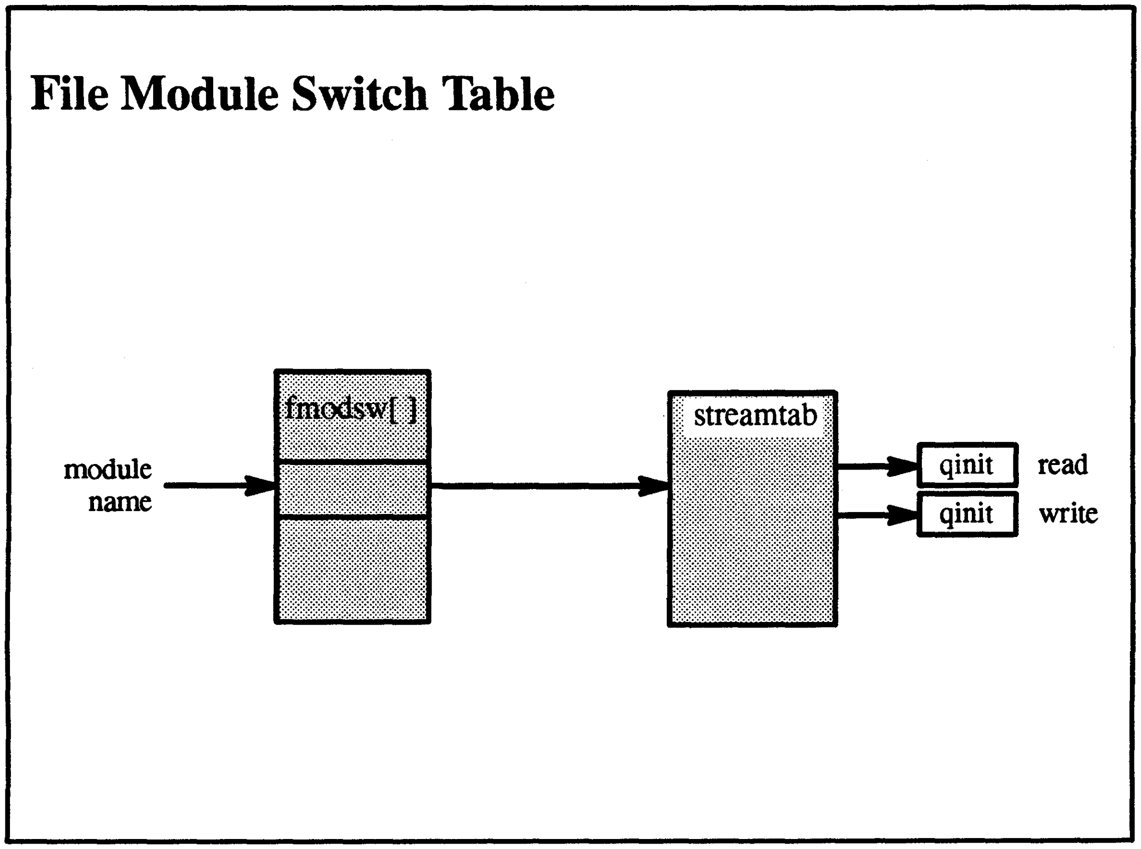
Each *dmodsw* entry also refers to an STHT (stream header table) structure, which is an array of STH structures, one per minor device. The STH structure, as we have seen, represents the stream head of a particular stream and is allocated when the stream (i.e. minor device) is opened.

Each stream appears to the rest of the operating system as if it were a character-special device. Thus associated with each streams device is an entry in the *cdevsw* table. However, this entry refers not to the entry points of the streams driver, but to the general stream-head entry points. Additional information is needed to find the entry points of the streams device itself and to find the STH structure identifying the specific instance of the stream.

This information is obtained from the *dmodsw* structure as we have just seen. The major portion of the device number is used twice: once to index the *cdevsw* table to determine that this is a stream, and again to index the *dmodsw* table to find the *streamtab* data structure and the STHT. Finally, the minor portion of the device number is used to index the STHT to find the STH that identifies the actual instance of the stream.

7–23. Implementation of Streams

File Module Switch Table

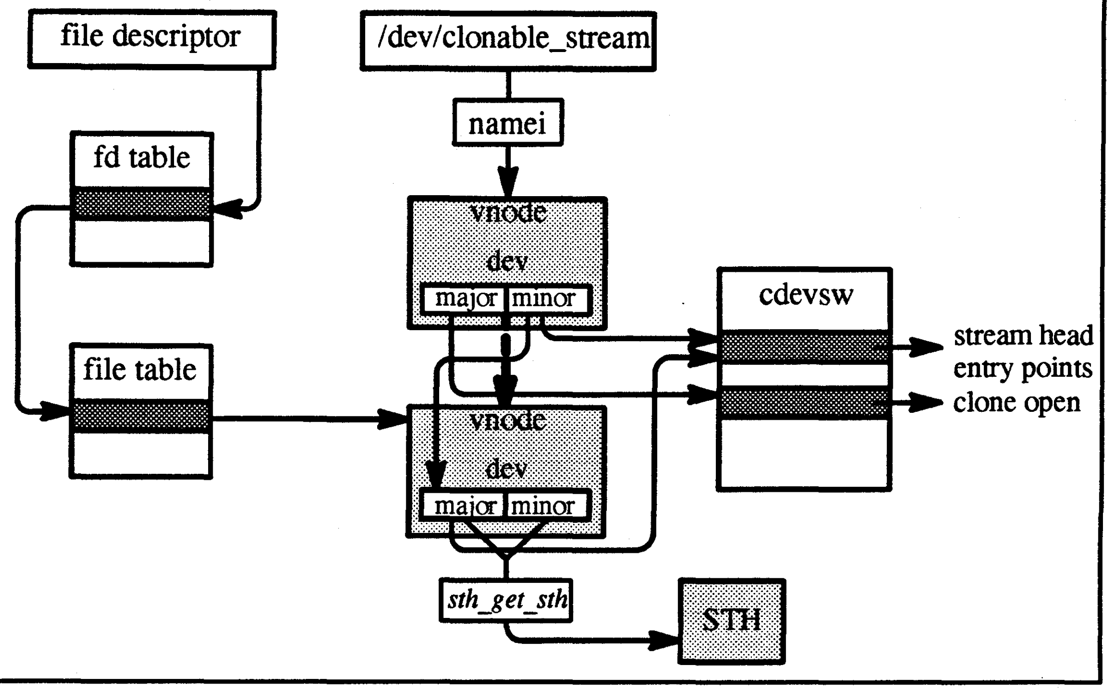


File Module Switch Table

The *file module switch table* is an array of *fmodsw* structures. It is accessed by the module name and refers to each module's streamtab structure, which in turn refers to the module's *qinit* structures. Thus individual streams modules are identified through the file module switch table.

7–24. Implementation of Streams

Cloning a Stream



.H2596 7-24. 276 © 1991 Hewlett-Packard

Cloning a Stream

The notion of *cloning* is an important concept that was introduced in SVR3 for the support of streams and has been extended in OSF/1 for use in any character-special file. Cloning is used in situations in which there is a varying number of logical devices.

For example, each connection over a streams implementation of TCP/IP is a logical device; each such logical device is represented by a separate minor device. Since the user of the logical device does not really care which minor device member is chosen, it is inconvenient to represent each logical device as a separate special file (in the /dev directory). Cloning allows one to use just one special file to represent the major device, and to have the minor device numbers automatically produced internally in response to opening the major device.

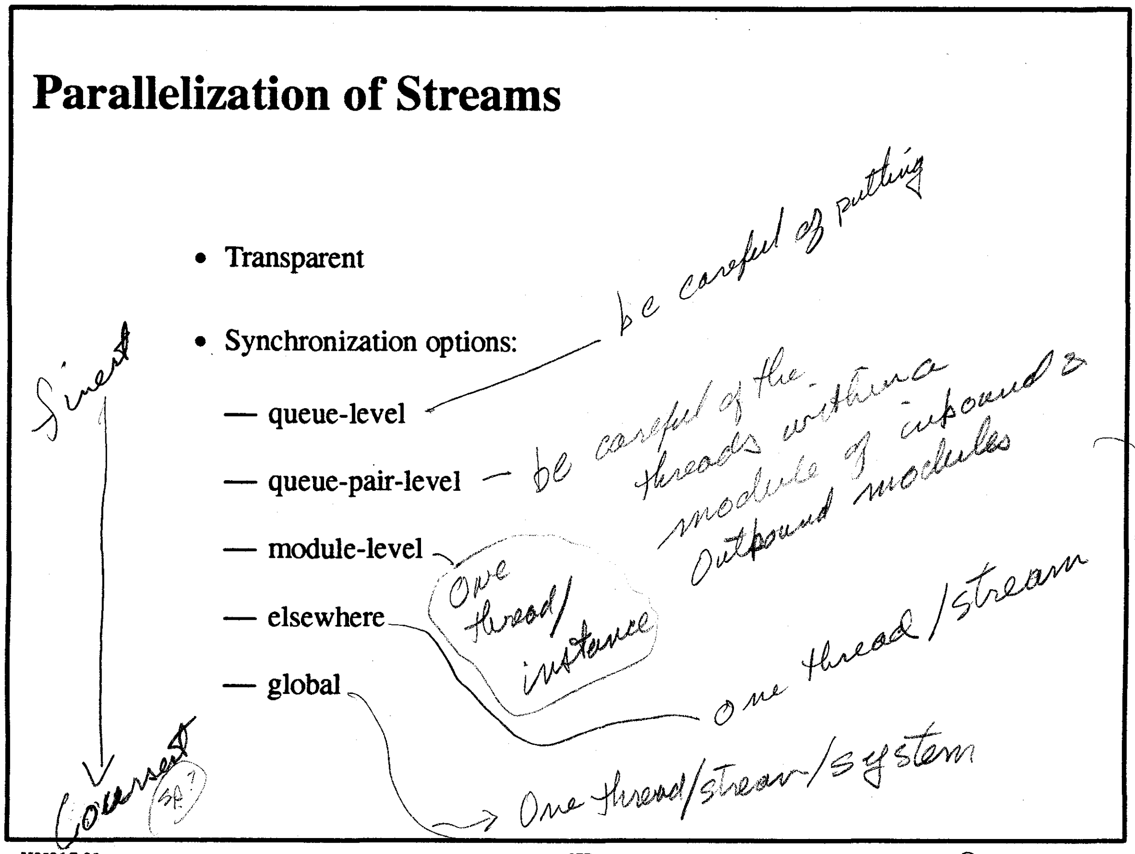
A clonable device is represented by a special file whose major device number is that of the *clone driver* and whose minor device number is equal to the major device number of the clonable device. Opening this special file results in a call to the open entry point of the clone driver, *clone_open*, which returns the error code ECLONEME.

The caller of this routine, *spec_open,* then calls *spec_clone,* which creates a new vnode, and then calls the open routine given in *cdevsw* as indexed by the original minor device number (i.e., the major device number of the clonable device).

For the case of streams, the call to open results in a call to *osr_open,* which sets up a stream head and the device driver module, and calls the device driver *open* routine with the clone flag set. The device driver then finds an available minor device number and returns it to the caller and the caller's caller, and so forth, who will eventually put this in the new vnode. The effect then is that all further I/O requests use the newly created vnode and access the newly created stream.

OSF/1 Specific - and the rest of the slides in this section

7–25. Parallelization



Parallelization of Streams

- Transparent
- Synchronization options:
  - — queue-level
  - — queue-pair-level
  - — module-level
  - — elsewhere
  - — global

be careful of putting

be careful of the threads within a module of inbound & outbound modules

One thread/instance

one thread/stream

One thread/stream/system

Finest

Coarsest (SP?)

Active at any time.

H2596 7-25. 277 © 1991 Hewlett-Packard

Parallelization of Streams

Streams are parallelized in a fully transparent manner: streams modules from an SVR3 system can be put into an OSF/1 kernel so as to allow fully parallel execution with essentially no changes to the code. Certainly some synchronization is necessary. This synchronization is implemented within the standard routines that are called for communication between streams and modules. The finest degree of parallelization is at the individual-queue level (SQLVL_QUEUE). Associated with the queue is a lock; only one thread may execute within the queue at a time. One may also request synchronization across a pair of queues within a module (SQLVL_QUEUEPAIR), across all instances of a module (SQLVL_MODULE), and across a group of modules (SQLVL_ELSEWHERE). For debugging purposes, one may request a single lock for the entire streams system, i.e. only one thread at a time may be executing anyplace (SQLVL_GLOBAL).

We first look at queue-level synchronization. Whenever a module is entered (e.g. via *putnext)*, a check is made to see if the queue is locked. If so, then instead of waiting for the queue to be unlocked, the request is queued on a *synchronization queue*, and the call returns immediately. Thus while a thread operates within a module, further requests to enter that module queue up on the synchronization queue. When the thread leaves the module, it must check if there are any requests in the synchronization queue and then handle each of these requests as if it had made them itself.

The synchronization queue is headed by an SQH data structure, which, for queue-level synchronization (SQLVL_QUEUE), is in the queue data structure. This contains a pair of simple locks, one that is the lock on the entire queue, and the other that is the lock for operations on the synchronization queue. Each request in the queue is represented by an SQ data structure, which refers to the queue being accessed, the routine being called, and the message being transmitted. The SQ data structure itself is allocated within the *mblk* data structure.

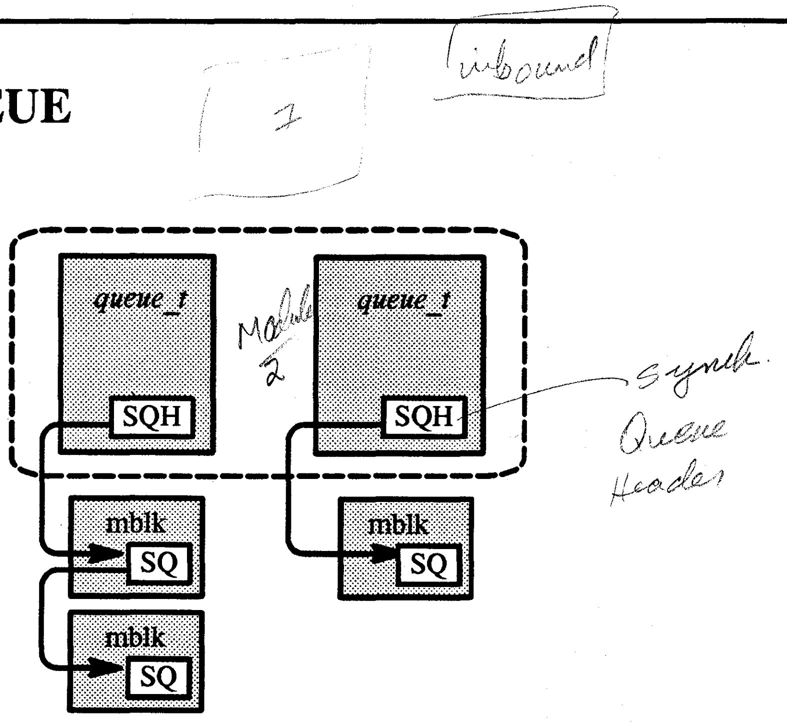
The technique for single-queue parallelization can be extended for coarser parallelization. To achieve queue-pair synchronization (SQLVL_QUEUEPAIR), one effectively merges the synchronization queues of the two individual queues. This is accomplished through the use of the *sq_parent* field of the SQH data structure. This field normally points to the SQH data structure itself, but it may point to another SQH data structure. In the latter case, the target SQH structure is used instead of the source SQH structure. So, for queue-pair parallelization, the write-side queue's SQH structure points to that of the read-side queue, and there is one set of blocks and one synchronization queue for both queues of the module.

For module-wide synchronization (SQLVL_MODULE), each queue in each instance of the module refers to an SQH structure in the module's *fmodsw* structure (or the *dmodsw* structure if it is a driver).

Queue Level

7–26. Parallelization

SQLVL_QUEUE



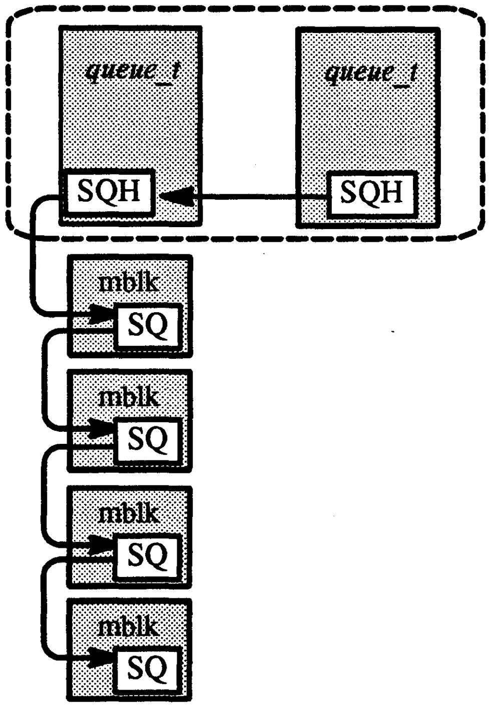
.H2596 7-26. 278 © 1991 Hewlett-Packard

SQLVL_QUEUE

In this picture queue-level synchronization is used. There is space in each *queue_t* for an SQH structure, which heads the queue's synchronization queue. For queue-level synchronization, each of the synchronization queues is totally independent of the others. Thus each queue may have no more than one thread active at a time, but there may be two active threads within each instance of the streams component.

7–27. Parallelization

SQLVL_QUEUEPAIR

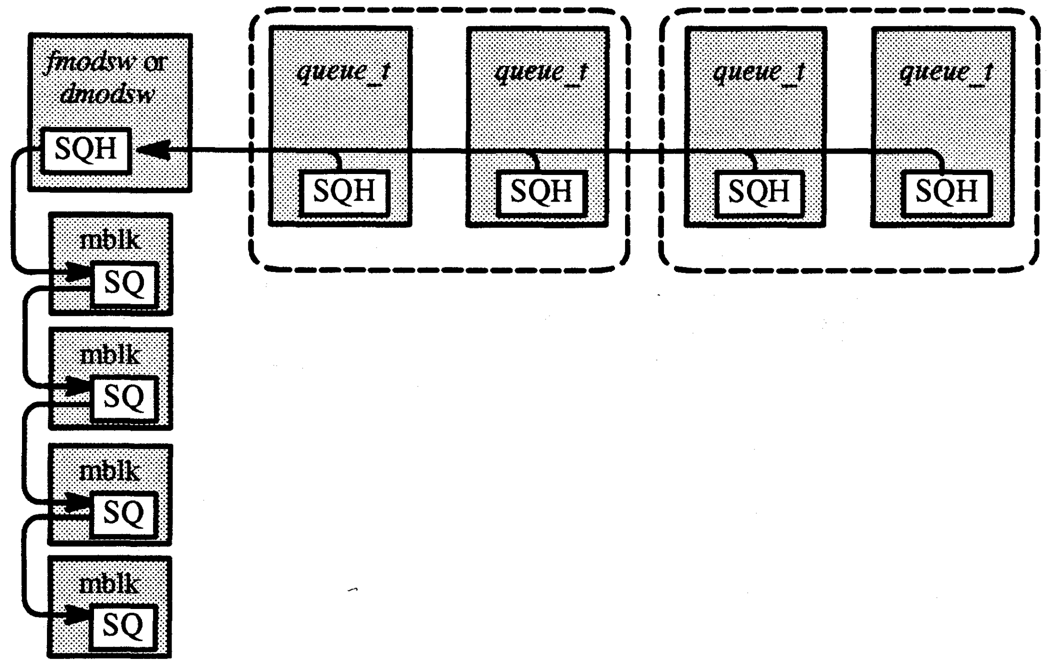


SQLVL_QUEUEPAIR

This picture illustrates queue-pair-level synchronization. The *sq_parent* field of one queue's SQH structure points to the other SQH structure, effectively merging the two synchronization queues. Thus we are assured that only one thread can be in either queue at a time.

7–28. Parallelization

SQLVL_MODULE



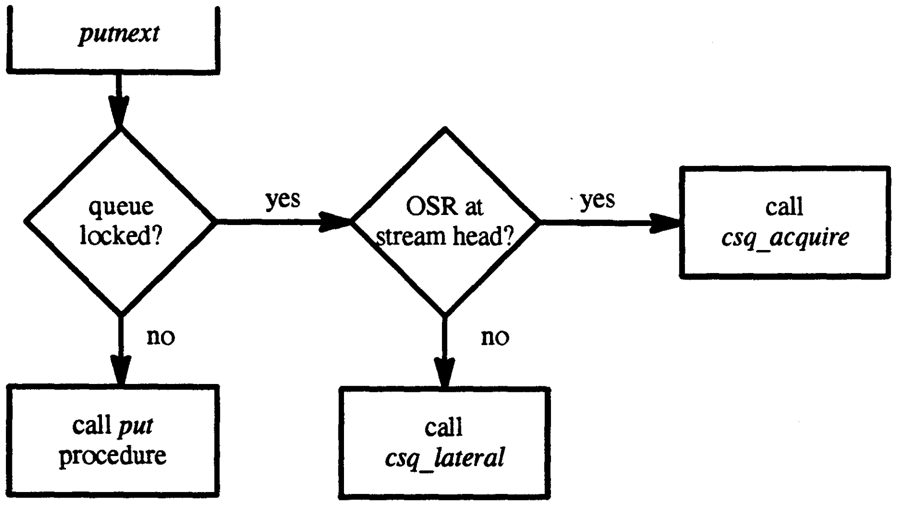
.H2596 7-28. 280 © 1991 Hewlett-Packard

SQLVL_MODULE

This picture shows module-level synchronization. The *sq_parent* fields of each queue's SQH structure points to the SQH structure stored in the *fmodsw* structure (modules) or the *dmodsw* structure (drivers). Thus the synchronization queues for all queues within all instances of a module are effectively merged: at most one thread can be active in any instance of the module at a time.

7–29. Parallelization

Implementation of Synchronization Queues



H2596 7-29. 281 © 1991 Hewlett-Packard

Implementation of Synchronization Queues

In order to enter a streams module (for example, as part of a *putnext* request), a thread encodes a request in the message's SQ structure and calls the *csq_lateral* routine. This routine checks the lock on the appropriate synchronization queue. If the lock is taken, the routine puts the request on the synchronization queue and returns. When a thread finishes a request inside of the module, as mentioned earlier, it checks the synchronization queue and handles any queued requests.

This arrangement could result in a race condition. The calling thread might determine that the queue is busy, but, before it can enqueue an SQ on the synchronization queue, the thread that owned the streams queue determines that the synchronization queue is empty and releases the lock. Thus the SQ will soon be enqueued, but no thread will be available to process it. To deal with this, *csq_lateral* checks the lock after it has enqueued the SQ, and, if it has been released, then processes the SQ itself.

When a user thread (executing in kernel mode in response to a system call) operating within a stream head calls *putnext* to send a message to the first module, it calls *csq_acquire* instead of *csq_lateral*. This routine blocks waiting for the lock. When the thread holding the lock is finished with its operation, instead of processing the synchronization queue itself, it hands the work over to the user thread blocked in *csq_acquire*.

Module 7 — Streams

Exercises:

1. a. List the three types of streams components.
   b. What happens when one stream is linked to another stream?
   c. What are the functions of a multiplexer?
2. a. What is the purpose of the *dblk* data structure?
   b. How is data prepended to the beginning of a message?
   c. From what routines is a module's *put* procedure called? In whose context are *service* procedures called?
3. a. When are *queue_t* structures allocated?
   b. Explain why it is necessary for streams-oriented system calls to be serialized. What mechanism is used to perform this serialization?
4. a. When a read or write system call is issued to an open stream, at what point does the flow of control first differ from the flow when an ordinary character-special device is called?
   b. Explain the role of the streams device driver in cloning.
5. a. List the synchronization options available in streams.
   b. When is it permissible for a thread to block within a streams call?
   c. Explain how synchronization queues differ from ordinary streams queues.

Advanced Question:

6. Synchronization queues add another set of queues to a streams pipeline. Explain what effects this might have on data in the pipeline. For example, does the use of synchronization queues affect the order of messages? Does it affect the number of messages that may exist within a pipeline?

Module 8—Sockets

Module Contents

Module Objectives

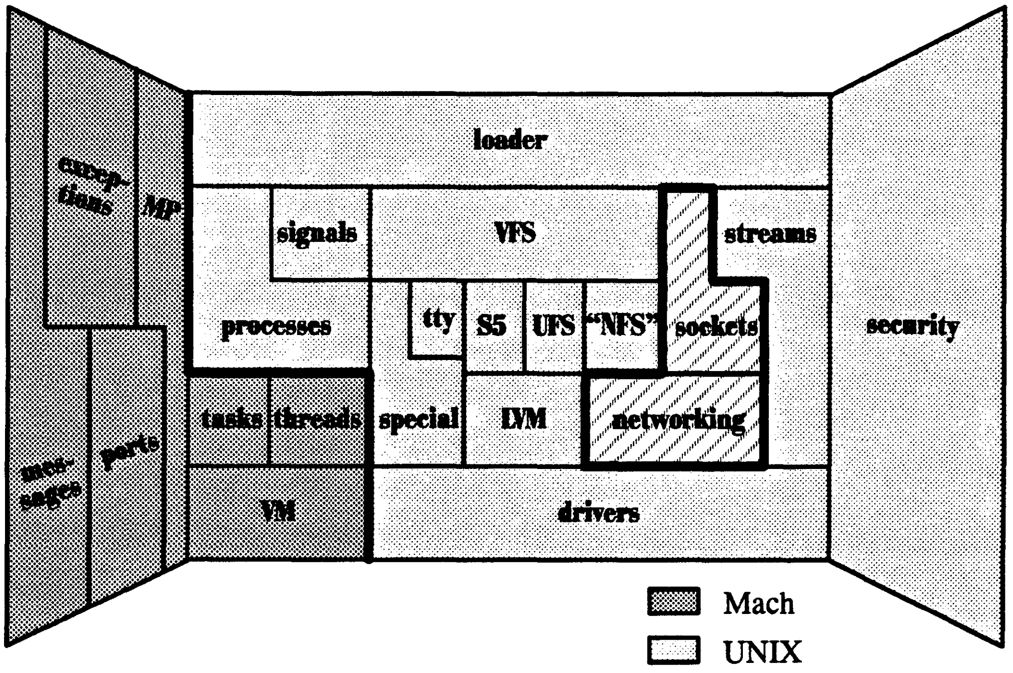
In order to demonstrate an understanding of sockets and their use in supporting networking in OSF/1, the student should be able to:

- explain the difference between communication using datagram sockets and communication using stream sockets
- describe how messages are represented by *mbufs*
- list the set of actions that may be taken when the list of free *mbufs* is exhausted
- explain how kernel threads are used in the socket networking subsystem
- explain how OSF/1 supports both a streams and a socket interface to networking

Module 8 — Sockets

8–1. The Big Picture

Networking in OSF/1



.H2596 8-1. 282 © 1991 Hewlett-Packard

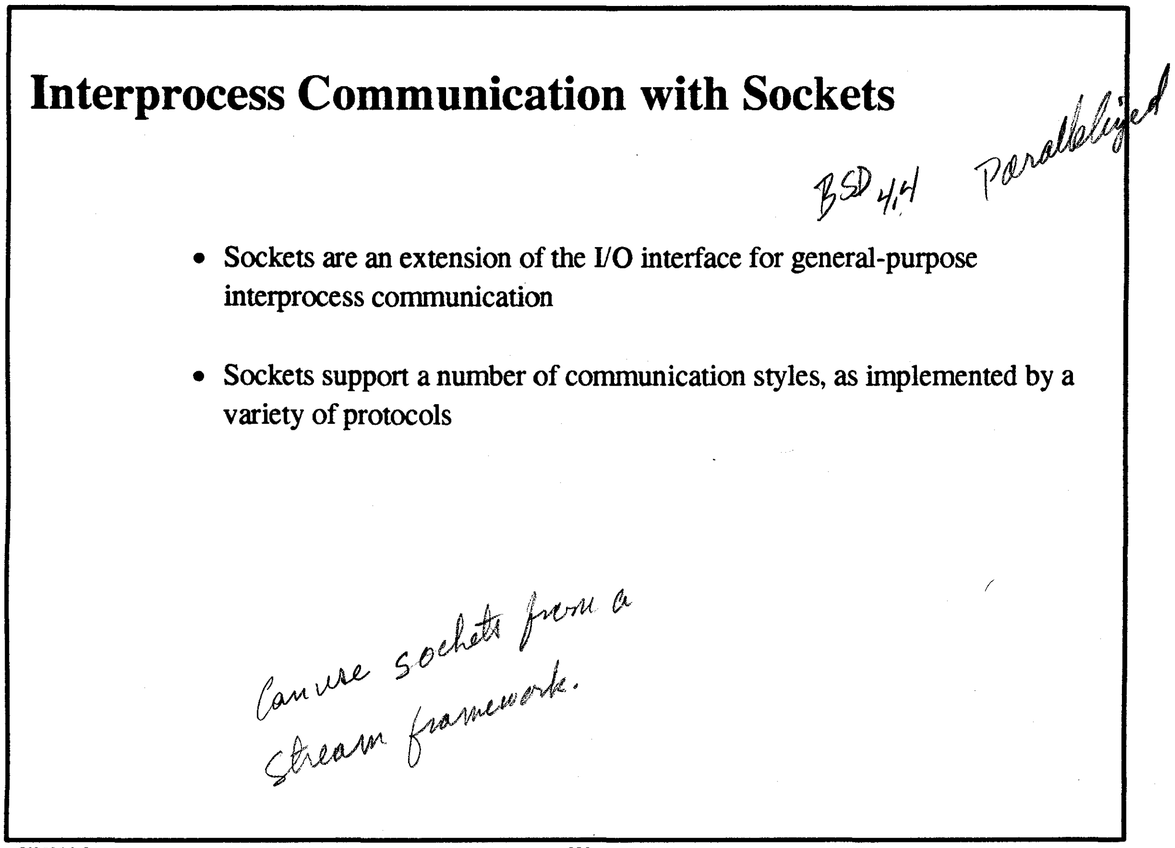
Networking in OSF/1

OSF/1 uses Berkeley's *socket* model to implement its communication protocols. The code is a parallelized version of that in 4.4BSD. The recommended user interface for networking is the X/Open Transport Interface (XTI), which is an enhancement of AT&T's Transport Layer Interface (TLI). While user code can communicate directly with the socket layer, XTI is more likely to be portable.

OSF/1 implements XTI in a user-level library that communicates with the operating system via the streams interface. Since OSF/1 implements the network protocols in the socket framework rather than in the streams framework, the system provides a conversion layer. This conversion layer is a stream whose device driver converts stream requests into socket requests.

Some of the material in this module is discussed in chapter 14 of Open Software Foundation, 1990a.

8–2. Sockets



Interprocess Communication with Sockets

BSD 4.4 Parallelized

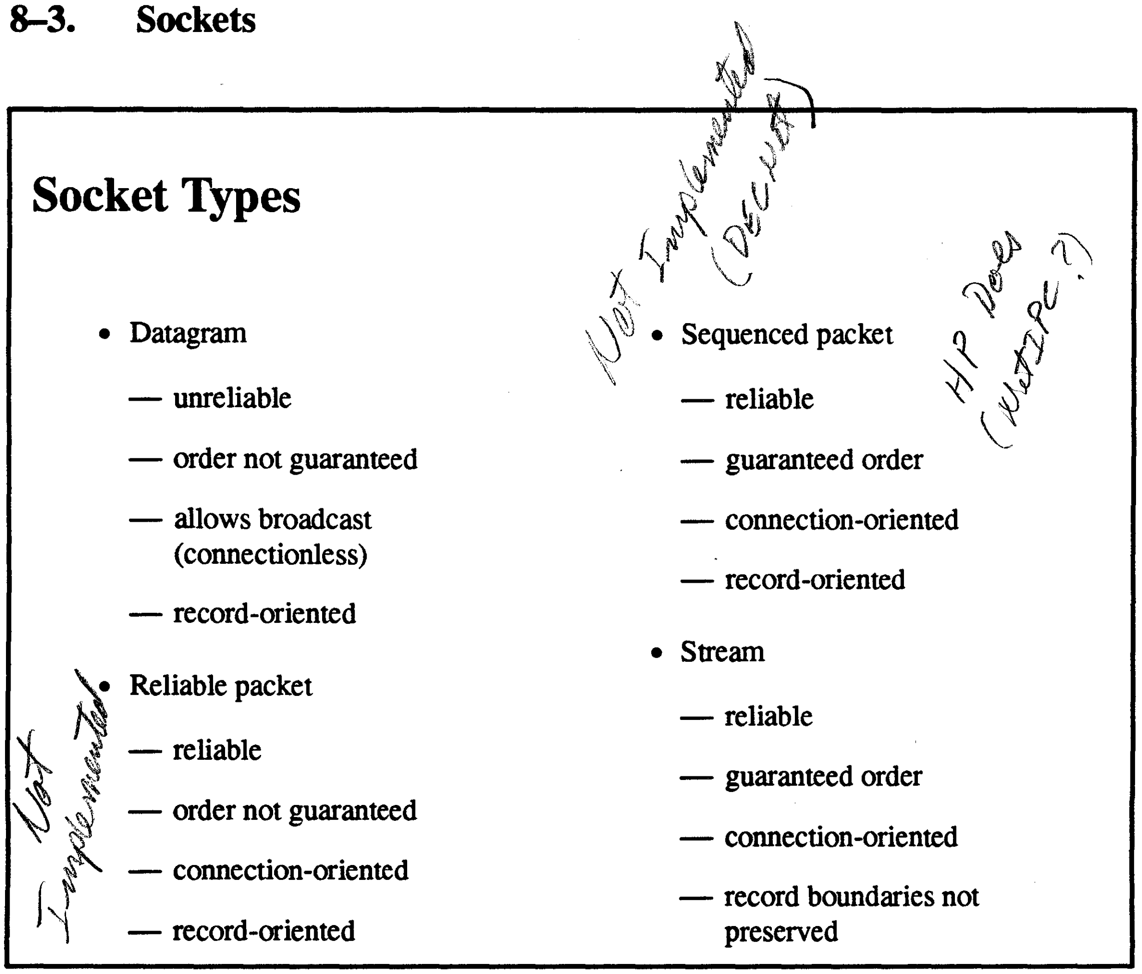
- Sockets are an extension of the I/O interface for general-purpose interprocess communication
- Sockets support a number of communication styles, as implemented by a variety of protocols

Can use sockets from a stream framework.

H2596 8-2. 283 © 1991 Hewlett-Packard

Interprocess Communication with Sockets

8–3. Sockets



Socket Types

- Datagram
  - — unreliable
  - — order not guaranteed
  - — allows broadcast (connectionless)
  - — record-oriented
- Reliable packet
  - — reliable
  - — order not guaranteed
  - — connection-oriented
  - — record-oriented

Not Implemented

- Sequenced packet
  - — reliable
  - — guaranteed order
  - — connection-oriented
  - — record-oriented

Not Implemented (DECnet)

HP Does (NetIPC?)

- Stream
  - — reliable
  - — guaranteed order
  - — connection-oriented
  - — record boundaries not preserved

.H2596 8-3. 284 © 1991 Hewlett-Packard

Socket Types

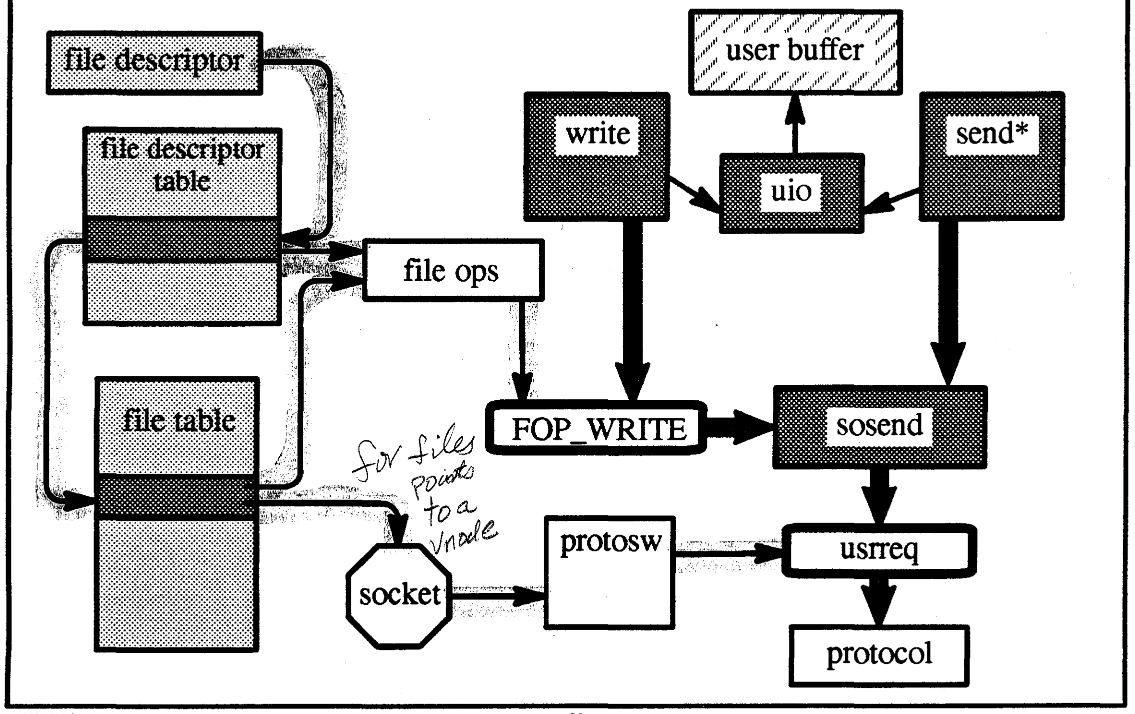
OSF/1 includes only protocols supporting datagram and stream-type sockets. The UNIX domain contains both a datagram and a stream protocol, as does the Internet domain. (UDP is the datagram protocol; TCP is the streams protocol.)

Flow of Control

8–4. Sockets

Data pointers

Writing with Sockets

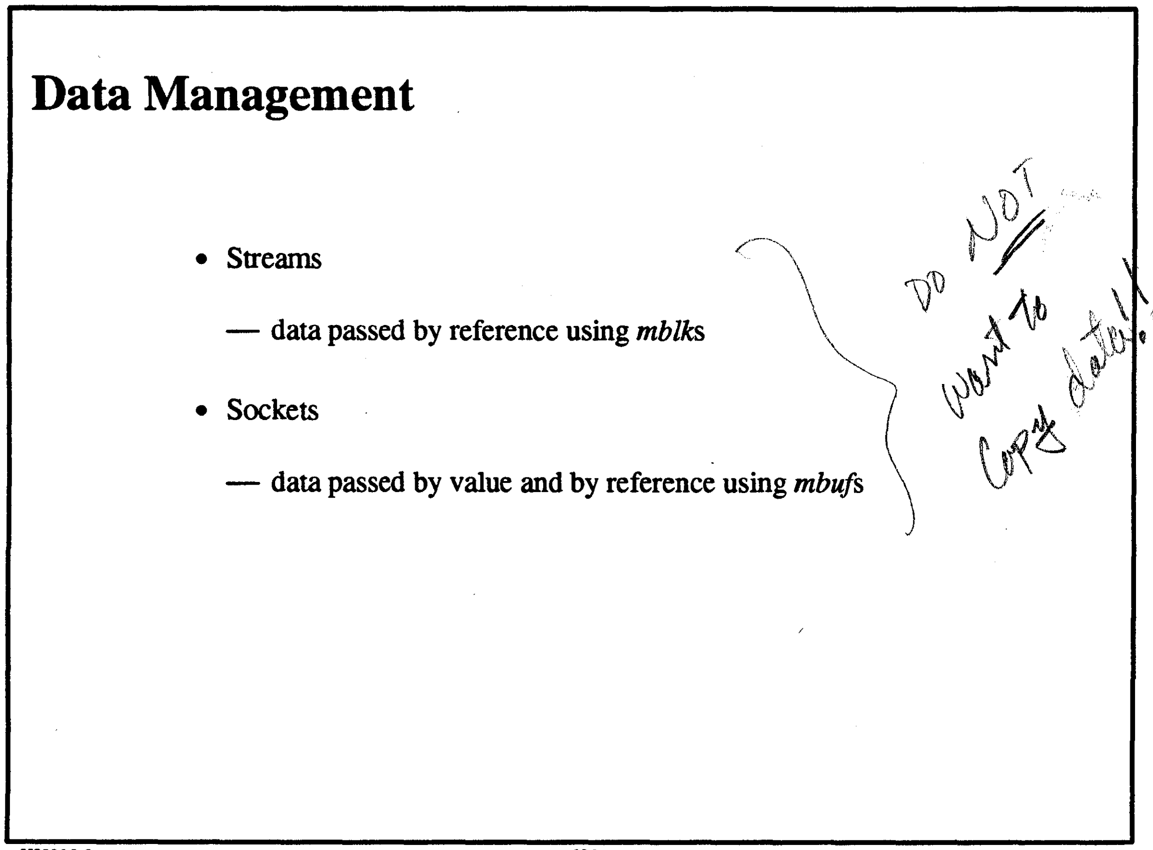


Writing with Sockets

In Module 5 we saw that the *fileops* array contained the vector of entry points for operations on files. In this case it contains the vector of entry points for operations on sockets.

The *protosw* contains the entry points into the selected protocol.

8–5. Mbufs



Data Management

- Streams
  - — data passed by reference using *mblks*
- Sockets
  - — data passed by value and by reference using *mbufs*

Do NOT want to copy data!!

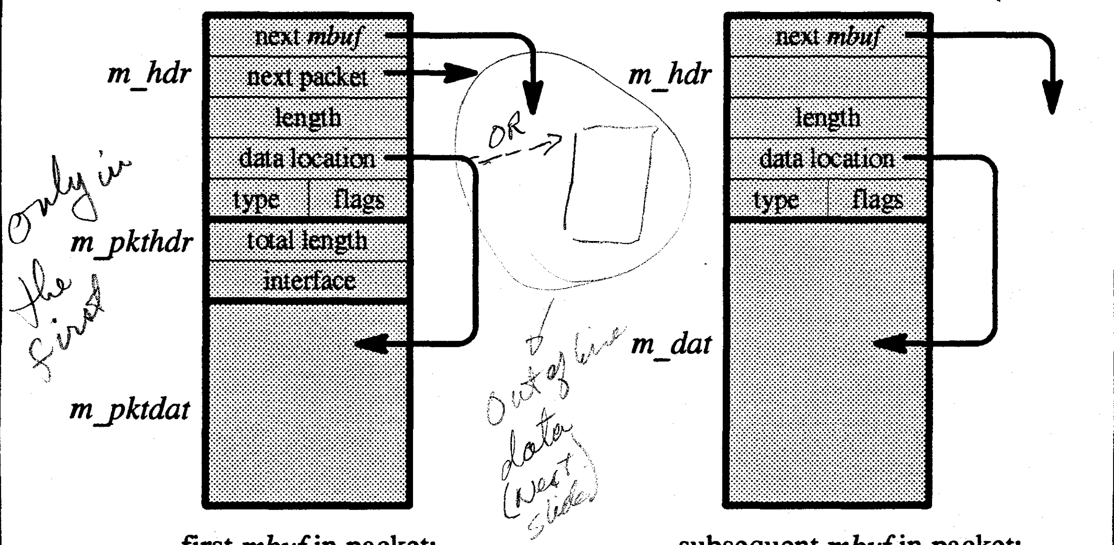
.H2596 8-5. 286 © 1991 Hewlett-Packard

Data Management

As with streams, data structures must be provided to facilitate the efficient movement of data through the various layers of the system. The data structures used with sockets, known as *mbufs*, are very similar to the data structures used with streams *(mblks)*. The major difference is that while with *mblks* all data is passed by reference, with *mbufs* small amounts of data can be passed by copying (though larger amounts of data are passed by reference).

8–6. Mbufs

The mbuf Structure, part 1



first *mbuf* in packet;
data in *mbuf*

subsequent *mbuf* in packet;
data in *mbuf*

H2596 8-6. 287 © 1991 Hewlett-Packard

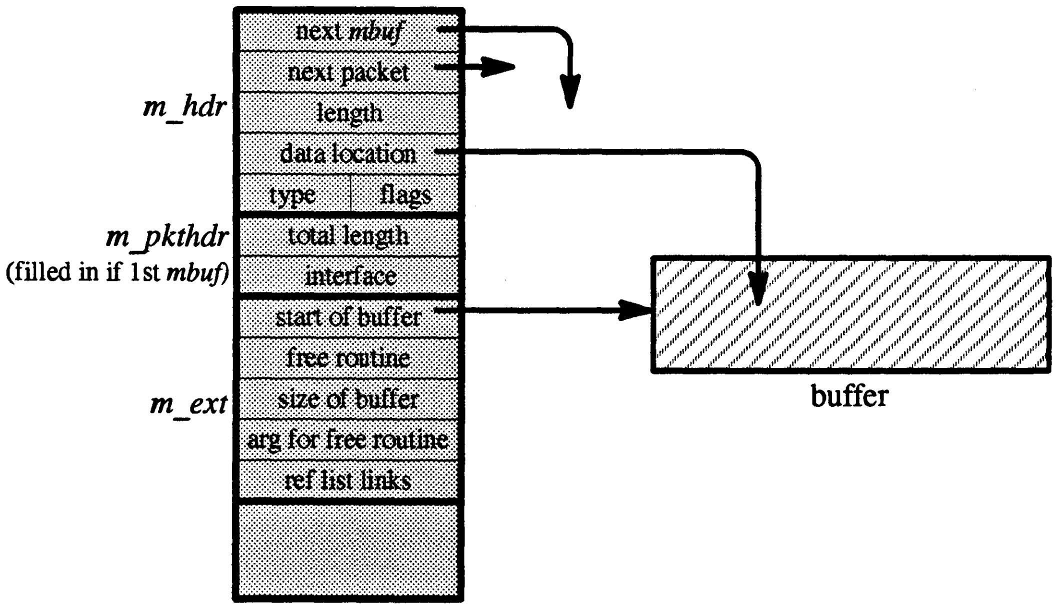
The mbuf Structure, part 1

Mbufs either contain a small amount of data (up to 108 bytes) or refer to a larger amount of data. As with *mblks*, a packet is represented by chaining a sequence of *mbufs*, and a queue of packets is represented by chaining the first *mbuf* of each packet. The header of each *mbuf*, besides containing the links in the various chains, describes the contents of the *mbuf* and includes a pointer to the first byte of data, wherever it may be. The first *mbuf* contains additional information: the length of the entire packet and possibly a reference to the network interface from which the packet came, or to which it is going.

8–7. Mbufs

No Reference Count!

The mbuf Structure, part 2



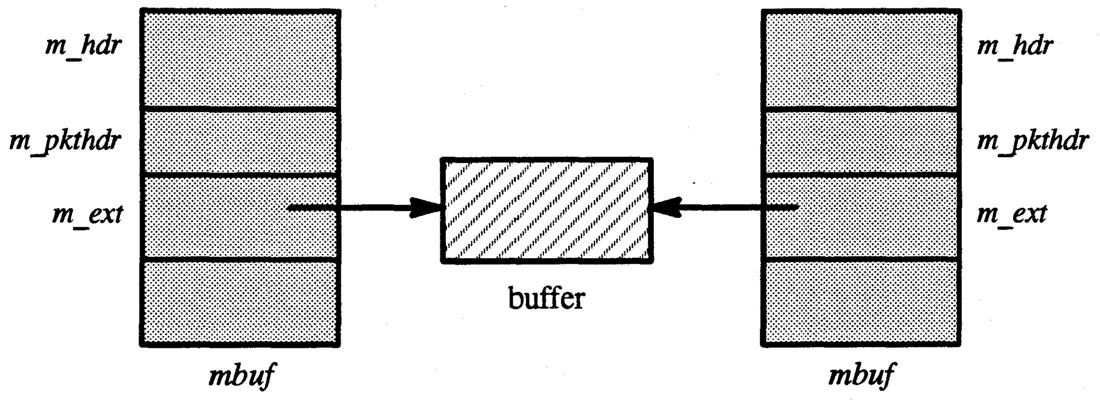
.H2596 8-7. 288 © 1991 Hewlett-Packard

The mbuf Structure, part 2

If an *mbuf* does not contain its data but instead refers to it, then the *mbuf* contains an *m_ext* structure to refer to the buffer.

8–8. Mbufs

Virtual Copy (Socket Style)



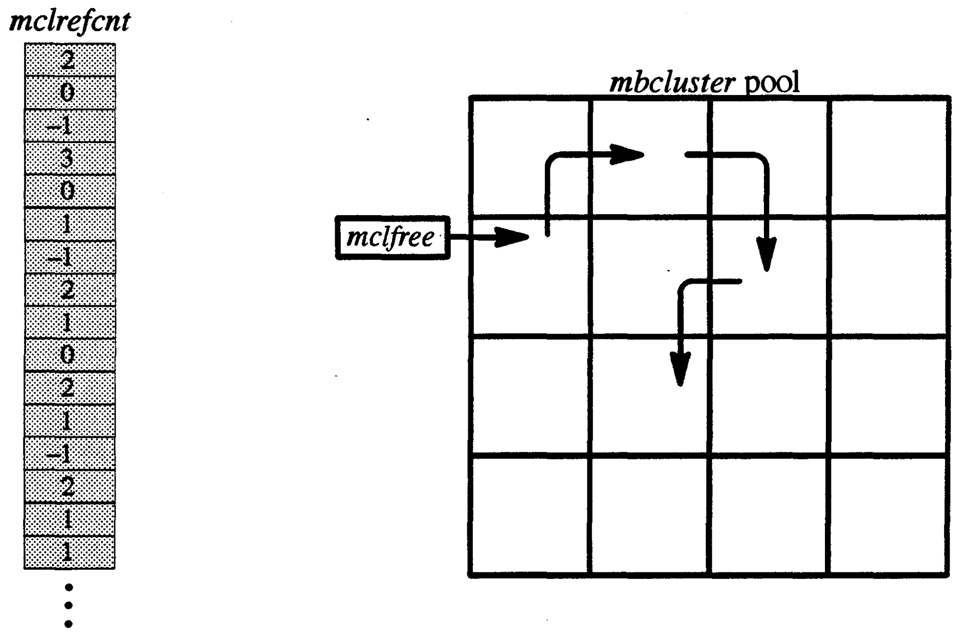
Virtual Copy (Socket Style)

Buffers may be passed by reference; different layers of the protocol may each contain references to the same buffer.

Every 5 sec. garbage collector will free extraneous buffers - dynamic number of buffers w/ high & low water-marks

8–9. Mbufs

The Cluster Pool and Reference Counts



.H2596 8-9. 290 © 1991 Hewlett-Packard

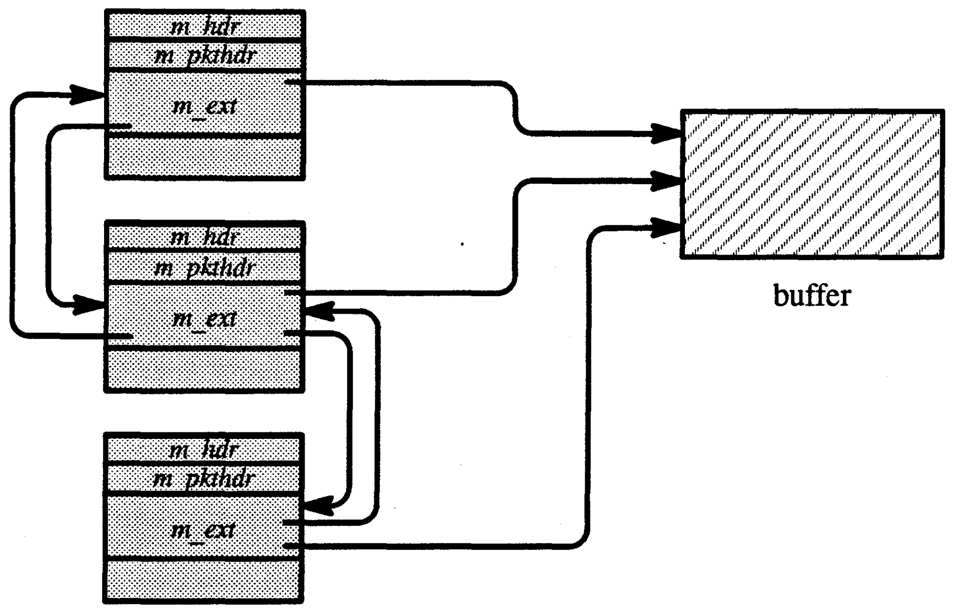
The Cluster Pool and Reference Counts

The system maintains a pool of *mbclusters*, which are buffers to which *mbufs* may refer. Thus, for example, to buffer data going to the network, the system allocates *mbclusters* to hold the data and *mbufs* to hold the protocol headers.

Free *mbclusters* are linked together in a free list headed by *mclfree*. The *mclrefcnt* array contains the reference counts to the *mbclusters*. A reference count of 0 indicates no references to a particular *mbcluster*. A reference count of −1 indicates that no real memory is backing up the associated virtual address.

8–10. Mbufs

Maintaining References



.H2596 8-10. 291 © 1991 Hewlett-Packard

Maintaining References

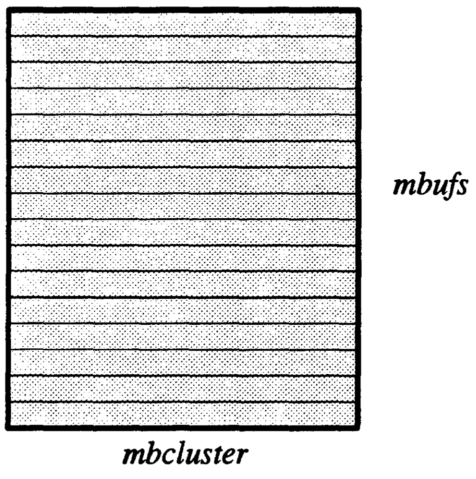
Buffers need not be allocated from the *mbcluster* pool. In particular, the streams system can allocate buffers and pass them on to the socket system via the XTISO driver. The only difficulty in doing this is maintaining the reference count. The *mclrefcnt* array cannot be used, since these buffers are not coming from the *mbcluster* pool.

All *mbufs* referring to the same buffer are doubly linked via their *m_ext* structures. Each *m_ext* structure contains the address of a storage liberation routine to be called when the last reference to a buffer goes away.

8–11. Mbufs

Mbufs from Mbclusters

16 mbufs/cluster



H2596 8-11. 292 © 1991 Hewlett-Packard

Mbufs from Mbclusters

The system allocates a fixed amount of virtual memory and a smaller amount of real memory for the pool of *mbclusters*. Free *mbclusters* (backed up with real memory) are linked into a free list. When the supply of free *mbclusters* becomes too low, it is replenished with pages from the free page list.

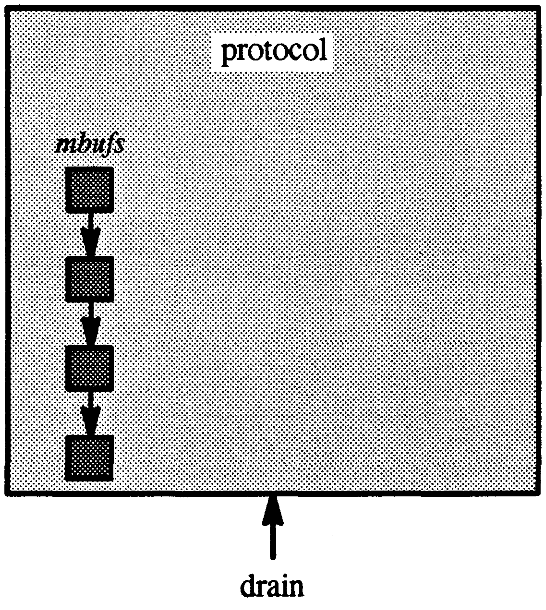
Mbufs are obtained by allocating *mbclusters* and breaking them up into *mbufs*. Free *mbufs* are linked together. When this supply of mbufs becomes too low, more *mbufs* are allocated from *mbclusters*.

The system invokes a *garbage collector* every five seconds to examine the *mbuf/mbcluster* situation and free up storage when necessary: if there are more than enough *mbufs*, then some are coalesced back into *mbclusters* and returned to the free *mbcluster* list. If there are more than enough *mbclusters*, then some are deallocated by returning their pages to the free page list.

The *mclrefcnt* array is used to aid the coalescing *mbufs* into *mbclusters*. When an *mbcluster* is on the free list, its reference count is 0. When it is broken up into *mbufs*, its reference count is set to 1. When an *mbuf* is allocated from the free list of *mbufs*, the reference count of the associated *mbcluster* is incremented by 1 (and decremented by 1 when the *mbuf* is freed).

8–12. Mbufs

Responding to Memory Shortages



.H2596 8-12. 293 © 1991 Hewlett-Packard

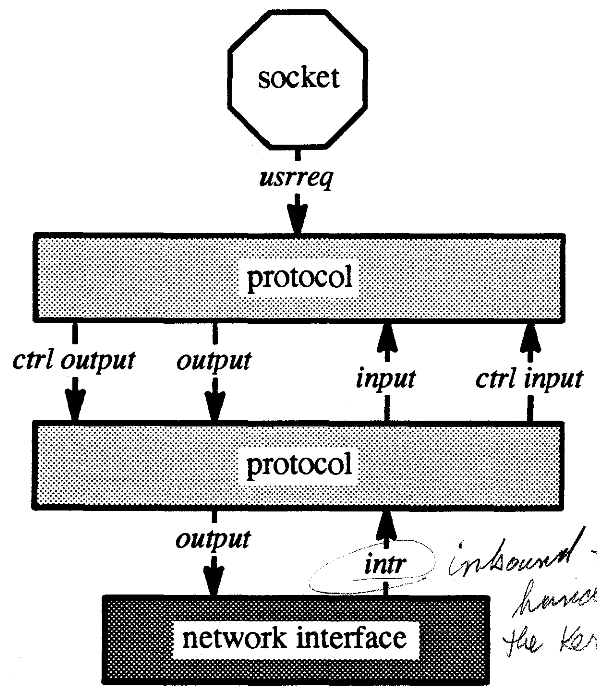
Responding to Memory Shortages

Many protocols hold on to data maintained in *mbufs* for a period of time. For example, the IP protocol maintains a reassembly queue to hold on to pieces of fragmented packets. When further pieces arrive the IP protocol reassembles them into a whole packet. If the operating system is extremely short of real memory, it calls each protocol's *drain* routine to ask the protocol to liberate as much memory as possible. In response to a call to its *drain* routine, the IP protocol releases the *mbufs* in all of its reassembly queues.

8–13. Implementation

IPC/Networking Control Flow

Not a stream -



inbound - interrupt handled by the kernel.

.H2596 8-13. 294 © 1991 Hewlett-Packard

IPC/Networking Control Flow

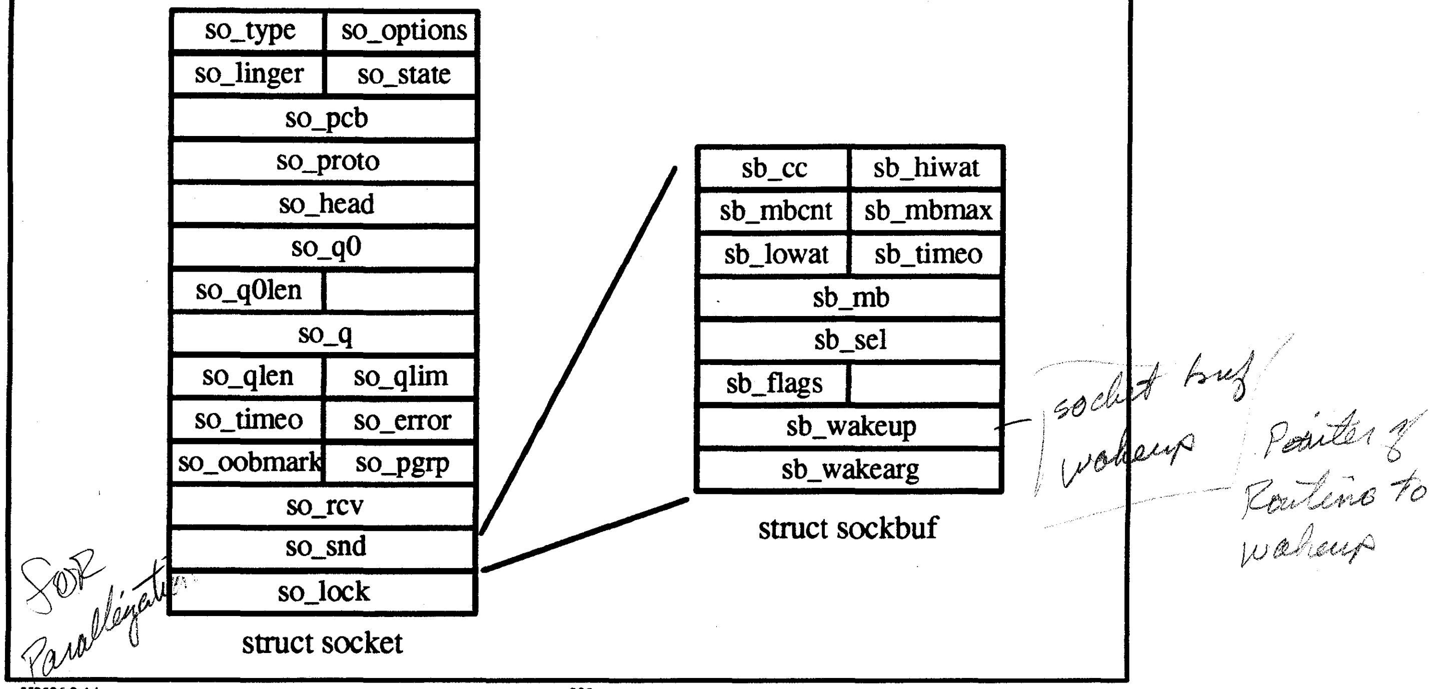
In a typical protocol stack, the top-level protocol might be TCP, the bottom-level protocol IP, and the network interface an *ethernet* interface. A user thread making a system call enters the socket layer and calls the top-level protocol's *usrreq* entry point. Control may continue via a call to the next level of protocol, which may then queue an output request by calling the network interface.

Packets coming from the network are dealt with in the interrupt context by the network-interface module, and then queued for processing by the bottom-level protocol. A kernel thread, one of a number of *netisr* threads, calls the bottom-level protocol's *intr* entry point and then processes the queued packets. The kernel thread might then continue by calling the top-level protocol's *input* entry point. This protocol processes the data, which is queued on the socket either immediately or sometime later.

The protocols will also be called periodically by *netisr* threads to handle timeouts. Because the *netisr* threads may block, there should be a few more of them than there are processors (so as to maximize the utilization of the available processors).

8–14. Implementation

Socket Data Structure



H2596 8-14. 295 © 1991 Hewlett-Packard

Socket Data Structure

All of the fields of the socket data structure are illustrated in the picture. Of particular importance to this discussion are the socket send and receive queues (*so_snd, so_rcv*) of type *struct sockbuf.* This data structure contains:

1. a byte count
2. a high-water mark (the maximum allowable size of the queue)
3. a message count
4. a maximum message count
5. a low-water mark (used for flow control: any thread blocked waiting for a space in the queue is woken when the queue size drops below the low-water mark)

6. a pointer to the queue itself (a linked list of *mbufs*)
7. the address of a routine to be called to wake up anyone blocked on the queue

In addition, there is a blocking lock on the entire socket structure.

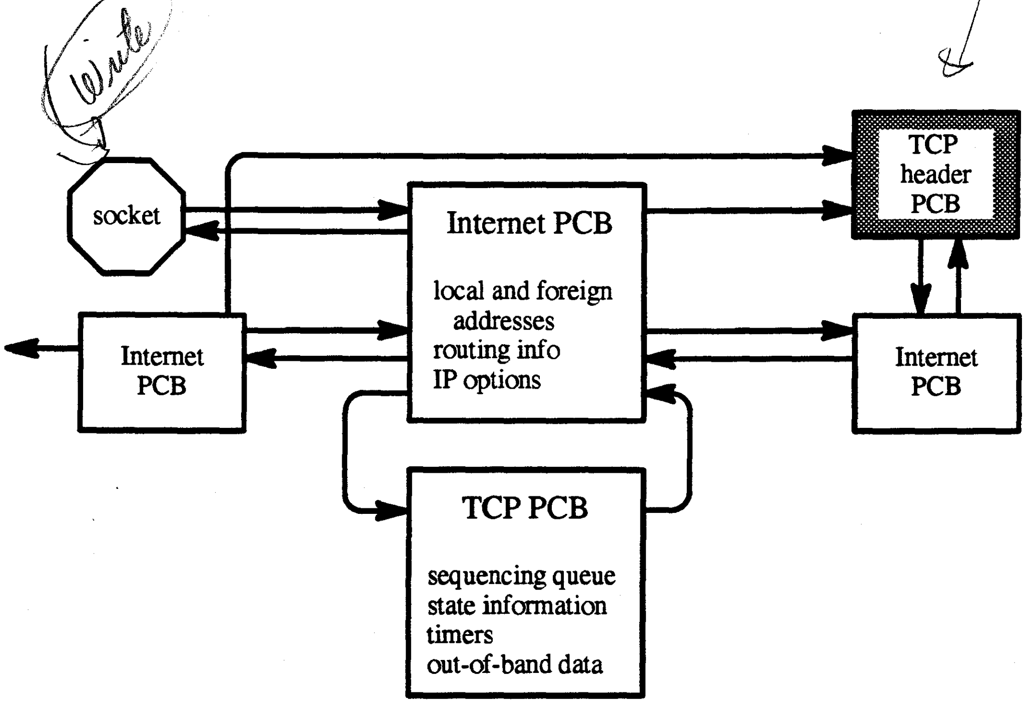
8–15. Implementation

Protocol Control Blocks

Write

Read - from kernel starts here & walks whole list looking for correct PCB

&

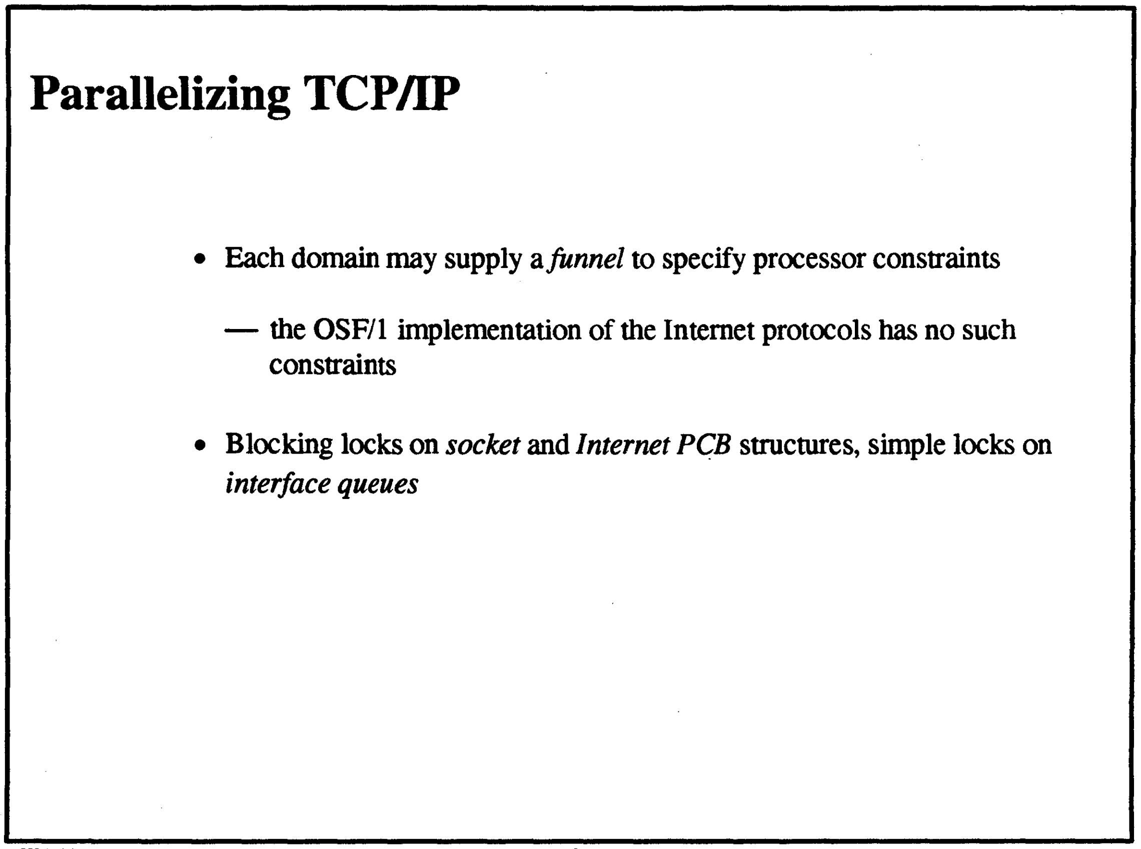


H2596 8-15. 296 © 1991 Hewlett-Packard

Protocol Control Blocks

This picture illustrates the protocol control blocks used by TCP/IP. Each connection is represented by a pair of protocol control blocks: an Internet PCB and a TCP PCB. The Internet PCBs of all the active TCP connections are doubly linked (necessitating a sequential search to associate a packet with a connection).

8–16. Implementation



Parallelizing TCP/IP

- Each domain may supply a *funnel* to specify processor constraints
  - — the OSF/1 implementation of the Internet protocols has no such constraints
- Blocking locks on *socket* and *Internet PCB* structures, simple locks on *interface queues*

H2596 8-16. 297 © 1991 Hewlett-Packard

Parallelizing TCP/IP

Associated with each family of protocols *(domain)* is a *funnel* structure, as discussed on page NO TAG.

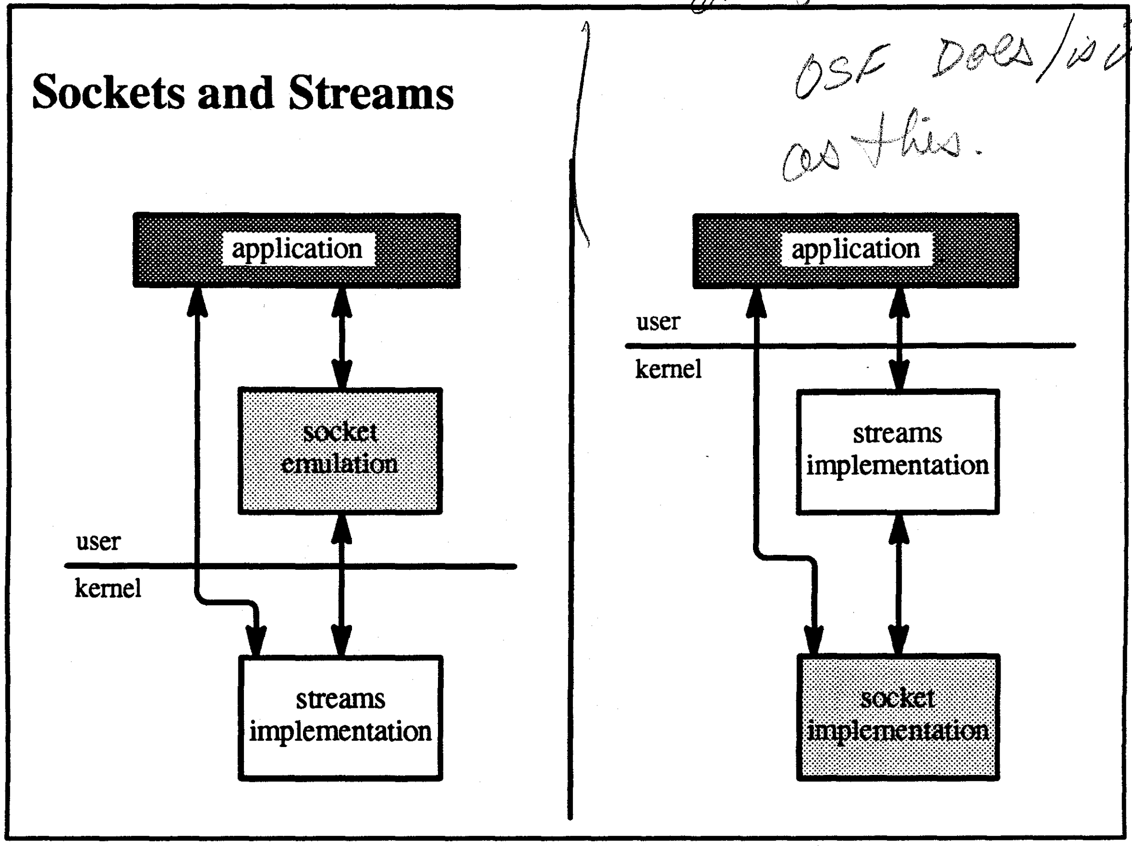
The Internet family of protocols (TCP/UDP/IP, etc.) have been parallelized. Blocking locks are in the *socket* and *Internet PCB* structures. The locking order is first to acquire the *socket lock* and then, when necessary, acquire the *Internet PCB* lock. Since these locks are blocking locks, they must be acquired by threads.

Simple locks are used on data structures such as the interface queues, which must be accessed in the interrupt context.

8–17. Sockets and Streams

Streams & Sockets could be one of two methods

OSF Does/is implemented as this.



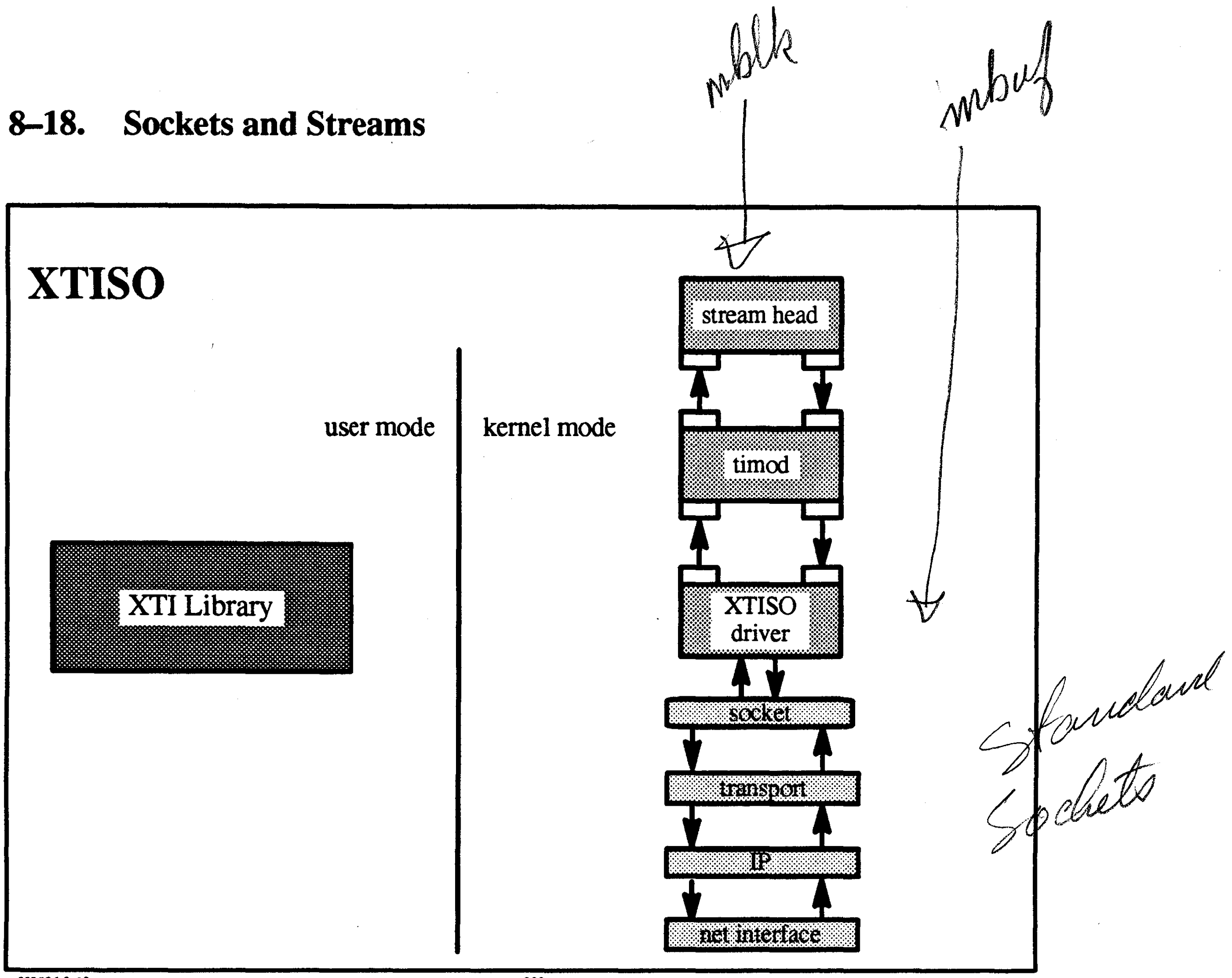
H2596 8-17. 298 © 1991 Hewlett-Packard

Sockets and Streams

A perhaps unfortunate fact of life is that there are two competing network interfaces: streams and sockets. There are probably more applications currently built on sockets than on streams, but this may be changing. A modern UNIX system should be able to support both interfaces.

SVR4 does so by providing a socket emulation library (in user mode). OSF/1, which supports both sockets and streams in the kernel, has no need for an emulation library, but instead provides a means whereby a protocol implemented in the socket framework can be accessed via the streams interface.

8–18. Sockets and Streams



.H2596 8-18. 299 © 1991 Hewlett-Packard

XTISO

XTISO (X/Open Transport Interface to Sockets) is a technique for supporting a streams-based XTI library with a socket-based protocol implementation. The XTISO stream consists of a standard stream-head module, a fairly simple *timod* module, and the XTISO driver module, which is the interface to the socket level. The XTISO driver takes *transport interface* (TPI) messages and converts them into operations on sockets. This transformation requires converting messages represented as *mblk*s into messages represented as *mbuf*s. These messages are passed to the socket layer by calling the socket code as if a socket system call had just been made.

Messages coming up from the transport layer are queued on the socket's *receive* queue. Normally the next step is to wake up any process waiting for this message. What happens instead is that the XTISO driver's *read service* procedure is enabled. This procedure is called to pull messages represented as *mbuf*s from the socket's *receive* queue, converts them into messages represented as *mblk*s, and converts these into a TPI message which is sent upstream.

The *timod* module converts messages sent to it from the XTI library via the stream head into the TPI format. This conversion primarily involves changing IOCTL-type messages into PROTO-type messages. Upstream messages are converted from PROTO-type into IOCACK-type as required.

Exercises:

1. What are the differences between communication through datagram sockets and communication through streams sockets?

2. a. Why do some *mbufs* contain data while others point to data that is external to the *mbuf*?

   b. How is it determined whether any *mbufs* refer to a buffer?

   c. List the set of actions that may be taken when the list of free *mbufs* is exhausted.

3. a. How are kernel threads used in the socket and networking subsystem?

   b. Which socket/networking data structures need locks? What types of locks are required? What is the precedence relation of these locks?

4. How does the XTISO driver serve as an interface between streams and sockets?

Module 9 — Logical Volume Manager

Module Contents

Module Objectives

In order to demonstrate an understanding of the concepts and implementation of the logical volume manager, the student should be able to:

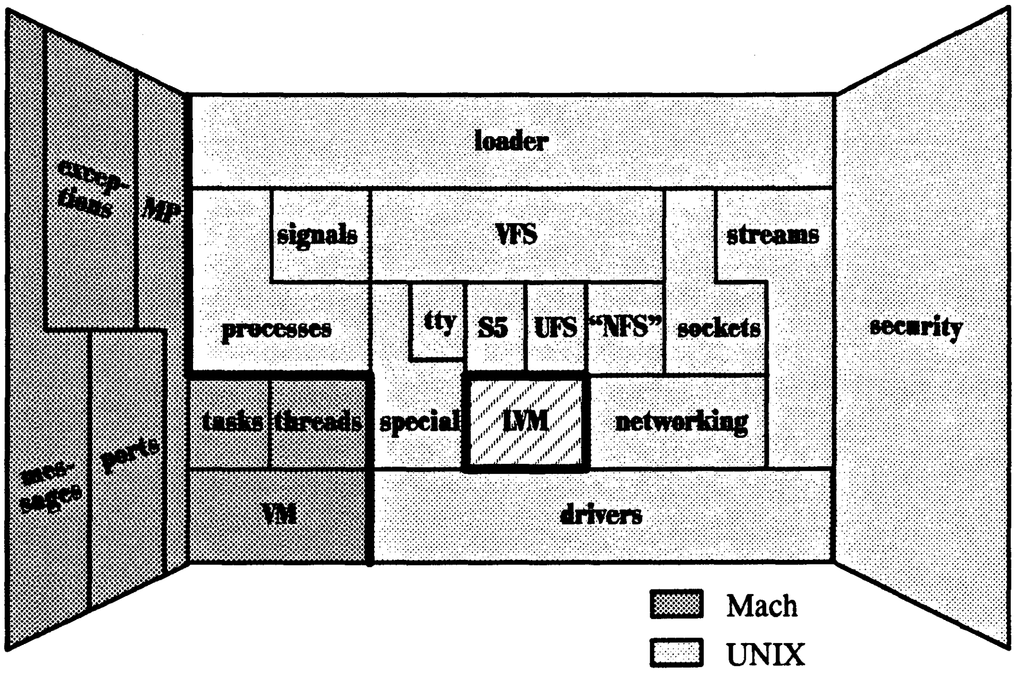
- list the functionality provided by the LVM but not provided by the standard file systems and disk device drivers

- explain how the LVM can be certain that it has an accurate description of a volume group, even when some of the underlying physical volumes are inaccessible

- list the benefits of mirroring

9–1. The Big Picture

Logical Volume Manager



H2596 9-1. 300

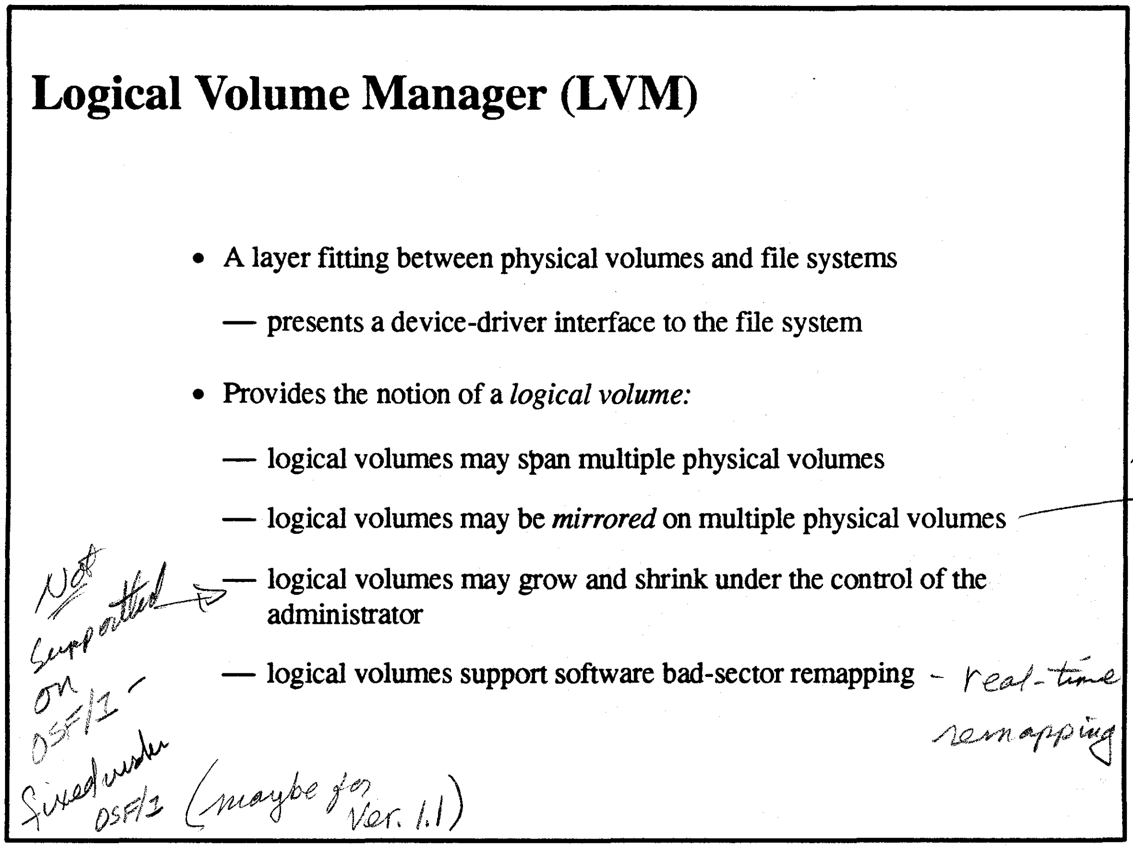
Logical Volume Manager

This material is discussed in chapter 15 of Open Software Foundation, 1990a.

Module 9 — Logical Volume Manager

From IBM's AIX

9–2. Role of the LVM



Logical Volume Manager (LVM)

- A layer fitting between physical volumes and file systems
  - — presents a device-driver interface to the file system
- Provides the notion of a *logical volume:*
  - — logical volumes may span multiple physical volumes
  - — logical volumes may be *mirrored* on multiple physical volumes — zero, one, or two Mirrors
  - — logical volumes may grow and shrink under the control of the administrator
  - — logical volumes support software bad-sector remapping - real-time remapping

Not supported on OSF/1 - fixed under OSF/1 (maybe for Ver. 1.1)

H2596 9-2. 301 © 1991 Hewlett-Packard

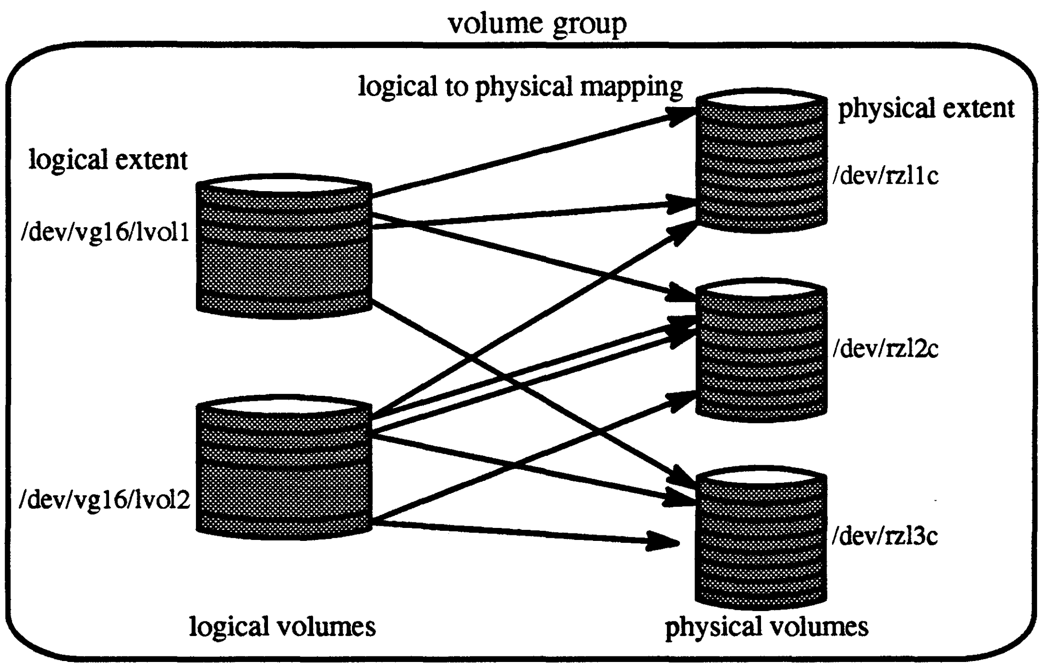
Logical Volume Manager (LVM)

UNIX files have always been limited by their inability to span multiple volumes. Since logical volumes can span multiple physical volumes, this restriction is pretty much removed in OSF/1.

The mirroring makes sure the images are consistent — Not that they all match the latest version. So, this is for disc failure not for protection of databases for system failure.

9–3. Role of the LVM

Logical Volume Manager Organization



H2596 9-3. 302 © 1991 Hewlett-Packard

Logical Volume Manager Organization

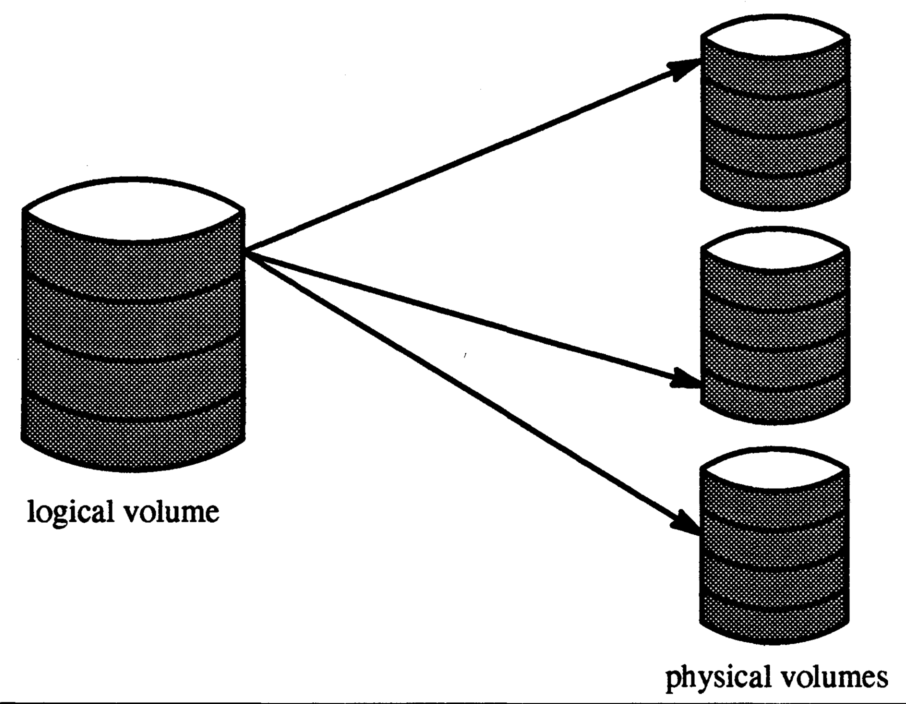
Logical volumes are organized within *volume groups* that contain both logical volumes and physical volumes. Logical volumes are divided into *logical extents*, the size of which may be any power of 2 between 1Mb and 256Mb. Each logical extent is mapped to one, two, or three *physical extents* on physical volumes. The size of physical extents is equal to the size of logical extents, which is the same throughout a volume group. Logical volumes appear to be real devices to most of the system, so they have a name as a special file within the /dev directory.

Each logical volume contains a single file system. The size of the logical volume may be easily changed by adding or removing logical extents and associating with them physical extents. In OSF/1 release 1, neither the S5 nor the UFS file system supports the notion of growth or shrinkage in a file system's underlying volume. However, in OSF/1 release 1.1 these file systems will be "growable."

If set up properly, the organization of logical volumes will not interfere with the organization of the UFS file system. Each UFS file system is built with the assumption that each cylinder group is composed of contiguous cylinders, but there is no built-in assumption that adjacent cylinder groups are actually near one another. Thus the UFS disk-allocation strategies will continue to work as long as cylinder groups do not cross logical extent boundaries.

9–4. Role of the LVM

Mirroring



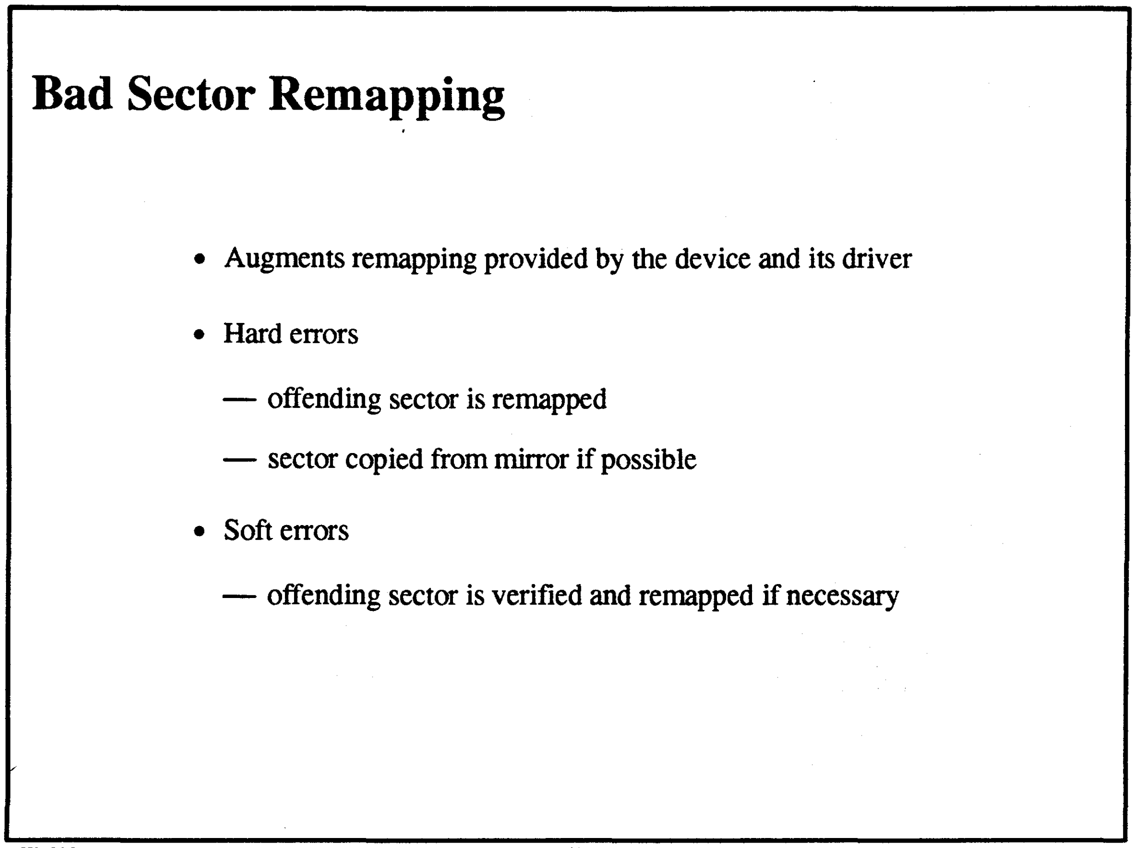
Mirroring

There are two motivations for *mirroring*: speed and crash recovery.

Mirroring can be used to speed *read accesses* to a logical volume: read requests to a mirrored logical extent can be translated into reads of a physical extent on the least busy physical volume. This approach is particularly useful for logical volumes that are "read-mostly," such as a logical volume containing binaries.

Most importantly, disk mirroring provides sufficient redundancy to survive crashes. If a physical volume is lost, the data contained in it can be recovered from copies maintained in other physical volumes—the mirrors. It is very important to insure that all mirrors containing the same data are identical. If an update was in progress at the time of a crash, only one mirror may have been updated. Recovery procedures are needed to reestablish the consistency of the data. Since it may not be known which mirror is the most recent, the recovery procedures select one mirror and copy it to the others. Thus the primary goal of recovery is to regain consistency among the mirrors.

9–5. Role of the LVM



Bad Sector Remapping

- Augments remapping provided by the device and its driver
- Hard errors
  - — offending sector is remapped
  - — sector copied from mirror if possible
- Soft errors
  - — offending sector is verified and remapped if necessary

.H2596 9-5. 304 © 1991 Hewlett-Packard

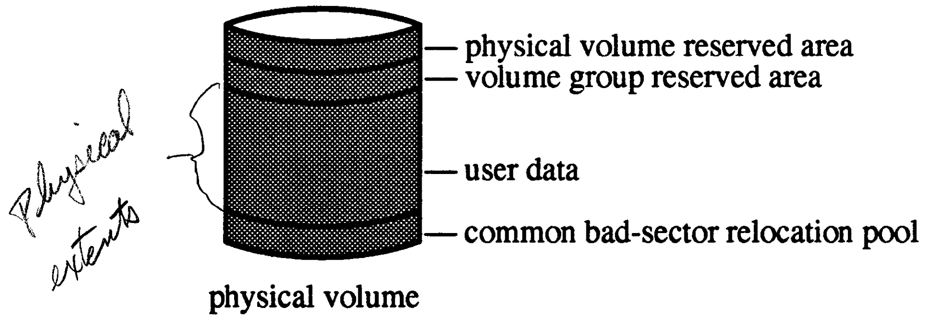
Bad Sector Remapping

The logical volume manager (LVM) augments the bad-sector remapping provided by the hardware. For mirrored volumes, the LVM can fix newly detected bad sectors by relocating the sector, reading the mirror, and writing the data into the relocated sector.

If a "soft" read error occurs (an error that was detected and corrected by the disk controller), the data is rewritten and verified. If a "hard" error now occurs, then the offending sector is remapped. Errors are not passed back to the file system unless a hard error occurs on a read from an unmirrored physical extent.

9–6. Data Structures

Logical Volume Manager: Physical Volumes


physical volume reserved area
volume group reserved area
user data
common bad-sector relocation pool
physical volume
Physical extents

.H2596 9-6. 305

Logical Volume Manager: Physical Volumes

A *physical volume* may be either an entire physical disk or a portion of a partitioned disk. That is, physical drives may be partitioned as they have always been in UNIX, or volume groups can effectively partition the physical volumes. By physical volume, from here on, we mean either a portion of a partitioned disk drive or the entire disk drive, depending upon how the drives are organized.

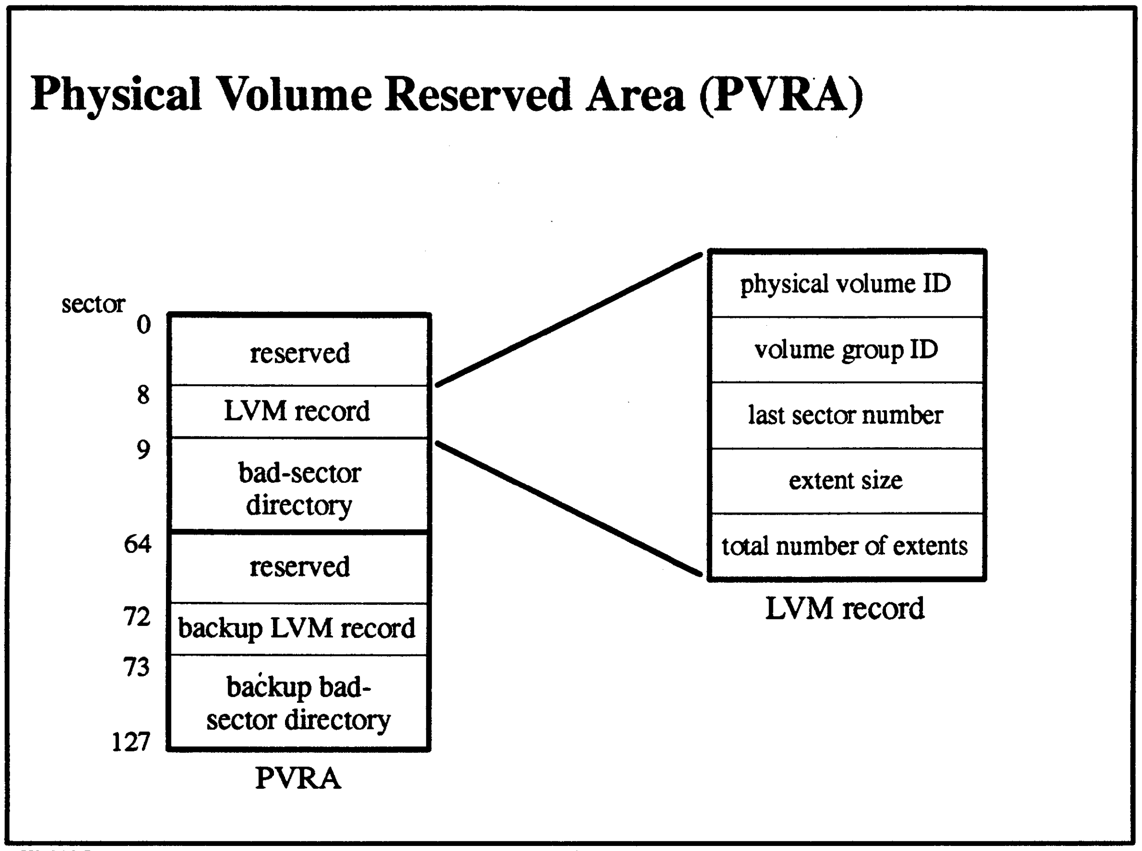
A certain amount of overhead is required within each physical volume. This consists of:

- physical volume reserved area
  - — this describes the individual physical volume
- volume group reserved area (VGRA)
  - — this describes the entire volume group

- common bad-sector relocation pool

As an option, space for bad sectors may be reserved at the end of each physical extent. This reduces the long seeks that would otherwise be required for remapping.

9–7. Data Structures



Physical Volume Reserved Area (PVRA)

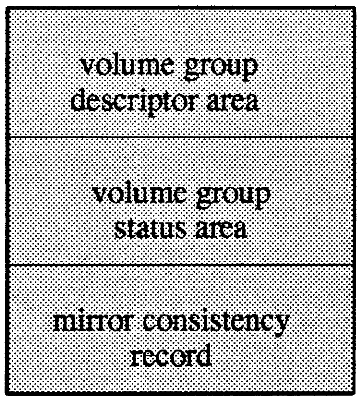
H2596 9-7. 306 © 1991 Hewlett-Packard

Physical Volume Reserved Area (PVRA)

- At a fixed location on the volume (*must* be good sectors!)
- Identifies the physical volume
- Gives its size
- Gives layout of rest of volume
- Contains bad-sector directory

9–8. Data Structures

Volume Group Reserved Area (VGRA)



volume group reserved area

.H2596 9-8. 307

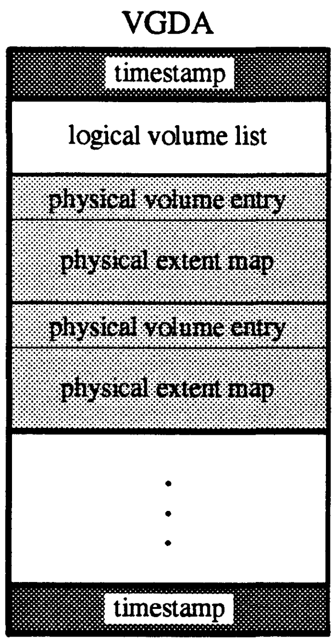
Volume Group Reserved Area (VGRA)

- Volume group descriptor area (VGDA)
  - — identifies all physical and logical volumes in volume group
  - — gives mapping (for entire volume group) of physical extent to logical extent
- Volume group status area (VGSA)
  - — lists missing/present status of each physical volume
  - — lists stale/ok status of each physical volume's physical extents
- Mirror consistency record (MCR)
  - — lists updates in progress

9–9. Data Structures

Volume Group Descriptor Area (VGDA)

update one timestamp before the write & update the other when finished. - if they don't match, this is BAD VGDA.



H2596 9-9. 308 © 1991 Hewlett-Packard

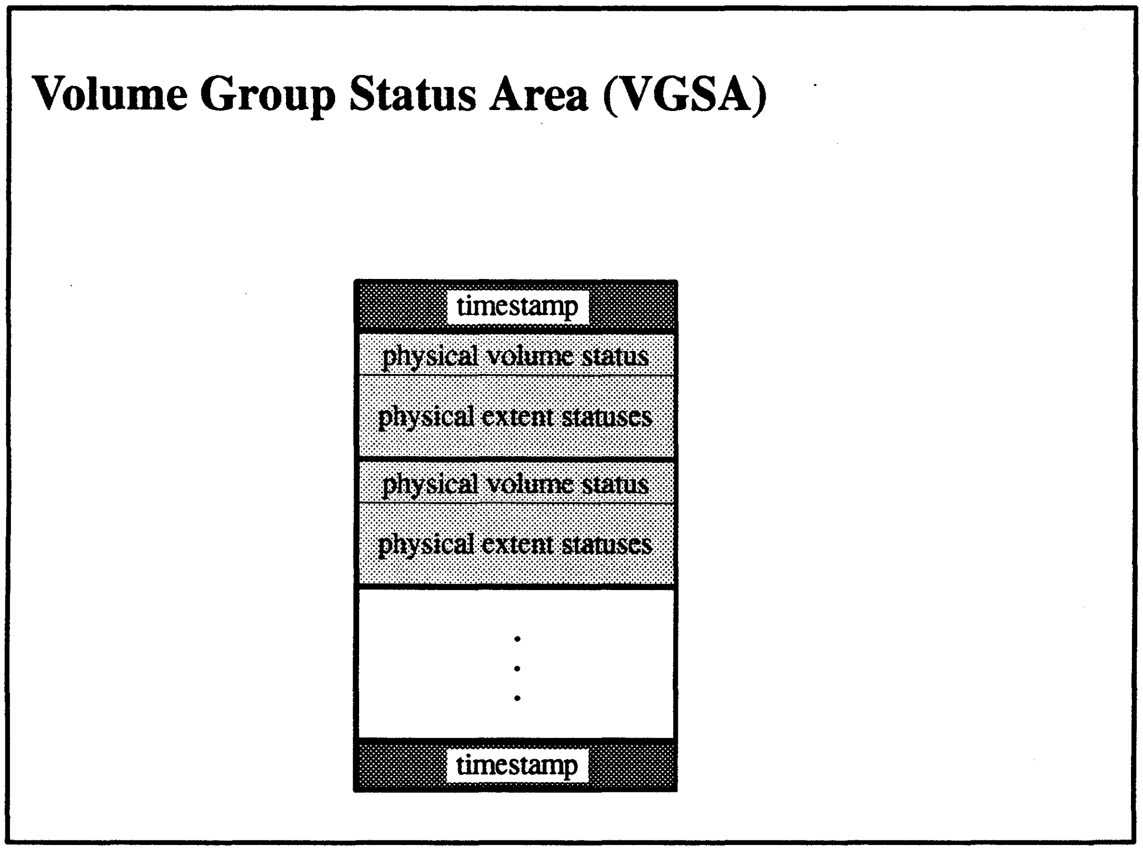
Volume Group Descriptor Area (VGDA)

Usually each physical volume contains two copies of the VGDA, one of which is up to date. However, if there are many physical volumes in the volume group, it would be excessively redundant for each physical volume to contain copies of the VGDA, so some may have no copies of it.

Each copy of the VGDA contains a timestamp. The copy with the most recent timestamp is considered to be the valid copy. When the VGDA is to be modified, the older copy of the two (per physical volume) is modified and thus becomes the newer version. To protect against failures while updating the VGDA, two timestamps are used, one at the beginning and one at the end of the area. The sectors containing the VGDA are written out synchronously and in order. When the VGDA is read, if the beginning timestamp and the ending timestamp are not equal, then a failure must have occurred during the update, and this copy of the VGDA is invalidated.

To guarantee consistency with the volume group, a *quorum* (more than half) of physical volumes must have identical VGDAs. If such a quorum is not available (i.e. some physical volumes are “down”), then no operations are permitted that would result in updating the VGDAs.

9–10. Data Structures


Volume Group Status Area (VGSA)

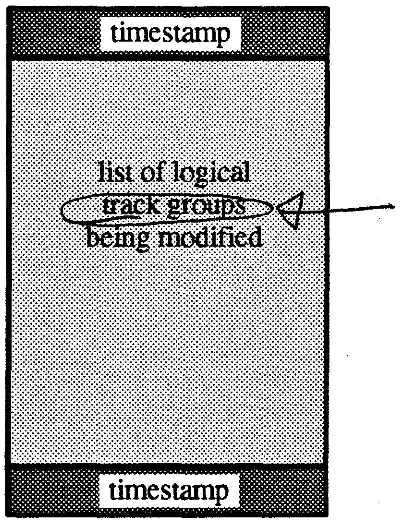
H2596 9-10. 309 © 1991 Hewlett-Packard

Volume Group Status Area (VGSA)

Each physical volume that has a VGDA has two copies of a volume group status area (VGSA) indicating whether the physical volume is available or missing. Like the VGDA, each of these copies is timestamped at the beginning and end. The VGSA is of particular importance after the system resumes operation following a crash, since the system must determine which of the volumes that had been available are now missing. In addition, this area indicates, for each physical extent within each volume, whether it is *stale* or *ok*. A physical extent on an available volume marked *stale* must be made *ok* by copying into it an *ok* version of its data.

9–11. Data Structures

Mirror Consistency Record (MCR)



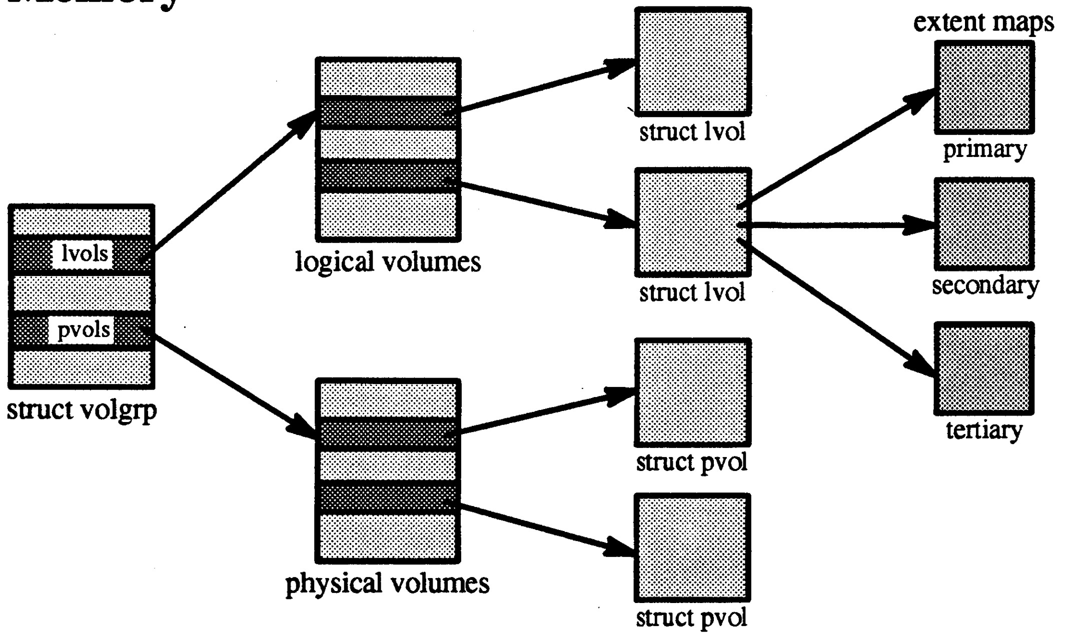
H2596 9-11. 310 © 1991 Hewlett-Packard

Mirror Consistency Record (MCR)

The mirror consistency record (MCR), like the VGSA, is maintained to help restore consistency following a crash. It indicates which portions of the logical volume were being modified at the time of the crash. Each physical extent is divided into *logical track groups*, as will be discussed.

9–12. Data Structures

Representing a Logical Volume Group in Primary Memory



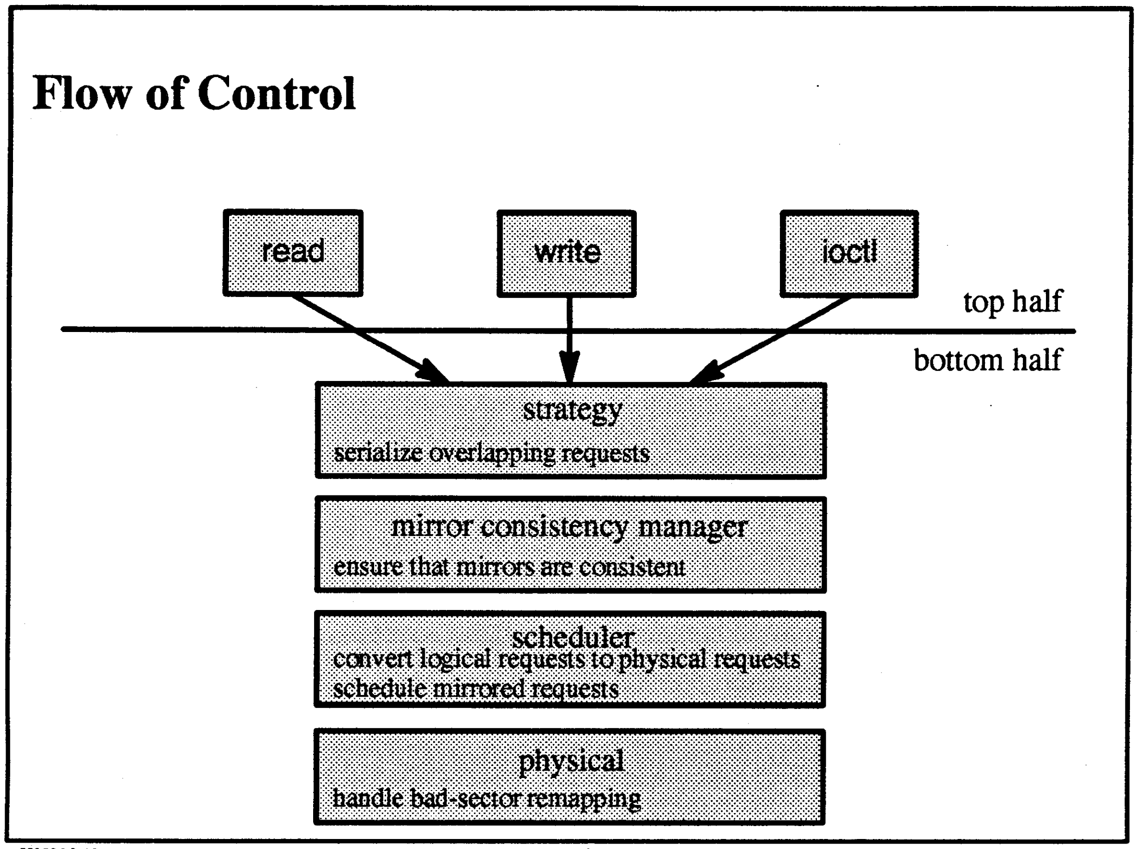
H2596 9-12. 311 © 1991 Hewlett-Packard

Representing a Logical Volume in Primary Memory

The extent maps are the inverse of those provided in the VGDAs, i.e., they provide mappings from the logical extent to the physical extent.

The *struct pvols* contain status information about each of the physical volumes and their physical extents.

9–13. Components and Flow of Control


Flow of Control
read
write
ioctl
top half
bottom half
strategy
serialize overlapping requests
mirror consistency manager
ensure that mirrors are consistent
scheduler
convert logical requests to physical requests
schedule mirrored requests
physical
handle bad-sector remapping

H2596 9-13. 312 © 1991 Hewlett-Packard

Flow of Control

The primary purpose of the *strategy layer* is to synchronize requests with respect to changes in progress at lower levels. For example, while a sector is being relocated, the strategy layer blocks all further requests for that sector. This sort of synchronization is accomplished by *serializing* requests at the strategy layer: whenever a new request arrives, it is held up until all earlier requests to overlapping sectors have been completed. When administrative commands change information in the VGDA regarding a particular logical volume, all operations on that volume are held up until the change is complete.

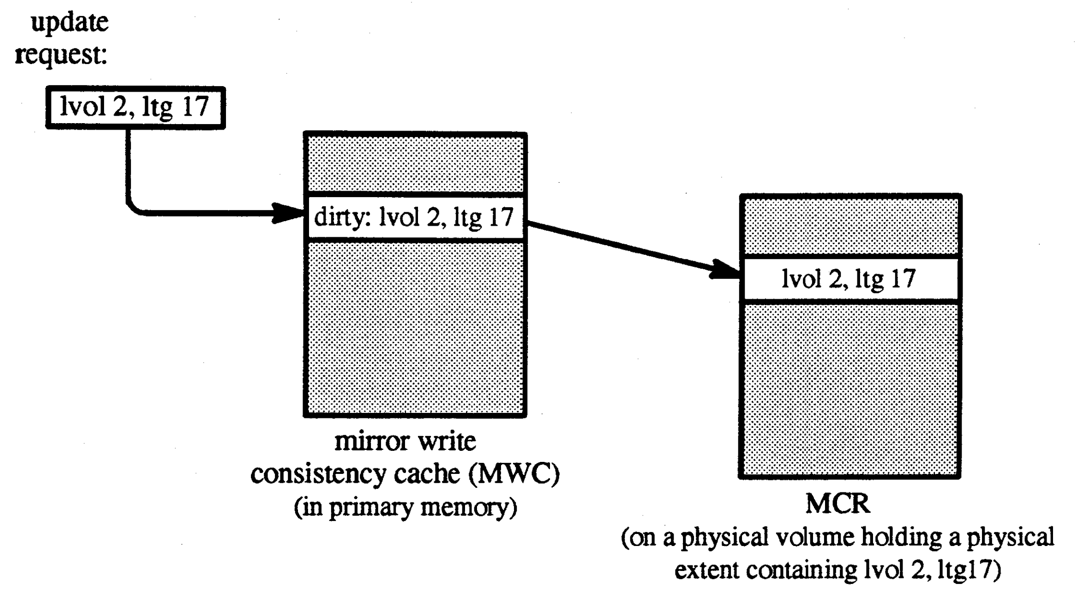
The *mirror consistency manager* maintains the consistency of mirrors. It keeps a list of update operations in progress, so that it can regain consistency in the event of a crash.

The *scheduler layer* is responsible for translating logical requests into physical requests. If the logical volume is mirrored, then each logical request may correspond to two or more physical requests.

The *physical layer* is responsible for bad sector relocation. It communicates directly with the real device driver and responds to disk errors by relocating sectors.

9–14. Components and Flow of Control

Consistency Management



H2596 9-14. 313 © 1991 Hewlett-Packard

Consistency Management

The basic approach to consistency management is to maintain a record on disk of the write operations currently in progress to mirrored extents. Thus, after a crash, those operations that were in progress can be identified and the mirrors involved can be made consistent with one another. A straightforward but amazingly expensive application of this approach would be to update this MCR before and after each disk write. The approach taken instead minimizes the number of extra disk writes at the expense of a longer crash recovery procedure.

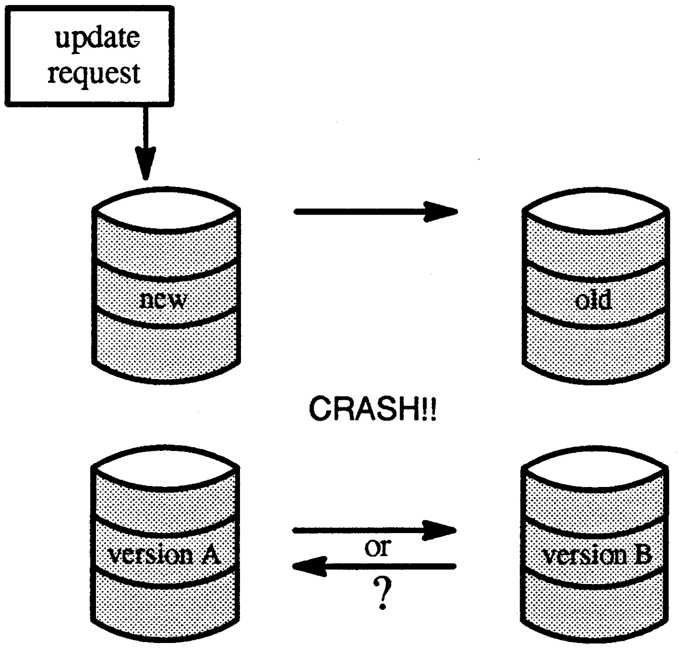
An up-to-date listing of updates in progress is kept incore. The on-disk record, however, lists update operations as being in progress for some time longer than they actually are in progress.

The incore data structure for representing update operations in progress is the *mirror write consistency cache* (MWC). This cache contains 62 entries. Each active entry is marked either "clean" or "dirty". The clean entries are arranged in LRU (least recently used) order. Whenever an update operation starts, an entry representing a *logical track group* (LTG) in a logical event is allocated in the MWC and marked "dirty". Each physical volume contains a recent copy of the MWC called the mirror consistency record. These copies are timestamped; the copy with the most recent timestamp is the valid one. Whenever a new entry is added to the MWC, the contents of the MWC are copied to the MCR of at least one of the physical volumes involved in the update. The MCRs contain no indication of "dirty" or "clean"; thus all entries marked "clean" in the MWC are interpreted as "dirty" in the MCR.

Each physical extent is divided into logical track groups of 32 pages each, where a "page" here is the disk block size. Each entry in the MWC and the MCRs corresponds to an LTG. Before any portion of an LTG is modified, an entry for it is allocated in the MWC and marked "dirty," and a new MCR reflecting the updated MWC is written. When the update completes (to all mirrors), the MWC entry is marked "clean" but no new MCR is written (thus if the system were to crash at this moment, the MWC would disappear and the most recent mirror consistency record would indicate that an update to this LTG is still in progress). If the LTG is updated again, then the MWC entry is changed to "dirty" but, again, no new MCR needs to be written. At some point there will be a period of no updates to the LTG, and its MWC entry will become the least recently used clean entry. When the next update request arrives for any other LTG, this entry is used to represent the new LTG and a new MCR will reflect this change, with the effect that the original LTG is no longer indicated as having an update in progress in the most recent MCR.

9–15. Components and Flow of Control

Consistency Management: Crash Recovery



H2596 9-15. 314 © 1991 Hewlett-Packard

Consistency Management: Crash Recovery

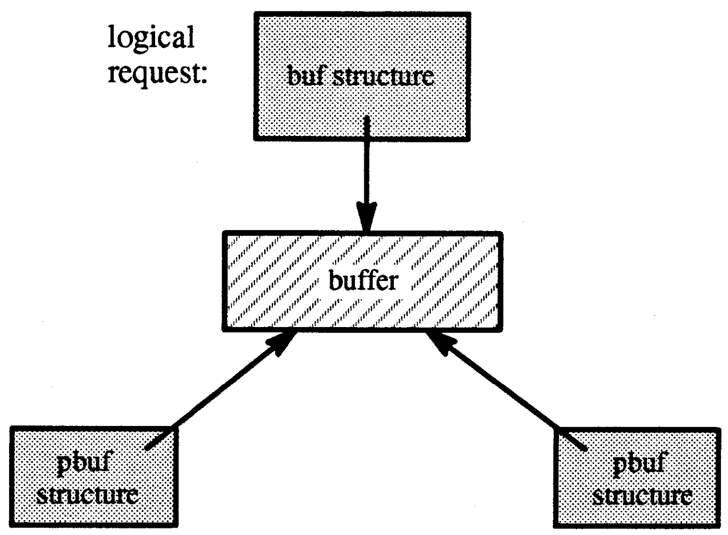
When a crash occurs, some number of update operations may be in progress. If a mirrored volume was in the process of being updated, the crash may cause the mirrors to be different. The mirror consistency manager must *restore consistency*, i.e. make mirrors identical with one another.

Crash recovery begins with locating the most recent MCR, which contains a list of the LTGs that were being modified. Since it is not known which mirror of each LTG contains the most recent information, the mirror consistency manager chooses one arbitrarily (actually, it issues a *read* request to the logical volume: the scheduler layer, using its rules for *read* scheduling, reads the data from a single LTG) and copies this LTG onto each of the others.

Another possible problem is that a physical volume may become unavailable (e.g., because of controller or media failures). This physical volume might contain only the most recent version of the MCR. If this volume was lost in a crash, then it cannot be known whether or not it contained the most recent MCR, so we must assume that it did. To cope with this, OSF/1 assumes the worst case: that *every* LTG on the volume was being modified at the time of the crash. We then must indicate that all associated mirrors may be inconsistent. This is done at the physical-extent level by marking all physical extents of this volume as "stale" in the VGSAs. Then, for all logical extents that were double-mirrored, resynchronization is done by copying one of the accessible physical extents to the other (the second physical extent is also marked stale). If the logical extent was single-mirrored, then only one accessible physical extent remains, which by definition is consistent. (Due to the drastic steps taken for recovery when a physical volume becomes unavailable (e.g., an entire disk might be copied), this procedure is performed only if explicitly permitted by the operator.)

9–16. Components and Flow of Control

The Scheduler Layer



H2596 9-16. 315 © 1991 Hewlett-Packard

The Scheduler Layer

The *scheduler layer* converts the logical request represented by a *buf* structure into one or more physical requests represented by *pbuf* structures. For a *write* request, all mirrors must be modified; in a *read* request it is only necessary to access a single mirror. Two general policies are used for scheduling the requests: a *parallel* and a *sequential policy.*

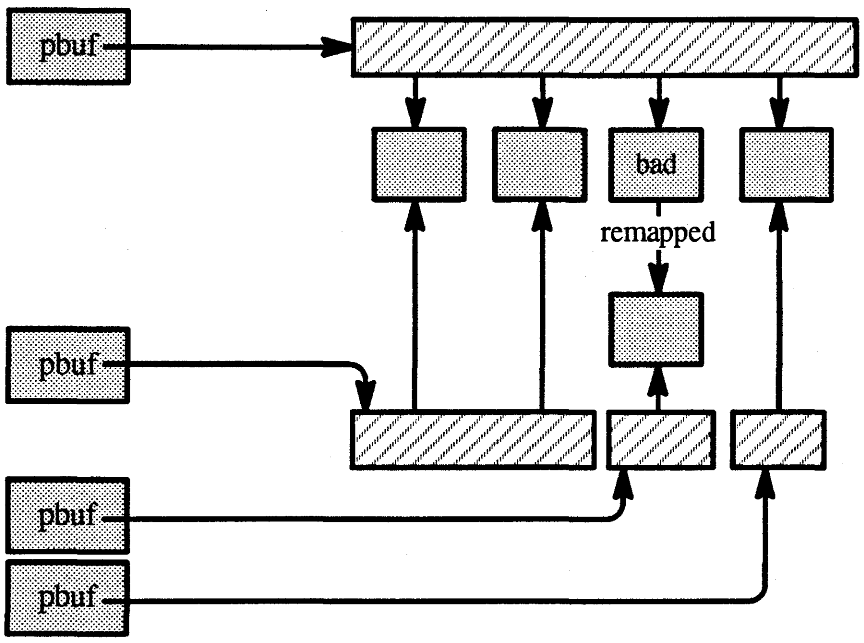
In the *parallel policy*, a *read* request is always sent to the physical volume with the fewest outstanding I/O operations. *Write* requests are issued in parallel to the physical volumes.

In the *sequential policy*, a *read* request is performed by attempting *read*s from mirrors in a predefined order. If the first *read* succeeds, then the operation completes; otherwise a *read* from the next physical volume is attempted, and so on. *Write* requests are performed sequentially; a *write* request to one physical volume is complete before a *write* request to the next physical volume begins.

Clearly, the parallel policy is more efficient than the sequential policy. However, the sequential policy is safer. For example, if the system crashes in the middle of an update, the updates of each of the mirrors are affected with the parallel policy, while with the sequential policy, only the update of one of the mirrors is affected.

9–17. Components and Flow of Control

The Physical Layer



H2596 9-17. 316 © 1991 Hewlett-Packard

The Physical Layer

The *physical layer*'s main responsibility is to handle bad-sector remapping. It must examine each physical request for references to any known bad sectors. If a request does refer to a bad sector, then (assuming only a single bad sector for ease of exposition) the physical layer breaks the request into three pieces: the first piece for the request up to the bad sector, the second piece for the relocated bad sector, and the third piece for the remaining portion of the request. The pieces are then treated as separate requests and are processed sequentially.

The device driver detects new bad sectors during an I/O operation and reflects the error back to the physical layer. There are essentially two types of errors: soft errors and hard errors. Soft errors have been detected and corrected by the disk controller. The physical layer attempts to test the sector by sending a "write-verify" request to the disk driver. If this succeeds, then nothing else need be done. However, if it fails, the sector is remapped.

If a non-mirrored *read* operation encounters a hard error, then there is no choice but to reflect the error to the caller. However, an entry is made in the bad-block directory indicating that relocation is desired and this is done the next time this sector is written.

If a mirrored *read* operation encounters a hard error, the scheduler layer performs a *read* to a mirror, then sends a *write* request to the original sector specifying that hardware relocation is desired. The physical layer passes this on to the device driver. If hardware relocation fails or is not supported, then software relocation is performed.

Exercises:

1. a. What functionality does the LVM provide that the standard file systems and disk device drivers do not?

   b. Which LVM-related information is replicated over most of the physical volumes? Which is only maintained on a single physical volume?

2. a. What is the purpose of timestamps in the on-disk data structures?

   b. How can the LVM be certain that the volume group descriptor information is valid even when some physical volumes are inaccessible?

3. a. Why is it necessary to serialize overlapping requests?

   b. What information is contained in the MCR?

   c. Suppose that, as part of crash recovery, it is discovered that the MCR contains a single entry and this entry pertains to a doubly mirrored volume. Assuming that all physical volumes are present, list the actions taken as part of the recovery of the logical volume group.

   d. Explain the differences between the parallel and sequential scheduling policies.

Advanced Question:

4. What must be changed in the UFS file system so that it can exploit the ability of logical volumes to grow?

Module 10 — Loader

Module Contents

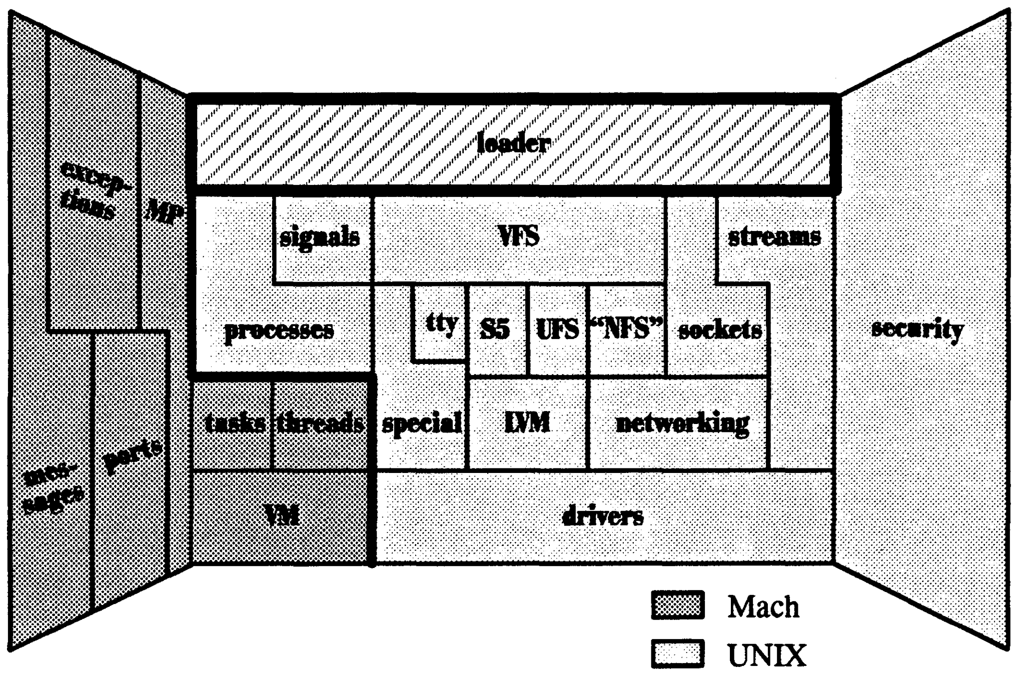
Module Objectives

In order to demonstrate an understanding of shared libraries, dynamic loading, and symbol resolution, the student should be able to:

- describe the functionality provided by the OSF/1 loader that is not provided by traditional *ld* and *exec*
- explain the role of packages in symbol resolution
- explain the phases of the run-time load procedure
- list the three techniques for implementing shared libraries and their advantages and disadvantages
- describe the differences between loading into the current process and loading into the kernel

10–1. The Big Picture

The OSF/1 Loader

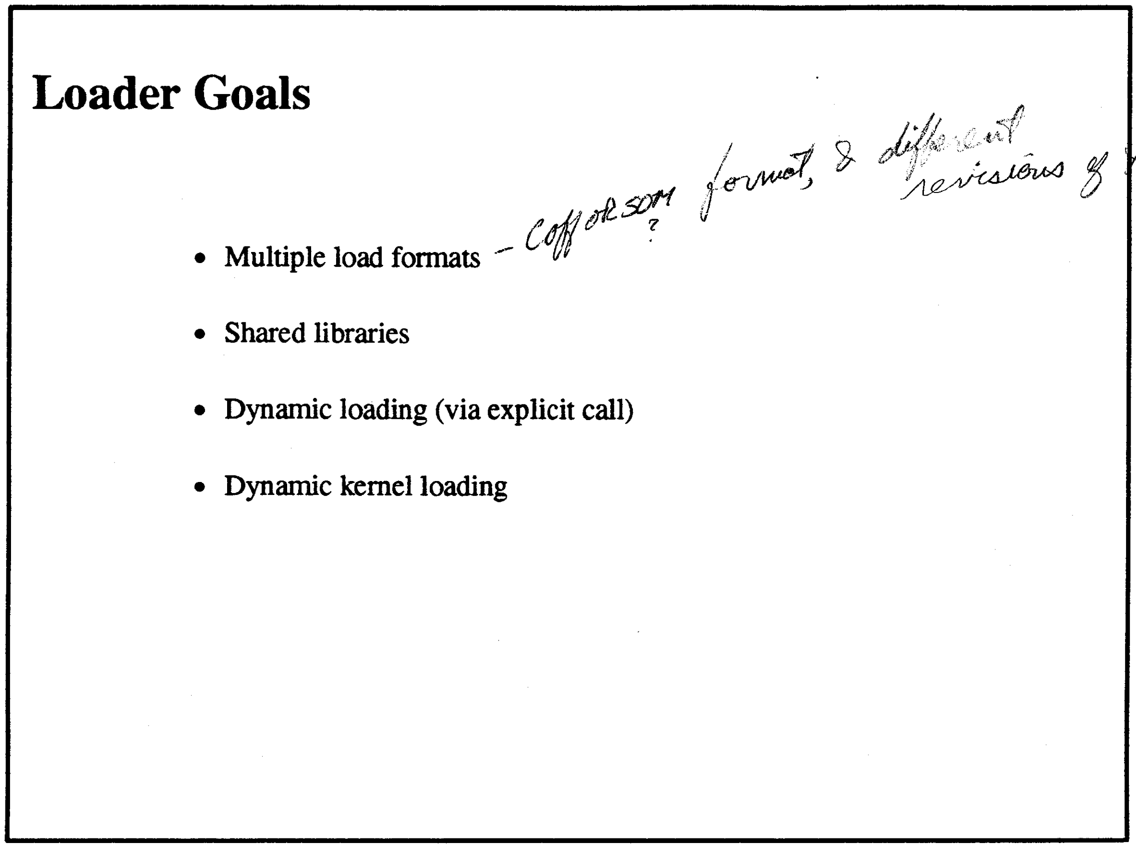


.H2596 10-1. 317 © 1991 Hewlett-Packard

The OSF/1 Loader

The material in this module is discussed in chapter 6 Open Software Foundation, 1990a. Additional information about the OSF/1 loader can be found in Allen, 1991.

10–2. Role of the Loader



Loader Goals

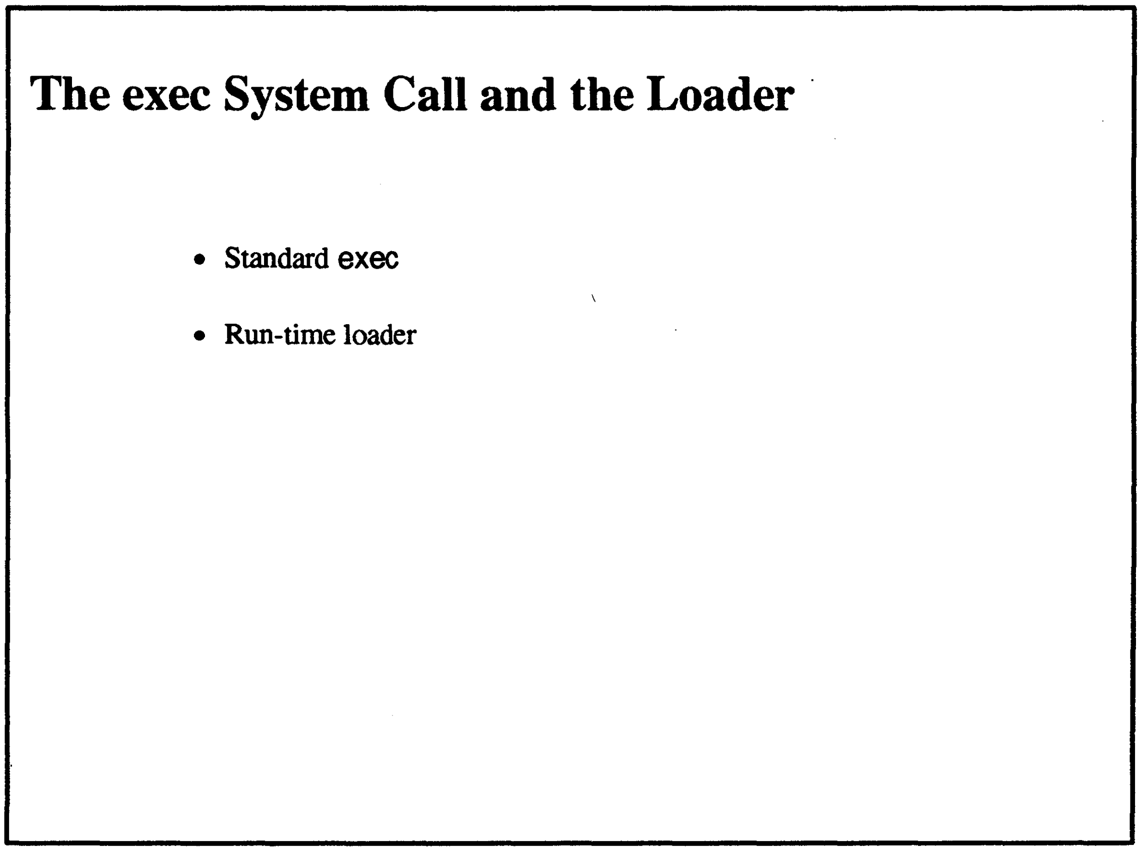
- Multiple load formats - Coff or SOM ? format, & different revisions of the loader.
- Shared libraries
- Dynamic loading (via explicit call)
- Dynamic kernel loading

H2596 10-2. 318 © 1991 Hewlett-Packard

Loader Goals

A *library* is a collection of *modules* contained in a file. A module is a component of an executable image. For example, a module might be a result of compiling or assembling a file containing a program in source form.

10–3. Role of the Loader



The exec System Call and the Loader

- Standard exec
- Run-time loader

H2596 10-3. 319 © 1991 Hewlett-Packard

The exec System Call and the Loader

As in any UNIX system, the exec system call can invoke a program. In older UNIX systems, such a program must have been fully bound and relocated. While OSF/1 certainly supports this mode of operation, the system also allows much of the binding and relocation to be postponed until run time.

When a program is exec'd, the system-call handler examines the program: if the program is in a recognizable load format, is fully relocated and contains no unresolved references, then it is loaded directly and control is passed to it on return to user mode (as usual). Otherwise, exec loads the (user-mode) run-time loader into the address space and passes control to it on return to user mode, giving it the name of the program to be loaded. The run-time loader then completes the load process. It copes with load formats unrecognized by the kernel, linking the image to shared libraries, loading additional modules as required, and relocating the entire image as necessary.

10–4. Role of the Loader

Constructing an Executable Image

- Linking
  - — matching symbols to packages
- Loading
  - — symbol resolution
    - ♦ pull in all necessary modules
  - — relocation

H2596 10-4. 320 © 1991 Hewlett-Packard

Constructing an Executable Image

The first step in the creation of an executable image is the construction of the component modules. These modules are created by *linking* together the object code produced by compilers and assemblers. Some of the symbols referenced in the resulting module might not be defined there, but are instead defined in some library. In older UNIX systems, the routines from the libraries that define these symbols would be copied and bound into the resulting module. With the OSF/1 loader, the linker may merely augment the symbol name with the library name (package name) that defines it, and postpone until later fetching the routine defining the symbol. Thus the result of linking is to create an *imported symbol table* that contains a list of unresolved symbol-name/package-name pairs.

The loading procedure may be completed at run time. The first step involves final *symbol resolution*. All of the packages mentioned in the imported symbol table must be tracked down and the routines containing the unresolved symbols must be extracted. This is in general an iterative procedure, since the routines so extracted may have their own imported symbol tables, causing further symbol resolution. As symbol resolution is performed, relocation is also performed.

10–5. Symbol Resolution

Packages and Libraries

- Two-dimensional name space

  — package name/symbol name

- Packages are mapped to libraries at run time

  — binaries need not contain path names

H2596 10-5. 321 © 1991 Hewlett-Packard

Packages and Libraries

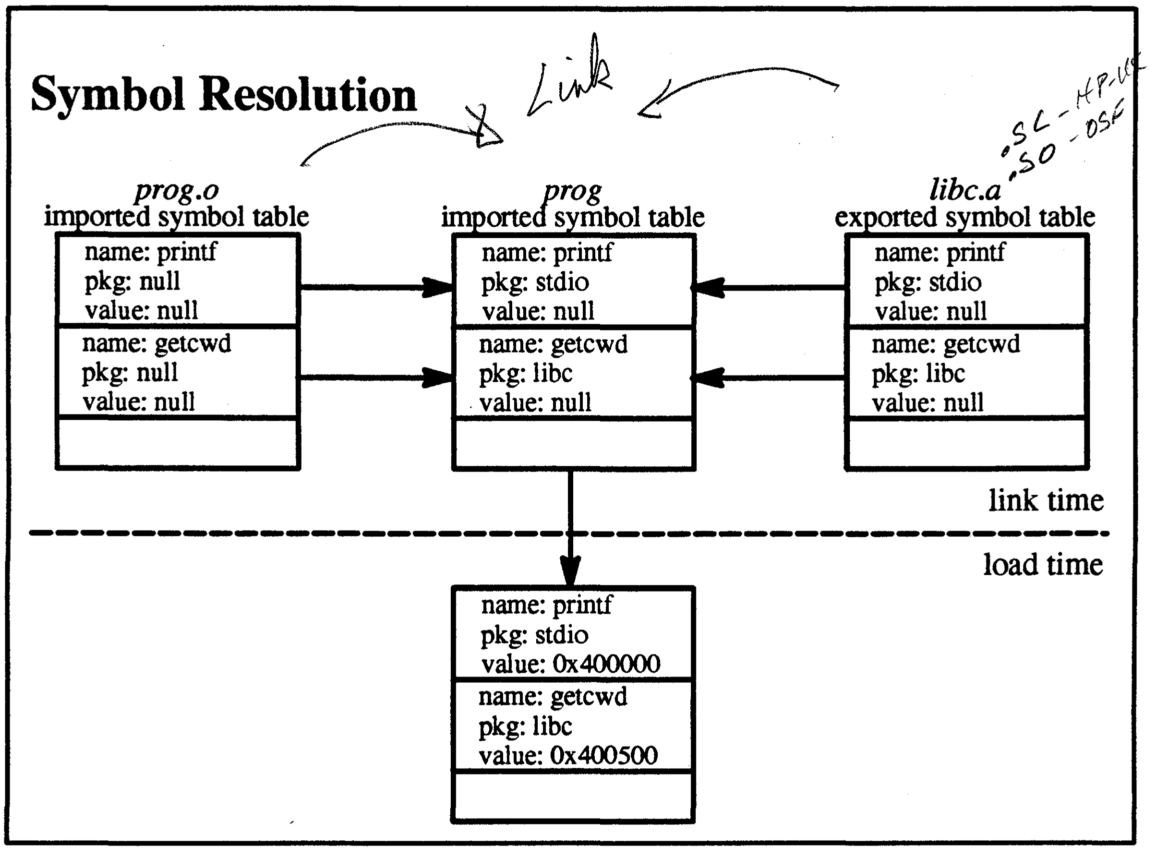
The concept of *packages* was invented to deal with the following issues:

- symbols should not be bound to library path names, since libraries may move
- naming conflicts may occur between symbols of different libraries
- symbol resolution should be *flexible*: the user should be able to alter resolution at compile, link, and load time

Packages are the *abstraction* of libraries: they normally correspond to libraries, but the programmer is free to split a library into sub-libraries by breaking it up into a number of packages. Currently, package names are assigned to symbols only at link time, but with sufficient compiler support they could also be assigned at compile time.

10–6. Symbol Resolution

Symbol Resolution



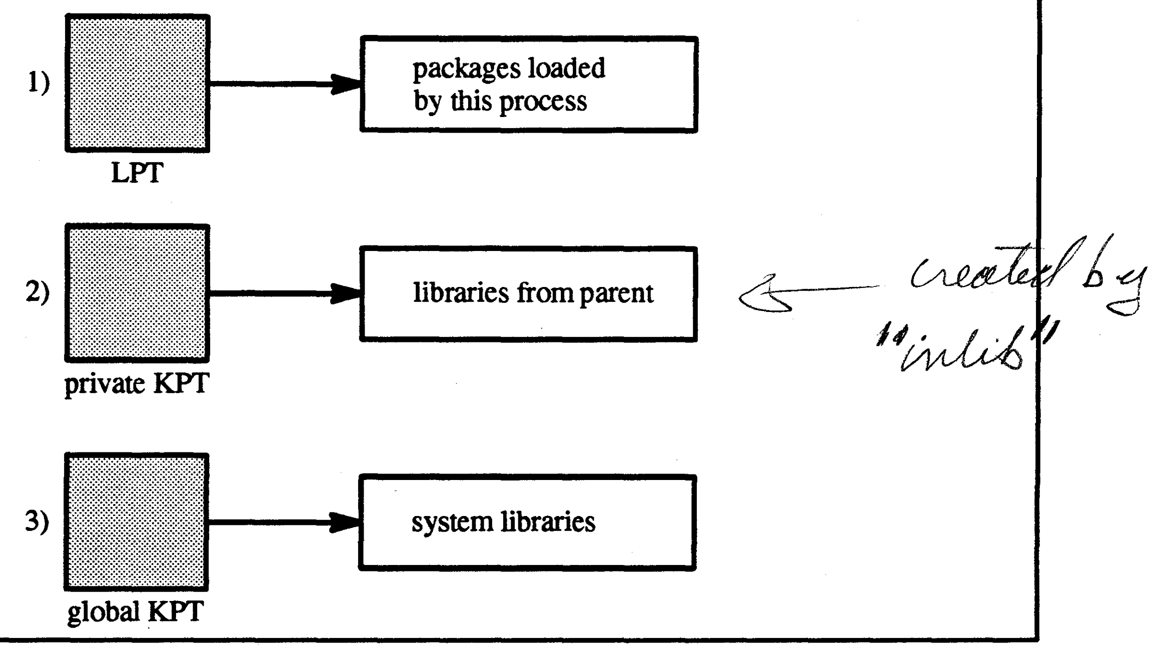
H2596 10-6. 322 © 1991 Hewlett-Packard

Symbol Resolution

Prog.o is produced by the compiler. Its imported symbol table contains the names of unresolved symbols, but little is known about these symbols. At link time, the *ld* program finds definitions for the symbols in libraries, but instead of extracting the code, it augments the imported symbol table by filling in the package for each of the symbols listed. Finally, at load time, the *ld* program somehow brings the desired packages into the address space and fills in the value fields of the imported symbol tables with the addresses of the component routines.

10–7. Symbol Resolution

Known Package Tables (KPTs)



H2596 10-7. 323 © 1991 Hewlett-Packard

Known Package Tables (KPTs)

Known package tables (KPTs) contain mappings from package names to actual code. They are used to translate a package-name/symbol-name pair into a symbol within a particular library. Each process has a sequence of KPTs which it searches to resolve a particular package-name/symbol-name pair. The order of search is:

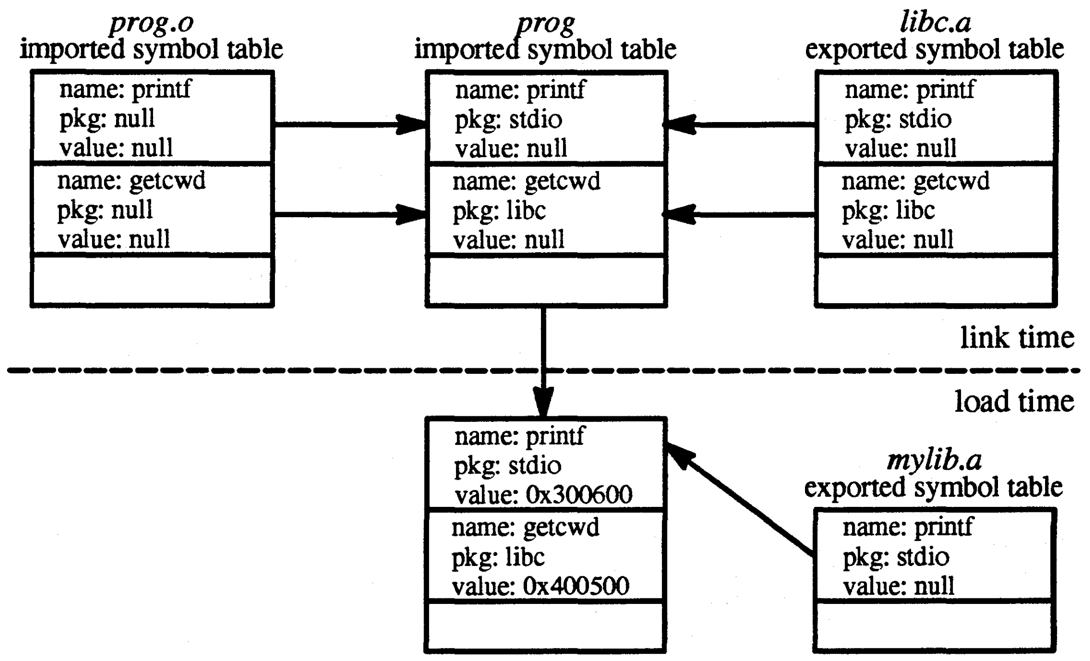
1. loaded package table (LPT): a per-process table referring to packages from modules that have been explicitly loaded into the program; this includes the loader itself, the "main" module (whose name was given in exec), modules loaded because they are dependencies of other modules, and modules that have been dynamically loaded

2. private known package table (private KPT): maintained by the user and inherited copy-on-write from the parent process (this is contained in anonymous memory that is retained across execs)

3. global known package table (global KPT): a system-wide table maintained by the system administrator (used to define the standard system libraries)

10–8. Symbol Resolution

Package Substitution



.H2596 10-8. 324 © 1991 Hewlett-Packard

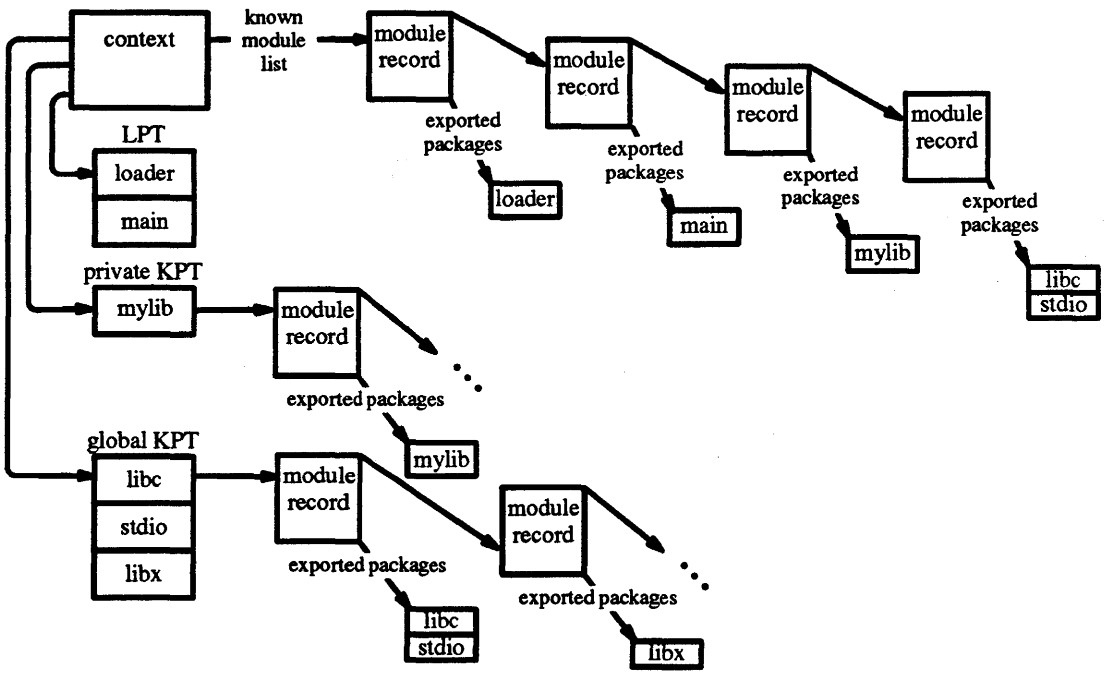
Package Substitution

By modifying the contents of the private KPT, the user can effect changes in symbol resolution. In this example, by *inlib*ing *mylib.a*, we have created a private package containing a redefinition of *printf*. Since at link time the symbol *printf* in our program has been associated with the package *stdio*, we have to call our private package *stdio* as well, so that it is used to define the *printf* mentioned in our program. Our private *stdio* package, containing only *printf*, will appear in the private package table.

At load time, since the program searches the private KPT before the global KPT, it uses the new version of *printf*. However, any reference to any other member of the standard *stdio* package is satisfied via the global KPT, since we have only replaced *printf* of the standard *stdio* package.

10–9. Data Structures and Flow of Control

The Run-Time Loader



The Run-Time Loader

This picture gives a simplified view of the data structures used by the loader. There is a *context* structure for each image maintained by the loader. Typically there will be just one context structure, representing the image in the current address space. However, the kernel loader, for example, would have a *context* structure for another image—the kernel image. The *context* structure is initialized to refer to the global KPT and the inherited private KPT. The LPT is initialized to refer to the loader itself. Each module loaded into the address space is represented by a *module record* structure, which is linked into the *known module list*, which is headed by the *context* structure. Each *module record* points to an array listing all of the packages contained in the module and made available to other modules (exported). Associated with the private and global KPTs are *module records* for each of the modules supplying packages for the associated KPT. Unlike the *module records* in the *known module list*, which contain detailed information about the module such as the imported symbol table, these *module records* only list the exported packages of the module.

Initially the *known module list* contains a *module record* for the loader. The next step is to append a *module record* for the main routine, i.e., the one given in the exec system call. The goal of the loader is now to build a complete *known module list*, containing all modules that are required for the image. It does this by identifying for each module record in the list the additional modules it needs and adding the record for these additional modules to the *known module list.*

The packages exported by these modules are added to the loaded package table. Modules are added to the *known module list* when they are needed to resolve symbols listed in a module's imported symbol table. *Module records* for these modules are appended to the end of the *known module list.* In some load formats, the names of these modules are supplied explicitly. The technique intended is that the symbol-name/package-name pairs given in the imported symbol table will be looked up in the loaded package tables, as discussed on page 10-9. The next step is for the loader to traverse the *known module list* and to map in (if not already mapped) the *regions* (e.g., text, data, BSS) of each module. At the same time the loader can determine the values of each module's exported symbols.

Finally, the loader traverses the *known module list* again and performs relocation in each module.

In summary, run-time loading consists of three phases:

- *discovery*—locating the desired modules based upon translating symbol-name/package-name pairs to routines in libraries
- *mapping*—map the modules into memory
- *relocation*—convert symbolic reference to actual addresses

10–10. Data Structures and Flow of Control

Multiple Load Formats

- Format-dependent loader
  - — format recognition
  - — construction of imported symbol table
  - — provision of exported symbol table
  - — mapping of regions
  - — relocation
  - — unloading

Multiple Load Formats

Associated with each object format is a set of routines known as the *format-dependent loader*. Adding a new object format merely involves writing another version of these routines. The primary duties of these routines include:

1. *format recognition*: whenever the loader encounters a new module, it calls upon each of the format-dependent loaders in turn, essentially asking them "Is this one of yours?"

2. *construction of imported symbol table*: each module's format-dependent loader is called upon to fill its imported symbol table

3. *provision of exported symbol table*: each module provides a list of exported symbols in format-dependent form. Thus each module also provides a routine to return the value of each of the symbols it defines

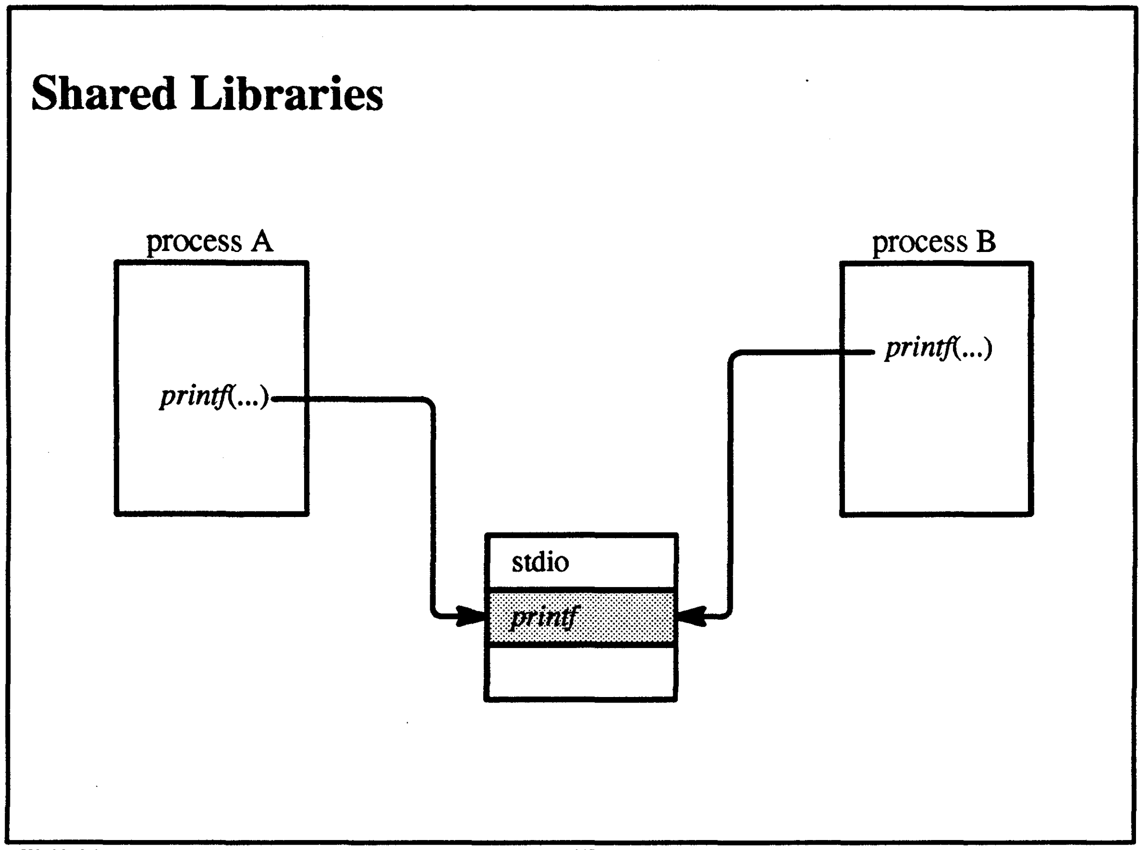
4. *mapping of regions*: each module's format-dependent loader maps the regions of the module into memory as required

5. *relocation*: the format-dependent loader performs all necessary relocation

6. *unloading*: if the module is to be unloaded, the format-dependent loader performs the necessary chores

A detailed discussion on the format-dependent portion of the loader can be found in chapter 7 of Open Software Foundation, 1990b.

10–11. The Run-time Image



H2596 10-11. 327 © 1991 Hewlett-Packard

Shared Libraries

Shared libraries is one of the most important features of the OSF/1 loader. There are a variety of ways to share code. The simplest is merely to map the shared routine into the address space. If relocation is required, the routine may be mapped copy-on-write and then appropriately modified to effect the relocation. However, if appreciable relocation is required, this technique cuts down on the amount of sharing that actually takes place.

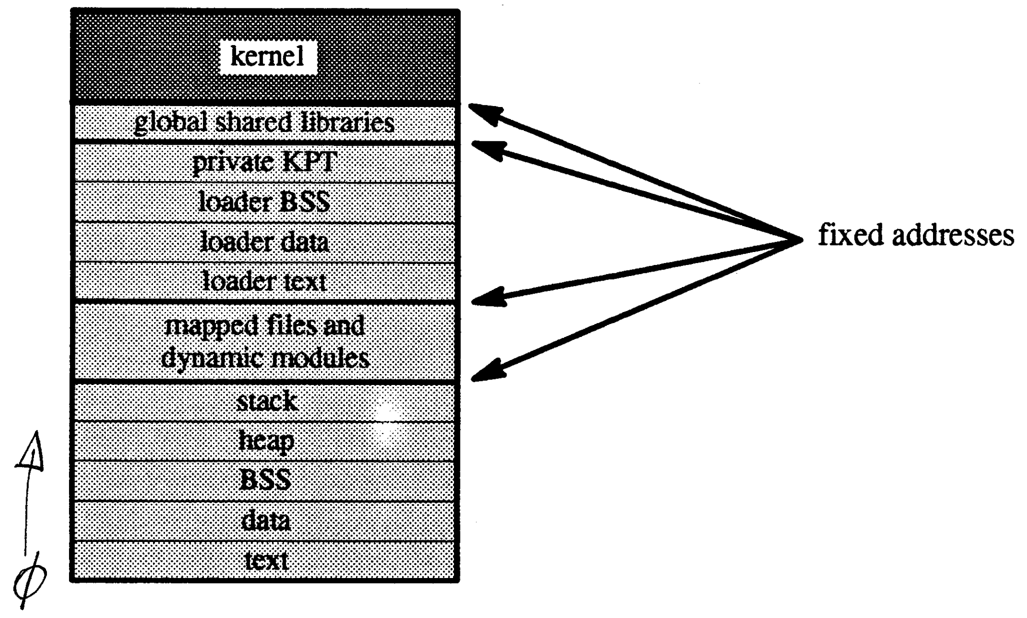
Another approach is to use position-independent code (PIC) to eliminate the need for pre-relocation. This technique is not currently supported by the OSF/1 compilers, but might be supplied by vendor-supplied compilers.

What OSF/1 does is to use pre-relocated shared libraries. Such libraries are combined into a single image which is pre-relocated to fit at a fixed location in the address space. They are thus available for linking to programs without any further relocation. The standard libraries are provided in this format and are linked in this form (as described on page 10–9) with the standard UNIX commands.

Note that symbol substitution is still possible, e.g., an alternative version of *printf* can be substituted at load time for the version in the shared library. Also, because final symbol resolution can occur at load time, the locations to which shared routines have been pre-relocated can be changed without the need for relinking.

10–12. The Run-time Image

Typical Address Space Layout

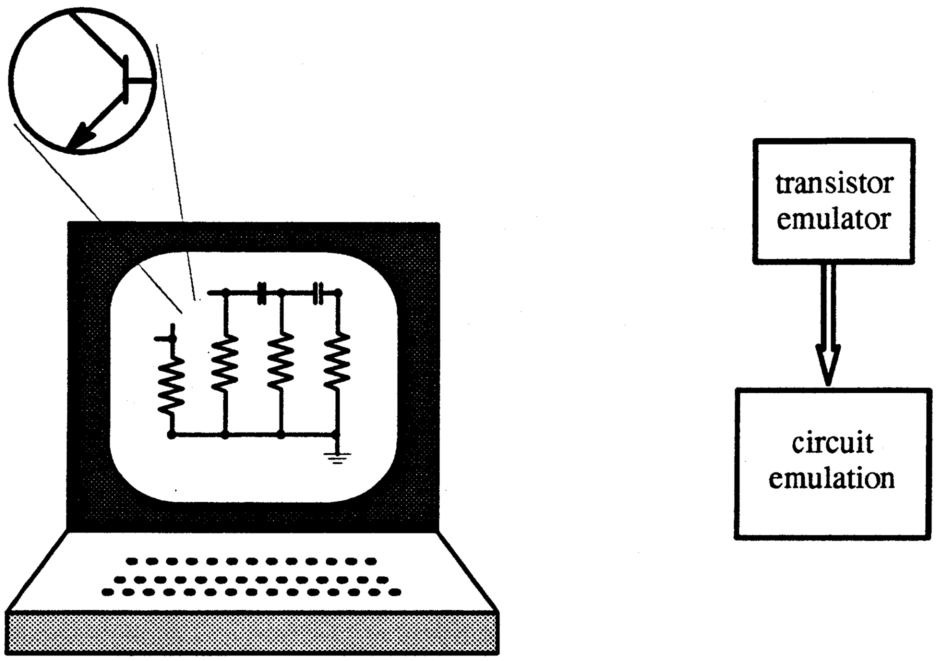


H2596 10-12. 328 © 1991 Hewlett-Packard

Typical Address Space Layout

10–13. Dynamic Loading

Run-Time Loading and Unloading



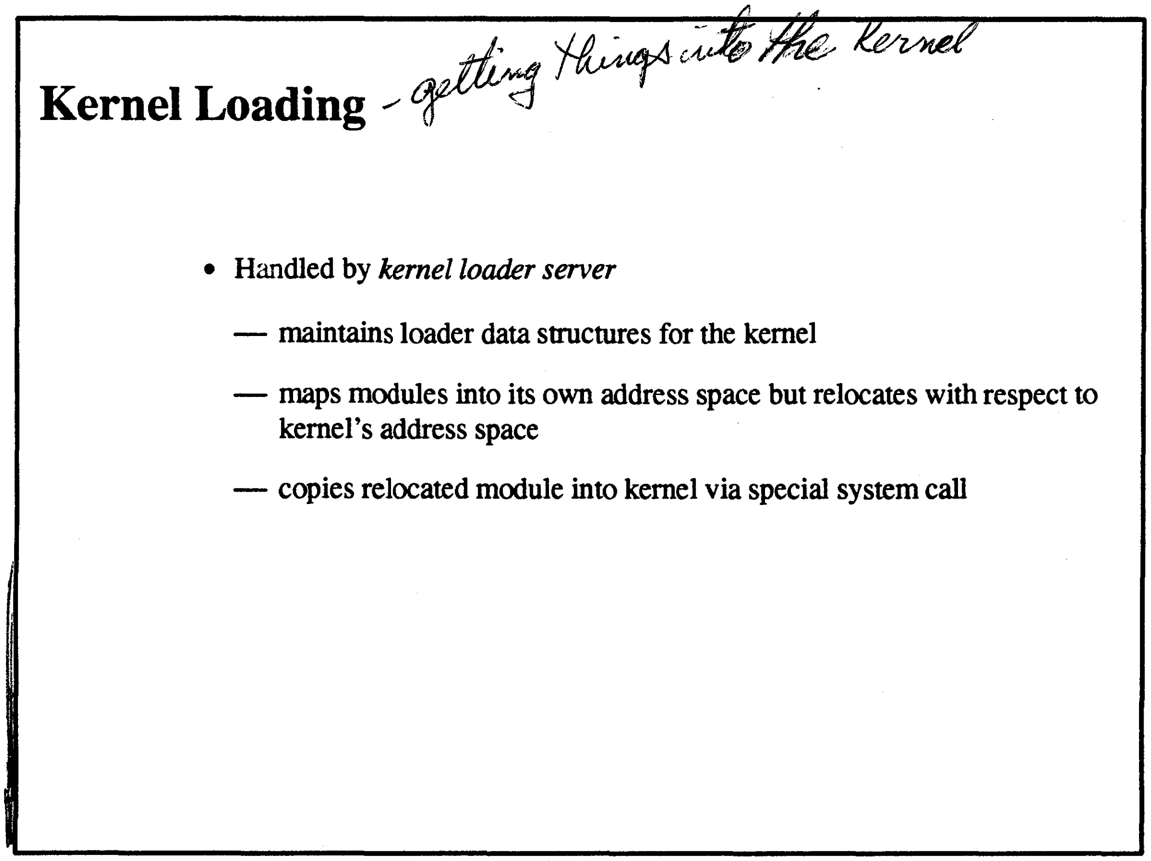
H2596 10-13. 329 © 1991 Hewlett-Packard

Run-Time Loading and Unloading

Modules may be explicitly loaded into or unloaded from a running program. For example, a user of a CAD/CAM application might put a transistor into a diagram. The CAD/CAM application might then call upon the loader to load a transistor-emulation module into the program. If the user subsequently decides to remove the transistor from the diagram, then the CAD/CAM application would call the loader to unload the emulation module as well.

To make this possible, the run-time loader remains in the address space even after the program starts up. The user program can call the loader via its *load* and *unload* entry points; all of the loader's data structures still exist, and they are updated by the loader to reflect the presence of the new module.

10–14. Dynamic Loading



Kernel Loading - getting things into the kernel

- Handled by *kernel loader server*
  - — maintains loader data structures for the kernel
  - — maps modules into its own address space but relocates with respect to kernel's address space
  - — copies relocated module into kernel via special system call

H2596 10-14. 330 © 1991 Hewlett-Packard

Kernel Loading

The OSF/1 loader can be used to load modules into or unload modules from the kernel. This feature is used in conjunction with dynamic configuration to support loadable/unloadable device drivers, streams modules and drivers, file systems, and protocols. Kernel loading is managed by a privileged user-mode task, the *kernel-loader server*. This server maintains the data structures describing the kernel address space (i.e., the same types of data structures that describe a user task).

Loading into the kernel is essentially identical to loading into a user task, except that the actual loading is done remotely: modules to be loaded into the kernel are mapped into the server's address space, but are relocated with respect to their final position in the kernel address space. Special system calls perform the actual loading into the system address space (such modules go into wired memory) and call the module's *configure* routine (so that it can link itself into kernel tables).

Exercises:

1. What functionality is provided by the OSF/1 loader that is not provided by the standard *ld* and exec?
2. a. Explain the role of packages in symbol resolution.
   b. How can one replace a routine supplied by a system library?
3. a. Why can't the LPT and the private KPT be combined into a single table?
   b. Why is it necessary to separate the format-dependent loaders from the format-independent loader?
4. a. List three techniques for implementing shared libraries.
   b. Why are loader text, data, and BSS kept separate from the standard text, data, and BSS?
5. a. List the actions taken to load a module into a running program.
   b. In what ways is loading into the kernel treated differently from loading into the current process?

Versioning for Dynamically Loaded Modules:

(1) A) Replace a file on system - Version string in Dynamic Loaded system (file)

B) Package name could have the version name.

---

(2) Shared Libraries? Semantics changes?
Package name - new library changes name.

Module 11 — Security

Module Contents

Module Objectives

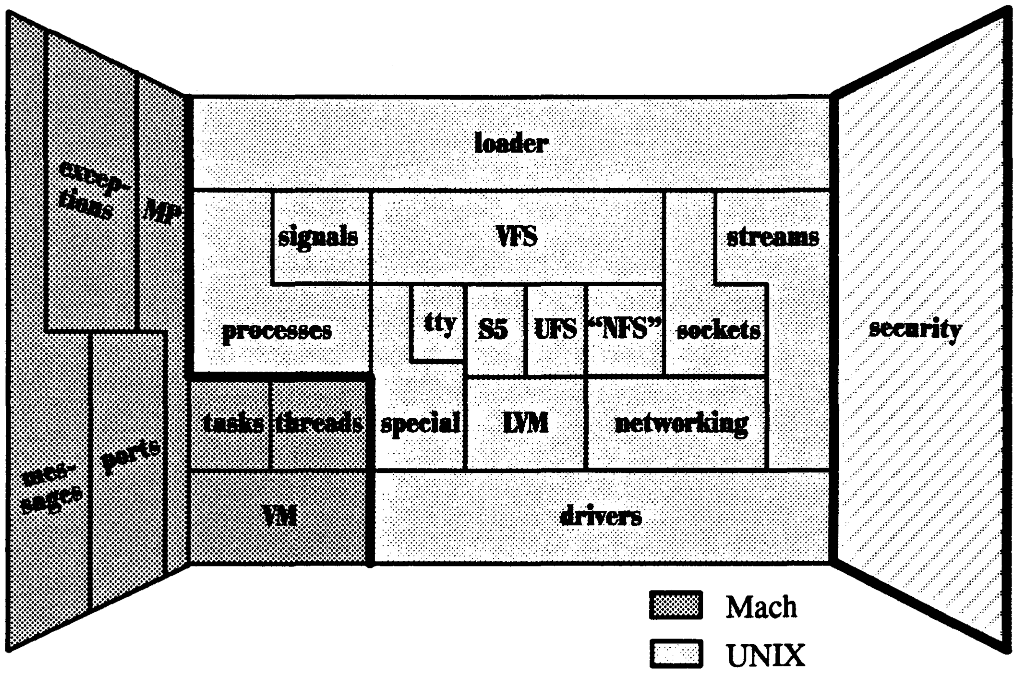
In order to understand security in the OSF/1 environment, the student should be able to:

- describe how OSF/1 is compliant with the Orange Book security model
- describe how audit information is collected
- explain how it is determined whether a particular subject has the desired access to a particular object
- explain how authorizations and privileges exceed traditional security measures

Module 11 — Security

11–1. The Big Picture

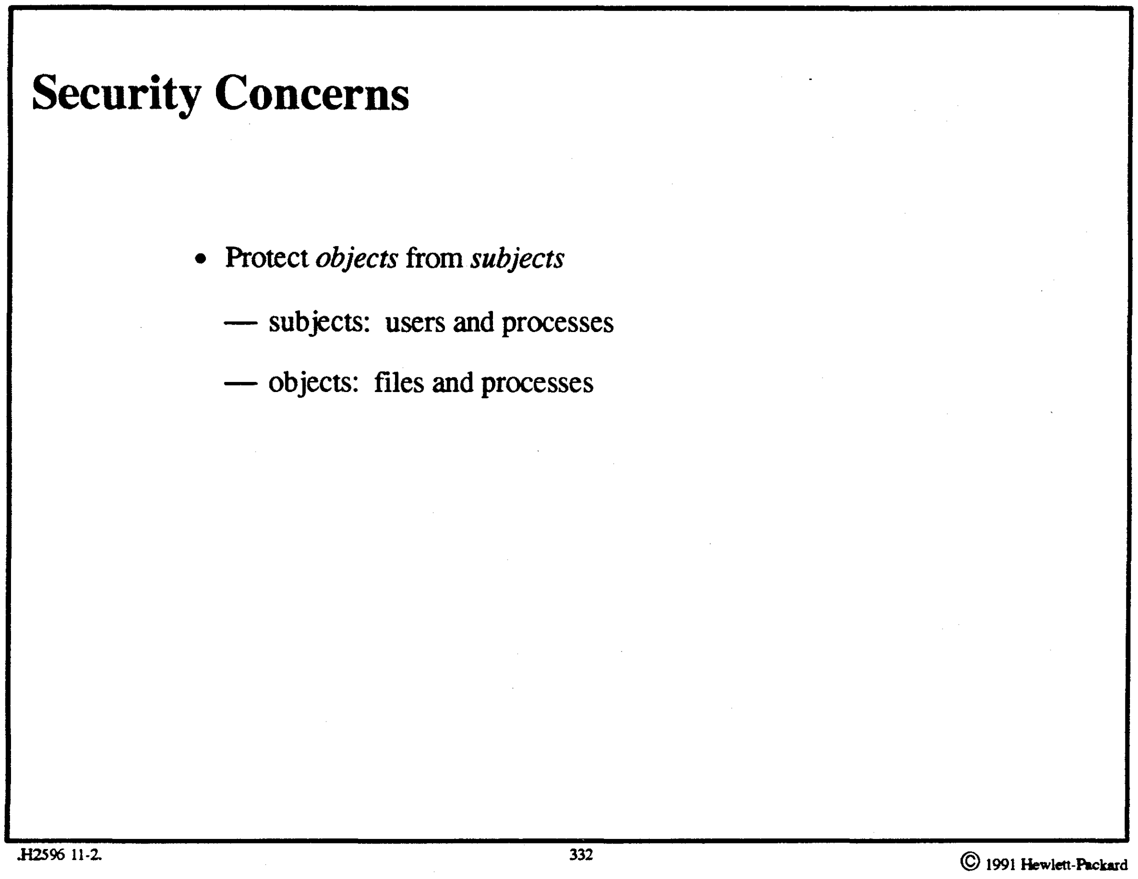
Security



.H2596 11-1. 331 © 1991 Hewlett-Packard

Security

11–2. Security Concerns



Security Concerns

- Protect *objects* from *subjects*
  - — subjects: users and processes
  - — objects: files and processes

.H2596 11-2. 332 © 1991 Hewlett-Packard

Security Concerns

11–3. Security Concerns

The "Orange Book" Model

- division D: minimal protection
- division C: discretionary protection
  - — class C1: discretionary security protection
  - — class C2: controlled access protection
- division B: mandatory protection
  - — class B1: labeled security protection
  - — class B2: structured protection
  - — class B3: security domains
- division A: verified protection
  - — class A1: verified design

.H2596 11-3. 333

The "Orange Book" Model

The U. S. Department of Defense (DoD) trusted-computer-evaluation criteria, known as the "Orange Book" model (so called because of the color of its cover), define criteria for classifying computer systems according to their degree of protection.

Division D contains those systems whose security features have been evaluated and have flunked.

Division C, a very minimal level of security, contains two classes. To be in class C1, a system must provide controls so that users can protect private information and keep others from *accidentally* reading or destroying data. The model is of cooperating users processing data at the same levels of security. Most UNIX systems should fit within this class easily.

To be in class C2, a system must meet all of the requirements for class C1 and, in addition, users of the system must be individually accountable for their actions through login procedures, auditing, and resource isolation. UNIX systems may relatively easily aspire to be in this class.

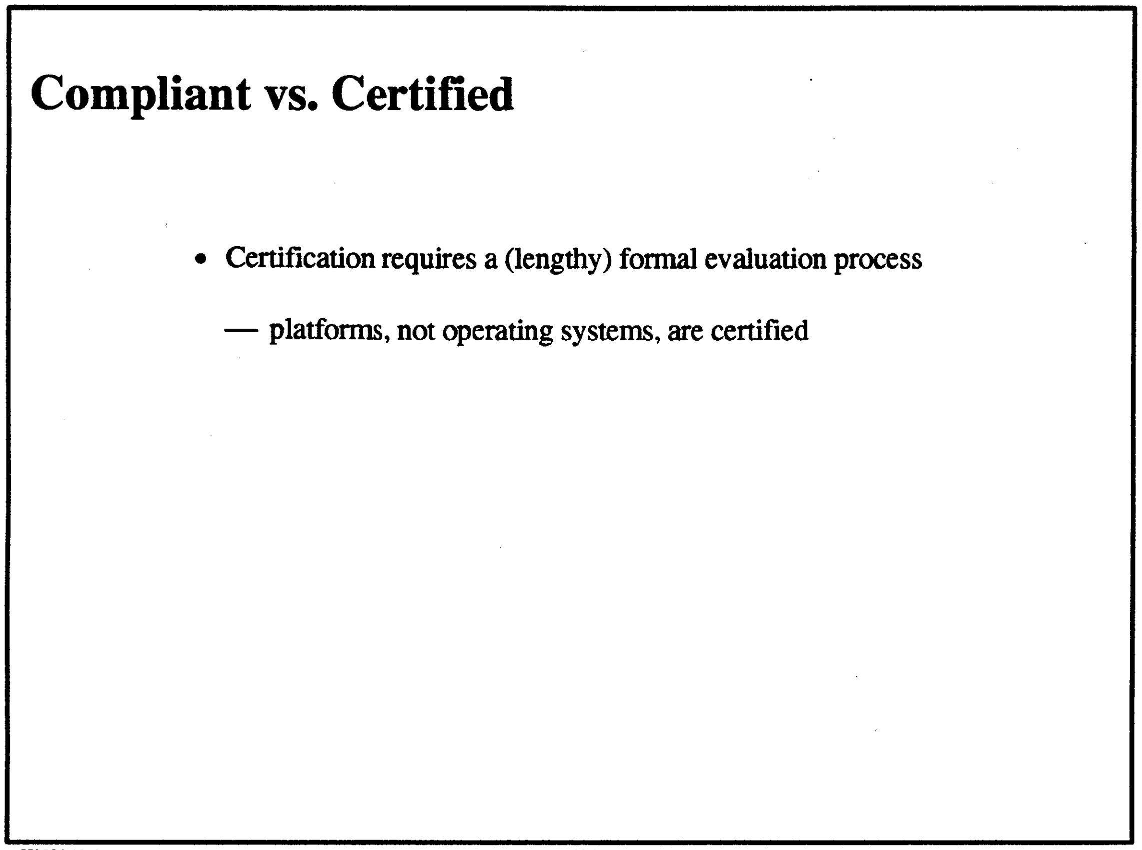
If a system is certified to be in division B, then it must be realistically considered secure. To be in class B1, a system must meet all the requirements for class C2. In addition, it must have an informal statement of the security model and must provide data labeling and mandatory access control over named subjects and objects. The capability must exist for accurately labeling exported information (e.g., in a defense environment, Top Secret printouts must be clearly labeled as such).

To be in class B2, a system must meet all the requirements for class B1 but, instead of an informal statement of the security policy model, there must be a clearly defined and documented *formal* security policy model. The discretionary and mandatory access controls of B1 must extend to *all* subjects and objects, much more thorough testing and review is required, and very stringent configuration management controls are necessary.

To be in class B3, a system must meet all requirements for class B2. In addition, its trusted computing base (TCB), i.e. that portion of the system that runs in privileged mode, must be small enough to be subjected to rigorous analysis and test. All accesses of subjects to objects must be mediated, the system must be tamper-proof, a *security administrator* must be supported, audit mechanisms must be expanded to *signal* security-relevant events, and detailed system recovery procedures must be in place.

The primary difference between class A1 and and class B3 is that the formally specified design must be formally verified. Going beyond class A1, if such classes were defined, might involve a formally verified implementation, which is considered beyond the state of the art.

11–4. Security Concerns



Compliant vs. Certified

- Certification requires a (lengthy) formal evaluation process

  — platforms, not operating systems, are certified

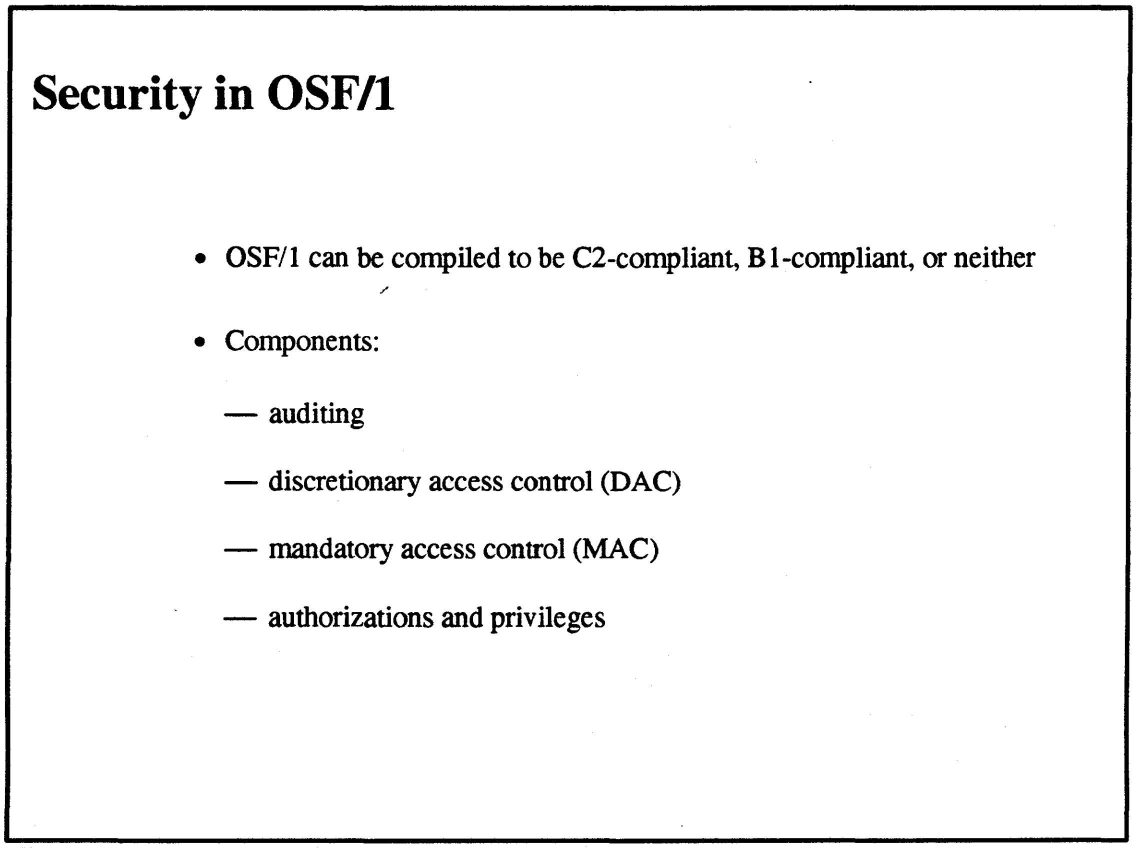
.H2596 11-4. 334 © 1991 Hewlett-Packard

Compliant vs. Certified

Certification requires a formal evaluation process. It is not merely the operating system that is being evaluated, but also the implementation of the operating system on a particular architecture with a given set of options. OSF supplies most of the voluminous documentation required for certification.

The "Orange Book" criteria are strictly for stand-alone systems. A system that is networked cannot be secure under these criteria. Thus, for example, a B1-certified system cannot contain NFS.

11–5. Security Concerns



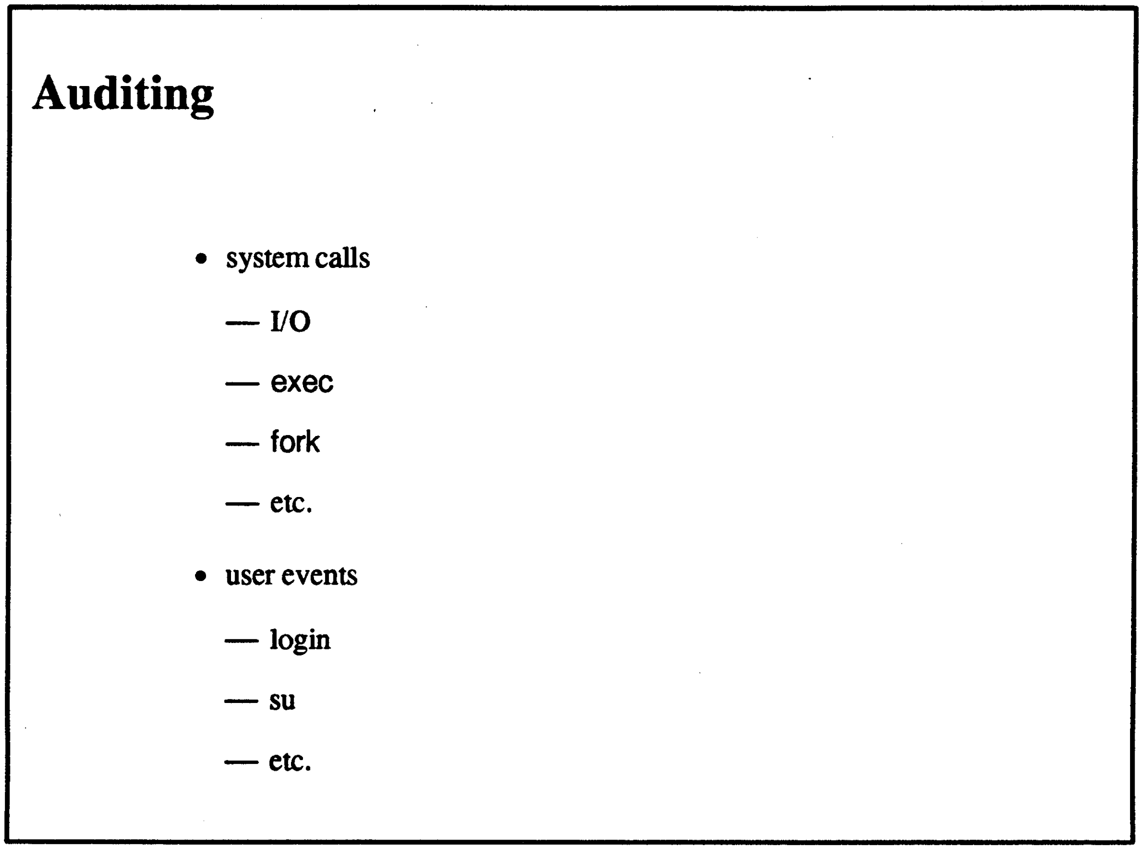
Security in OSF/1

- OSF/1 can be compiled to be C2-compliant, B1-compliant, or neither
- Components:
  - — auditing
  - — discretionary access control (DAC)
  - — mandatory access control (MAC)
  - — authorizations and privileges

H2596 11-5. 335

Security in OSF/1

11–6. Auditing



Auditing

- system calls
  - I/O
  - exec
  - fork
  - etc.
- user events
  - login
  - su
  - etc.

H2596 11-6. 336 © 1991 Hewlett-Packard

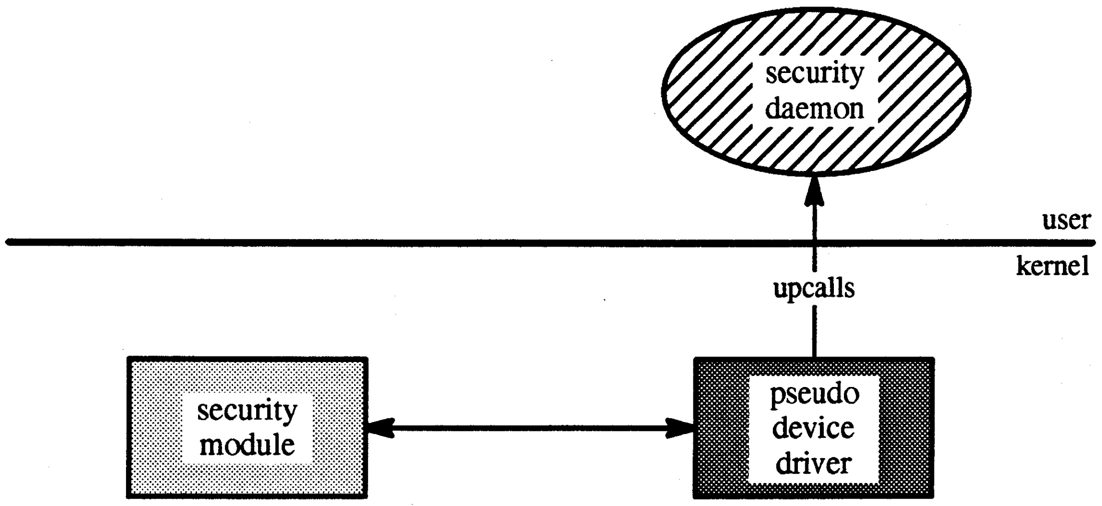
Auditing

Auditing may be used selectively; i.e., under the control of the system administrator, selected system calls and user events may be recorded.

Stub routines inserted in the control path collect information for system calls. The status of each system call is maintained in the *audit_info* structure, which is allocated on the kernel stack. When an audited system call completes, the *audit_info* structure is put into a buffer maintained by the *audit device driver*. This driver makes the information available to the *audit daemon*, which compacts it and stores it in a database.

11–7. Auditing

Kernel/Daemon Communication



H2596 11-7. 337 © 1991 Hewlett-Packard

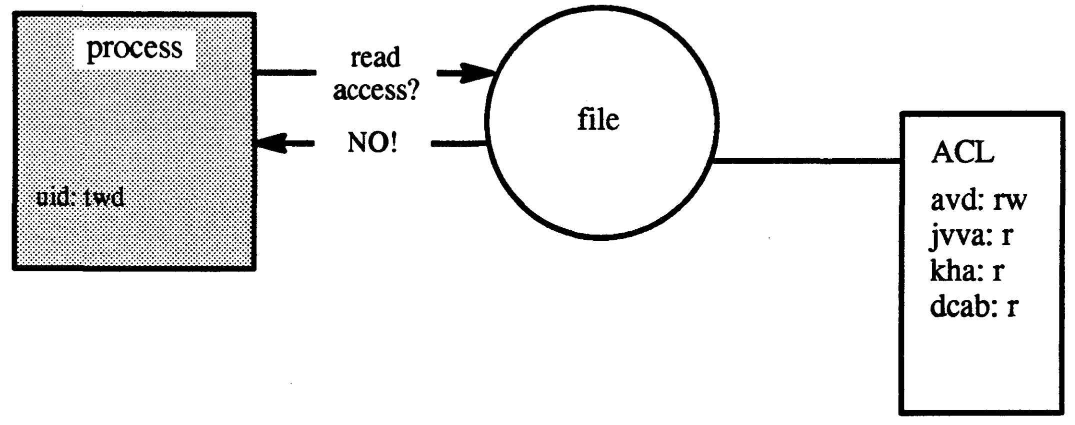
Kernel/Daemon Communication

A fair amount of the processing performed by the security system is done within user-level *daemons*. The security architecture was designed for general UNIX systems: it does not assume the communication facilities of OSF/1. Thus communication between the daemons and the kernel must be implemented in a way that can be easily ported to any UNIX system. The interface chosen is that of a *pseudo device*. The security daemons may perform standard read, write, and ioctl system calls on their associated pseudo devices. These devices are represented in the kernel as *pseudo-device drivers*.

Upcalls, e.g. requests sent to the daemons by the kernel, are implemented by having the daemon make a read system call. The call blocks until the pseudo-device driver has an upcall to make. When the read returns in the daemon, the result contains the *upcall* request.

11–8. Access Control

Discretionary Access Control (DAC)

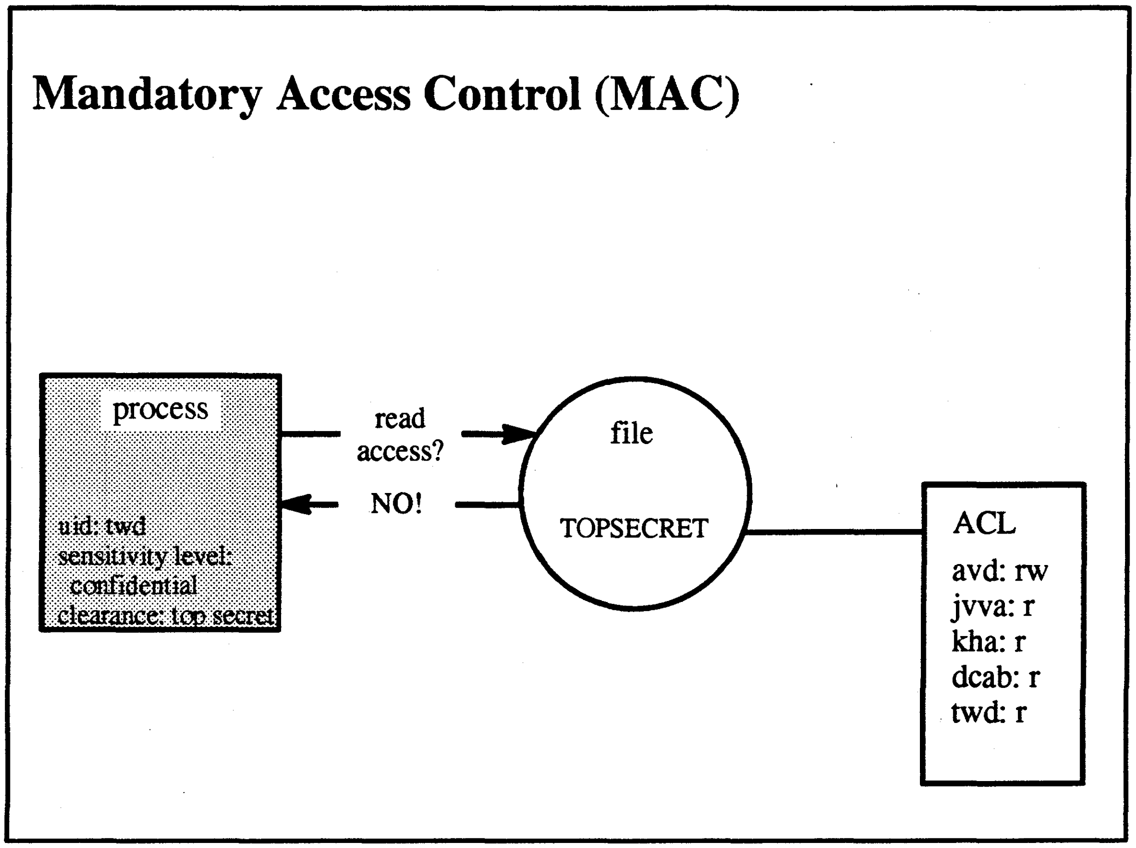


H2596 11-8. 338 © 1991 Hewlett-Packard

Discretionary Access Control (DAC)

By *discretionary access control* we mean the ability of an object's creator/owner to specify who has what sort of rights to the object. In OSF/1, the traditional UNIX security policy (a discretionary policy) has been augmented with the use of *access control lists* (ACLs). Associated with each object (i.e. file) is a list of all users allowed to access it and what their access rights are. ACLs extend the normal UNIX discretionary policy by increasing its flexibility. For example, with ACLs it is easy to prohibit access to certain individuals.

11–9. Access Control



H2596 11-9. 339 © 1991 Hewlett-Packard

Mandatory Access Control (MAC)

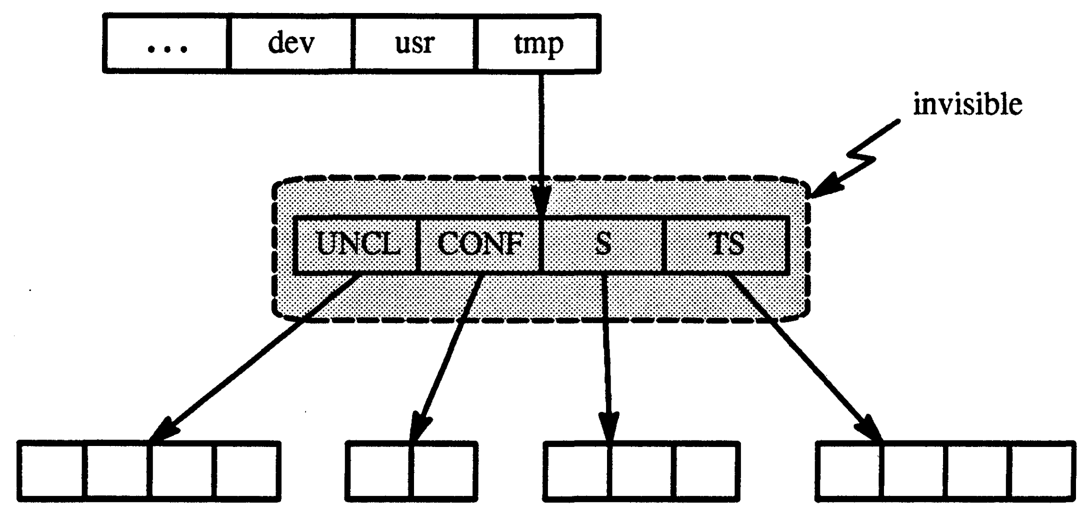
Mandatory access control involves enforced restrictions on objects that cannot be changed at the discretion of the creator or owner or user. Subjects are assigned *levels of trust* (clearances) and objects are assigned *degrees of sensitivity.* Both notions are represented with *sensitivity labels* and combine a hierarchical classification with a non-hierarchical set of categories (or compartments).

DoD security classifications are an example of a hierarchical classification—Unclassified < Confidential < Secret < Top Secret < Eyes-Only. Combined with this is the non-hierarchical notion of compartments, e.g. NATO, NORAD. Thus, to view a Top Secret NATO document, it is not enough to be cleared for Top Secret; one must also be working within NATO.

Various rules are established governing access to information. For example, one cannot modify a Secret file while executing in a Top Secret domain, but one can read a Secret document while in a Top Secret domain.

11–10. Access Control

Mandatory Access Control: Multilevel Directories



H2596 11-10. 340 © 1991 Hewlett-Packard

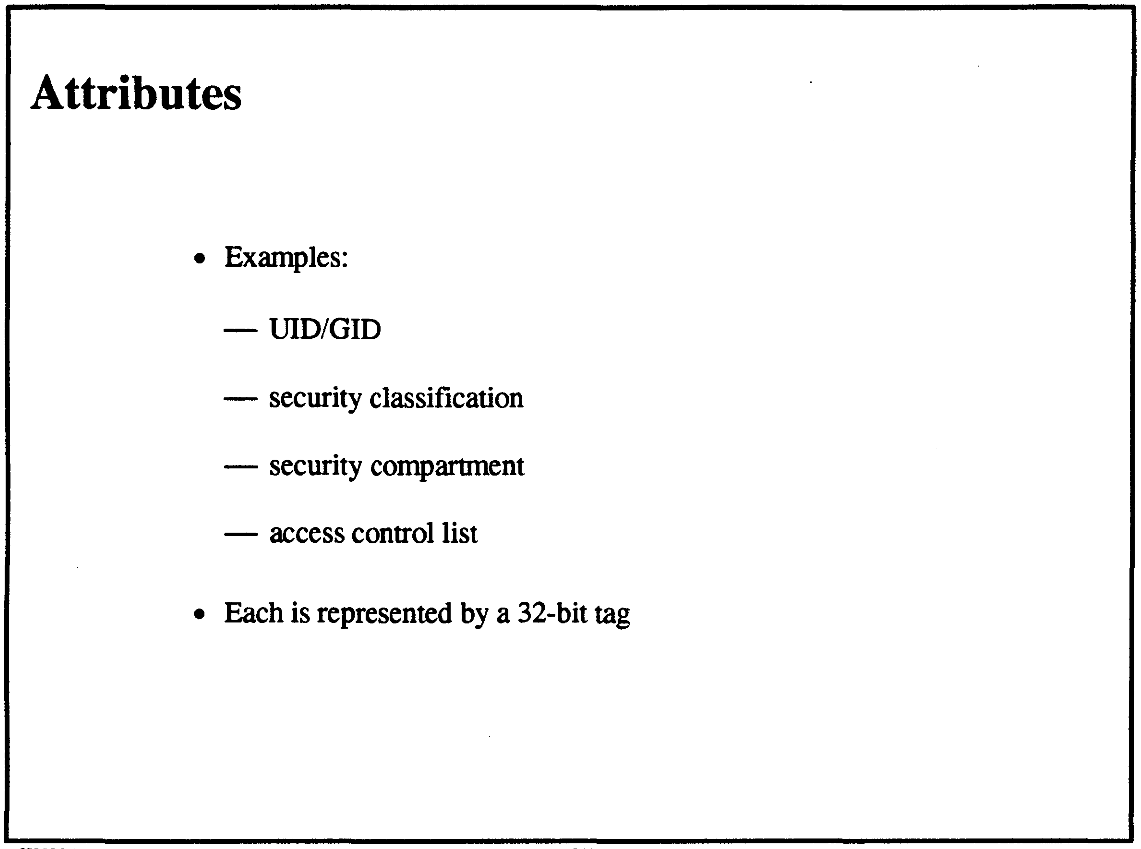
Mandatory Access Control: Multilevel Directories

Public directories (such as /tmp) need special treatment when used in conjunction with the mandatory access control policy. For example, various programs such as the C compiler use the /tmp directory to hold their temporary files. Even though the contents of these files may be securely protected, the existence of a temporary file as well as its name could be easily discovered by anyone performing an *ls* on the /tmp directory. Such information about Top Secret temporaries, for example, should not be known to those operating in the Secret domain.

Furthermore, if a direcdtory is, for example, Top Secret, then a process whose sensitivity level is Secret wouldn't be able to add files to it, since this would require modifications to the directory. If the directory was Secret, then a Top Secret process couldn't write to it, because this would allow the leakage of information to a lower sensitivity level.

One approach to dealing with this problem might be to change all applications so that they do not use public directories but instead find directories at appropriate security levels. A better approach is to deal with the problem transparently. A directory such as /tmp may be set up as a *multilevel* directory. It is then transparently split into a number of subdirectories, one for each security classification. Thus a reference to the file /tmp/xyz by a process executing as Top Secret would be translated into a reference to the file /tmp/topsecret/xyz.

11–11. Access Control



Attributes

- Examples:
  - — UID/GID
  - — security classification
  - — security compartment
  - — access control list
- Each is represented by a 32-bit tag

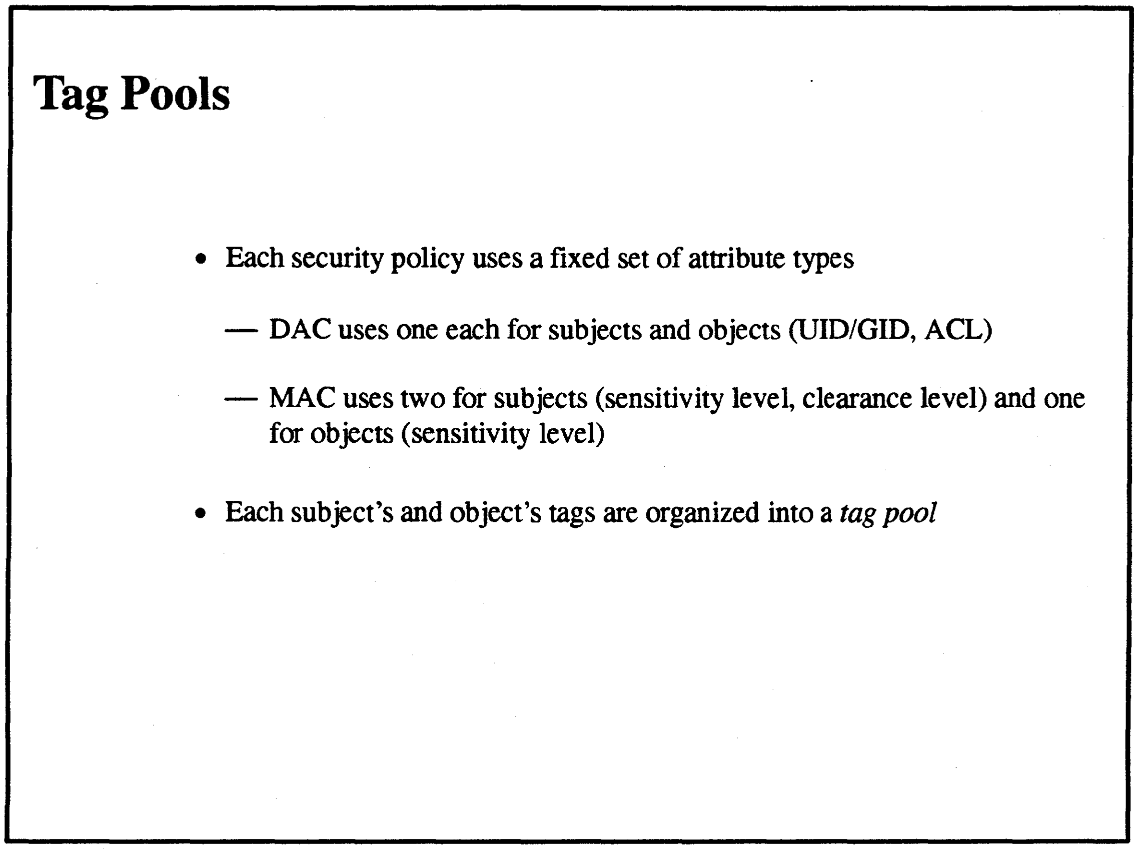
H2596 11-11. 341

Attributes

Each security policy must deal with a set of *attributes* on subjects and objects. These attributes can be very complicated; for example, an ACL can be arbitrarily long. A direct representation of such attributes would be too complex for the kernel to manipulate easily. Instead, each attribute is represented by a 32-bit tag (thus when a new ACL is used, a new 32-bit tag is created).

Every security policy has a policy daemon that is responsible for translating tags to attributes and vice versa. Each such daemon maintains a database to aid this process; it must ensure that tags are unique. Inside the kernel, the policy modules deal only with tags. Since they are not capable of interpreting the tags, they can only make equality comparisons; any other manipulation must be forwarded to the user-level policy daemon.

11–12. Access Control



Tag Pools

- Each security policy uses a fixed set of attribute types
  - — DAC uses one each for subjects and objects (UID/GID, ACL)
  - — MAC uses two for subjects (sensitivity level, clearance level) and one for objects (sensitivity level)
- Each subject's and object's tags are organized into a *tag pool*

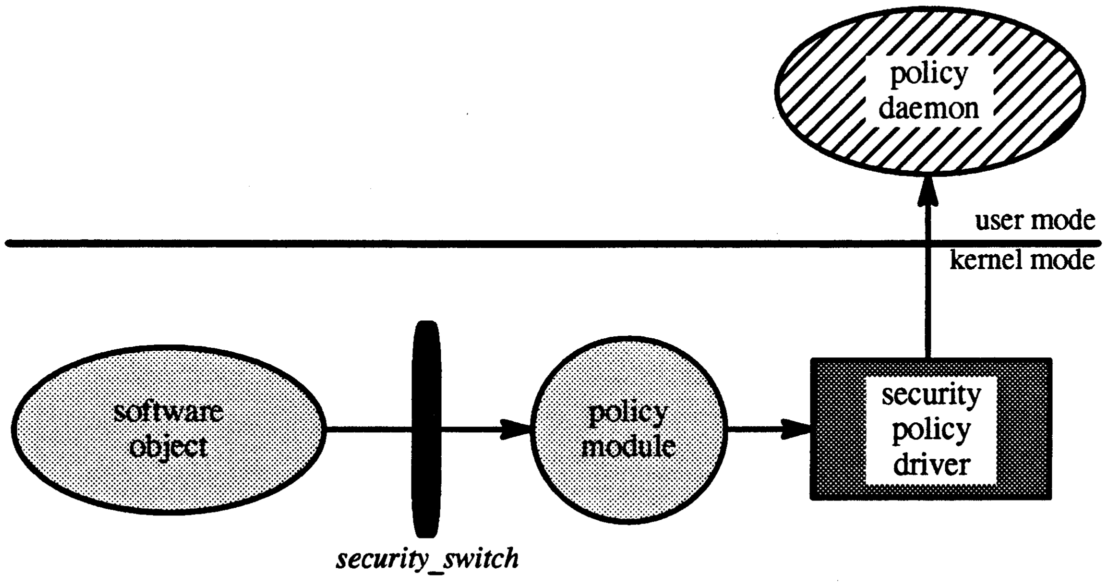
.H2596 11-12. 342

Tag Pools

Each subject and object must be associated with its attributes with respect to each security policy. In OSF/1, each subject must have three tags (one for DAC and two for MAC) and each object must have two tags (one each for DAC and MAC). The collection of tags associated with a subject or an object is known as a *tag pool.*

11–13. Access Control

Security Policy Architecture



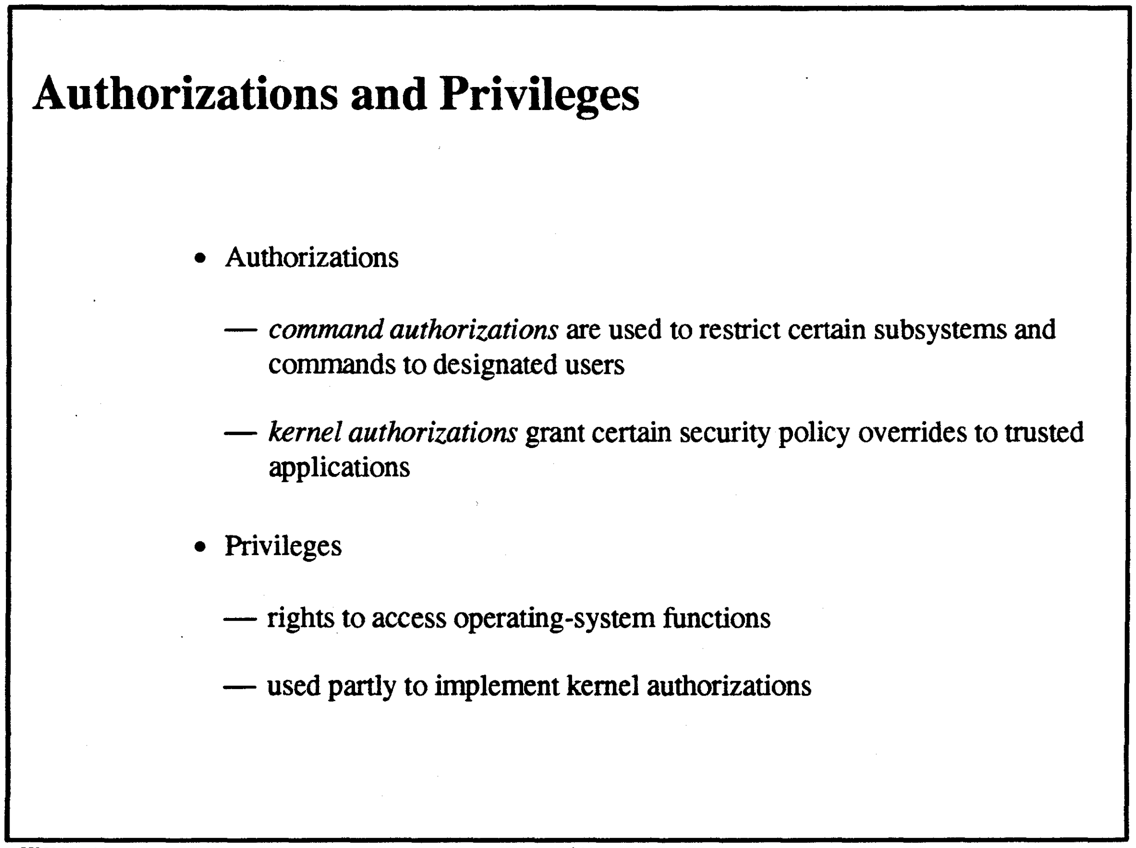
H2596 11-13. 343 © 1991 Hewlett-Packard

Security Policy Architecture

Whenever a security decision has to be made or action taken, each security policy must be consulted. In OSF/1, there are two such policies, DAC and MAC. Thus, for example, if a process attempts to open a file, both DAC and MAC are consulted to ensure that both will allow the access. If either says no, the access is denied.

The security decisions are implemented through the *security switch.* Each security policy provides a *policy module* in the kernel whose entry points are contained in the security switch. When a subject attempts to access an object, the macro SP_ACCESS is called, which calls the access entry point of each policy (via the security switch), passing to the policy module the subject's attributes and the object's attributes. Each policy module maintains a cache of recent security decisions, which consists of relations on attributes as represented by tags. If the decision cannot be made based upon the contents of the cache, then the policy module forwards the request via an *upcall* to its policy daemon, which then makes the decision.

11–14. Authorizations and Privileges



Authorizations and Privileges

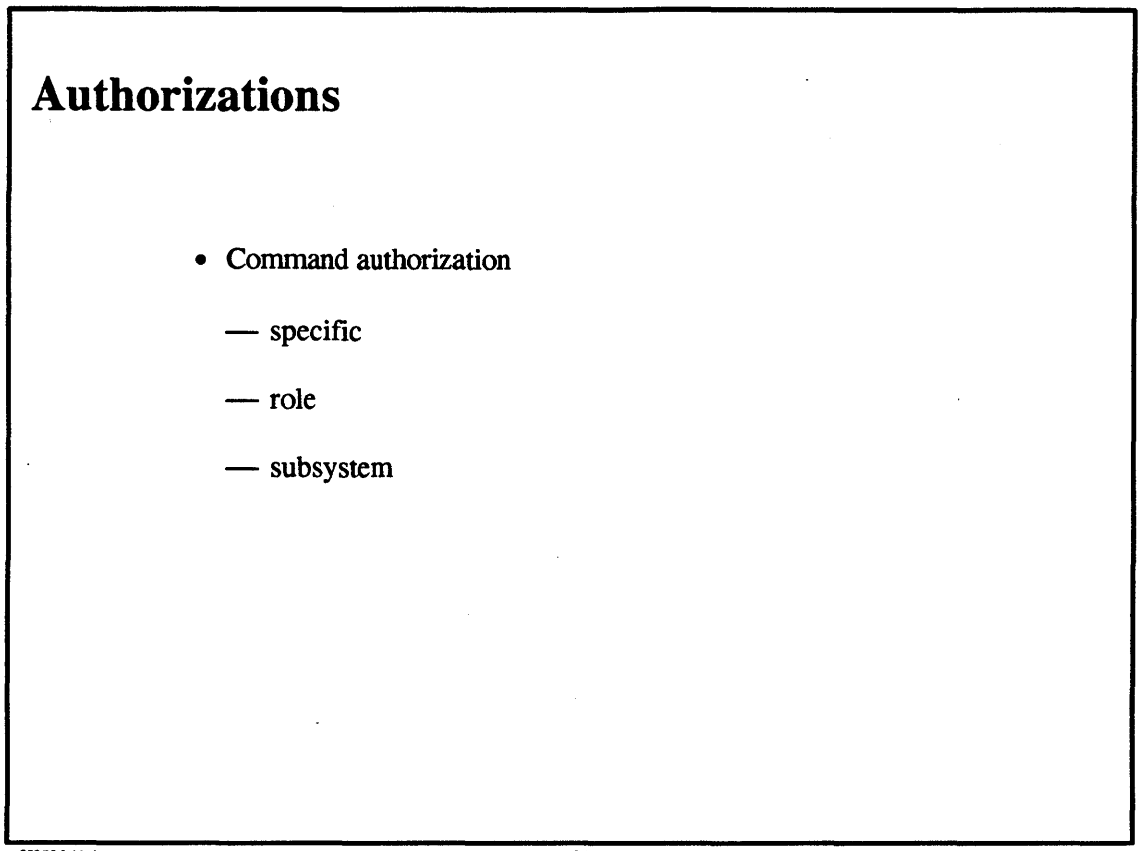
- Authorizations
  - — *command authorizations* are used to restrict certain subsystems and commands to designated users
  - — *kernel authorizations* grant certain security policy overrides to trusted applications
- Privileges
  - — rights to access operating-system functions
  - — used partly to implement kernel authorizations

.H2596 11-14. 344 © 1991 Hewlett-Packard

Authorizations and Privileges

A manage of an object finds out if the subject process is *authorized* by checking a list of authorized subjects, whereas a *privilege* is an actual property of the process, stored with the process.

11–15. Authorizations and Privileges



Authorizations

- Command authorization
  - — specific
  - — role
  - — subsystem

H2596 11-15. 345 © 1991 Hewlett-Packard

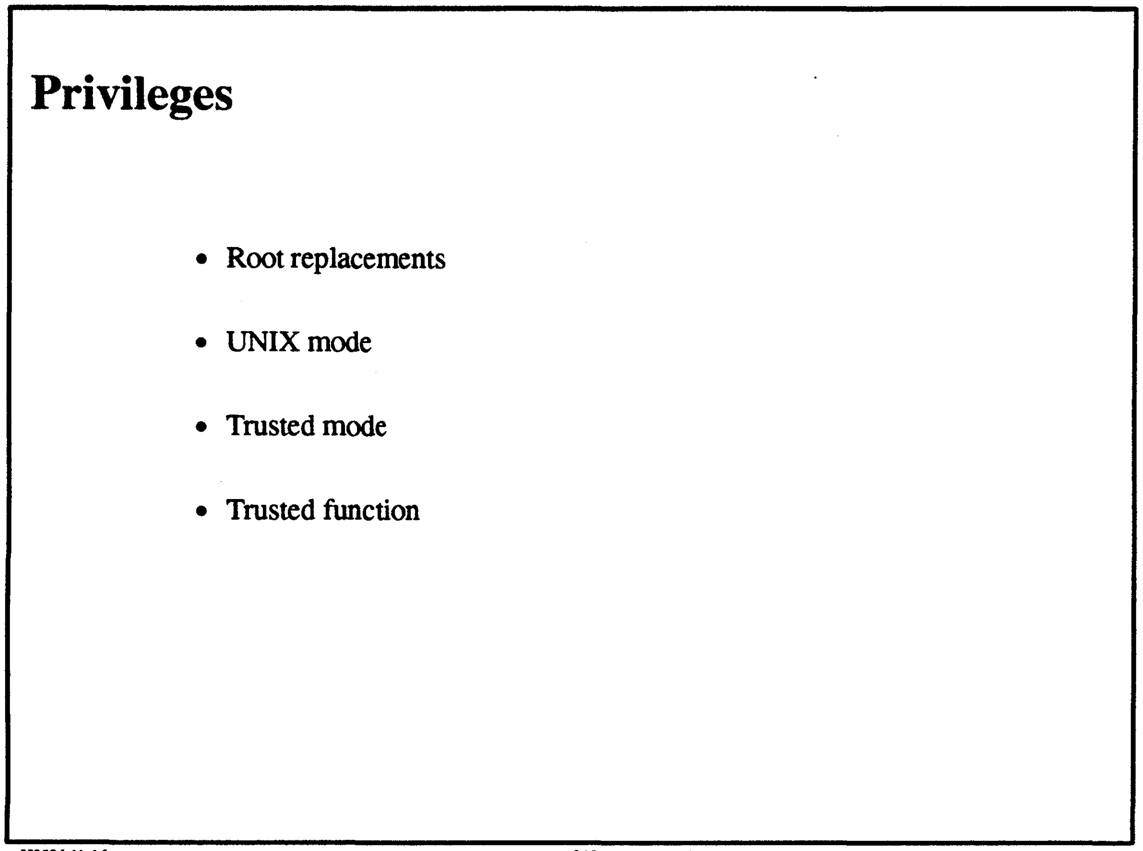
Authorizations

Command authorizations are associated with particular users and are maintained by user-level code and databases. These authorizations are broken into the following categories:

- specific
  - — allows the user to execute a command to perform a specific functions. E.g., the *mknod* authorization allows the user to invoke the *mknod* command to create special files
- role
  - — allows the user to perform tasks associated with some specific system role. E.g., the *isso* authorization allows the user to administer the security system

- subsystem
  - — grants the user additional rights in certain subsystems. E.g., the *lp* authorization allows the user to use the administrative and command options of the *lp* subsystem

11–16. Authorizations and Privileges



Privileges

- Root replacements
- UNIX mode
- Trusted mode
- Trusted function

H2596 11-16. 346 © 1991 Hewlett-Packard

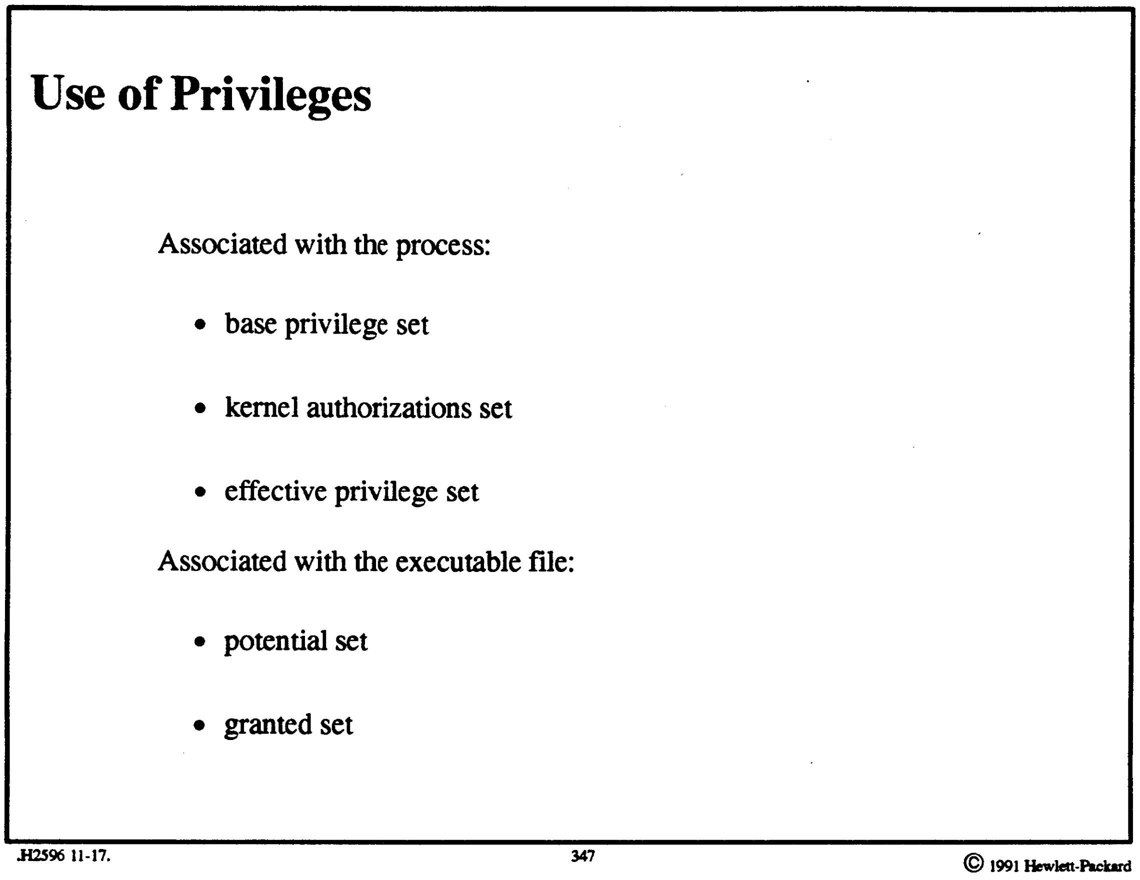
Privileges

Instead of the superuser/mere-mortal dichotomy of UNIX, OSF/1 uses a much finer breakdown of privileges, dividing them into the following categories:

- root replacements
  - — a breakdown of the privileges once reserved for the root into a set of rights that can be individually granted; e.g., the *sysattr* privilege allows one to invoke system calls that change system attributes such as the time of day.
- UNIX mode
  - — privileges that ordinary users have in UNIX but may be restricted in OSF/1; e.g., one must have the *execsuid* privilege to execute SETUID programs.

- trusted mode
  - privileges allowing a process to operate in modes that gain it special treatment with respect to trusted system features; e.g., the *suspendaudit* privilege allows a process to stop the kernel from collecting audit records.
- trusted function
  - privileges allowing a process to define new trusted functions; e.g., the *writeaudit* privilege allows a process to append records to the audit trail.

11–17. Authorizations and Privileges



Use of Privileges

Associated with the process:

- base privilege set
- kernel authorizations set
- effective privilege set

Associated with the executable file:

- potential set
- granted set

H2596 11-17. 347 © 1991 Hewlett-Packard

Use of Privileges

The *base privilege set* is the set of privileges that is always granted to a process when it execs a file.

The *kernel authorization set* is the set of privileges for which the process's user is authorized.

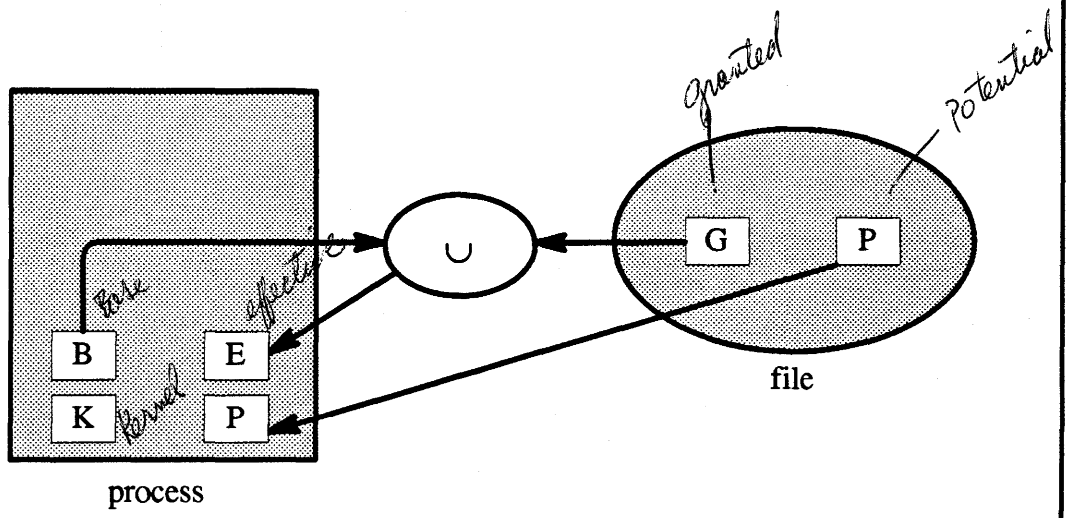
The *effective privilege set* is the set of privileges that are currently being used when the kernel checks for privileges.

The *potential set* is the set of privileges that a program may use.

The *granted set* is the set of privileges that is placed in a process's effective privilege set when the file is exec'd.

11–18. Authorizations and Privileges

Exec'ing a File



H2596 11-18. 348 © 1991 Hewlett-Packard

Exec'ing a File

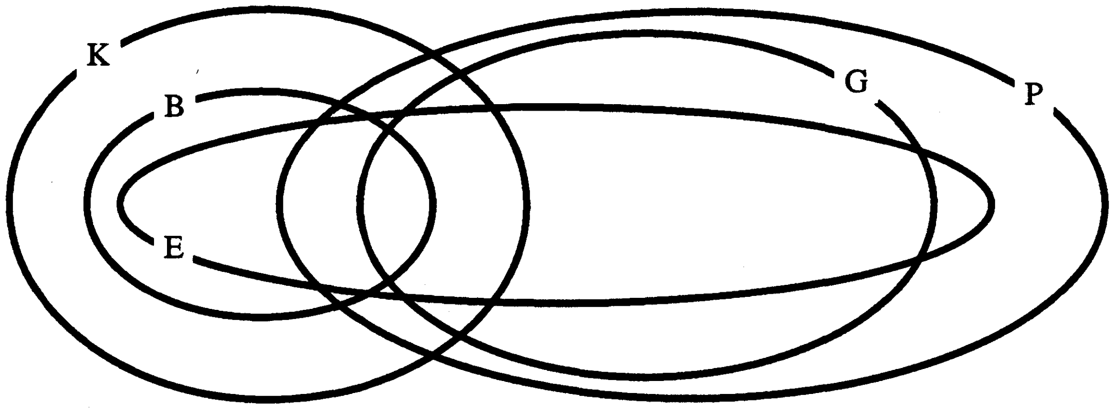
Privilege sets are represented as bit vectors. Associated with each process are four such bit vectors, for the *base privilege set*, the *kernel authorization set*, the *effective privilege set*, and the *potential set.*

Each inode contains bit vectors for the *granted* and *potential* sets.

When a user execs a program provided by another user, we have two mutually suspicious parties. Each party, i.e., the owner of the process and the owner of the executable file, provides an initial set of privileges that form the initial effective set of this joint venture. This effective set may be enlarged, but only subject to constraints provided by both parties. Privileges can be added to the effective set that are either in the potential set or the base set. The base set can itself be enlarged, but only be adding to it privileges that are in both the kernel authorization set and the potential set. This protects both parties in the event that another file is exec'd. The base set used with this new file would contain only those privileges allowed by both parties. This technique prevents privileges from being combined in unforeseen ways.

11–19. Authorizations and Privileges

Privilege Set Relationships



K: kernel authorizations
B: base privilege set
E: effective privilege set

G: granted set
P: potential set

H2596 11-19. 349 © 1991 Hewlett-Packard

Privilege Set Relationships

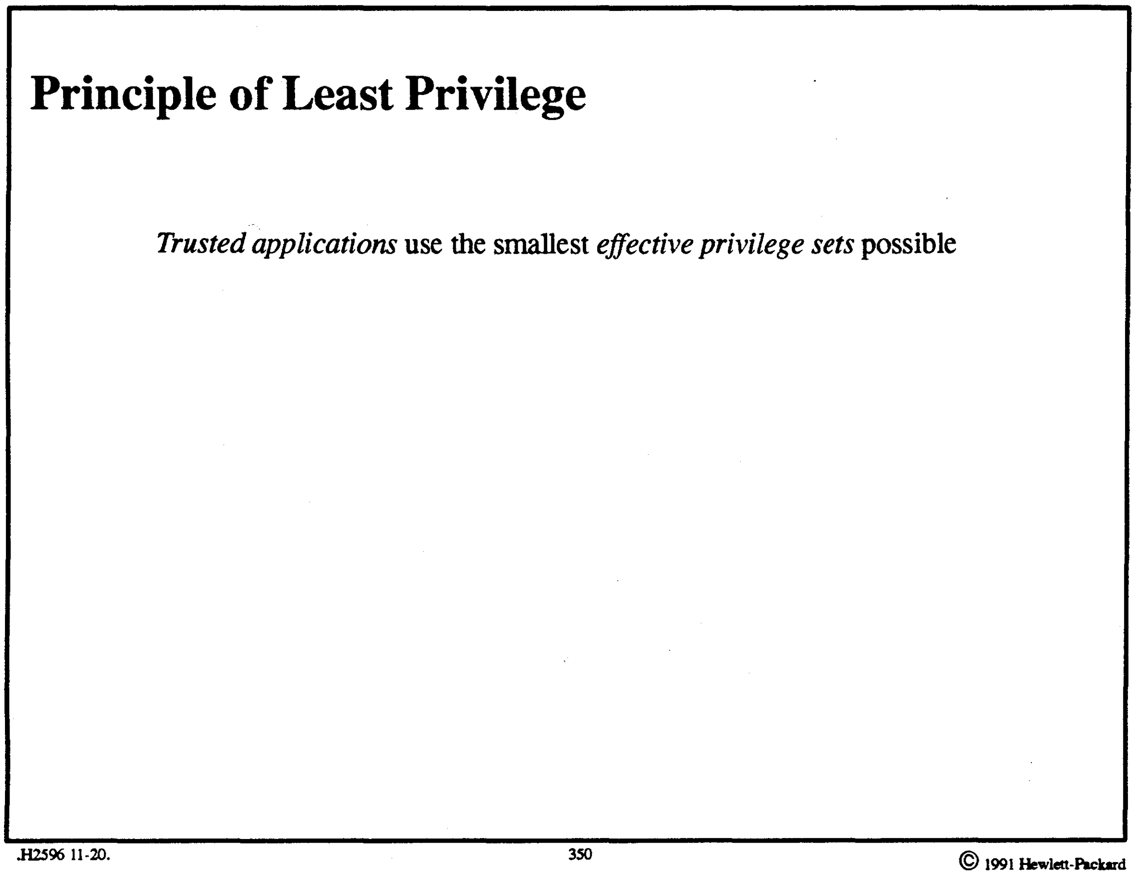
A set of kernel authorizations defines the limit of what privileges this process is allowed to have. It is a subset of the privileges available to the user. The potential set limits the privileges of a process executing the program contained in a file.

$B \subseteq K$

$E \subseteq (P \cup B)$

$G \subseteq P$

11–20. Authorizations and Privileges



Principle of Least Privilege

Trusted applications use the smallest *effective privilege sets* possible

.H2596 11-20. 350 © 1991 Hewlett-Packard

Principle of Least Privilege

Good practice dictates that the effective privilege set should be kept as small as possible. The setpriv system call allows a process to adjust its sets *K*, *B*, and *E*, subject to the constraints:

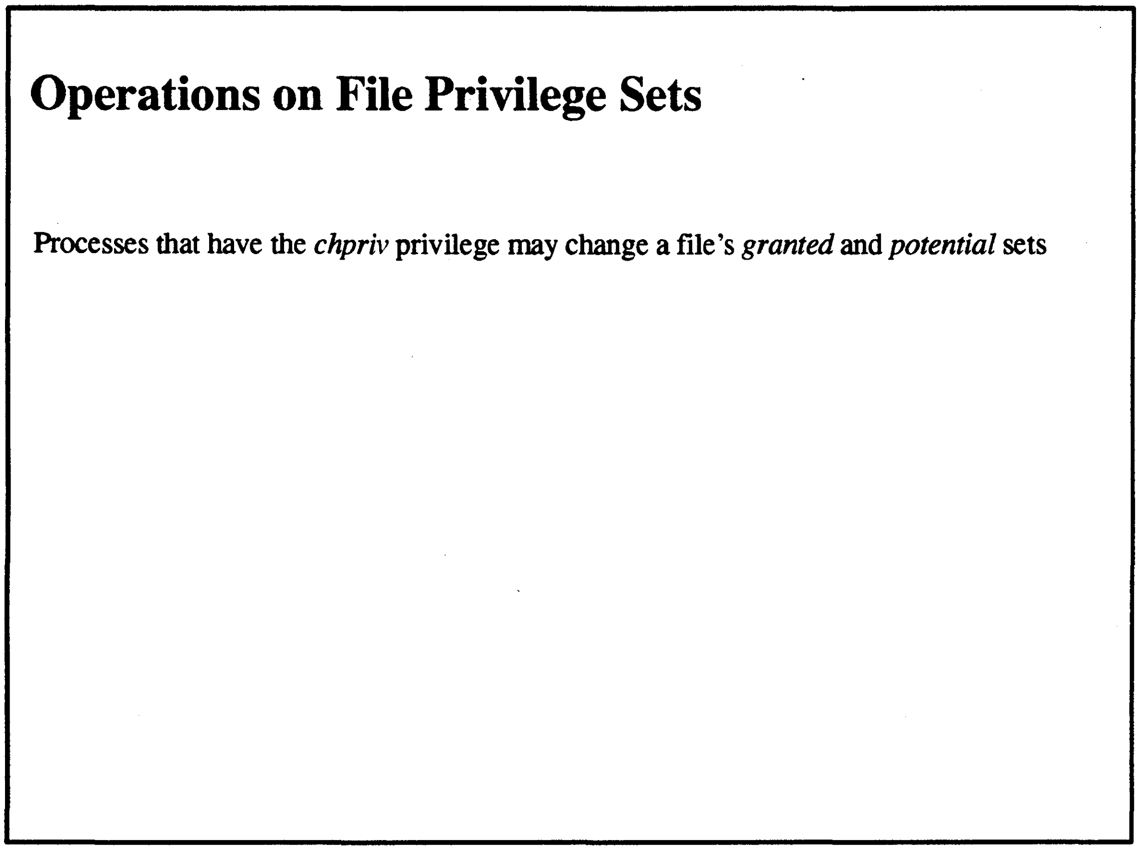
$$K' \subseteq K$$

$$B' \subseteq (K \cap P) \cup B$$

$$E' \subseteq P \cup B$$

(where *K'*, *B'*, and *E'* are the sets *K*, *B*, and *E* after a setpriv request).

11–21. Authorizations and Privileges



Operations on File Privilege Sets

Processes that have the *chpriv* privilege may change a file's *granted* and *potential* sets

H2596 11-21. 351 © 1991 Hewlett-Packard

Operations on File Privilege Sets

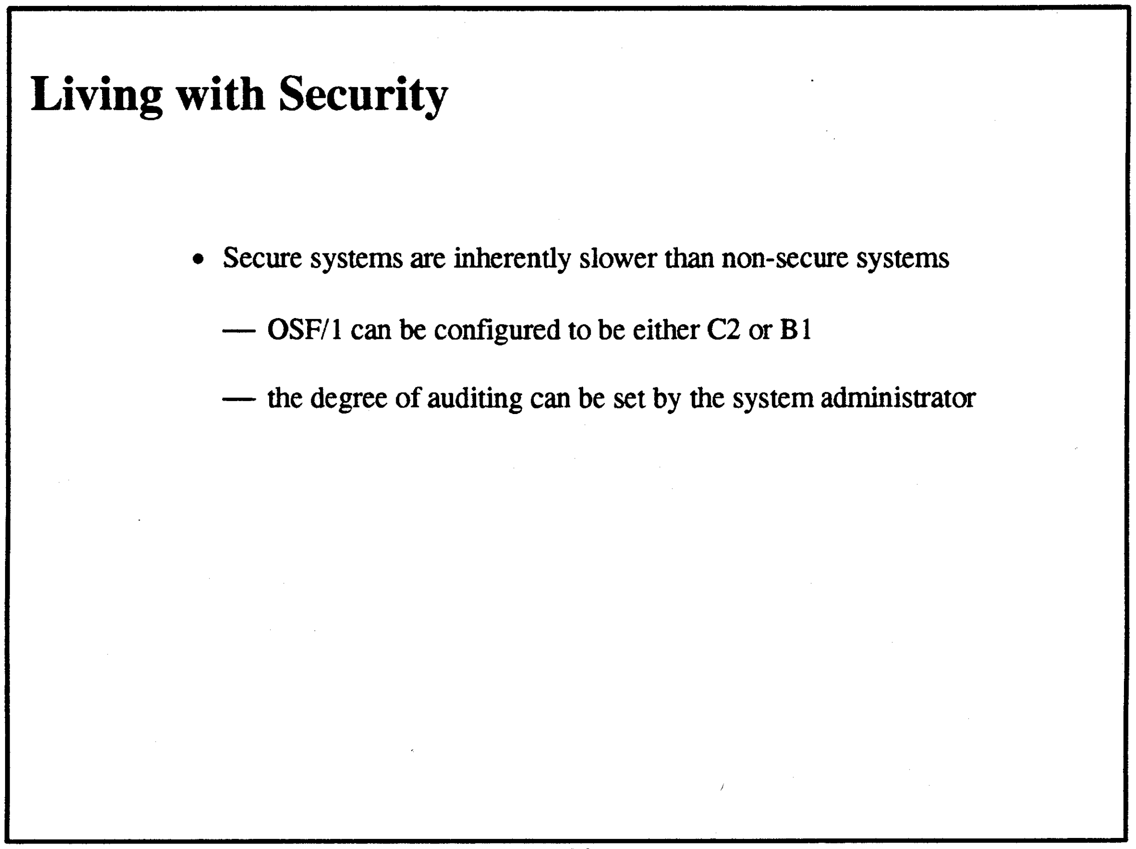
Using the chprv system call, a process may propagate to files only those privileges for which the process is authorized:

$$P' \subseteq K$$

$$G' \subseteq P \cap K$$

If an executable file is modified, then all privileges are removed from its granted and potential sets. This is analogous to the effects of modifying a *setuid* file in standard UNIX.

11–22. Living with Security



Living with Security

- Secure systems are inherently slower than non-secure systems
  - — OSF/1 can be configured to be either C2 or B1
  - — the degree of auditing can be set by the system administrator

H2596 11-22. 352

Living With Security

Not all installations will desire the B1 security features in OSF/1. The system can be configured to be either C2 or B1. This decision is implemented as a compile-time option; it is not a run-time option because too many tests would be necessary. However, the degree of auditing is selectable at run time.

Exercises:

1. a. To which security class are most UNIX systems probably compliant?

   b. To which security classes can OSF/1 be made compliant?

   c. What is the difference between compliant and certified?

2. a. In whose context is audit information collected?

   b. How is audit information collected in an audit file?

3. a. In what ways are ACLs more flexible than standard UNIX file protection?

   b. If DAC and MAC disagree on an access decision, how is the issue resolved?

   c. How is it determined whether a particular subject has the desired access to a particular object with respect to a particular access control policy?

4. a. Explain the difference between how an authorization is implemented and how a privilege is implemented.

   b. The effective privilege set is restricted to be a subset of the potential set unioned with the base privilege set. Why isn't the kernel authorizations set used instead of the base privilege set?

   c. List the kernel data structures that were modified to support the OSF/1 security features.

5. Why isn't security compliance a run-time or boot-time option?

Appendix — Answers to Exercises

Module 1

1. topic 1-1
2. topic 1-5
3. topic 1-7
4. a. topic 1-10
   b. topics 1-6 and 1-7
   c. topic 1-8
   d. topic 1-9
   e. topic 1-12
5. topics 1-18 through 1-22
6. Mach provides the facilities to allow the efficient transfer and sharing of information across address space boundaries. This is particularly useful for efficient communication with servers. In addition, Mach provides support for multiple threads of control within an address space.
7. OSF/1 includes the logical volume manager, support for dynamic configuration, support for shared libraries and dynamic loading, and B1–level security.

Module 2

1. a. topic 2-3
   b. topic 2-3
   c. topics 2-4, 2-5, 2-6
   d. topic 2-4
2. topics 2-7 through 2-11
3. a. topic 2-20
   b. topic 2-19, topic 2-28

   c. topic 2-29

   d. topic 2-24

4. a. topic 2-33

   b. topic 2-34

   c. topic 2-34

   d. topic 2-37

5. a. topic 2-38

   b. topic 2-39

   c. topic 2-41

6. a. topic 2-45

   b. topic 2-45

   c. topic 2-49

   d. topic 2-43

   e. topic 2-43

7. a. topic 2-51

   b. topic 2-51

8. topic 2-52

9. Separate *u_task* and *proc* structures are maintained in OSF/1 primarily because the separateness of these two structures is inherent in the Berkeley UNIX source code. There is no particular reason that they could not be merged, but there is no compelling reason to go to the effort of doing so.

10. OSF/1's kernel threads are cheaper than UNIX's kernel processes because kernel threads have no private address space associated with them. Thus the system can switch from the context of any task into the context of a kernel thread without changing address maps.

11. a) There are three primary reasons for nonpreemptibility in kernel mode. The first, which is not a problem in multiprocessor-safe operating systems such as OSF/1, is that certain data structures, if not accessed in the interrupt context, might have no synchronization to protect them other than the assurance that any thread in kernel mode will not be preempted. A second problem, which does affect OSF/1, is that a thread might be updating a data structure that can be accessed in the interrupt context and thus has a class of interrupts masked off. If this thread is preempted (because the mechanism causing preemption, e.g. clock interrupts, has not been masked off), then one of two things might happen, both of them bad. (1) Interrupts remain masked off when the system switches to the preempting thread; this is not good because the interrupt will be masked off much too long

and may perhaps interfere with interrupt-masking done by the preempting thread. (2) Interrupts become unmasked as the system enters the preempting thread's context; thus the system might now enter the previously masked interrupt context and access the data structure that was in the midst of modification by the preempted thread. This data structure, being in the middle of an update, is in an inconsistent state that is totally unexpected by the interrupt handler. The third reason is also a problem with OSF/1: if the preempted thread is holding a spin lock, the preempting thread might attempt to take this lock. This thread will of course spin, with no hope of taking the lock, until the preempted thread is allowed to execute again.
b) The constraints on preemption points are: they must be executed only in the context of a thread, this thread must not be holding any locks, and no interrupts may be masked.

Module 3

1. a. topic 3-3
   b. topic 3-3
2. a. topic 3-6
   b. topics 3-7, 3-8
   c. topics 3-7, 3-8
   d. topics 3-7, 3-8
   e. topics 3-7, 3-8
   f. topics 3-11 through 3-13
3. a. topics 3-14, 3-15
   b. topics 3-14, 3-15
4. The main problem with utilizing copy-on-write techniques to improve the implementation of UNIX system calls such as write is that the best use of these techniques requires page alignment of data structures such as buffers. Since typical UNIX programs are not written with such alignment requirements in mind, it is unlikely that very many buffers would actually be properly aligned. If such alignment problems could be dealt with, then any UNIX system call that transfers large amounts of data could be improved. In particular, this means I/O-related system calls operating on files, devices, sockets, and streams.

Module 4

1. a. topics 4-2, 4-29
   b. topic 4-2

2. a. topic 4-5
   b. topics 4-6, 4-11
   c. topic 4-29
   d. topic 4-5
3. a. topic 4-23
   b. topic 4-23
   c. topic 4-25
   d. topic 4-18
   e. topics 4-16—4-21
   f. topic 4-18
   g. topic 4-27
   h. topic 4-28
4. a. topics 4-29—4-50
   b. topics 4-30—4-31
   c. topic 4-38
   d. topics 4-43—4-48
   e. topic 4-49
5. a. topic 4-54
   b. topic 4-54
   c. topics 4-54—4-57
   d. topics 4-54—4-59
6. If we want to enjoy the advantages of lazily evaluating the allocation of backing store, then we must deal with the problem that a thread's execution may fail at an arbitrary point in time. The best we can hope for is that the extent of the damage be limited: for example, that it be necessary to terminate only one task. If this sort of behavior is intolerable, we may have to use a more conservative preallocation backing-store policy.
7. The primary difficulty in replacing the vnode pager with an external pager is that this external pager, being the "pager of last resort," must never encounter page faults itself. Thus one technique might be to wire the external pager's pages into primary memory.

Module 5

1. topic 5-3
2. a. topics 5-7, 5-8
   b. topics 5-11, 5-12
   c. topic 5-14
3. a. topic 5-16
   b. topic 5-19
   c. topics 5-25—5-30
4. a. topic 5-37
   b. topic 5-39
5. a. topic 5-43
   b. topics 5-46—5-47
6. a. topic 5-50
   b. topic 5-51
   c. topic 5-56
   d. topic 5-60
7. a. topic 5-62
   b. topic 5-64
   c. topic 5-66
   d. topic 5-68
   e. topic 5-73
   f. topic 5-77
8. All three locks are necessary. The lock on the file table entry is necessary to protect the offset stored there from being used prematurely by another thread. This ensures that I/O system calls executed by threads sharing a file table entry are atomic. The lock on buffers from the buffer cache is necessary to prevent a buffer from being stolen for some other purpose while one thread is using it. A simple lock is required for updates to the vnode because, for example, two threads concurrently reading the same file, but using different file table entries, might update the access time of the vnode.

9. One might argue that, if it is known that a directory is being searched and the result of this search will be used very soon as part of a delete operation, the directory should be searched while holding a write lock. The primary reason that this is not done is that it is highly unlikely that there will be two concurrent updates to the same directory. Thus the optimistic approach described on page 5–75 works out very well in what is by far the more usual case. The fact that concurrent updates, if they occur, can be quite expensive is thus of little consequence.

Module 6

1. a. topic 6-3
   b. topic 6-4
2. a. topic 6-6
   b. topic 6–6
3. topic 6-8
4. a. topic 6-18
   b. topic 6-20
   c. topic 6-23

Module 7

1. a. topics 7-3—7-5
   b. topics 7-8, 7-9
   c. topics 7-10, 7-11
2. a. topics 7-15—7-17
   b. topic 7-15
   c. topics 7-12, 7-13
3. a. topic 7-19
   b. topic 7-20
4. a. topic 7-21
   b. topic 7-24
5. a. topic 7-25

   b. topic 7-29

   c. topics 7-25—7-29

6. The use of synchronization queues has no effect on the order of message processing within a streams module: the synchronization queues preserve the order of calls to the module's procedures, and thus any ordering constraints imposed on messages by the code within a streams module is preserved. However, it is certainly the case that with synchronization queues there might be more messages within a streams pipeline than there would be without such queues. For example, while one thread is executing within a module, other threads might queue more requests on this module's synchronization queue than are allowed by the high-water limit on the module's normal streams queue. In practice this is unlikely to be a problem: the only situation in which it could be a problem is if the rate at which messages are processed by a module is slower than the rate at which tmessages are arriving to the module. Given that streams threads execute nonpreemptively and without blocking, this situation is highly unlikely.

Module 8

1. topic 8-3

2. a. topic 8-5

   b. topic 8-10

   c. topic 8-2

3. a. topic 8-13

   b. topic 8-16

1. pages topic 8-18

Module 9

1. a. topic 9-2

   b. topics 9-6—9-11

2. a. topics 9-9, 9-13

   b. topic 9-9

3. a. topic 9-13

   b. topic 9-11

   c. topic 9-15

   d. topic 9-16

4. Only two data structures in the UFS file system depend upon the size of the entire file system: the superblock and the cg summary. These of course must be modified to reflect the larger file system and the modified superblock must be copied to all of its alternative locations.

Module 10

1. topic 10-2
2. a. topics 10-5—10-8
   b. topic 10-8
3. a. topics 10-7—10-9
   b. topic 10-10
4. a. topic 10-11
   b. topics 10-11, 10-12
5. topic 10-13
   a. topic 10-14

Module 11

1. a. topic 11-3
   b. topic 11-5
   c. topic 11-4
2. a. topic 11-6
   b. topic 11-7
3. a. topic 11-8
   b. topic 11-13
   c. topic 11-13
4. a. topic 11-14
   b. topic 11-18
5. topic 11-22

Bibliography

Allen, L., H. Singh, K. Wallace, M. Weaver. 1991. "Program Loading in OSF/1." *Conference Proceedings of 1991 Winter USENIX Technical Conference.*

AT&T. 1989. *UNIX System V/386 Release 3.2: Streams Programmer's Guide.* Englewood Cliffs, NJ: Prentice Hall.

Bach, Maurice J. 1986. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice Hall.

Black, D., R. Rashid, D. Golub, C. Hill, and R. Baron. 1989. "Translation Lookaside Buffer Consistency: A Software Approach." *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems.*

Black, D. 1991. "Processors, Priority, and Policy: Mach Scheduling for New Environments." *Conference Proceedings of 1991 Winter USENIX Technical Conference.*

Comer, Douglas E. 1991. *Internetworking with TCP/IP, Vol. 1: Principles, Protocols, and Architecture.* Englewood Cliffs, NJ: Prentice Hall.

Golub, D., R. Dean, A. Forin, R. Rashid. 1990. "UNIX as an Application Program." *Conference Proceedings of 1990 Summer USENIX Technical Conference.*

Juszczak, Chet. 1989. "Improving the Performance and Correctness of an NFS Server." *Conference Proceedings of 1989 Winter USENIX Technical Conference.*

Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, MA: Addison-Wesley Publishing Company.

Macklem, R. 1991. "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol." *Conference Proceedings of 1991 Winter USENIX Technical Conference.*

Open Software Foundation. 1989. *Application Environment Specification (AES): Operating System Programming Interfaces Volume.* Englewood Cliffs, NJ: Prentice Hall.

Open Software Foundation. 1990a. *The Design of the OSF/1 Operating System.* Cambridge, MA: Open Software Foundation.

Open Software Foundation. 1990b. *System Extension Guide.* Cambridge, MA: Open Software Foundation.

Tevanian, A., R. Rashid, D. Golab, D. Black, E. Cooper, and M. Young. 1987. "Mach Threads and the UNIX Kernel: The Battle for Control." *Conference Proceedings of 1987 Summer USENIX Technical Conference.*

Glossary

address space	a set of virtual locations, such as those locations that can be referenced by a task or process.
authorizations	indications of whether a particular user is allowed to use a particular command or subsystem, perform a particular role, or gain a particular privilege.
blocking lock	a lock which a thread waits for by yielding the processor.
bogus memory	type of memory in a parallel architecture that does not guarantee atomicity of reads of aligned words.
buffer cache	the collection of buffers maintained in the kernel for use in accessing files and block special devices.
channel	address of a relevant data structure that specifies an awaited event.
concurrency	multiple threads are in progress at one time; their execution might be multiplexed on a single processor.
cooked mode	a terminal mode in which input lines can be edited, and certain characters cause signals to be sent to the process group.
copy-on-write	optimization using lazy evaluation in which copying is postponed until a task actually modifies a page.
devices	hardware.
disposition	indicates whether or not the sleep is interruptible by a signal.
exceptions	deviations to a thread's flow of control that are caused by actions of the thread itself (such as addressing errors, arithmetic errors, etc.).
external	outside the kernel.
file handle	data that is used to identify a file. After a client opens a file, the server gives it a file handle, which the client gives to the server to speed subsequent accesses.
funnel	a kernel data structure used to represent the parallel/sequential constraints of a particular subsystem.
handoff scheduling	a form of thread scheduling in which one thread gives its processor to another.

idempotent	when the effect of performing an operation once is the same as performing it multiple times, the operation is idempotent.
inode	the (ondisk and incore) data structure that describes a file (both S5 and UFS).
internal	inside the kernel.
lazy evaluation	technique of postponing everything until the last possible moment, since if you put it off long enough, maybe you won't have to do it.
local port	port that the message comes back through.
logical volume	an abstraction that behaves like a disk drive to file system code, but is in fact a collection of separate regions of real disk drives (physical volumes).
lookup cache	cache of the most recent component-name-to-vnode translations.
memory object	a "thing" that can be mapped into a task's address space. It might be temporary storage (e.g., UNIX's BSS and stack), a file, or an object defined by user-provided servers.
memory object manager	responsible for supplying initial values for a range of virtual memory and for backing up virtual memory when the physical memory cache becomes full. One may be used, for example, to map files into the address spaces of tasks, to provide shared memory in a distributed system, or to implement a transaction-management system.
message	a collection of data to be sent through a port to the task that has receive rights for the port.
microkernel	a simple, pure Mach kernel with no built-in UNIX (or other operating system) functionality. Such functionality would be provided by user tasks.
mmap	a system call that is used either to map a file into a process's address space or to create an anonymous memory region.
multithreaded	composed of a number of threads.
NICFREE	number of incore free blocks. Equal to 100.
package	an abstraction of a library.
parallelism	the simultaneous execution of multiple threads; requires multiple processors.
parallelization	the act of making a system parallelized.
physical volume	a real disk drive or a portion of a disk drive.

pmap	the data structure and code encapsulating the architecture-dependent portion of the virtual memory system.
port	a protected queue of messages or an object reference.
port set	two or more ports whose message queues have been consolidated into a single queue by the server task.
priority depression	option to the thread_switch system call; a calling thread's priority is "depressed" to the worst possible value for a given period of time, and is then restored.
privileges	properties of a process that gain it special treatment by the operating system.
process	an address space, one or more threads of control and additional information necessary to represent a UNIX context.
processor allocation	distributing the processors of a multiprocessor among the various applications.
processor set	mechanism for processor allocation.
processor sharing	scheduling or multiplexing processors.
raw mode	the terminal mode in which incoming characters are passed immediately to user threads and outgoing characters are sent to the terminal with no further processing.
read-ahead	reading the next unit of data at the same time as the current unit of data.
read-write lock	a lock that can be taken as either a read lock, allowing multiple readers by no writers, or as a write lock, allowing a single writer and no readers.
remote port	port for sending messages.
search cache	a cache in the inode that contains the offset at which the last search terminated.
sharing	what one is taught in nursery school.
simple lock	a spin lock.
socket	a data structure representing the end point of a communication
spin lock	a lock which a thread waits for by repeatedly testing a bit.
stream	the kernel analog of a shell pipeline.
submap	a data structure representing a portion of the kernel address space which is probably managed by a single subsystem.

swapping	unwiring or wiring the kernel stack.
task	a holder of capabilities, such as address space and communication channels.
thread	usual notion of thread of control.
thread pool	a collection of threads used to handle events generated in the interrupt context.
timed pause	when a thread calls thread_switch with the wait option, it can be suspended for a fixed period of time and then automatically woken up.
translation-lookaside buffer	a hardware cache which translates virtual addresses to real addresses.
upcall	a call from a lower level of a system to a higher level (e.g. from kernel mode to user mode).
virtual copy	an optimized copy operation.
virtual file system	the abstraction of the file system concept: the layer of the kernel which provides the standard interface to the real file systems.
vnode	an abstraction of a file; it contains generic information about files and refers to the file-system-specific information on individual files. It also refers to an array of entry points called vnodeops, which provides access to the various operations.
write-behind	delaying the update of a file until sometime after the write system call has been completed.
write-through cache	a buffer cache that requires that the data it buffers be written onto the disk before the system call returns.
zone	a collection of fixed-size blocks: a separate zone is created for each kernel data structure that is so managed. A zone is initialized with a pre-allocated free list, an allocation size, and a maximum size.

** For HP Internal Reference Only **

Manufacturing Part Number

H2596-90001

Customer Order Number

NONE

Printed in USA