

INSTITUTE FOR ADVANCED PROFESSIONAL STUDIES

Technology Consultation and Training Worldwide

955 MASSACHUSETTS AVENUE

CAMBRIDGE, MASSACHUSETTS 02139-3107

(617) 497-2075 • FAX: (617) 497-4829 • email@iaps.com

OSF/1 Internals

Volume I

For the Technical Staff of

Digital Equipment Corporation

Colorado Springs

Release 1.0

Amsterdam • Boston • Dallas • London • Los Angeles • Paris • San Francisco • Tokyo • Washington, DC

Copyright Notice

The material in this binder is either Copyright 1992 by the Institute for Advanced Professional Studies or Open Software Foundation, or reproduced for use in this course by IAPS with permission from the copyright holder.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying or otherwise, without the prior written permission of the Institute for Advanced Professional Studies.

Additional copies of these materials are available strictly through the Institute for Advanced Professional Studies, 955 Massachusetts Avenue, Cambridge, MA 02139.

The ideas and designs set forth in the course materials are the property of the Institute for Advanced Professional Studies. These materials are not to be distributed to third persons without the express written permission of IAPS.

Lunch @ 12
Finish @ 3

Contents

Overview

Course Description	0-1
Prerequisites	0-1
Audience	0-1
Course Goals	0-2
Exercises	0-2
Agenda	0-2
Recommended Readings	0-2

Slide Conventions	0-4
-------------------------	-----

Module 1 — Introduction

Objectives	1-1
What is OSF/1?	1-2
Organization of the OSF/1 Kernel	1-8
Threads and Parallelism	1-12
Introduction to Mach	1-20
The Extensible Kernel	1-36
Exercises	1-46

Module 2 — The Process Abstraction

Objectives	2-1
Processes	2-6
System Calls in OSF/1	2-16
Synchronization and Thread Management	2-26
Signals and Exception Handling	2-64
Threads	2-78
Scheduling	2-86
Thread Pools	2-104
Zoned Memory Allocation	2-106
Exercises	2-108

Module 3 — Messages and Ports

Objectives	3-1
------------------	-----

Messages	3-4
Ports	3-8
Flow of Control	3-28
Exercises	3-32

Module 4 — Virtual Memory

Objectives	4-1
Lazy Evaluation	4-4
VM Components	4-6
Memory Objects	4-30
Copying and Sharing	4-58
The Pmap Module	4-102
Exercises	4-120

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 principles of the following:

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

Familiarity with the operational principles 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 pages 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 through Module 8

Day 4: Module 9 through Module 11

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 bibliography). Two recommended books on UNIX are Bach, 1986

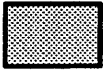
Overview

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.

Slide Conventions



port



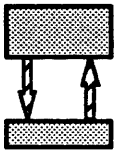
task



task has send rights



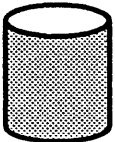
task has receive rights



a task with an unspecified number of additional *vm_map_entries*.



thread



disk



buffer



pmap



encloses an indirect reference to a routine

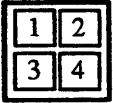
helvetica font

system call

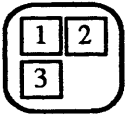
Slide Conventions



indicates the flow of control



boxes with square corners contain all of the incore pages



boxes with rounded corners contain all of the pages associated with the *vm_object*



bug

Slide Conventions

Module 1 — Introduction

Module Contents

1. What is OSF/1?	1-2
The components of the OSF/1 package	
What the course covers	
Where the technology originated	
Why this technology was chosen	
2. Organization of the OSF/1 Kernel	1-8
3. Threads and Parallelism	1-12
Concurrency vs. parallelism	
Types of hardware for OSF/1	
4. Introduction to Mach	1-20
Fundamental abstractions	
Basic system calls	
5. The Extensible Kernel	1-36
Network shared memory	

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

1-1. What is OSF/1?

What is OSF/1? — ~~no~~ AT&T license required

4.3 Reno BSD

- Parallelized 4.4BSD UNIX

- Mach kernel ^{Mach 2.5} written for Parallel

- p* • Logical volume manager — developed @ IBM - Logical Volumes

- p* • Streams early UNIX S5V3

- Extensible loader

- Dynamic configurability

- B1-compliant } speed issue
or C2

BFFS
S5
new NFS - not used w/DEC

Module 1 — Background and Introduction

Student Notes: 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).

Module 1 — Background and Introduction

1-2. What is OSF/1?

UNIX

- Compliance
 - POSIX 1003.1 → *system calls*
 - SVID Issue 2 (goal)
 - XPG Issue 3 *Xopen*
- 4.4BSD framework
 - processes
 - file systems
 - terminals
 - sockets

Module 1 — Background and Introduction

Student Notes: 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.

Module 1 — Background and Introduction

1-3. What is OSF/1?

Mach *~ on the cumbered*

UNIX support

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

Extensibility

- microkernel architecture

1-3.

© 1990, 1991 Open Software Foundation

OSF/1 MK

Module 1 — Background and Introduction

Student Notes: 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.

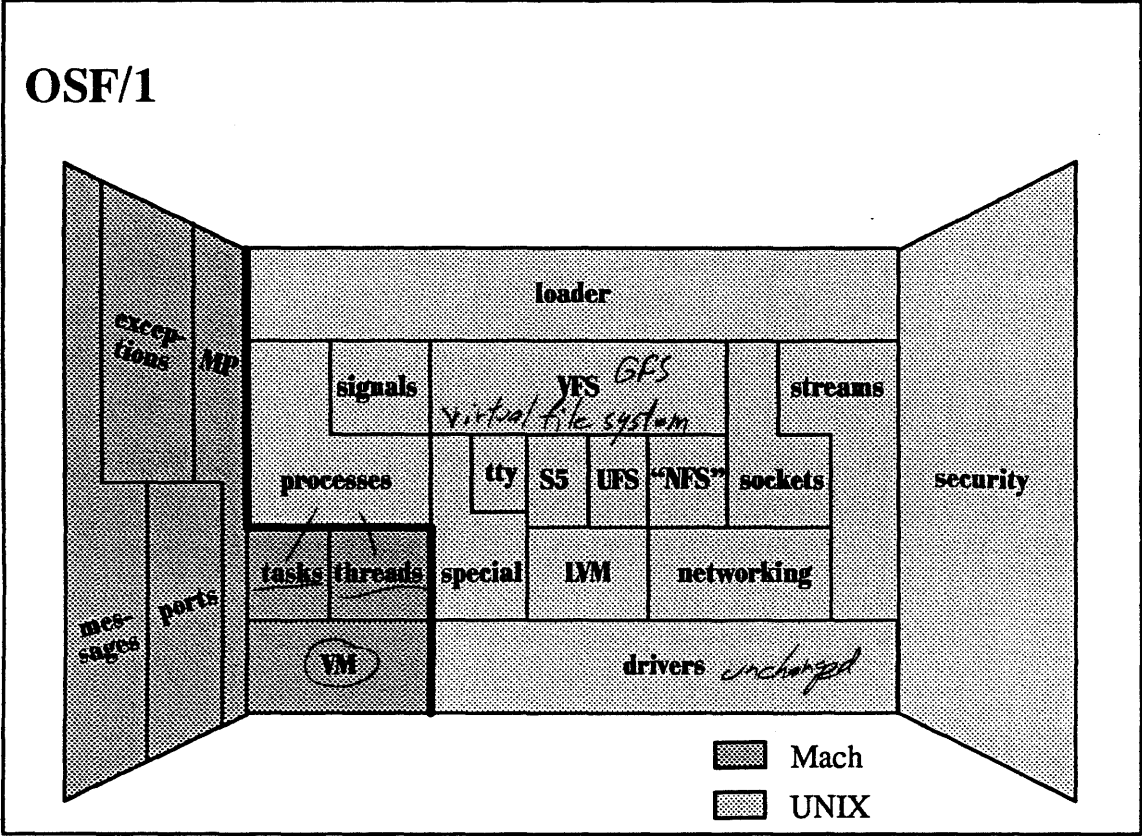
A key point to remember is that Mach fosters continued improvements. In particular, as part of the *microkernel project*, all non-Mach portions of the system will be moved from the kernel to various user-level tasks, to produce a very simple, pure Mach kernel in which user tasks provide many of the operating-system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

Task - addr space
Threads of control in addr space

Module 1 — Background and Introduction

1-4. Organization of the OSF/1 Kernel



Module 1 — Background and Introduction

Student Notes: OSF/1

Module 1 — Background and Introduction

1-5. Organization of the OSF/1 Kernel

UNIX with Mach

- UNIX in the kernel *OSF/1 IK*
- UNIX in a single task *OSF/1 MK* *> user's choice*
- great for debugging
- UNIX as a set of tasks

Module 1 — Background and Introduction

Student Notes: 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.

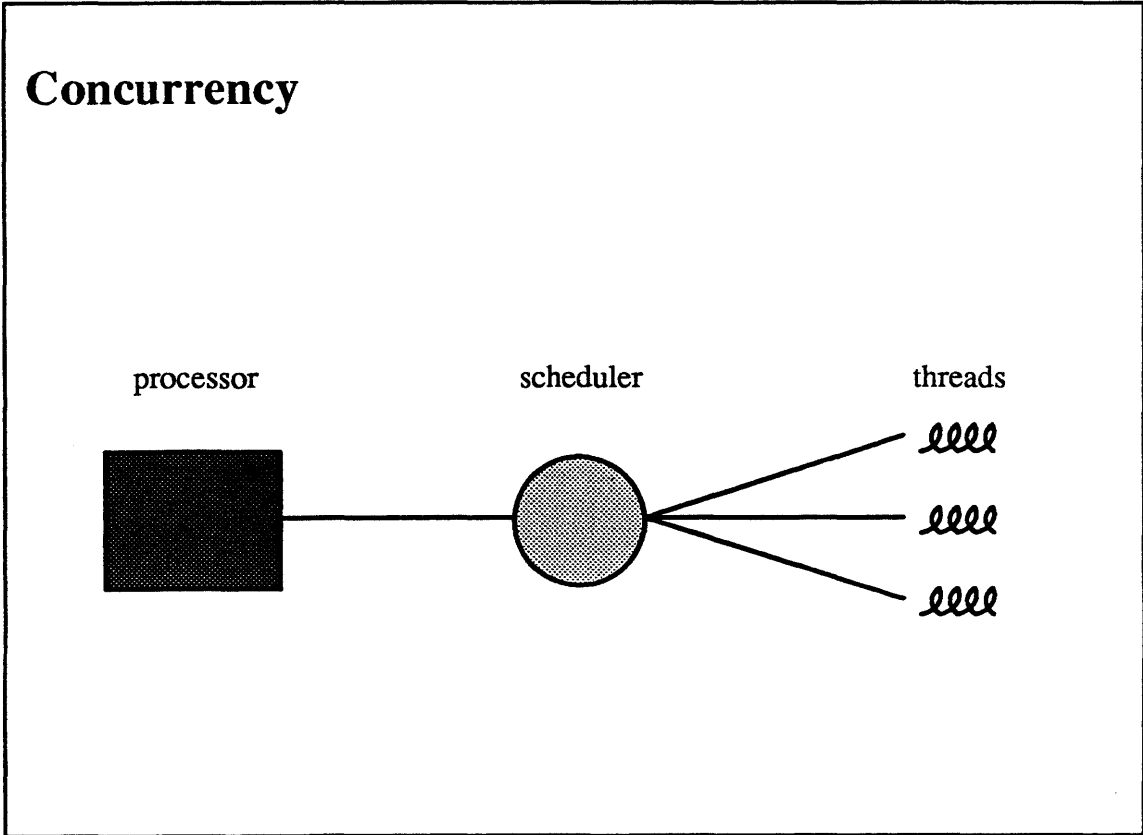
Current research at CMU, OSF, and elsewhere is directed towards providing UNIX functionality efficiently with user-level tasks supported by a “pure” Mach kernel. The first approach provided the UNIX functionality in a single task. This arrangement provided a pageable, interruptible, multithreaded UNIX, 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.

Module 1 — Background and Introduction

1-6. Threads and Parallelism



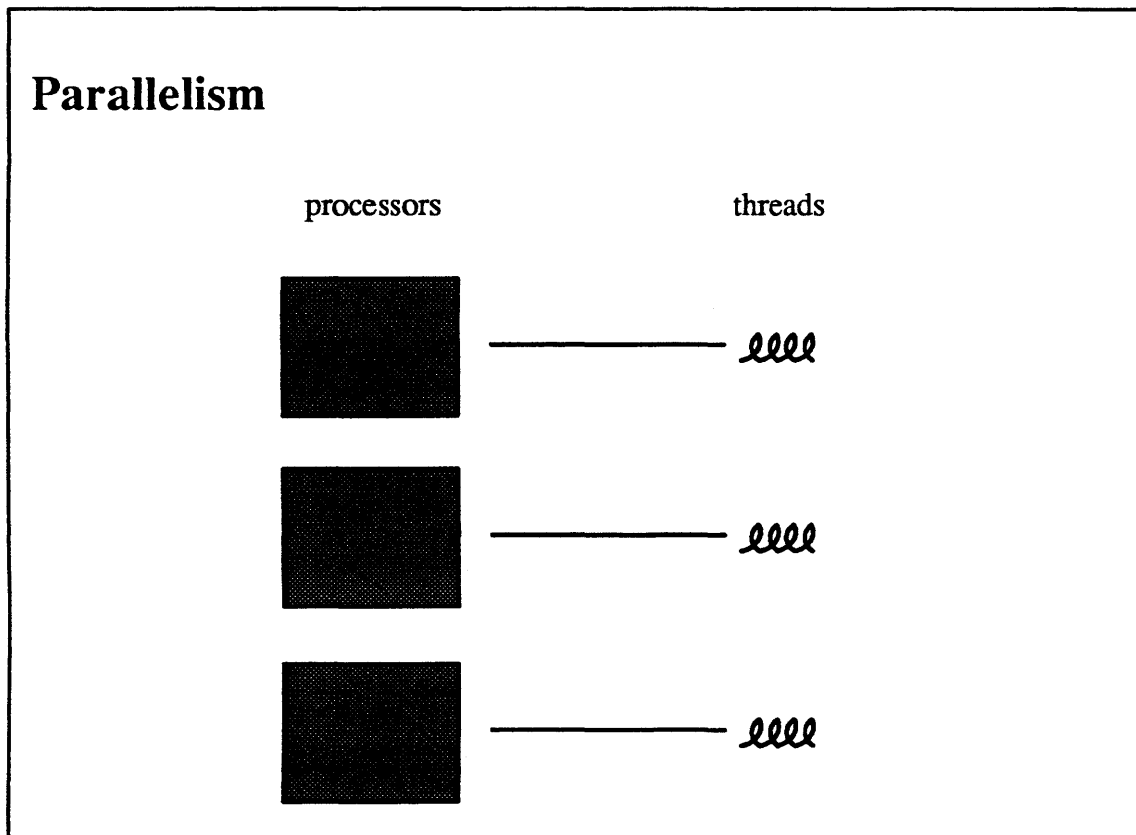
Module 1 — Background and Introduction

Student Notes: Concurrency

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

Module 1 — Background and Introduction

1-7. Threads and Parallelism



1-7.

© 1990, 1991 Open Software Foundation

Module 1 — Background and Introduction

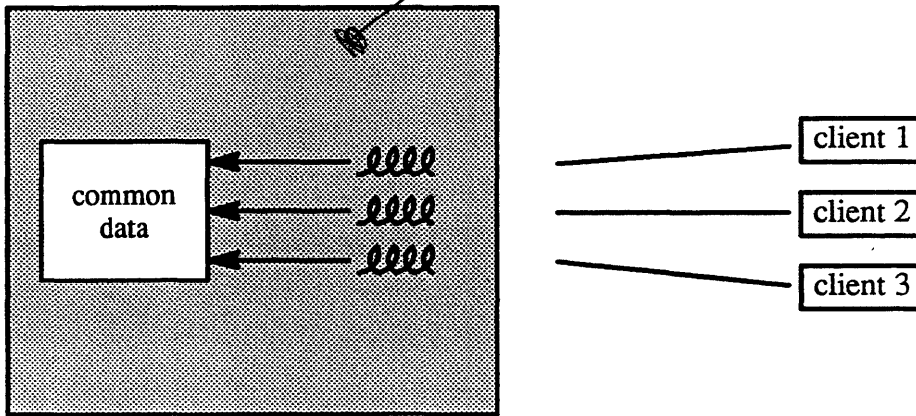
Student Notes: Parallelism

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.

Module 1 — Background and Introduction

1-8. Threads and Parallelism

Multithreaded Process: Server



1-8.

© 1990, 1991 Open Software Foundation

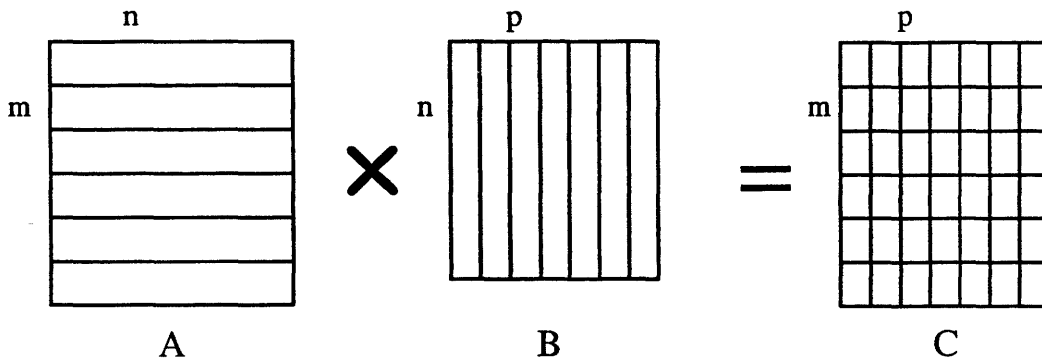
Module 1 — Background and Introduction

Student Notes: 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

1-9.

© 1990, 1991 Open Software Foundation

$$\frac{m \times p}{t}$$

Module 1 — Background and Introduction

Student Notes: 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.

Module 1 — Background and Introduction

1-10. Introduction to Mach

Mach

Fundamental abstractions

- tasks
- threads
- messages
- ports — *entire comm path, like pipe*
- memory objects

Module 1 — Background and Introduction

Student Notes: 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.

file
stack

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)
```

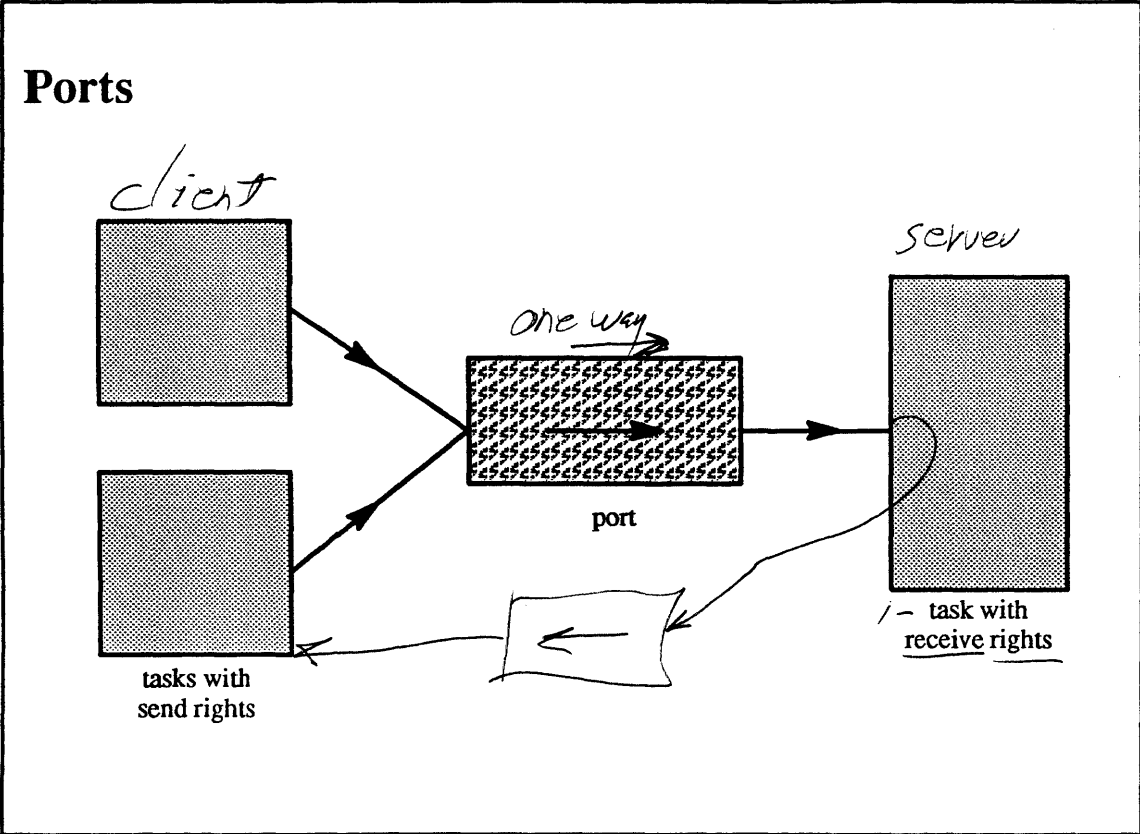

Module 1 — Background and Introduction

Student Notes: 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.

Module 1 — Background and Introduction

1-12. Introduction to Mach



Module 1 — Background and Introduction

Student Notes: 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.

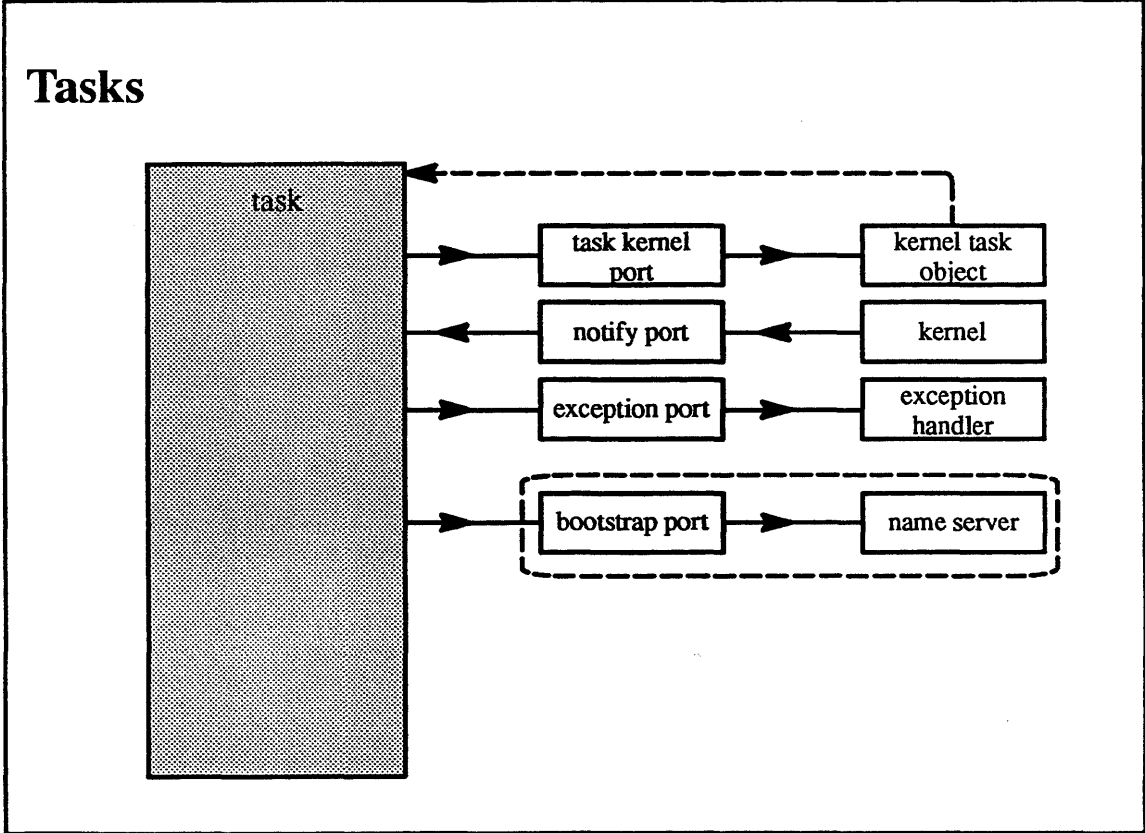
Ports in OSF/1 are most commonly used as object references: send rights on a port represent the name of the associated object.

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

Module 1 — Background and Introduction

1-13. Introduction to Mach



Module 1 — Background and Introduction

Student Notes: 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.

Each task has four ports associated with it:

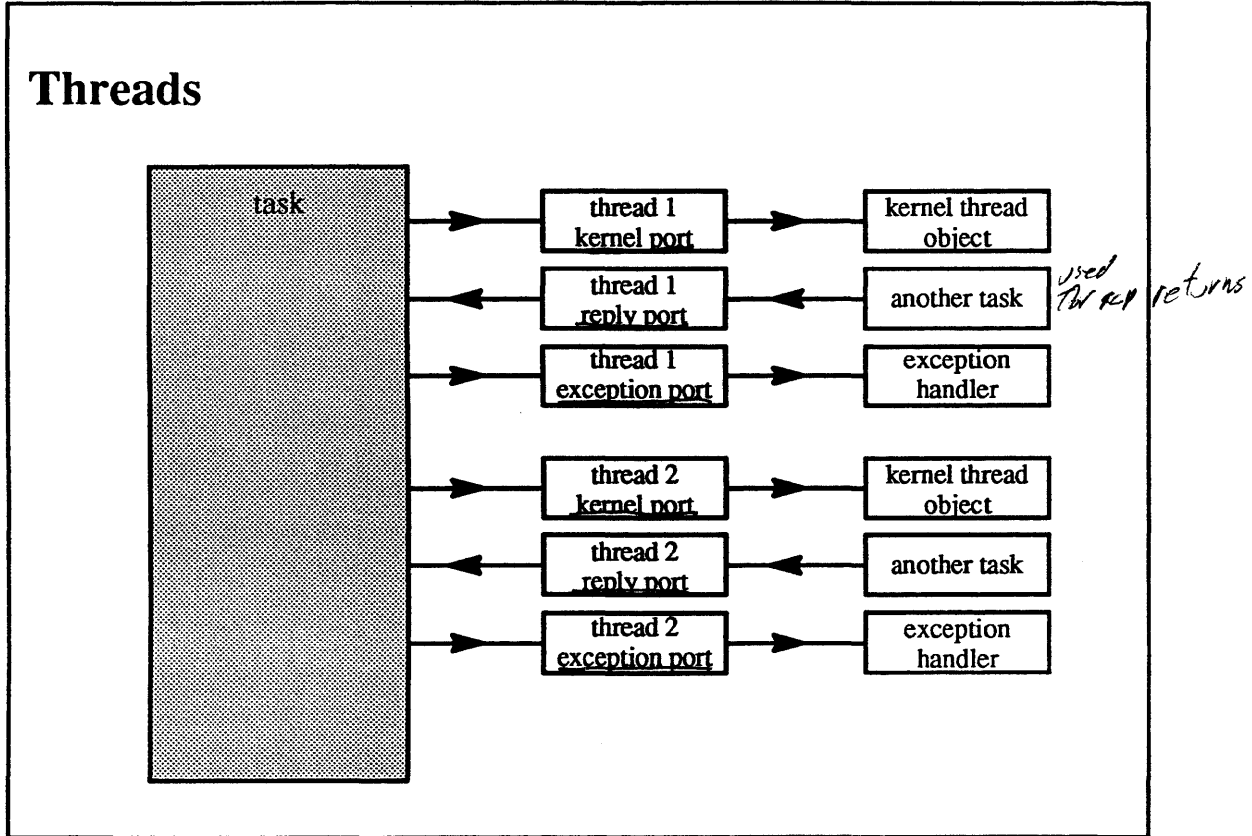
1. Task kernel port: *Object reference kernel is a task* essentially the name of the task. In order to perform a system call that affects a task, the calling thread must have send rights to the task kernel port of this task. Thus threads in other tasks may issue system calls on a task's behalf if their 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.
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. *MK only*

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.

Module 1 — Background and Introduction

1-14. Introduction to Mach



1-14.

© 1990, 1991 Open Software Foundation

Module 1 — Background and Introduction

Student Notes: 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.

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)`

1-15. Introduction to Mach

Virtual Memory in Mach

System calls:

`vm_allocate(target_task, address, size, anywhere)` *-fast, nothing done*
`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)`

Module 1 — Background and Introduction

Student Notes: Virtual Memory in Mach

Virtual memory is a property of the task.

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

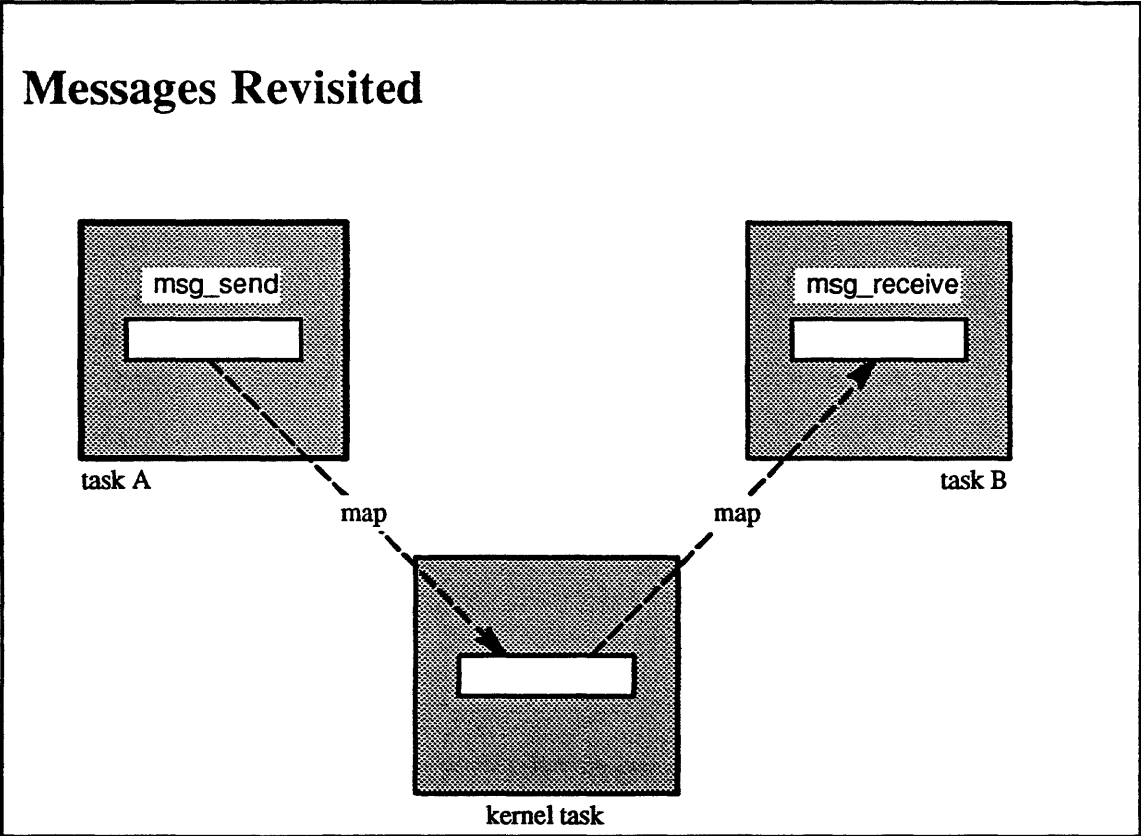
Tasks may inherit virtual memory from their parents, either shared or copied.

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.

Module 1 — Background and Introduction

1-16. Introduction to Mach



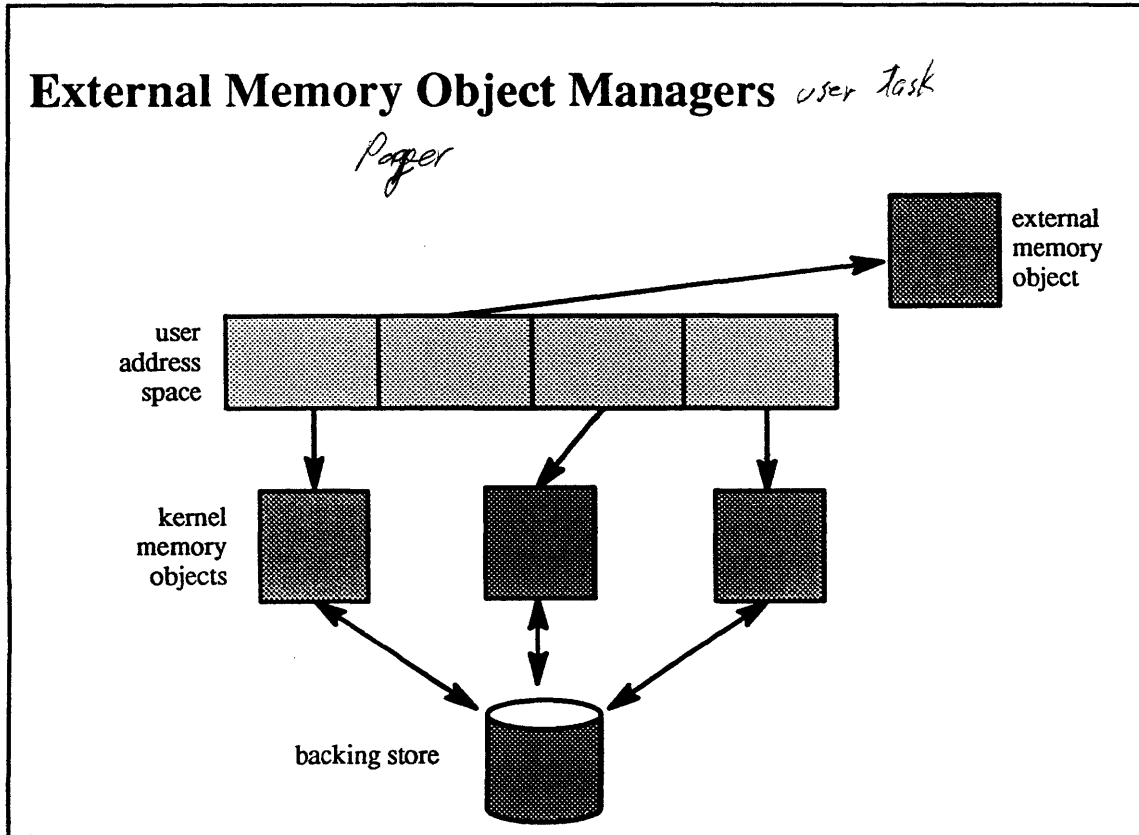
Module 1 — Background and Introduction

Student Notes: 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`.

Module 1 — Background and Introduction

1-17. Introduction to Mach



1-17.

© 1990, 1991 Open Software Foundation

Module 1 — Background and Introduction

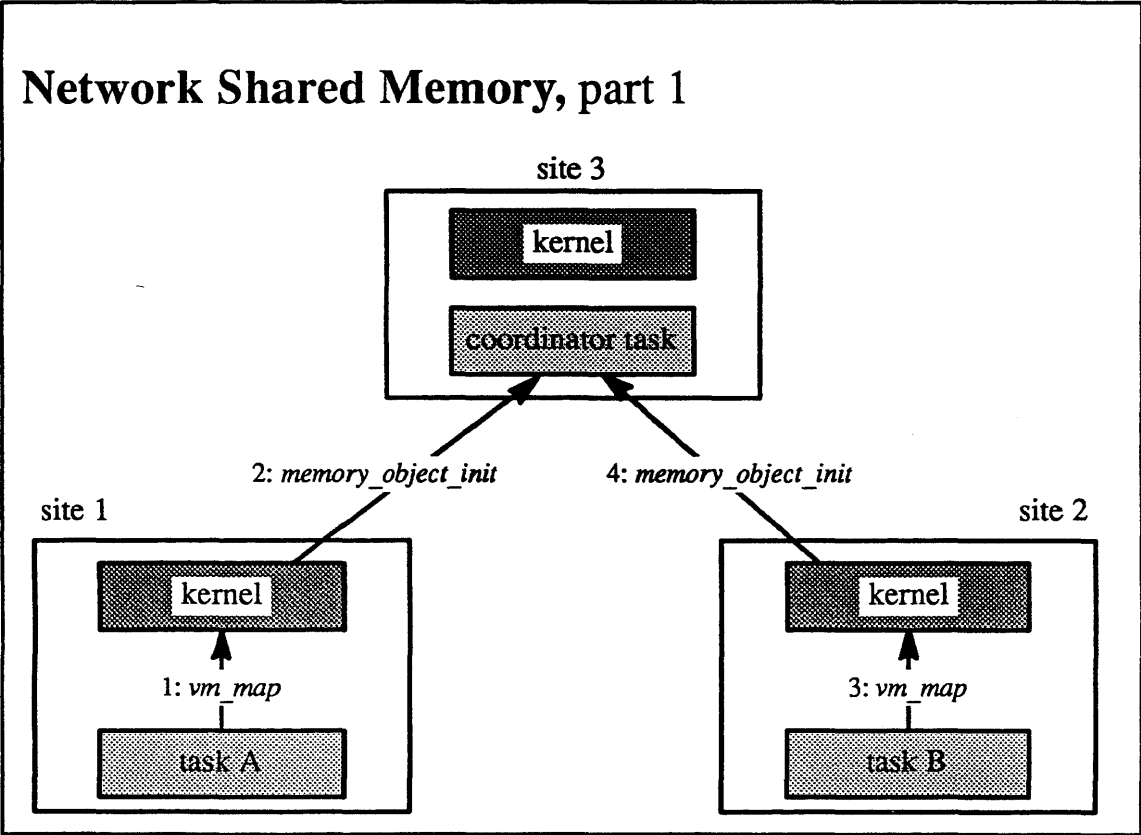
Student Notes: External Memory Object Managers

- 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.

Module 1 — Background and Introduction

1-18. The Extensible Kernel



Module 1 — Background and Introduction

Student Notes: Network Shared Memory, part 1

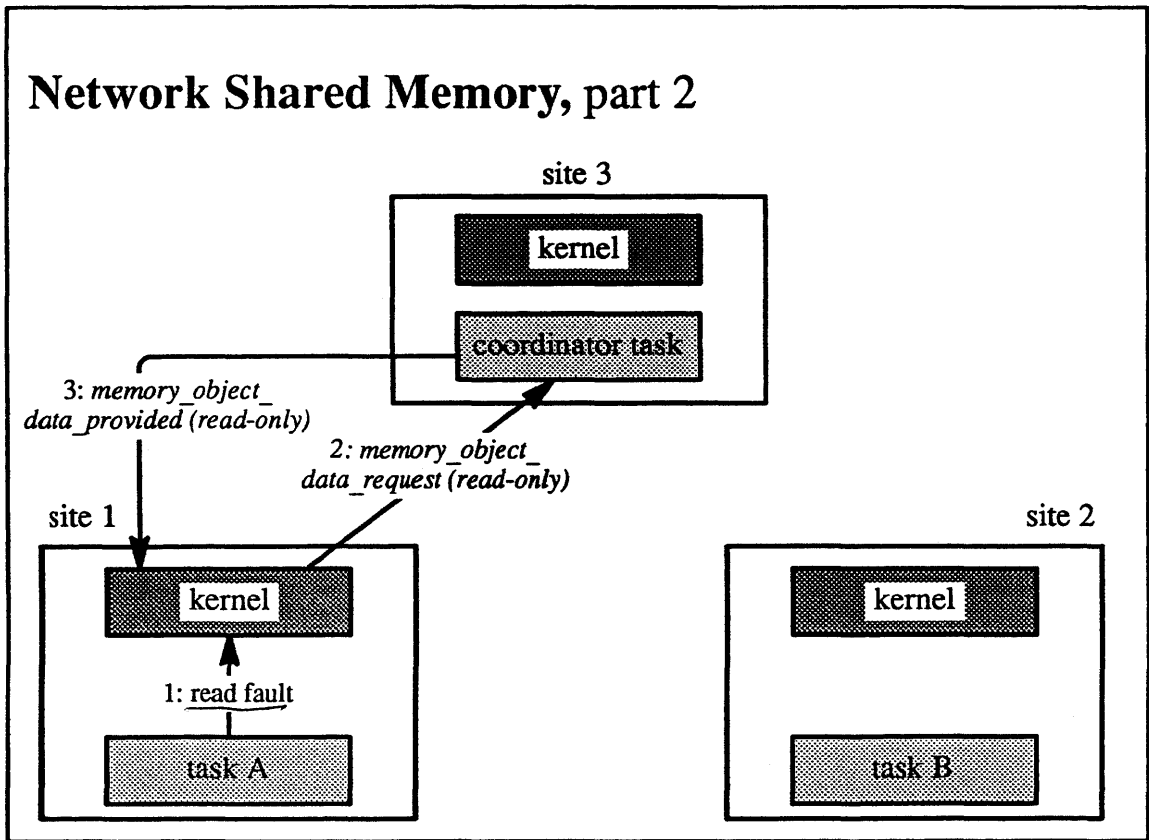
This example shows how the Mach facilities of OSF/1 might provide the abstraction of shared memory to threads running on different machines. Here, site 1 and site 2 are two different machines; the coordinator, the provider of the “shared memory,” might be on a third machine.

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.

Module 1 — Background and Introduction

1-19. The Extensible Kernel



1-19.

© 1990, 1991 Open Software Foundation

Module 1 — Background and Introduction

Student Notes: Network Shared Memory, part 2

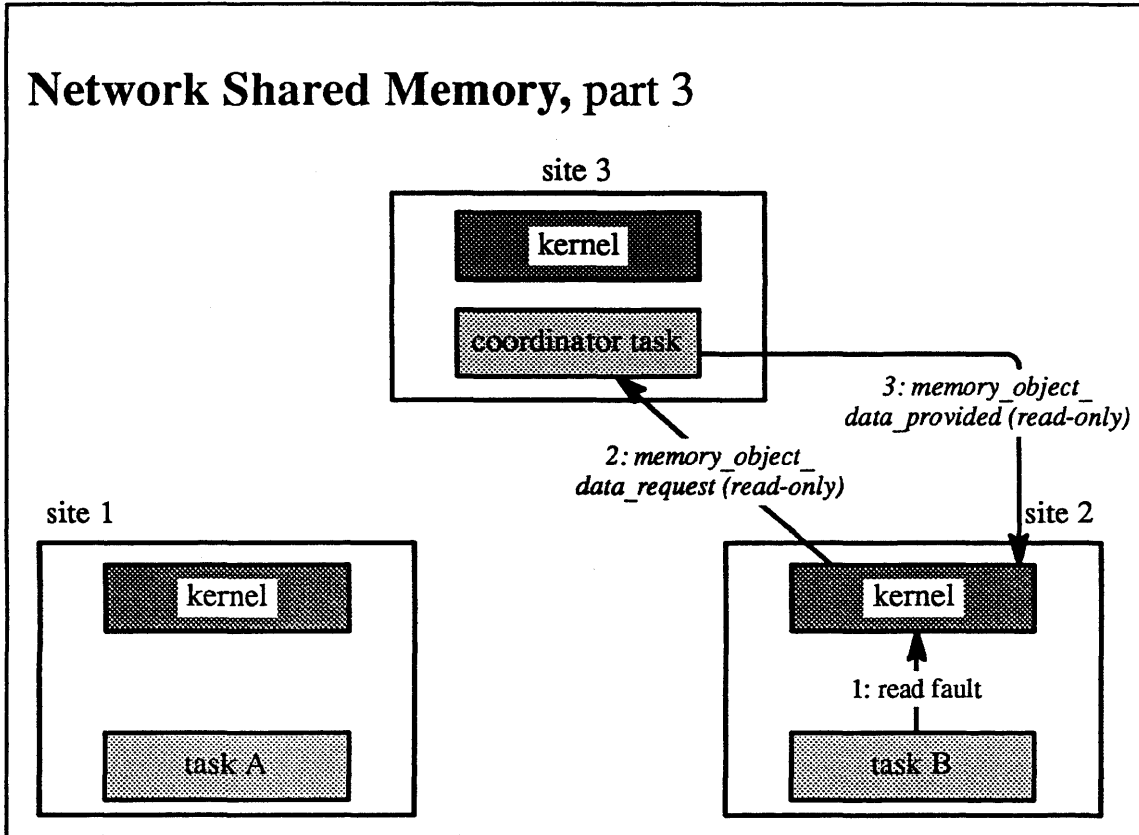
A thread running on site 1 attempts to read from one of the pages in the object maintained by the coordinator. Since the page is not resident at its site, a page fault occurs. The local kernel handles the fault and forwards it to the coordinator (by sending the coordinator an *memory_object_data_request*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

Module 1 — Background and Introduction

1-20. The Extensible Kernel



1-20.

© 1990, 1991 Open Software Foundation

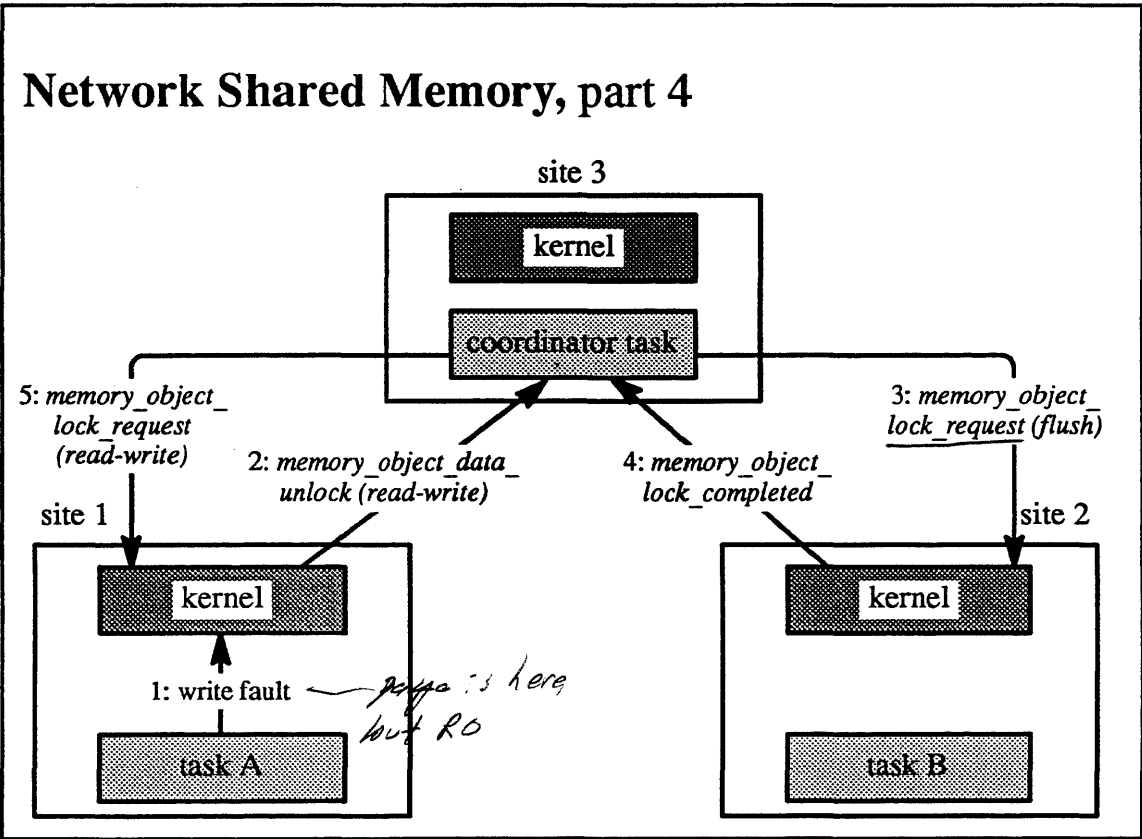
Module 1 — Background and Introduction

Student Notes: 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.

Module 1 — Background and Introduction

1-21. The Extensible Kernel



1-21.

© 1990, 1991 Open Software Foundation

Module 1 — Background and Introduction

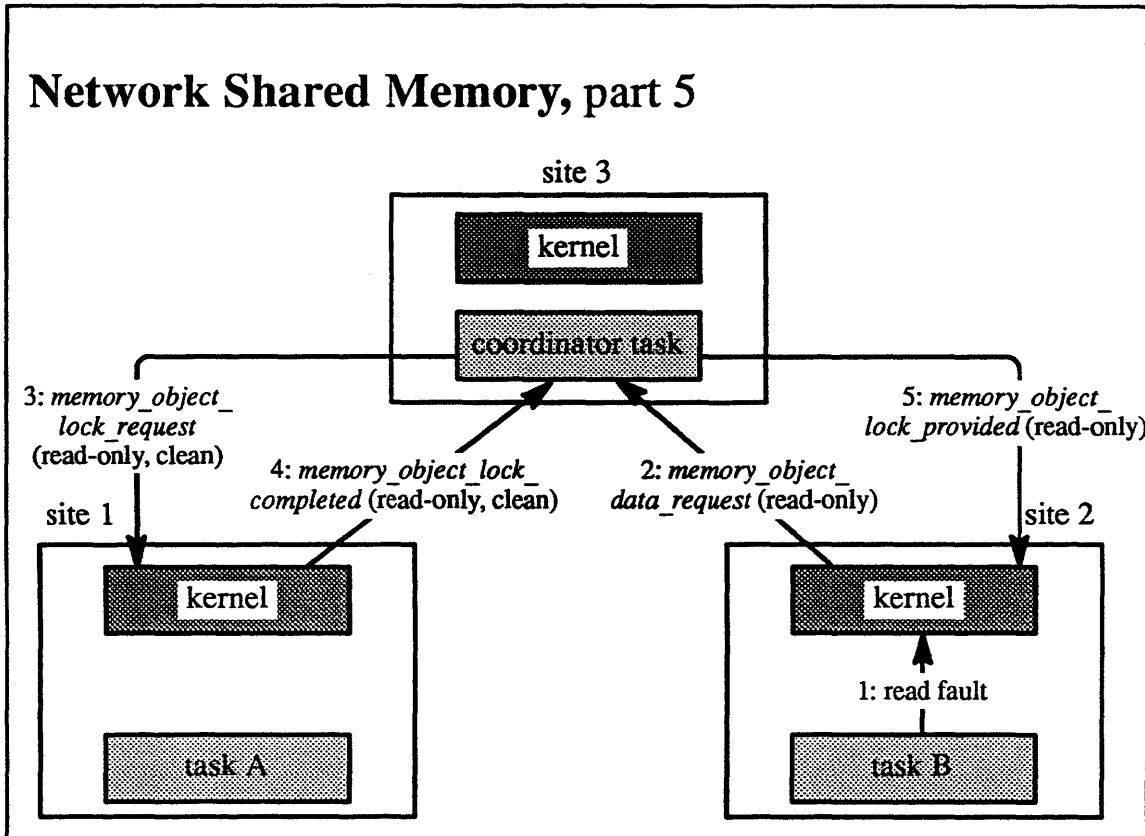
Student Notes: Network Shared Memory, part 4

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.

Module 1 — Background and Introduction

1-22. The Extensible Kernel



1-22.

© 1990, 1991 Open Software Foundation

Module 1 — Background and Introduction

Student Notes: 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

1. Processes	2-6
Extending traditional processes to multithreaded processes	
Representing processes in OSF/1	
2. System Calls in OSF/1	2-16
UNIX system calls	
Mach system calls	
3. Synchronization and Thread Management	2-26
Synchronization in standard UNIX	
The problems introduced by multiprocessors	
Synchronization primitives in OSF/1	
Managing threads through state transitions	
4. Signals and Exception Handling	2-64
Integrating signals into multithreaded processes	
The Mach exception mechanism	
5. Threads	2-78
The system call interface	
Utilizing threads with the POSIX-threads library	
6. Scheduling	2-86
Scheduling policies	
Multiplexing of threads	
Processor sets	
7. Thread Pools	2-104
8. Zoned Memory Allocation	2-106

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

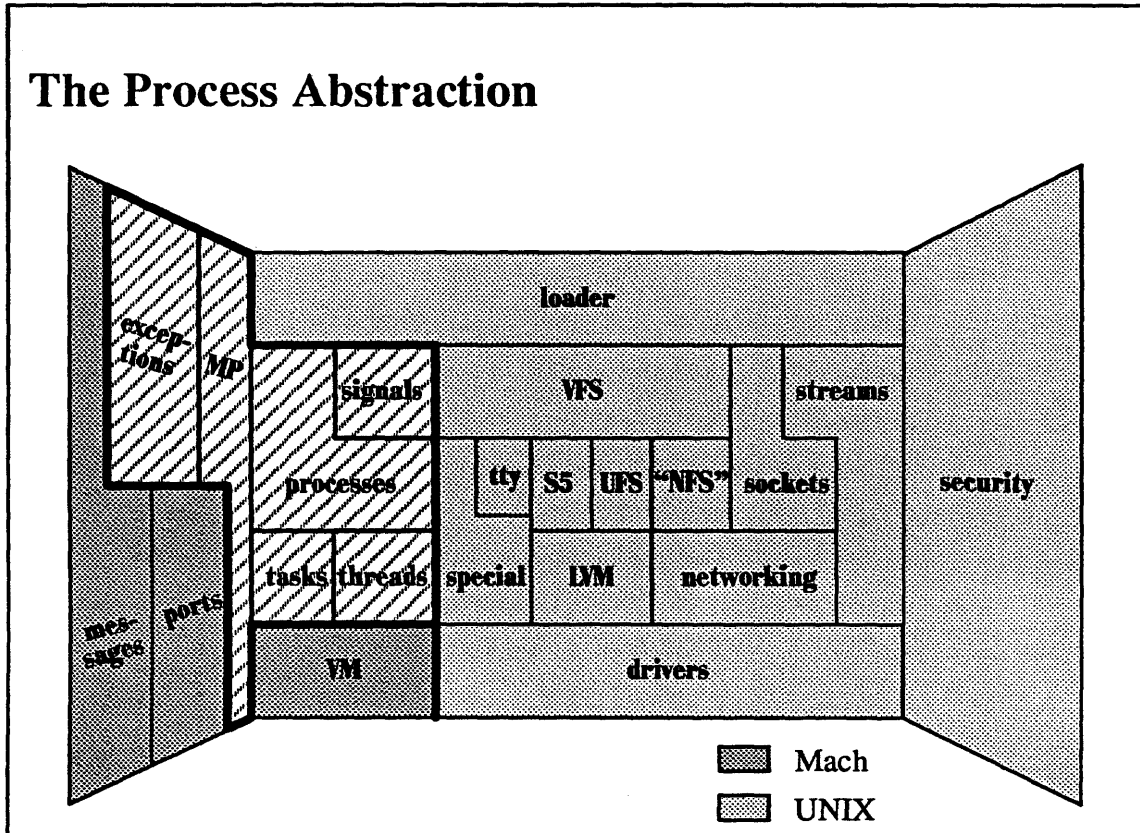
Module 2 — The Process Abstraction

- 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

Module 2 — The Process Abstraction

2-1. The Big Picture



2-1.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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

pid

- 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

Module 2 — The Process Abstraction

Student Notes: 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 *groupids*, 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.

Module 2 — The Process Abstraction

2-3. Processes

Multithreading the UNIX Process

System calls:

- fork
- exec
- error codes

Signals: Synchr/Asynchr

- who gets them? - which thread, the oldest thread or 1st thread
- sigpause/sigsuspend

Standard libraries: - use special lib

- return pointers to static data (e.g. gethostbyname)
- access shared data structures (e.g. stdio)

Handwritten notes:
One thread - fork
destroy all other threads - exec
} depends on 1 thread
- look at -errno, use C rt lib which understands threads - POSIX 1003

for each UNIX Process there is a MACH task

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

Student Notes: 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.

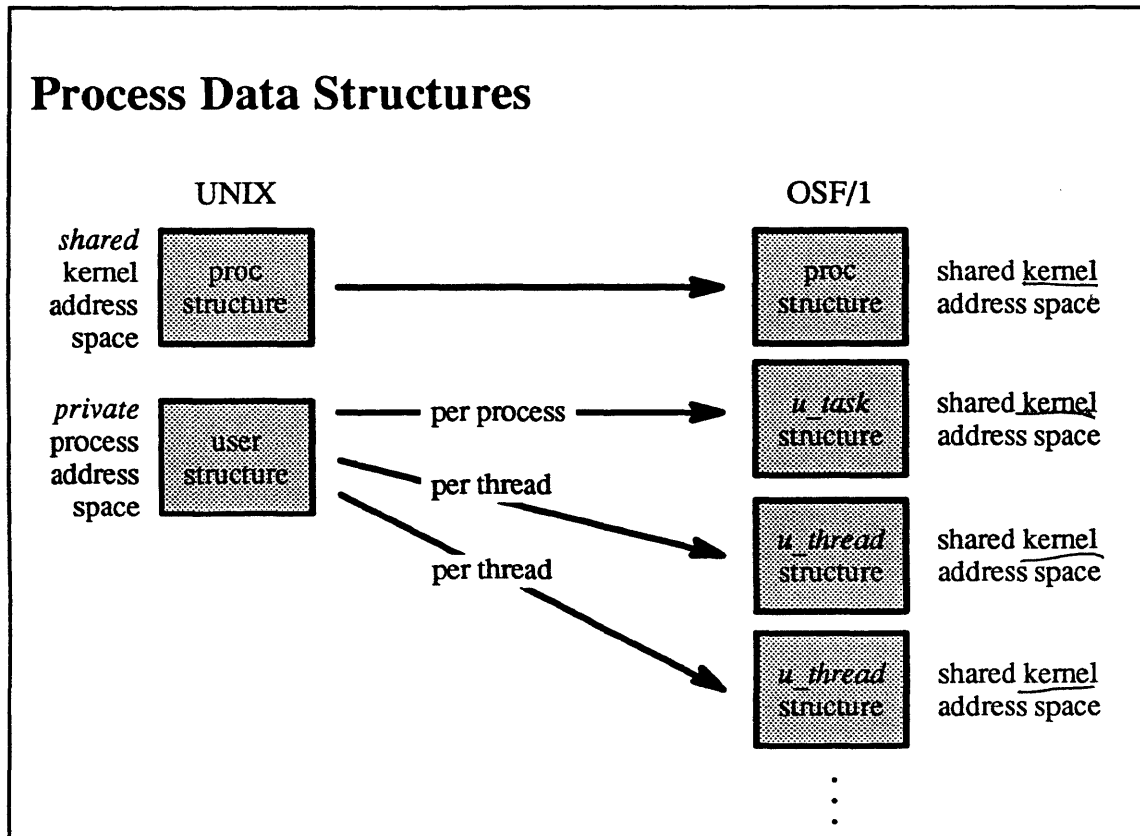
Both types of components are allocated from the system address space, not the process address space (the original UNIX scheme of allocating user structure at a fixed location in the process address space is not possible because of the multiple *u_thread* components).

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.

Module 2 — The Process Abstraction

2-5. Processes



2-5.

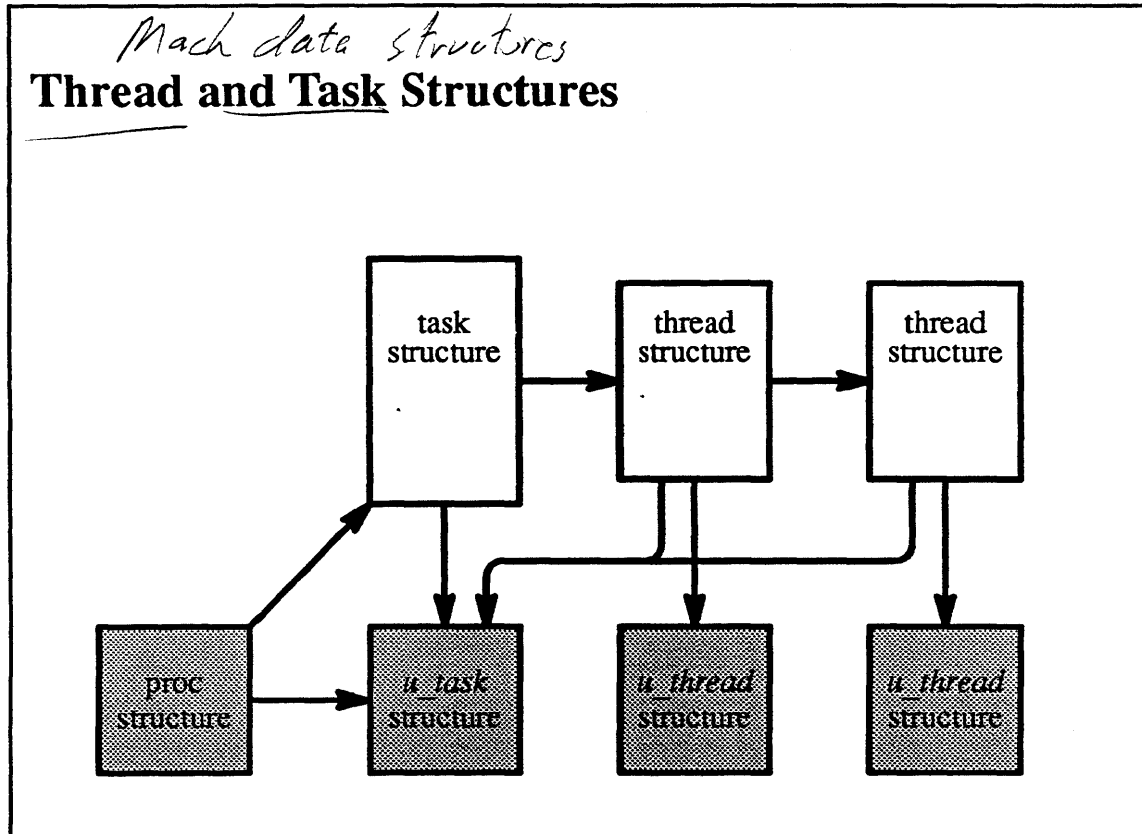
© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: Process Data Structures

Module 2 — The Process Abstraction

2-6. Processes



2-6.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: Thread and Task Structures

Module 2 — The Process Abstraction

2-7. System Calls in OSF/1

System Calls

- Mach system calls — *not portable, like rpc*
- UNIX system calls

Module 2 — The Process Abstraction

Student Notes: System Calls

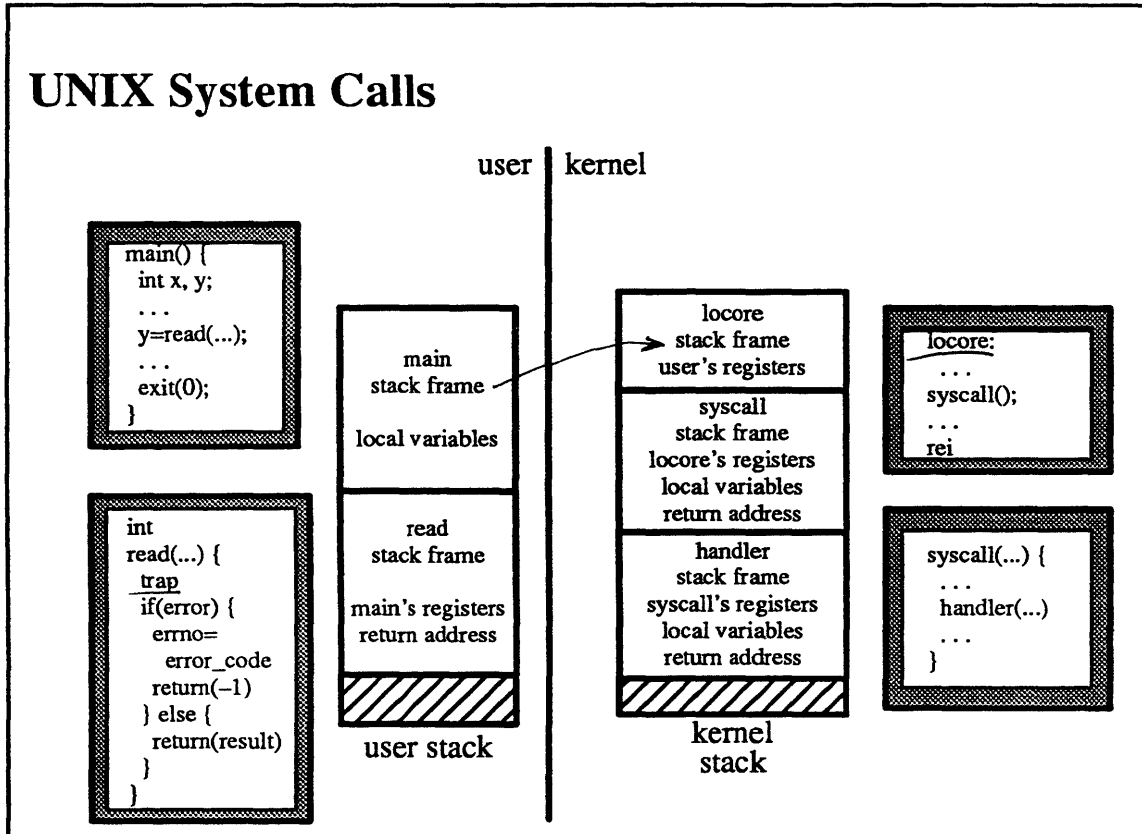
System calls are the sole interface between the user and the operating system. From the user's point of view, a system call is a subroutine call, but the body of this subroutine call involves a switch from unprivileged user mode to privileged system mode. Accomplishing this switch requires an architecture-specific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

Module 2 — The Process Abstraction

2-8. System Calls in OSF/1

UNIX System Calls



2-8.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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*.

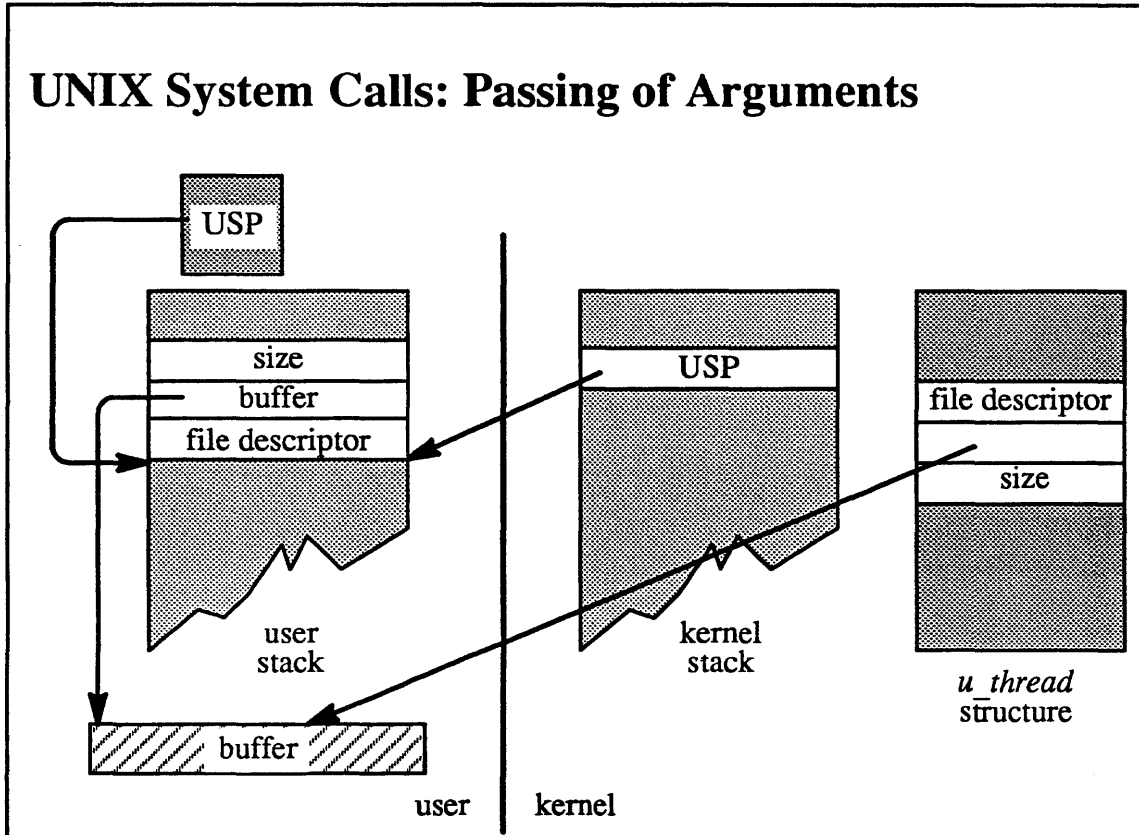
Syscall copies the arguments of the system call to the process's *u_thread* structure, then calls the appropriate system call handler in the kernel.

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.

Module 2 — The Process Abstraction

2-9. System Calls in OSF/1



2-9.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

Student Notes: 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.

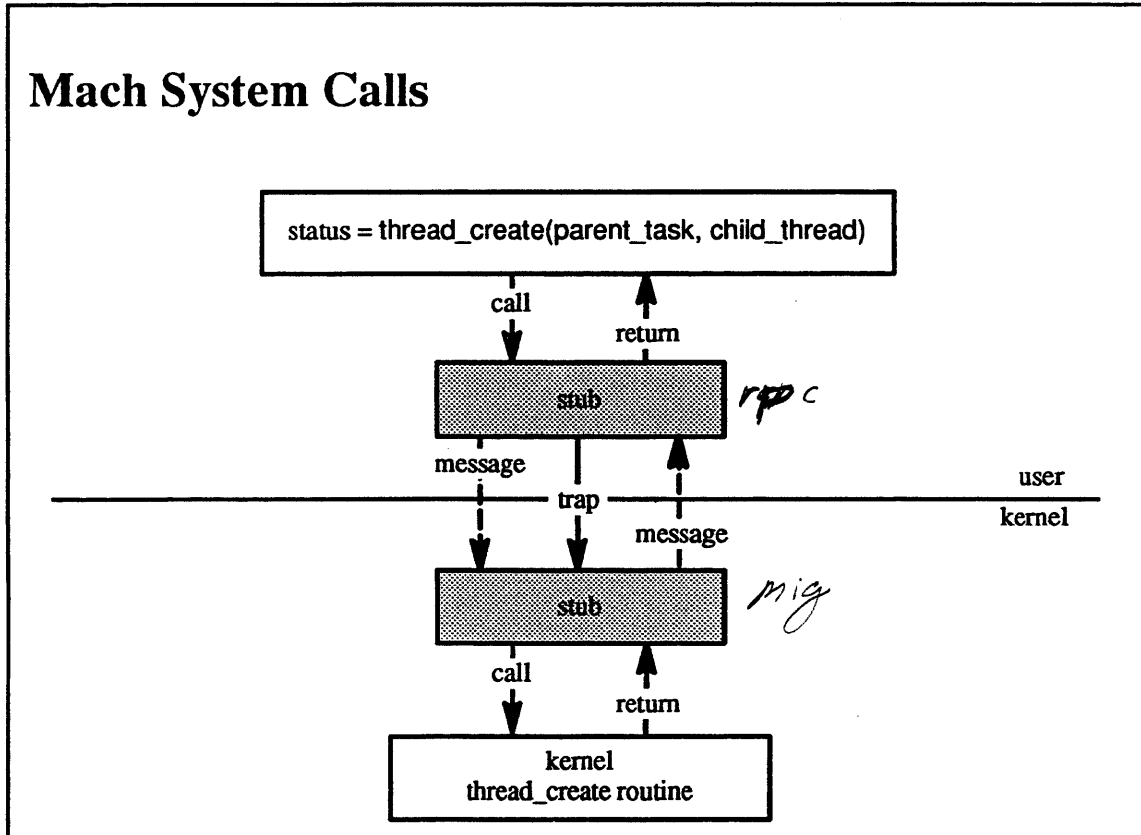
If an error occurred in the system call, UNIX requires that the library routine (e.g. write) return -1 and that the error code be found in the global variable *errno*. This is achieved via cooperation between the operating system and the library routine: the carry bit of the program status word is used 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!

Module 2 — The Process Abstraction

2-11. System Calls in OSF/1



2-11.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

Student Notes: 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 *switch*, 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.

2-13. Synchronization and Thread Management

UNIX Synchronization: Sleep and Wakeup

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

Module 2 — The Process Abstraction

Student Notes: UNIX Synchronization: Sleep and Wakeup

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

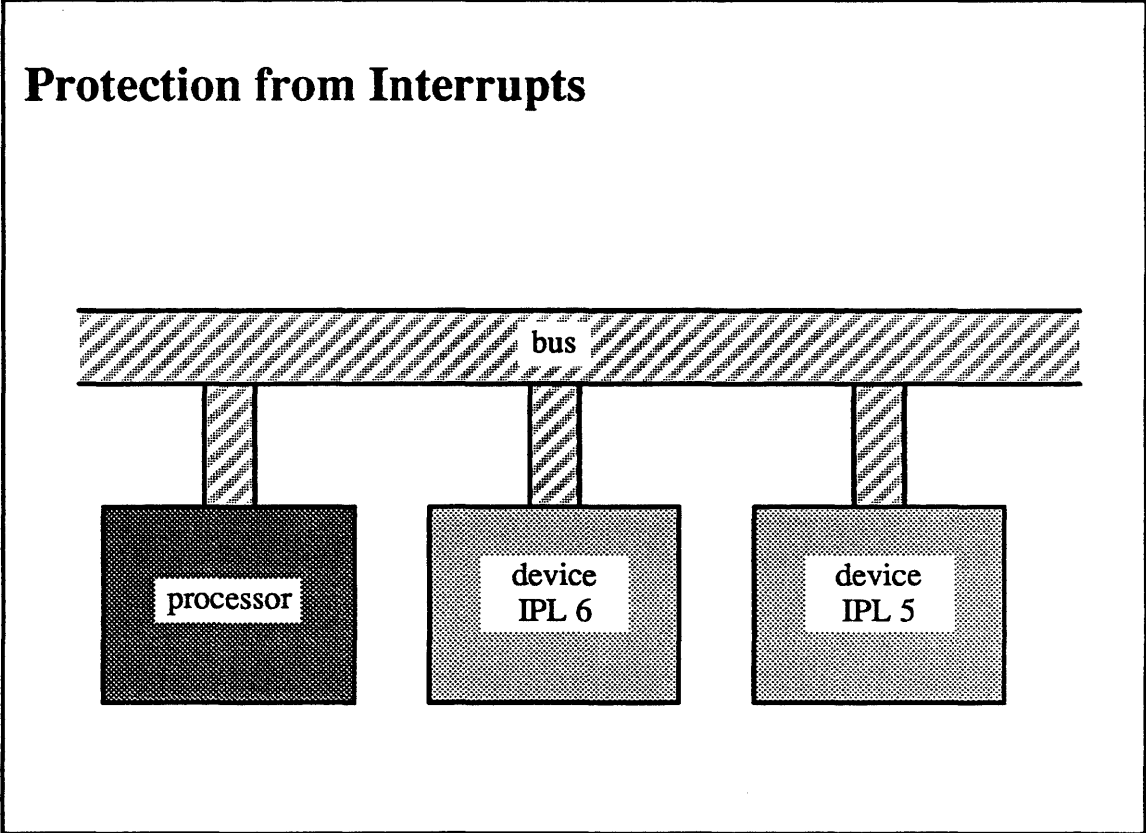
A number of things are overloaded on top of disposition:

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 longjumps 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.)

Module 2 — The Process Abstraction

2-14. Synchronization and Thread Management



Module 2 — The Process Abstraction

Student Notes: 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);
```


Module 2 — The Process Abstraction

Student Notes: 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 *wakeups* 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*.

Module 2 — The Process Abstraction

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

Module 2 — The Process Abstraction

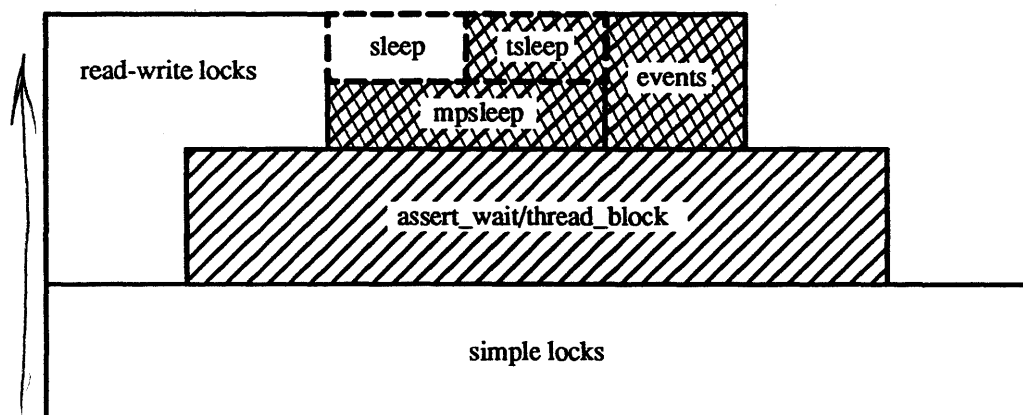
Student Notes: OSF/1 Synchronization

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.

Module 2 — The Process Abstraction

2-17. Synchronization and Thread Management

Synchronization Primitives in OSF/1



not parallelized



interruptibility and timeout are options



interruptibility is an option



interruptibility is not an option

Module 2 — The Process Abstraction

Student Notes: Synchronization Primitives in OSF/1

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

Module 2 — The Process Abstraction

2-18. Synchronization and Thread Management

Simple Locks *spin lock*

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

```
;
```

```
simple_lock_init(lock)
```

```
simple_lock(lock)
```

```
simple_unlock(lock)
```

```
simple_lock_try(lock)
```

Module 2 — The Process Abstraction

Student Notes: 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

<pre>simple_lock(...); if (should_sleep) { simple_unlock(...); sleep(...); } else simple_unlock(...);</pre>	OR	<pre><i>Worse</i> simple_lock(...); if (should_sleep) { sleep(...); simple_unlock(...); } else simple_unlock(...);</pre>
---	----	--

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


Module 2 — The Process Abstraction

Student Notes: 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.

Module 2 — The Process Abstraction

2-20. Synchronization and Thread Management

Combining Unlock with Sleep, part 2

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

*we want to
go to sleep*

=

```
simple_unlock + sleep
```

Module 2 — The Process Abstraction

Student Notes: 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);
```

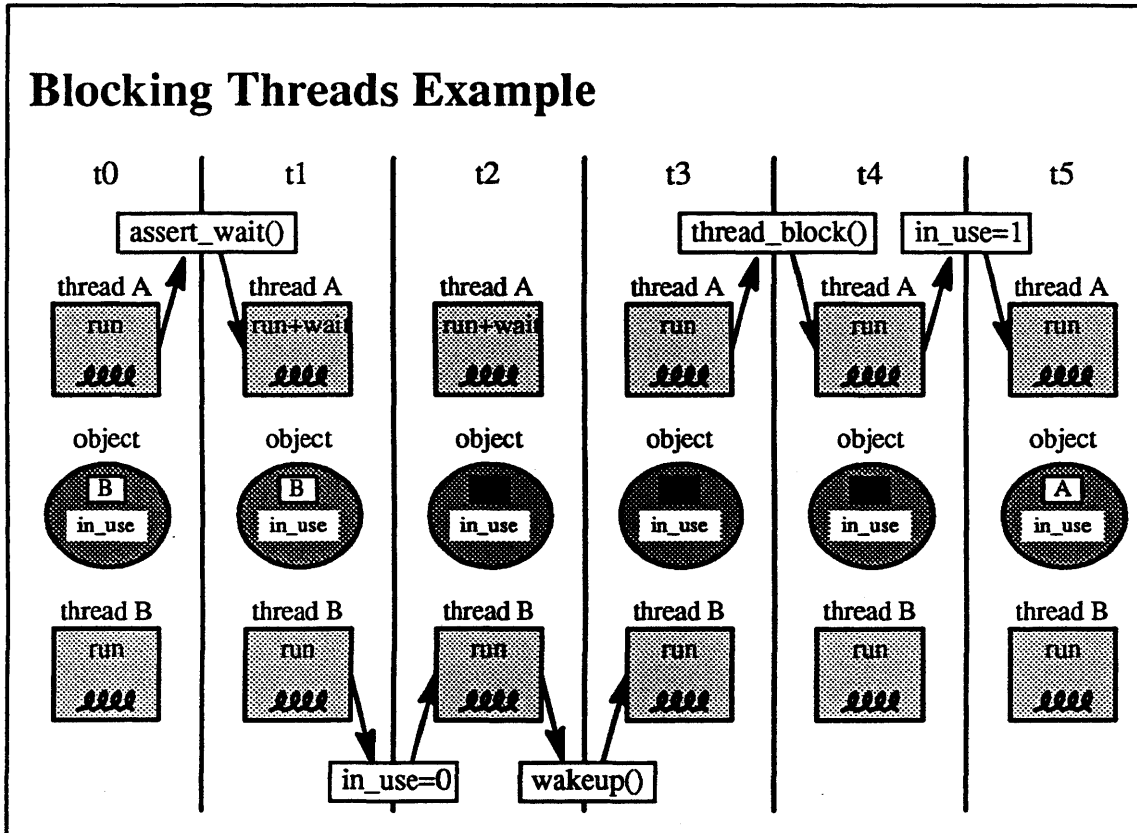
Module 2 — The Process Abstraction

Student Notes: 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.

Module 2 — The Process Abstraction

2-22. Synchronization and Thread Management



2-22.

© 1990, 1991 Open Software Foundation

run - about to go to sleep

Module 2 — The Process Abstraction

Student Notes: Blocking Threads Example

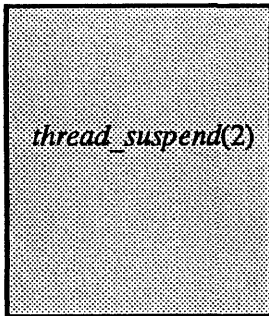
Initially thread B is using the object (has set its *in_use* field) and both thread A and thread B are running (or runnable). Thread A is attempting to use the object but finds by examining the *in_use* field that the object is being used by another thread. It indicates its intention to wait for the 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.

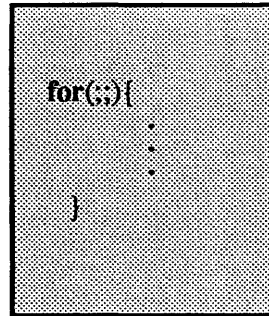
The effect of thread B's call to *wakeup* is to wake up thread A. However, thread A has not gone to sleep yet, so the wait bit is cleared in its state vector. Thread A subsequently calls *thread_block*. Since the wait bit is no longer set, thread A returns from *thread_block* immediately (if the call 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 1 running
on processor A



thread 2 running
on processor B

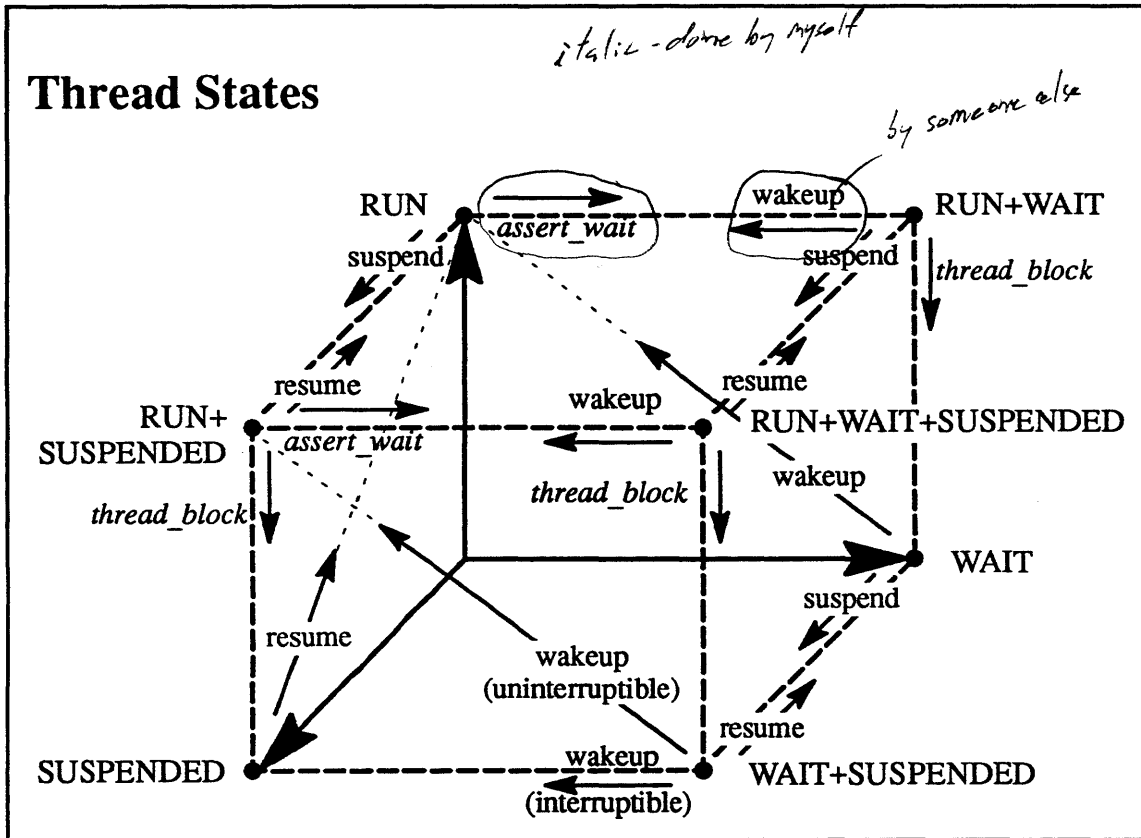
Module 2 — The Process Abstraction

Student Notes: 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).

Module 2 — The Process Abstraction

2-24. Synchronization and Thread Management



add swapped out states

Module 2 — The Process Abstraction

Student Notes: 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)

tsleep(chan, disposition, wmesg, timeout) ^{^T}

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

Module 2 — The Process Abstraction

Student Notes: UNIX-Style Sleep

UNIX-style *sleeps* 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. *Sleeps* 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

Module 2 — The Process Abstraction

Student Notes: Waking Up

Wakeup routines:

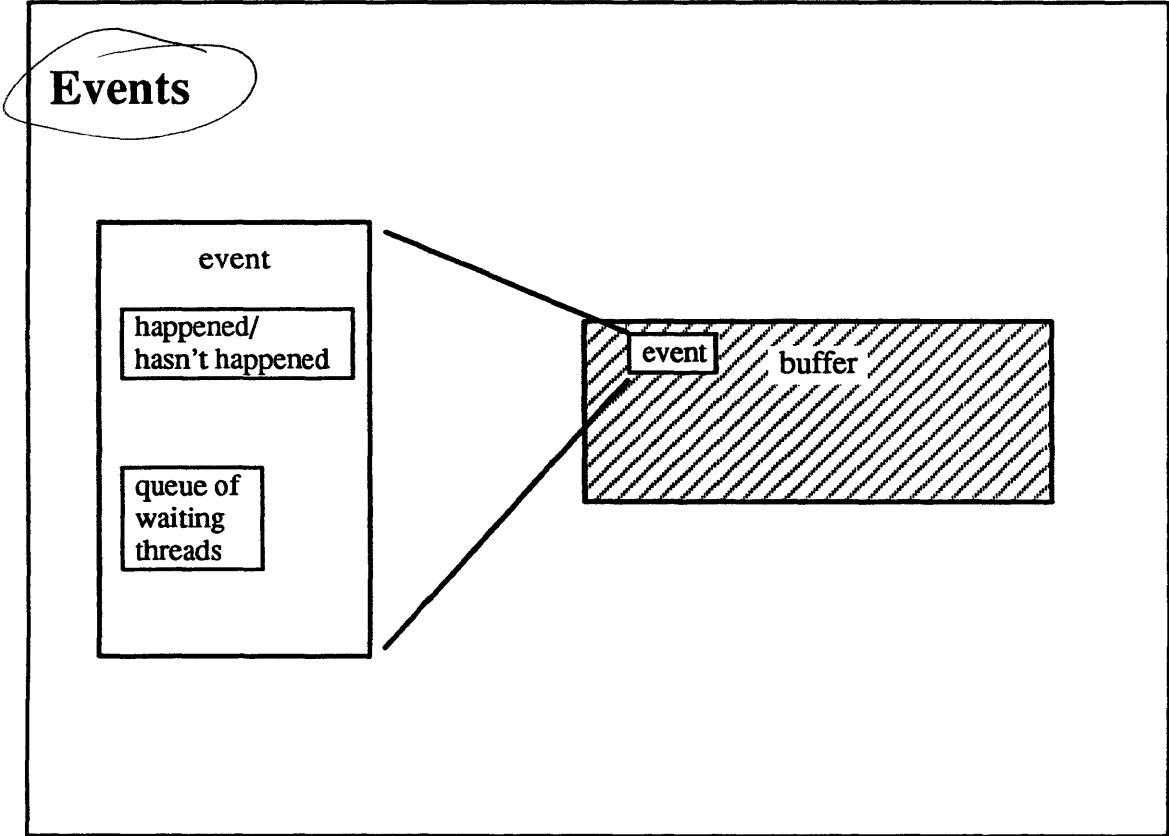
- *clear_wait*: wakes up a particular thread. If the *interruptible-only* flag is set, then the thread is awakened only if it is in an interruptible sleep (this flag is used, for example, to wake up a thread conditionally in response to a signal).
- *thread_wakeup_one*: wakes up the first thread waiting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

Module 2 — The Process Abstraction

2-27. Synchronization and Thread Management



2-27.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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

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)

Module 2 — The Process Abstraction

Student Notes: 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.

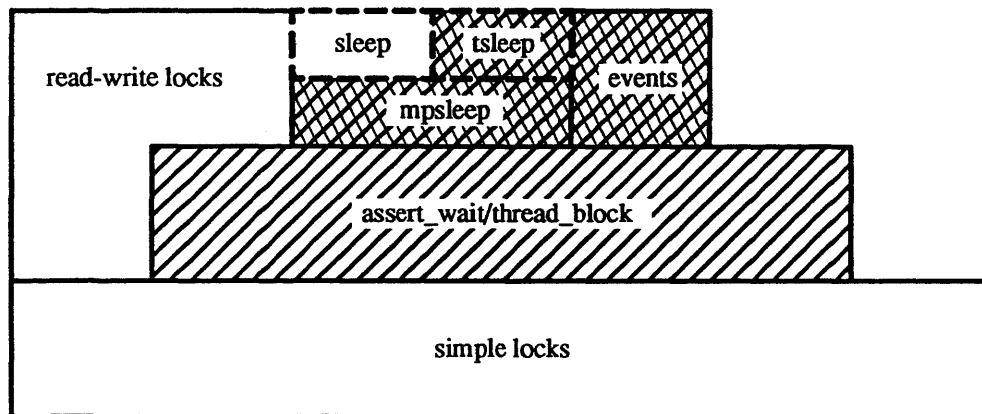
In some situations, it may be convenient to use a read-write lock *recursively*, i.e., a thread may “take” a lock even if it already has it. This notion is useful in situations in which a thread possesses a lock but is calling a routine that causes it to take the lock again (if the lock is not set to 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.

Module 2 — The Process Abstraction

2-29. Synchronization and Thread Management

Synchronization in OSF/1: Summary



not parallelized



interruptibility and timeout are options



interruptibility is an option



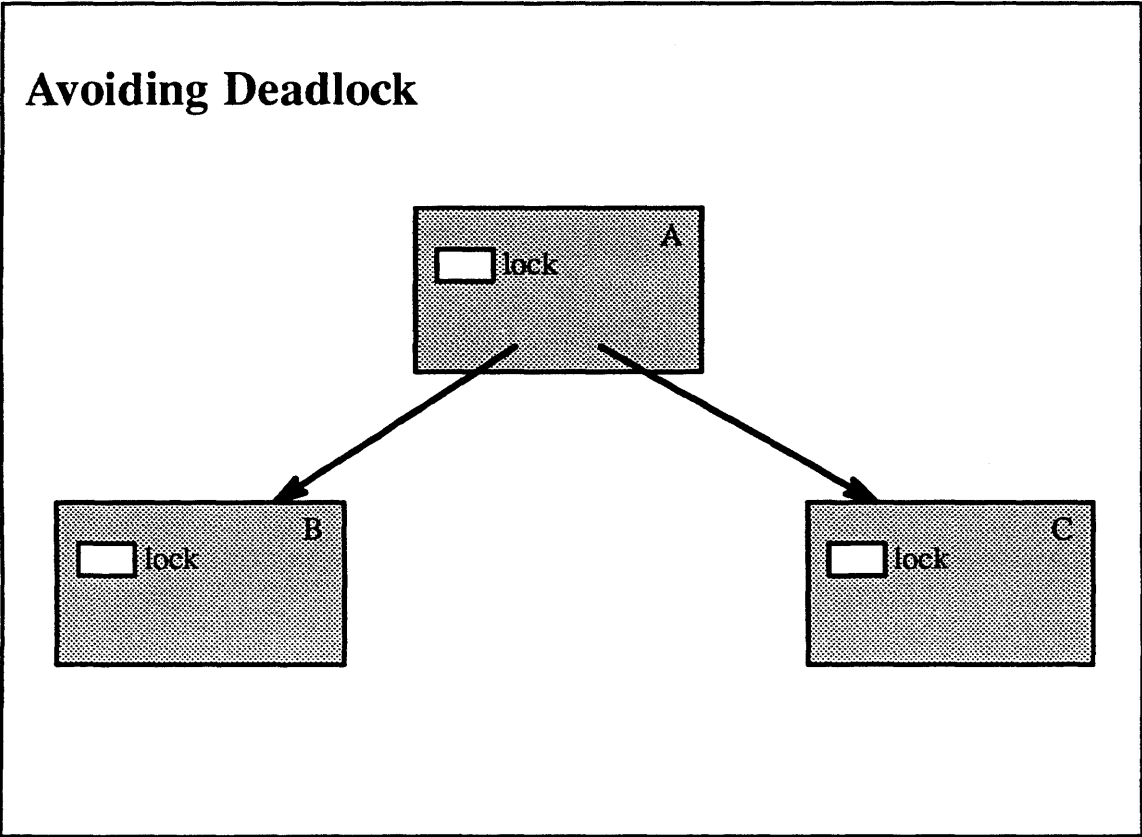
interruptibility is not an option

Module 2 — The Process Abstraction

Student Notes: 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



Module 2 — The Process Abstraction

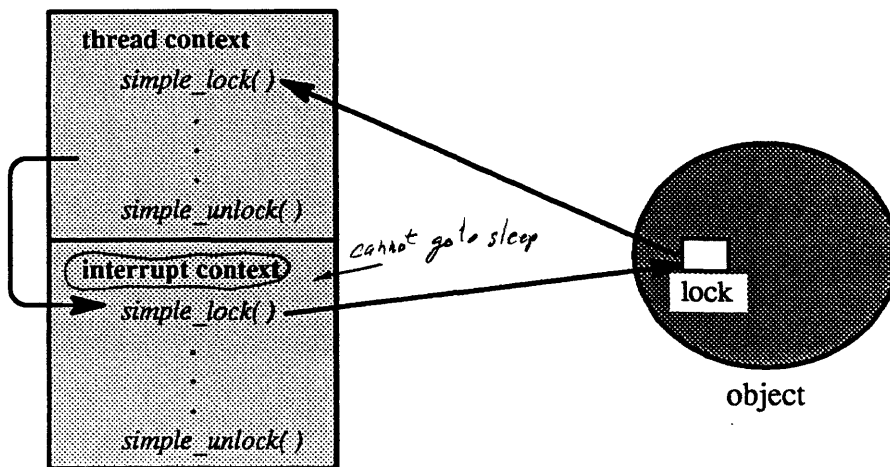
Student Notes: Avoiding Deadlock

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.

Module 2 — The Process Abstraction

2-31. Synchronization and Thread Management

Taking Locks in the Interrupt Context



2-31.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

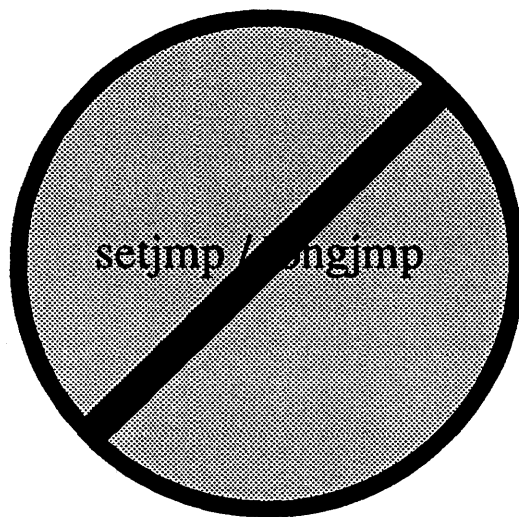
Student Notes: Taking Locks in the Interrupt Context

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



Module 2 — The Process Abstraction


Student Notes: Signals and Blocked Threads

All UNIX synchronization calls (*sleep*, etc.) return to their caller even if they have been interrupted by a signal. The *sleep* 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 <i>sleep</i> timed out

2-33. Signals and Exception Handling

Signals

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


Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

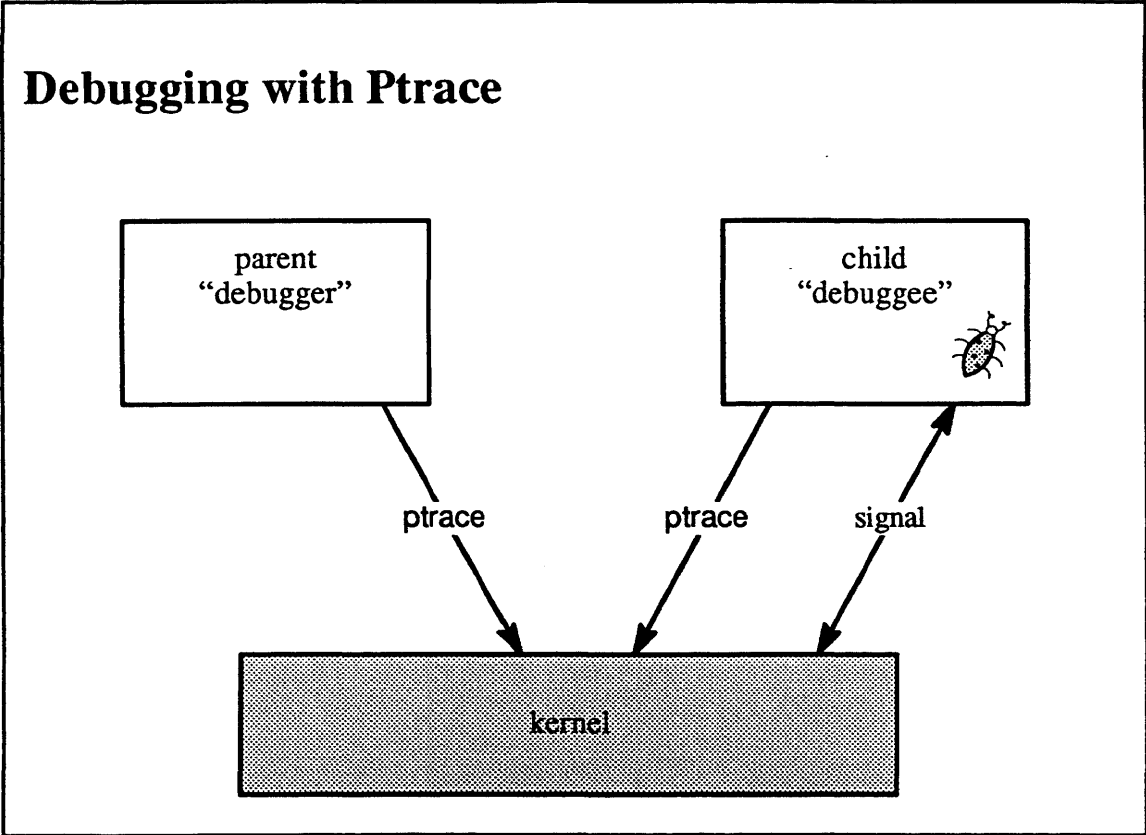
Student Notes: Signals and Multithreaded Processes

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.).

Module 2 — The Process Abstraction

2-35. Signals and Exception Handling



2-35.

© 1990, 1991 Open Software Foundation

proc in system

Module 2 — The Process Abstraction

Student Notes: 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.

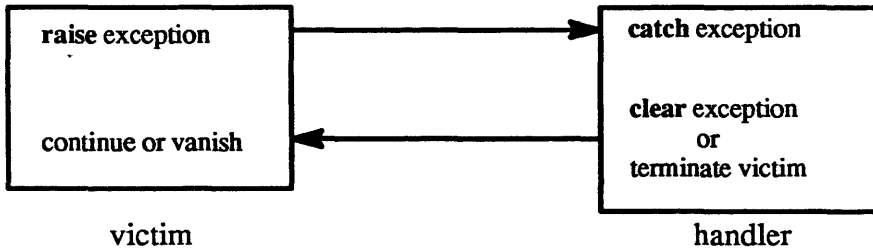
The parent debugger process may wait for a child to stop via the `wait` system call. The parent may send requests to the child by issuing `ptrace` calls with positive arguments. With `ptrace`, it may examine and modify the child’s memory and registers, and control the child’s response to signals. The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

Module 2 — The Process Abstraction

2-36. Signals and Exception Handling

Exception Handling in Mach

ex: div by 0



Module 2 — The Process Abstraction

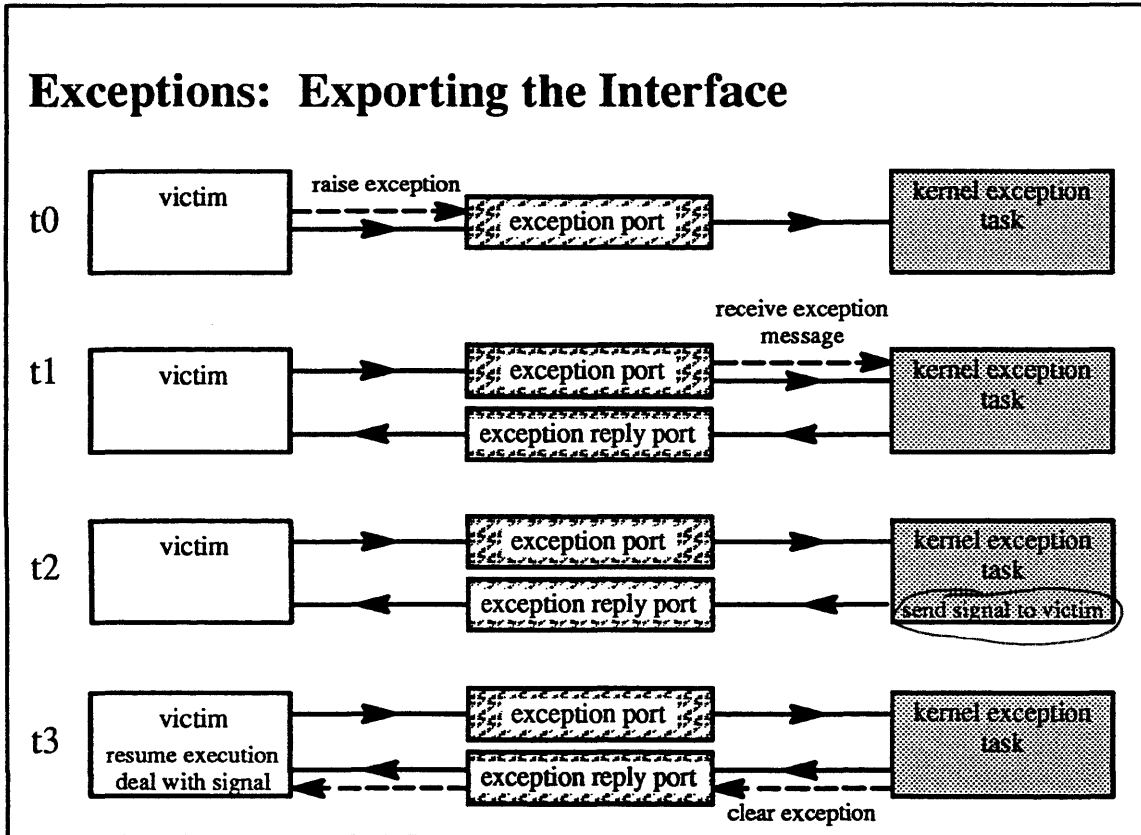
Student Notes: Exception Handling in Mach

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

Module 2 — The Process Abstraction

2-37. Signals and Exception Handling



2-37.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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.

Module 2 — The Process Abstraction

2-38. Threads

Creating a Thread

not what user sees

- Step #* 1. thread_create
 - 2. thread_set_state
 - 3. thread_resume
- } Mach
go

Module 2 — The Process Abstraction

Student Notes: 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`

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

Student Notes: Terminating a Thread

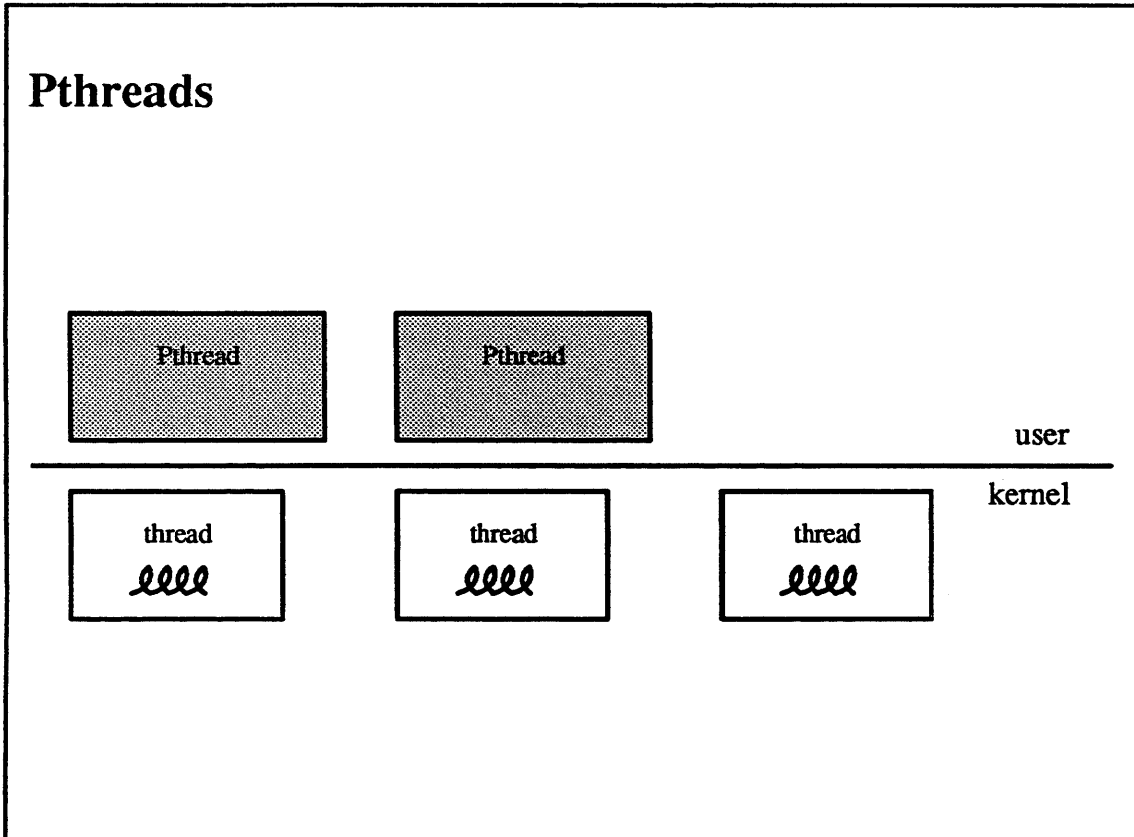
Terminating another thread is straightforward: the victim thread is stopped at a clean point, i.e., a point at which it is holding no locks, and then eliminated.

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.

Module 2 — The Process Abstraction

2-41. Threads



2-41.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

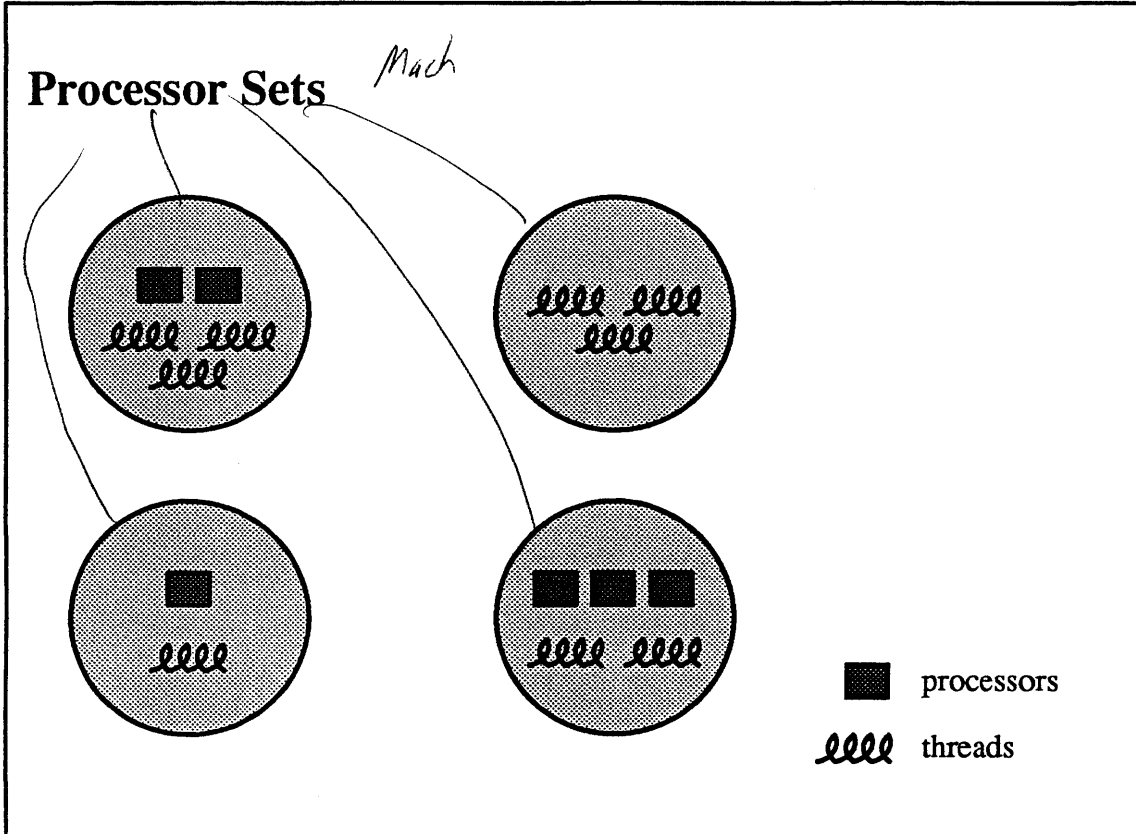
Student Notes: Scheduling

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.

Module 2 — The Process Abstraction

2-43. Scheduling



2-43.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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.

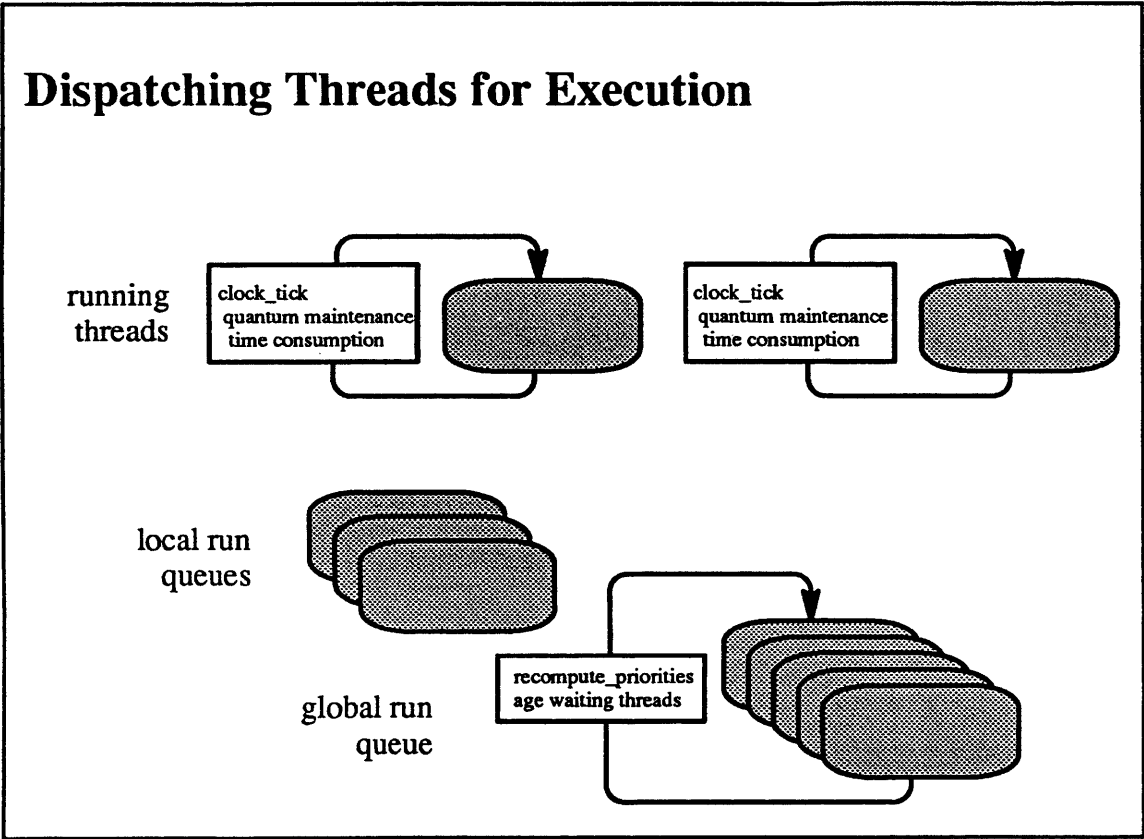
Processors are partitioned into *containers* called *processor sets*: each container holds zero or more processors, and each processor is in exactly one container. Threads are also assigned to these containers: threads may run only on a processor and its container (processor set). By default, there is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



Module 2 — The Process Abstraction

Student Notes: 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.

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 *~ Real time*

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

Student Notes: Time-Shared Threads

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

$$\textit{sched_average} = \textit{current_usage} * \textit{load} + (5/8) * \textit{sched_average}$$

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.

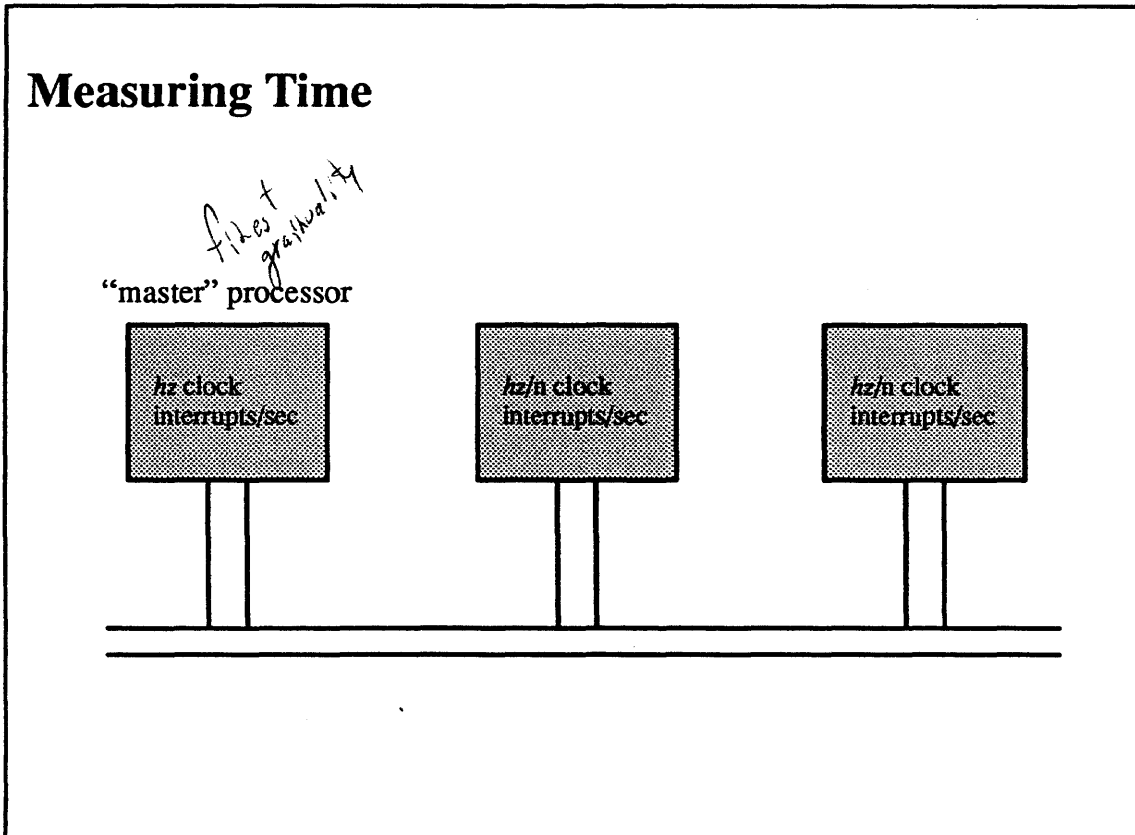
OSF/1 uses a distributed approach to compute this average efficiently: the *sched_average* computation is done in the clock-interrupt context for the currently running threads. Every two seconds all threads in the global run queues are "aged" by multiplying their *sched_averages* by $(5/8)^n$, where 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.

Module 2 — The Process Abstraction

2-47. Scheduling



2-47.

© 1990, 1991 Open Software Foundation

- ① Time out reqs
- ② Per thread/processor stats
- ③ Scheduling

Module 2 — The Process Abstraction

Student Notes: 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

Module 2 — The Process Abstraction

Student Notes: Time Slicing

The quantum for fixed-priority threads is settable for each thread. However, for time-shared threads, the time quantum is typically $\frac{1}{10}$ 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 $\frac{1}{10}$ 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 $\frac{1}{10}$ 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 — *for spin lock*
- Priority depression — *Setter*

Module 2 — The Process Abstraction

Student Notes: 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 `switch` system call returns an indication of whether another runnable thread is waiting to use the caller’s processor. The `switch_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

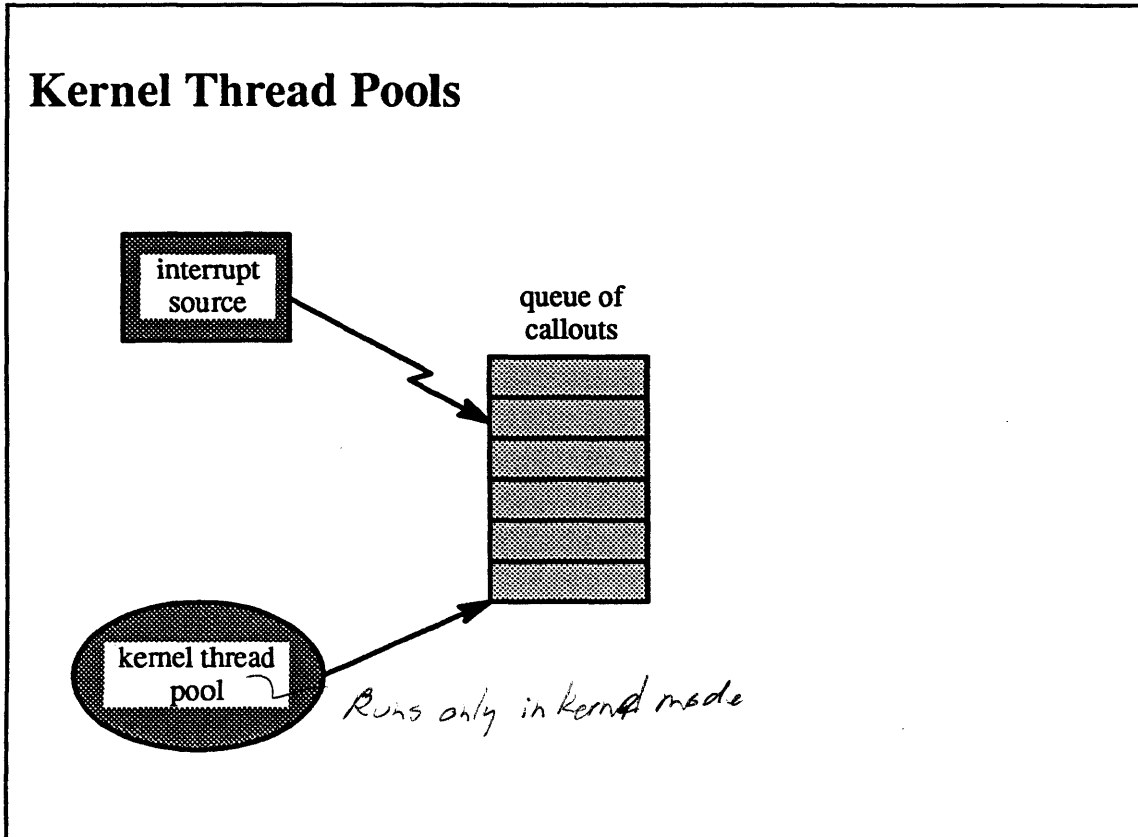
Module 2 — The Process Abstraction

Student Notes: 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



2-51.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: Kernel Thread Pools

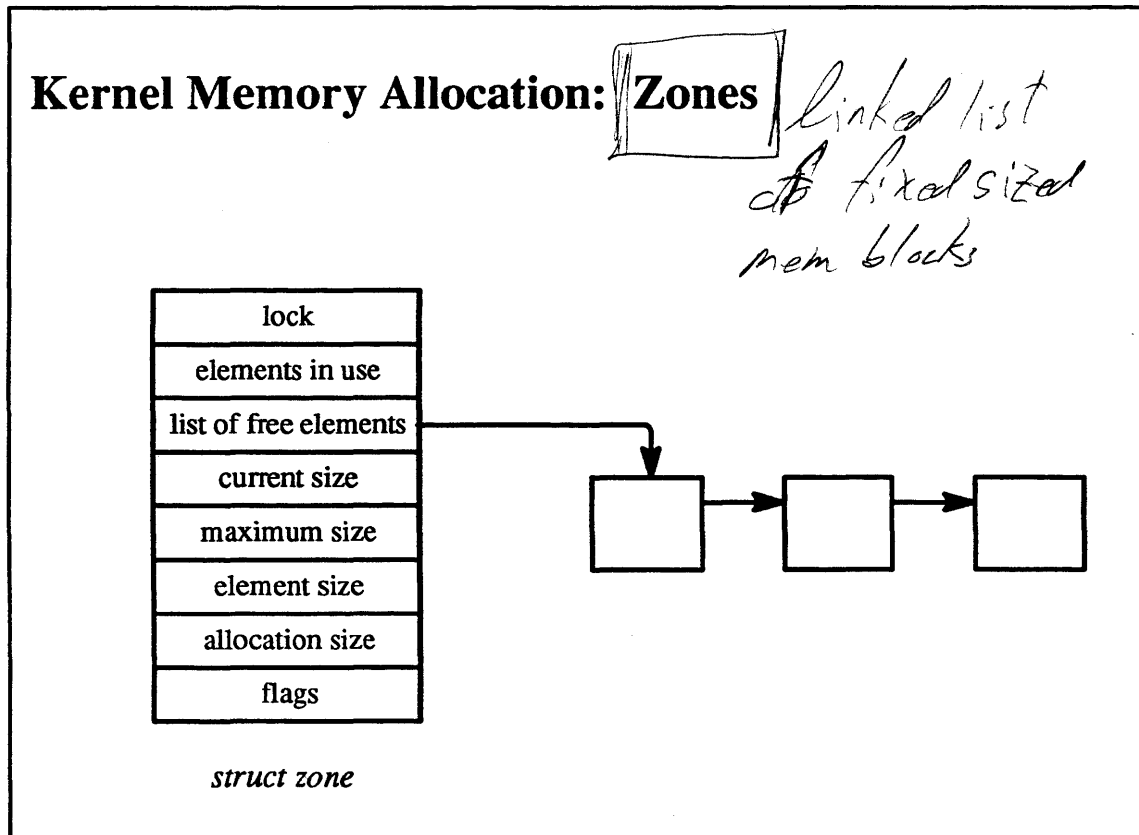
Kernel thread pools are used in a number of places to perform actions in a thread context that would otherwise be performed in the interrupt context. These pools are particularly useful for multiprocessors but may be used for uniprocessors as well.

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).

Module 2 — The Process Abstraction

2-52. Zoned Memory Allocation



2-52.

© 1990, 1991 Open Software Foundation

Module 2 — The Process Abstraction

Student Notes: 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?

Module 2 — The Process Abstraction

- 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 2 — The Process Abstraction

Module 3 — Messages and Ports

Module Contents

1. Messages	3-4
Representation	
Contents	
2. Ports	3-8
Representation	
Port sets	
Naming ports	
Ports as object references	
Port destruction	
Backup ports	
3. Flow of Control	3-28
Sending a message	
Receiving a message	

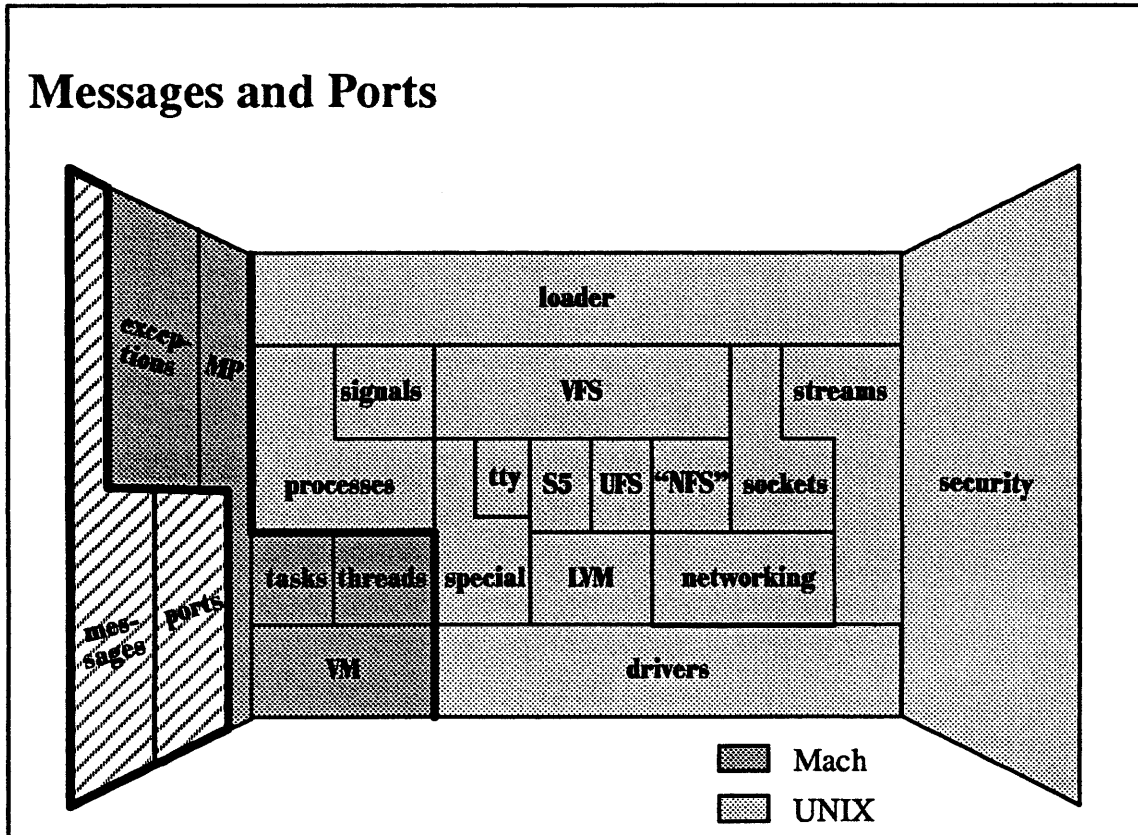
Module Objectives

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



3-1.

© 1990, 1991 Open Software Foundation

Module 3 — Messages and Ports

Student Notes: Messages and Ports

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

Module 3 — Messages and Ports

3-2. Messages

Messages

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

Module 3 — Messages and Ports

Student Notes: 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.)

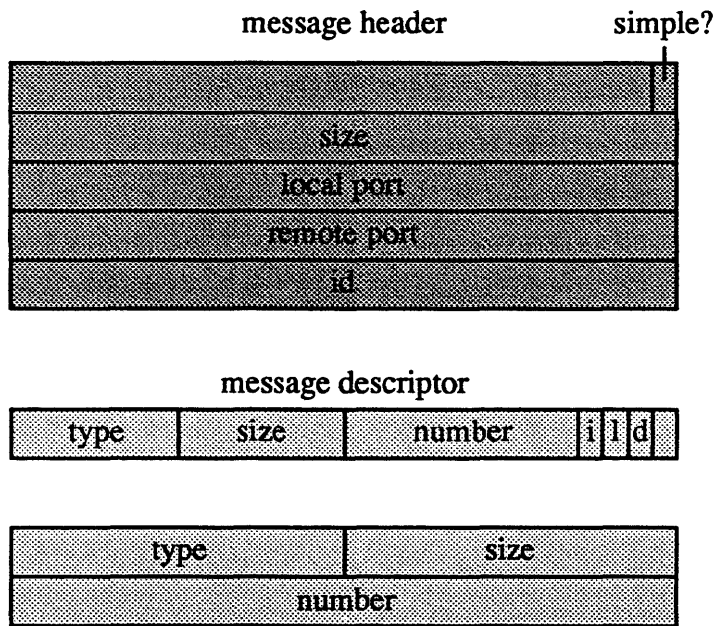
The message must contain a reference to the *destination port*, which is the port through which the message is transferred, and may contain a reference to a *return port* through which a reply can be sent.

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.

Module 3 — Messages and Ports

3-3. Messages

Message Data Structure



Module 3 — Messages and Ports

Student Notes: Message Data Structure

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

Message header:

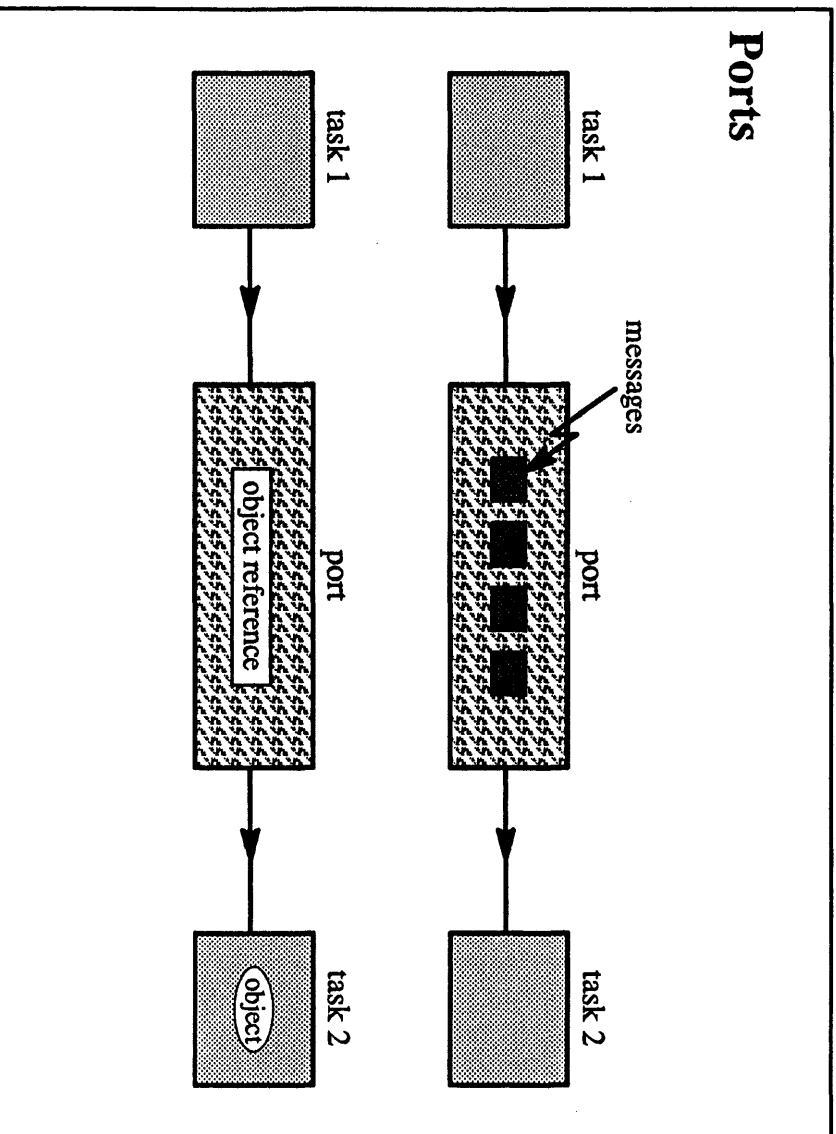
- *simple?* no ports or out-of-line data?
- *size*: total bytes (except for out-of-line data)
- *local port*: optional port through which a reply might be sent
- *remote port*: port for sending message
- *id*: application-specific id

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

Module 3 — Messages and Ports

3-4. Ports



3-4.

© 1990, 1991 Open Software Foundation

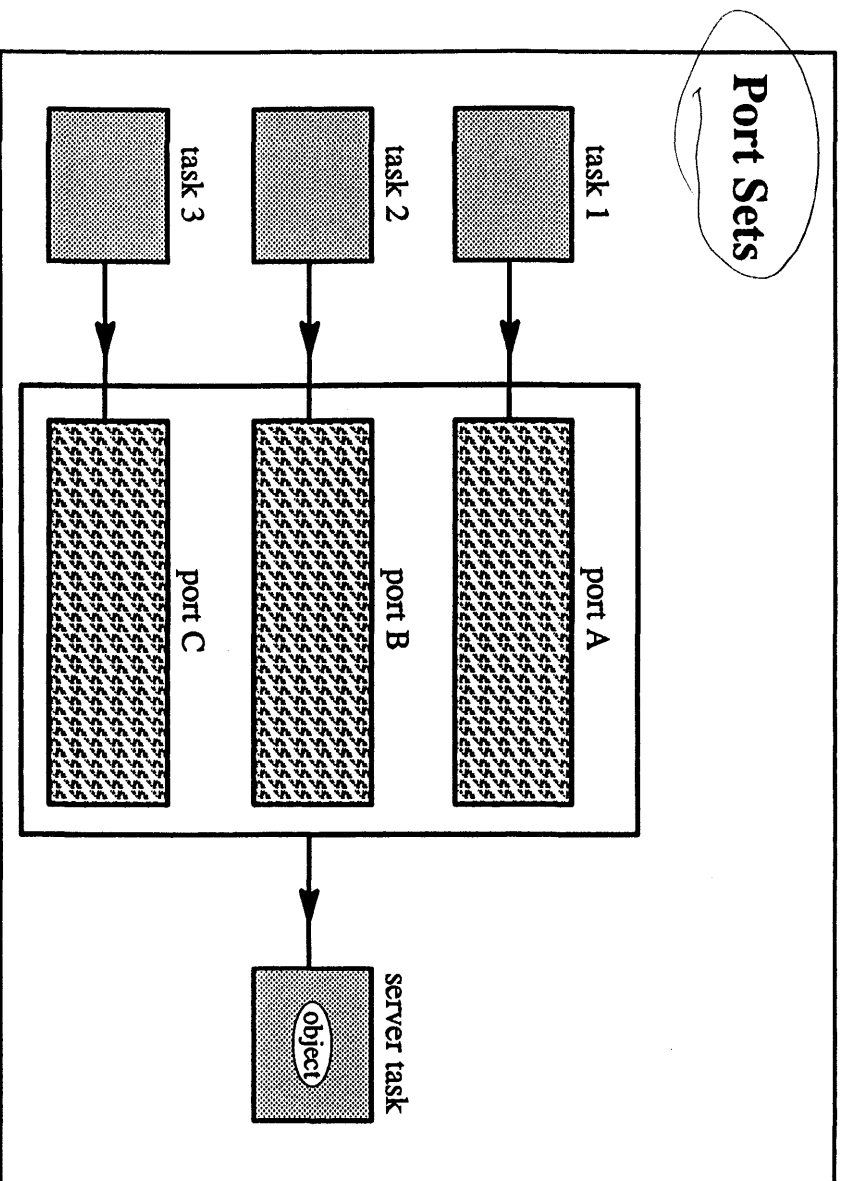
Module 3 — Messages and Ports

Student Notes: 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.

Module 3 — Messages and Ports

3-5. Ports



3-5.

© 1990, 1991 Open Software Foundation

Module 3 — Messages and Ports

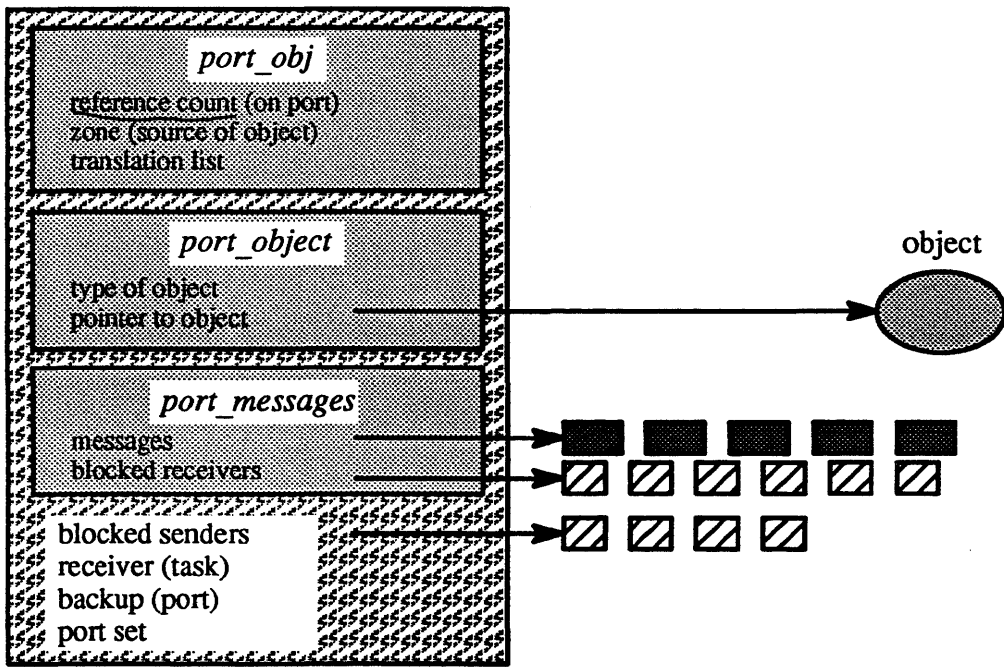
Student Notes: Port Sets

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.

Module 3 — Messages and Ports

3-6. Ports

The Kernel Port Structure



3-6.

© 1990, 1991 Open Software Foundation

Module 3 — Messages and Ports

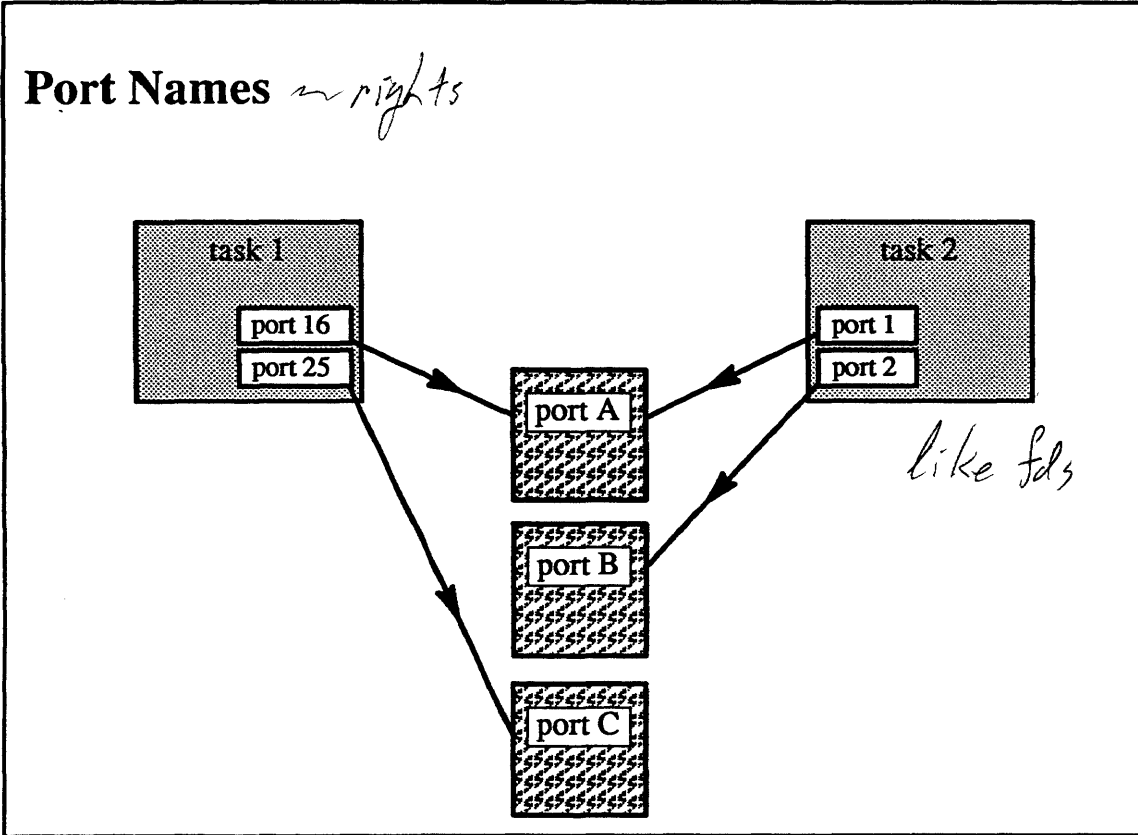
Student Notes: Kernel Port Structure

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.

Module 3 — Messages and Ports

3-7. Ports



3-7.

© 1990, 1991 Open Software Foundation

Module 3 — Messages and Ports

Student Notes: 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

Port Name Translation

<u>task that owns the port right</u>	local name [#] for port right
type: <u>send, receive, port set</u>	pointer to object/port

4 lists →

Module 3 — Messages and Ports

Student Notes: Port Name Translation

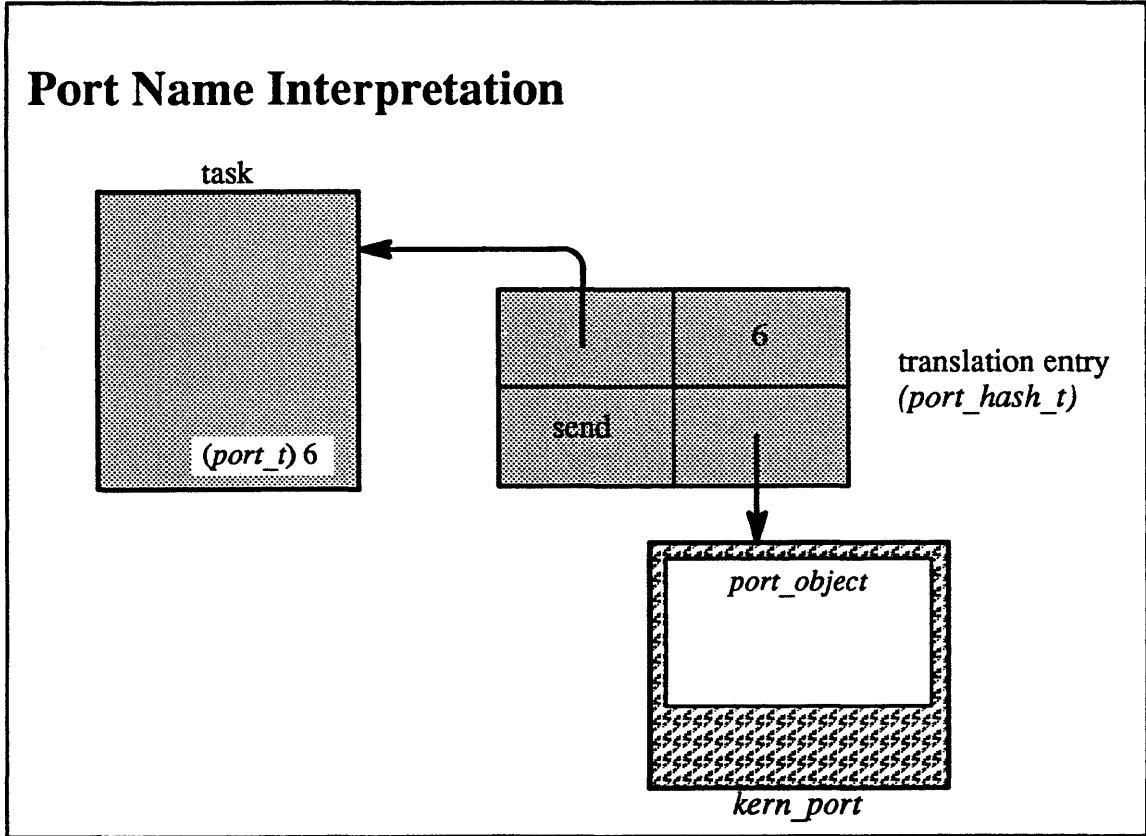
The first two lists are doubly linked and are used just to keep track of the *port rights* associated with each object. The second two lists are actually hash tables and are used for efficient translation from external to internal names and vice versa.

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)

Module 3 — Messages and Ports

3-9. Ports



3-9.

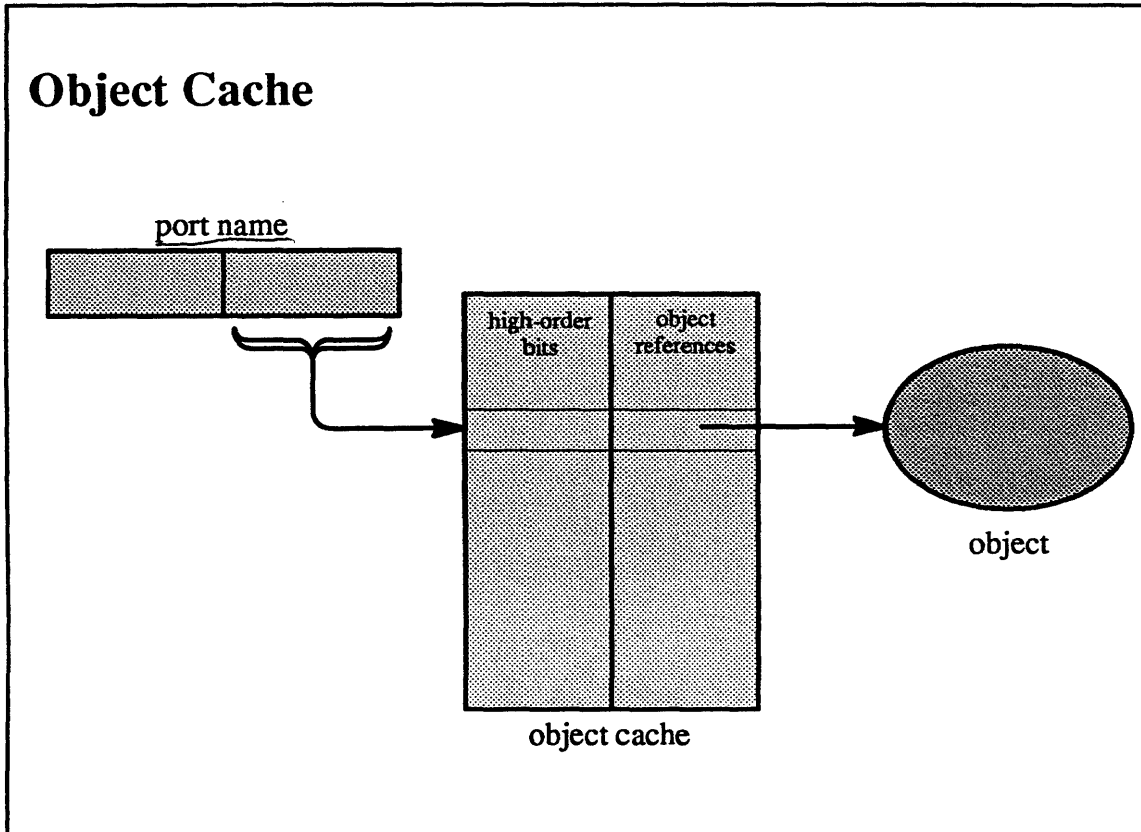
© 1990, 1991 Open Software Foundation

Module 3 — Messages and Ports

Student Notes: Port Name Interpretation

Module 3 — Messages and Ports

3-10. Ports



3-10.

© 1990, 1991 Open Software Foundation

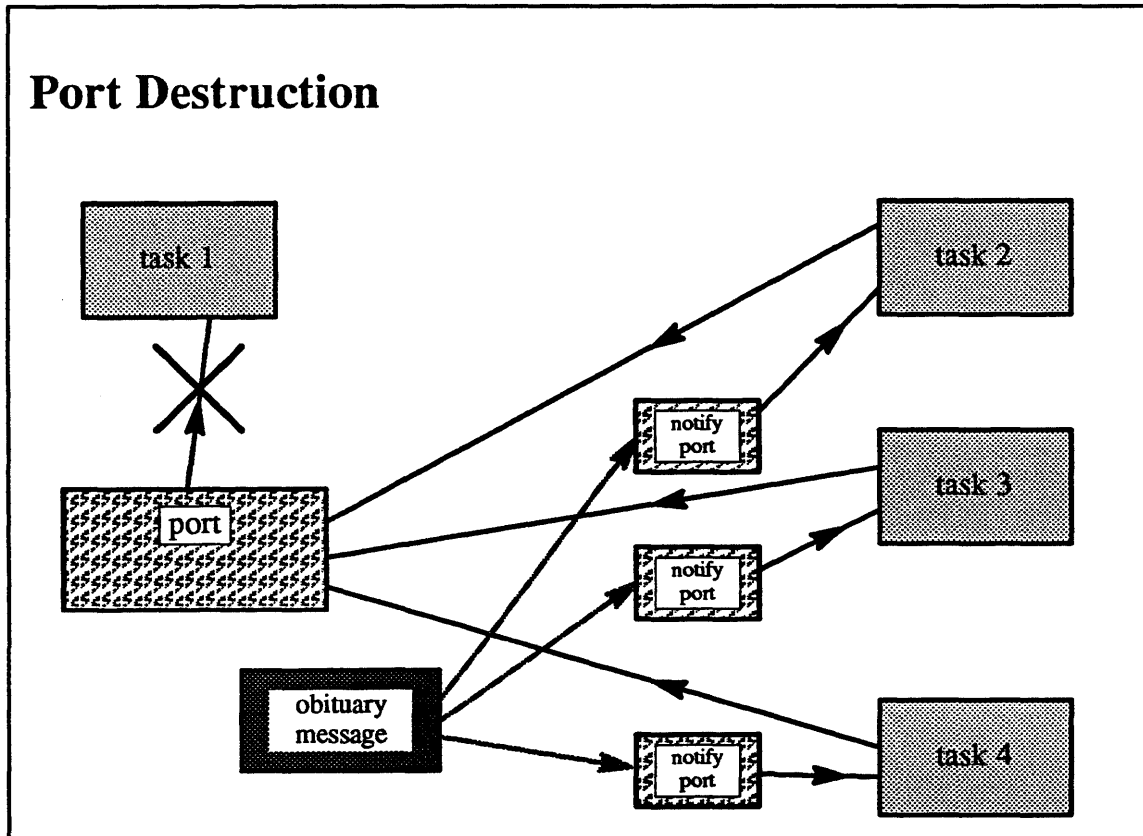
Module 3 — Messages and Ports

Student Notes: 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.

Module 3 — Messages and Ports

3-11. Ports



3-11.

© 1990, 1991 Open Software Foundation

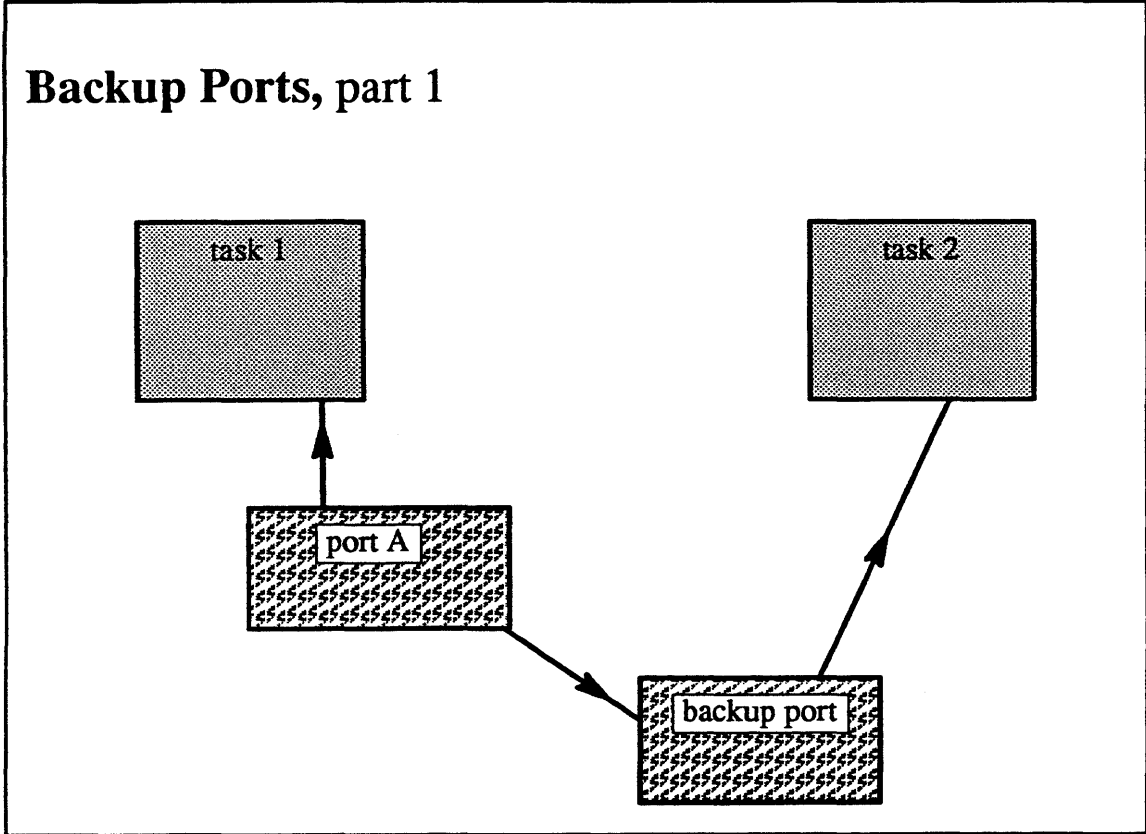
Module 3 — Messages and Ports

Student Notes: 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.

Module 3 — Messages and Ports

3-12. Ports



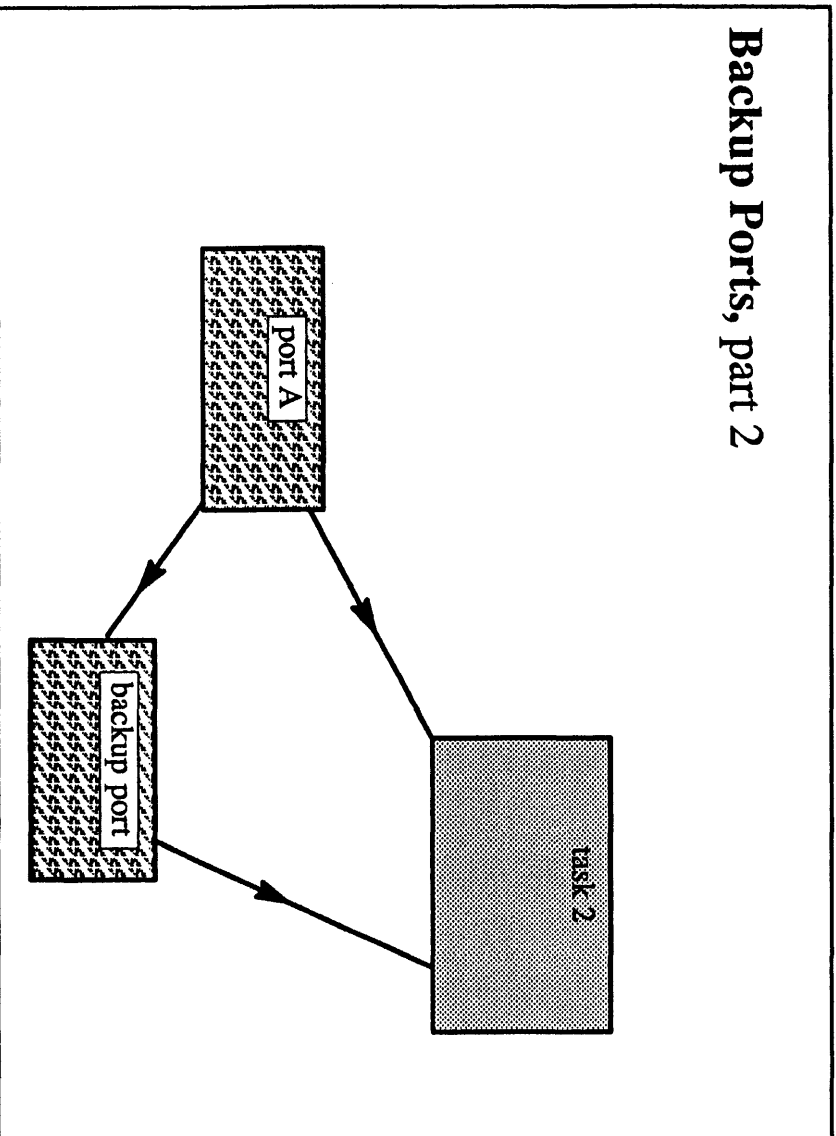
Module 3 — Messages and Ports

Student Notes: 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



3-13.

© 1990, 1991 Open Software Foundation

Module 3 — Messages and Ports

Student Notes: Backup Ports, part 2

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

Module 3 — Messages and Ports

3-14. Flow of Control

msg_send *very few traps to Mach*

- *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

- *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

Module 3 — Messages and Ports

Student Notes: `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

- *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

Module 3 — Messages and Ports

Student Notes: 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.

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

1. Lazy Evaluation	4-4
2. VM Components	4-6
Data structures	
Representing an address space	
Separating architecture-independent from architecture-dependent aspects	
3. Memory Objects	4-30
Vnode pager	
External memory object managers	
Paging and swapping	
4. Copying and Sharing	4-58
Virtual copy operation	
Read/write sharing	
Permanent memory objects	
External memory objects	
5. The Pmap Module	4-102
Required functionality	
A typical architecture	
TLB shutdown	

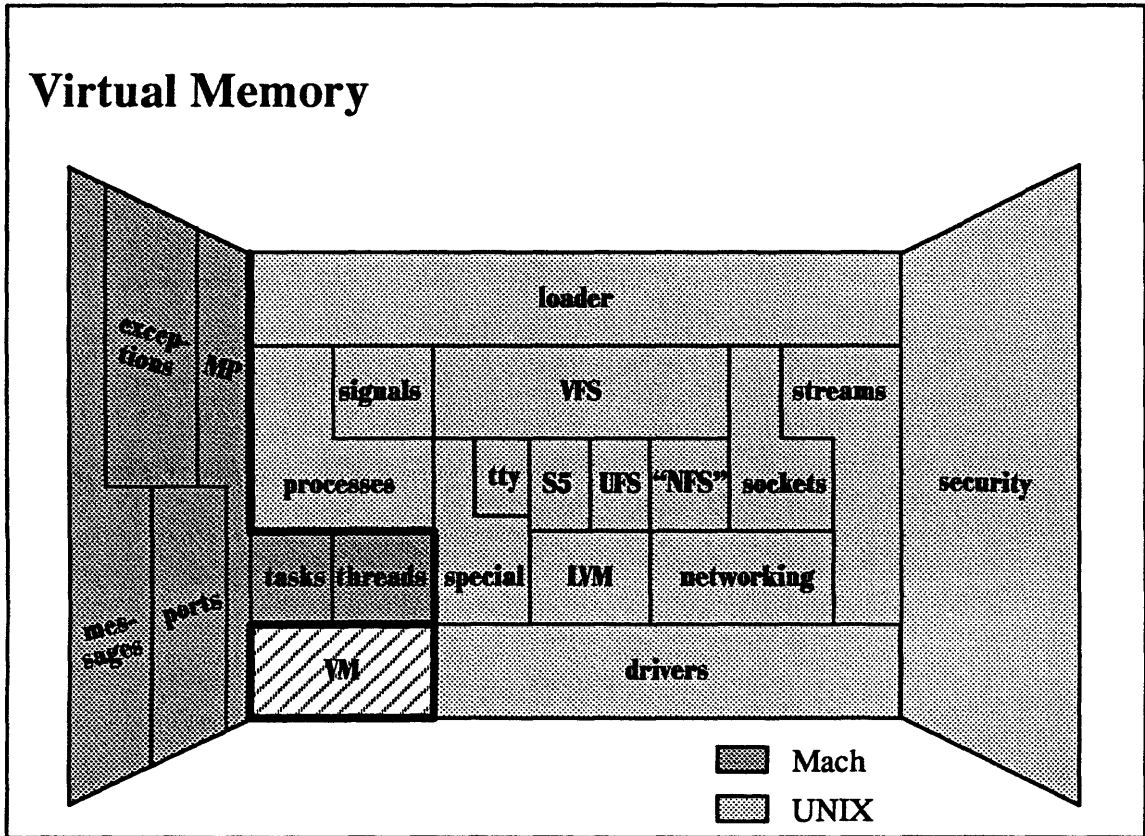
Module Objectives

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



Module 4 — Virtual Memory

Student Notes: 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.

Very portable

Module 4 — Virtual Memory

4-2. Lazy Evaluation

Lazy Evaluation *← not use for kernel stack*

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

Module 4 — Virtual Memory

Student Notes: 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:

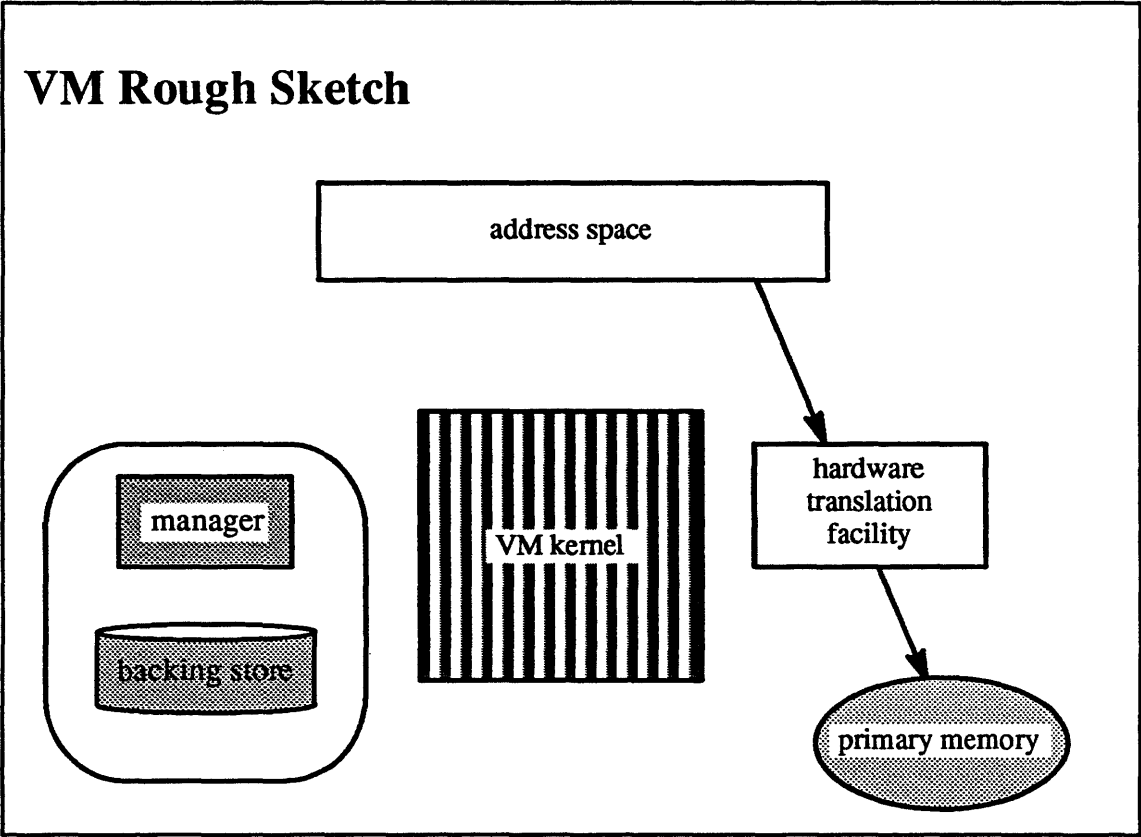
- no physical address maps are created until they are needed to satisfy a reference
- no pages are allocated until they are needed
- no page is copied until two copies are necessary
- no backing store is allocated until it is needed

. Good news - Faster

. Bad news - could run out

Module 4 — Virtual Memory

4-3. VM Components



4-3.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

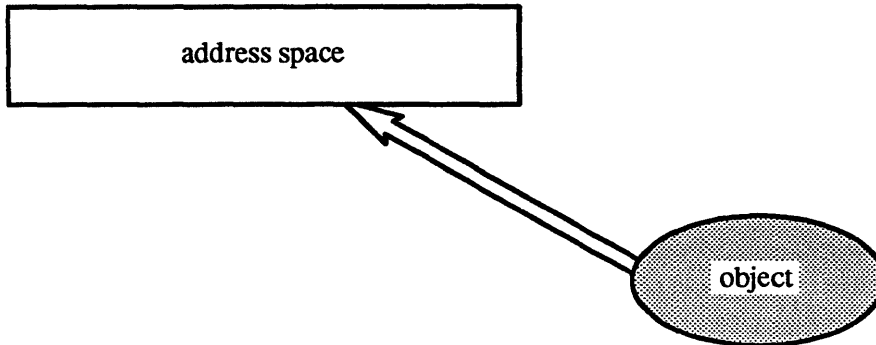
Student Notes: VM Rough Sketch

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



4-4.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: 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:

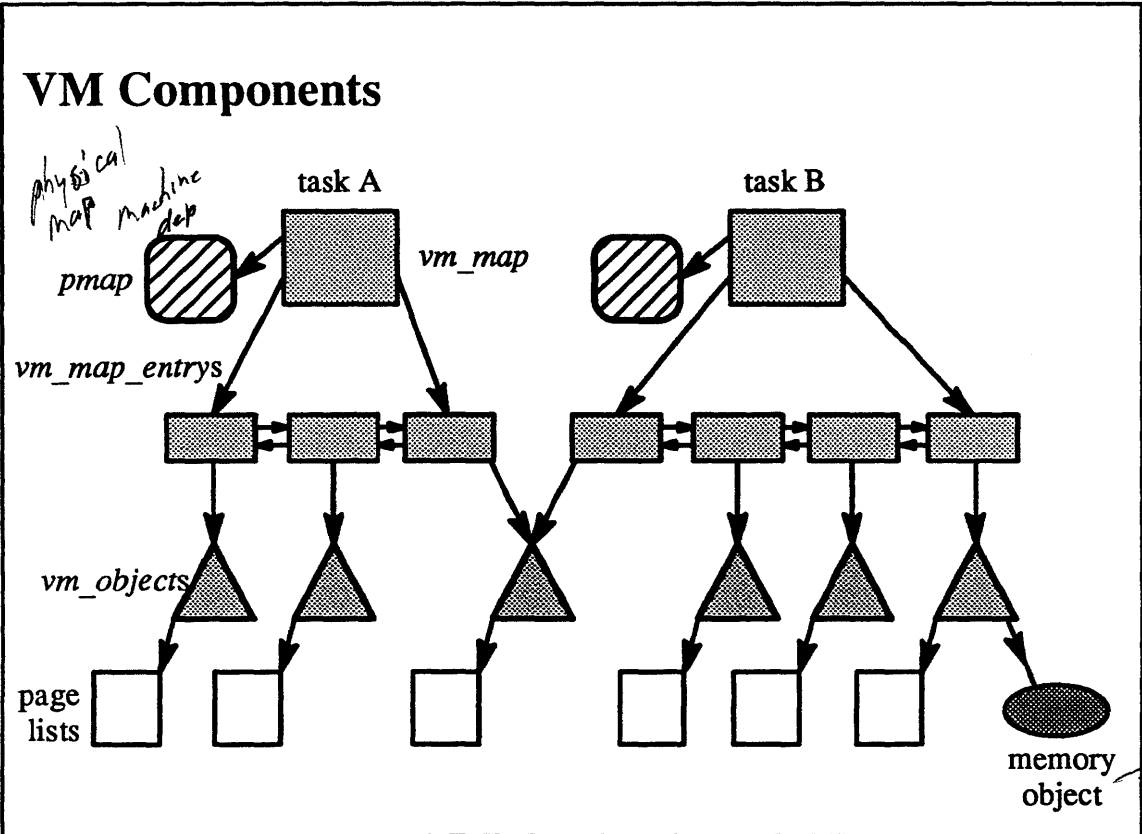
1. temporary: it has no name and hence no permanent existence.
2. a file: it has a name and hence a permanent existence, but should changes made to the address space be reflected to the file?
3. a user-provided object: a user process provides the contents of the object and manages its modifications.

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

Module 4 — Virtual Memory

4-5. VM Components



Module 4 — Virtual Memory

Student Notes: VM Components

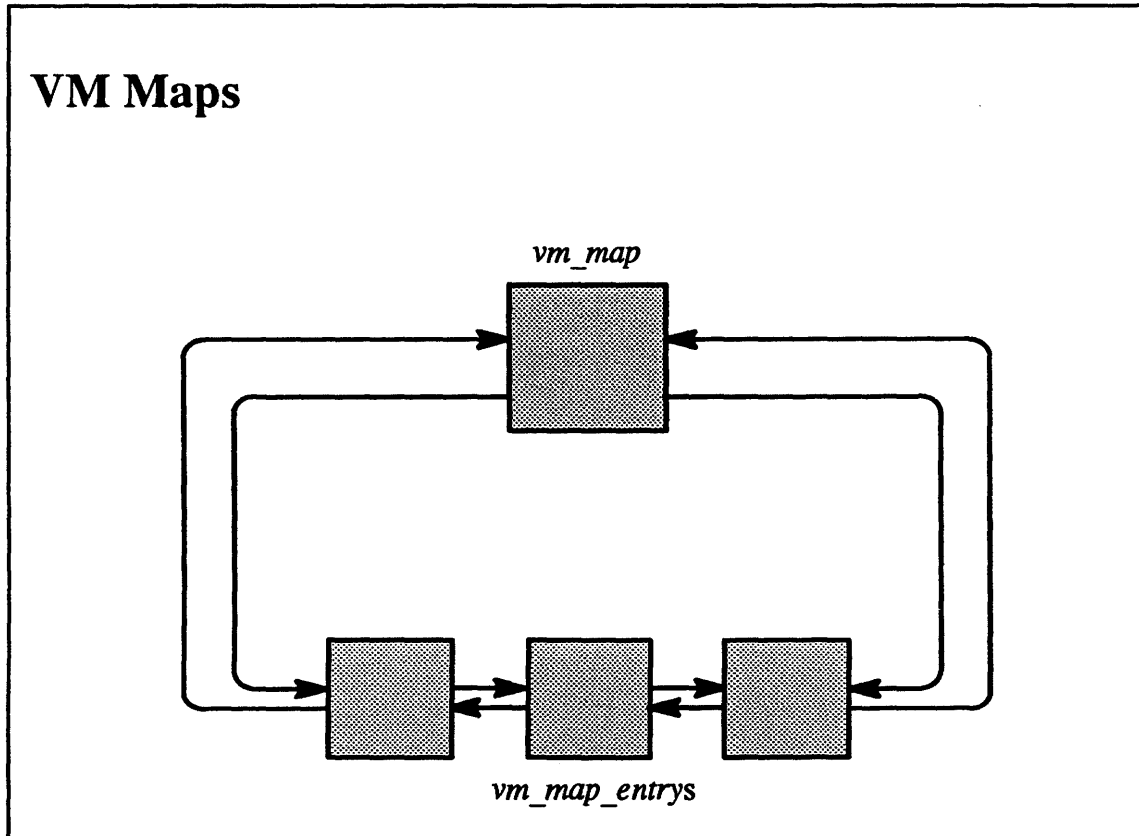
Each task has a map to represent its address space, consisting of a header, the *vm_map* structure, and a linked list of structures, each a *vm_map_entry* representing a continuous range of addresses.

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.

Module 4 — Virtual Memory

4-6. VM Components



4-6.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-7. VM Components

vm_map_entry

wired count	inheritance	
maximum protection	current protection	
object type	offset	
object	flags	
previous entry	start address	next entry
	end address	

vm_map_entry

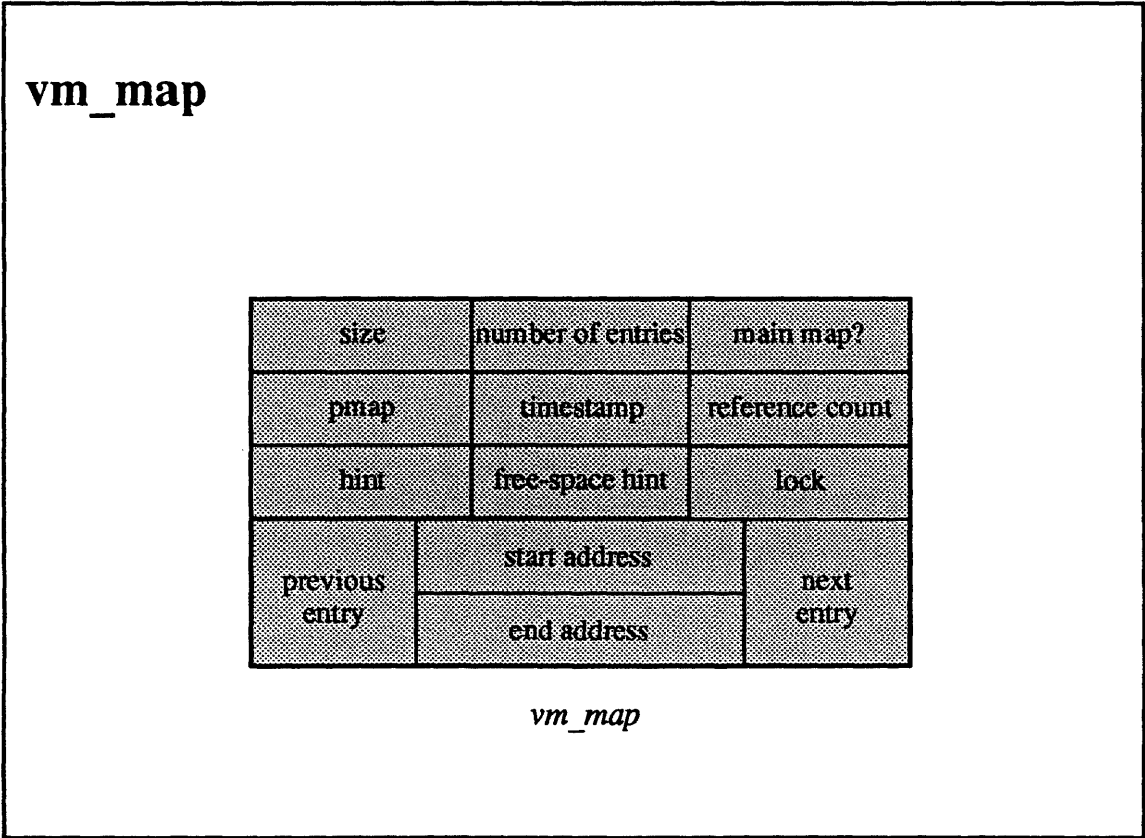
Module 4 — Virtual Memory

Student Notes: `vm_map_entry`

- Previous entry, next entry:
 - links in chain of `vm_map_entries`
- 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)
- 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)

Module 4 — Virtual Memory

4-8. VM Components



4-8.

© 1990, 1991 Open Software Foundation

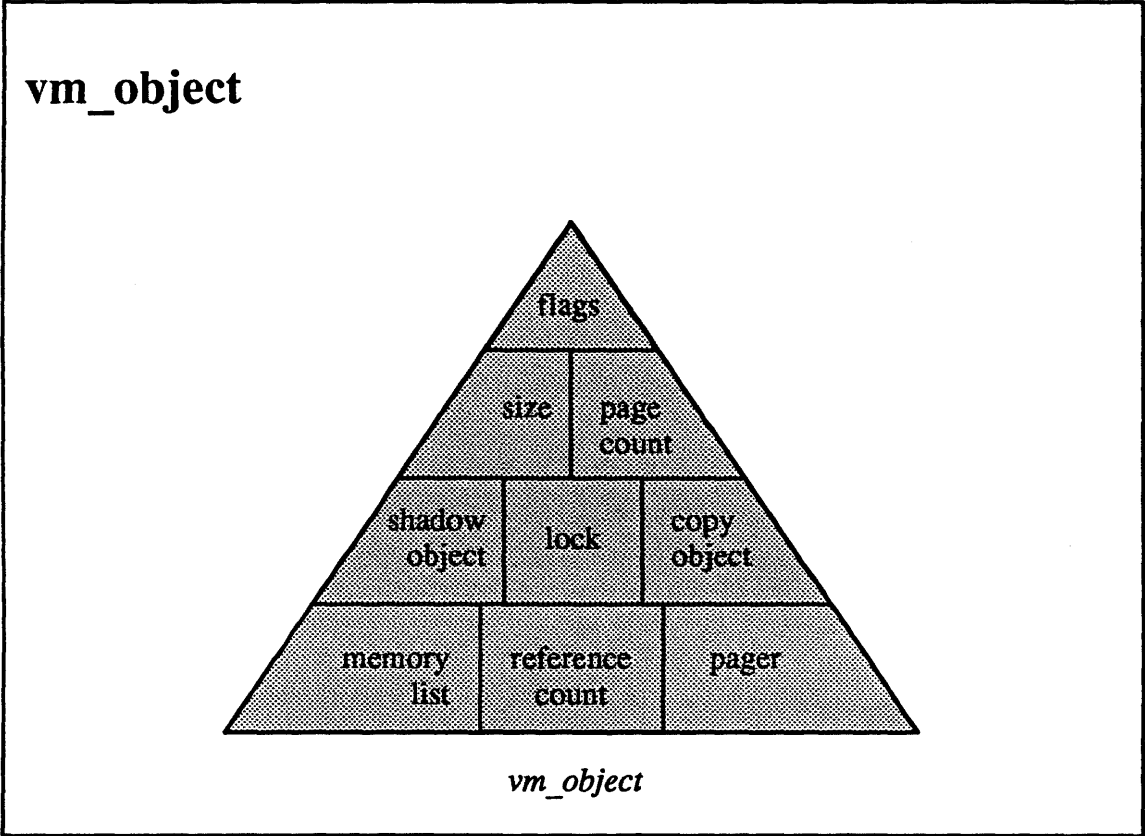
Module 4 — Virtual Memory

Student Notes: `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 *where?* (a good place to start for the next lookup)
- Free-space hint:
 - pointer to the first hole in the address space

Module 4 — Virtual Memory

4-9. VM Components



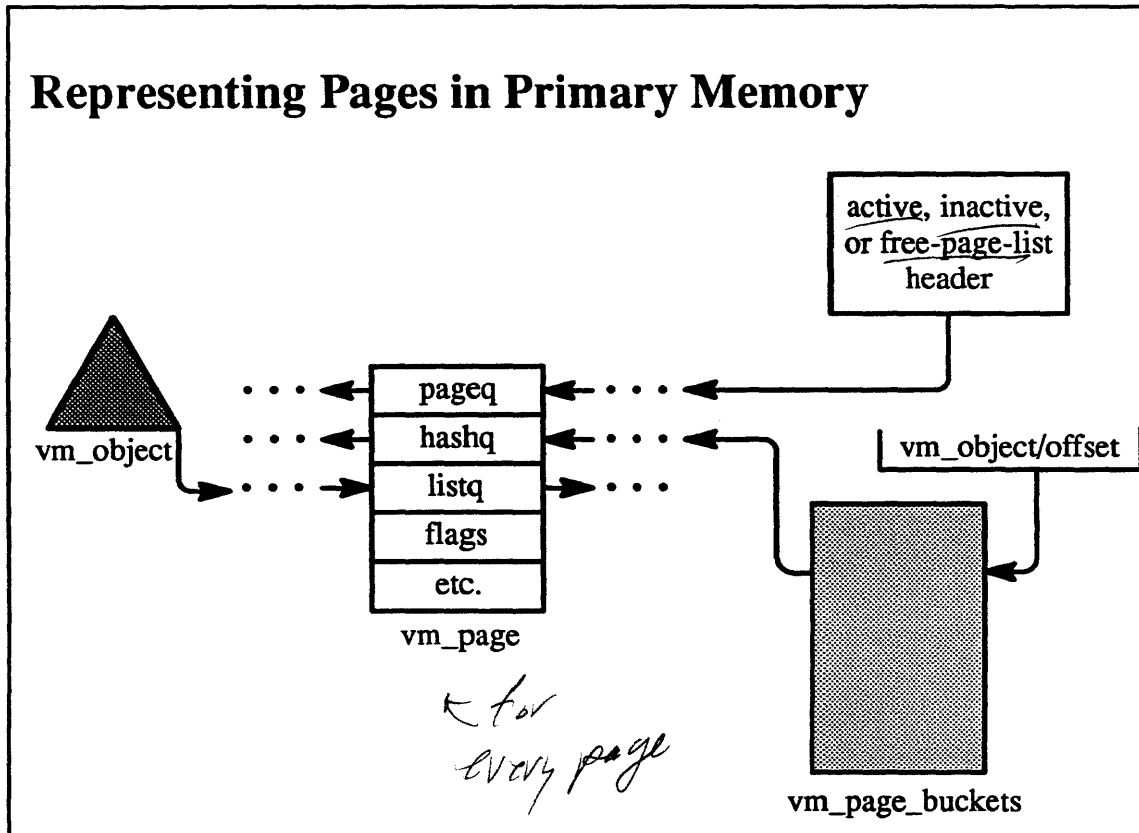
Module 4 — Virtual Memory

Student Notes: 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

Module 4 — Virtual Memory

4-10. VM Components



4-10.

© 1990, 1991 Open Software Foundation

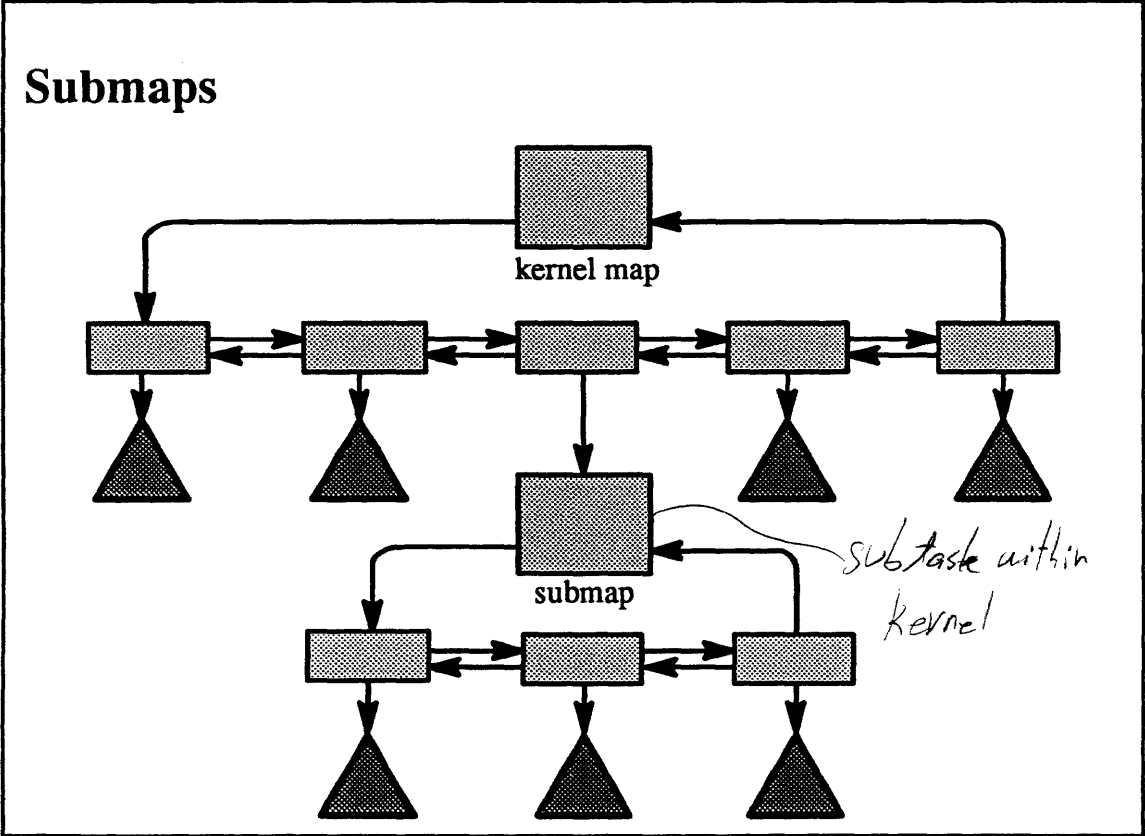
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-11. VM Components



4-11.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

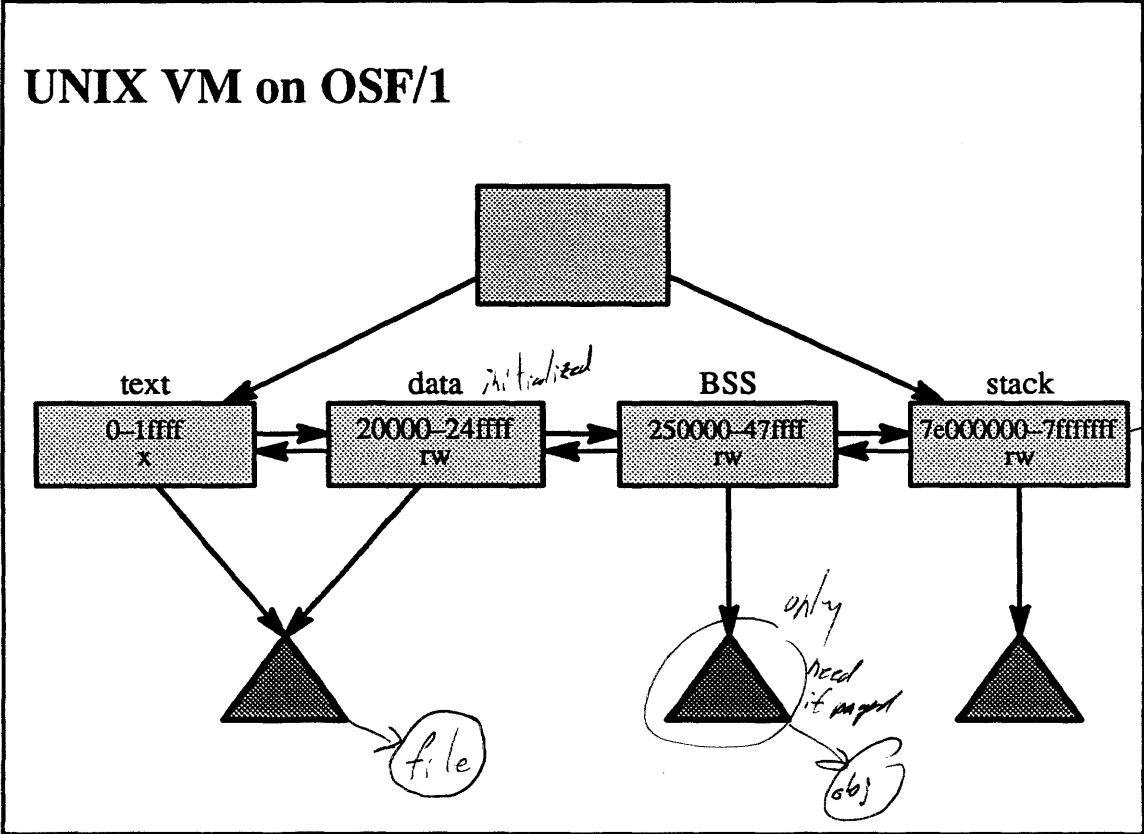
Student Notes: Submaps

Since the list of *vm_map_entries* 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.

Module 4 — Virtual Memory

4-12. VM Components



Module 4 — Virtual Memory

Student Notes: UNIX VM on OSF/1

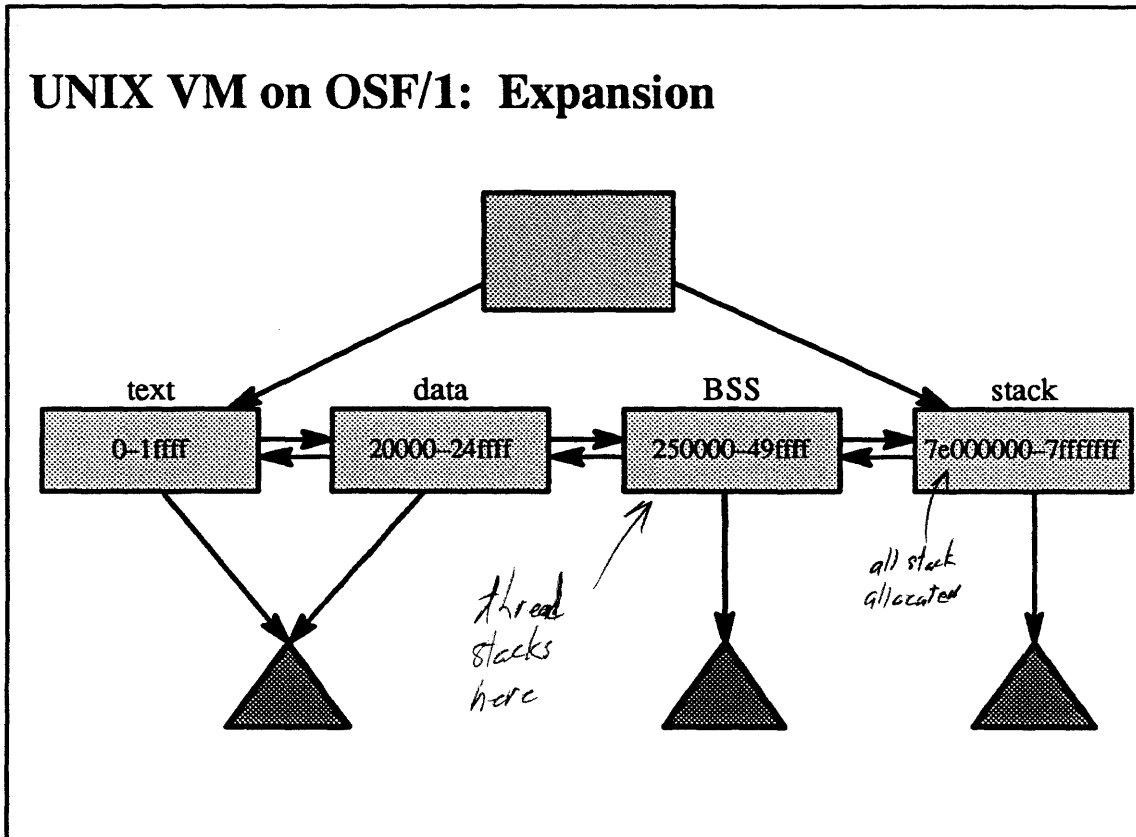
This picture shows how the address space is initially set up for a UNIX process. A *vm_object* for an executable file is mapped into both the text and the data sections. The executable portion of the file is mapped in the text region, and then the initialized data from the file is mapped on the data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

Module 4 — Virtual Memory

4-13. VM Components



4-13.

© 1990, 1991 Open Software Foundation

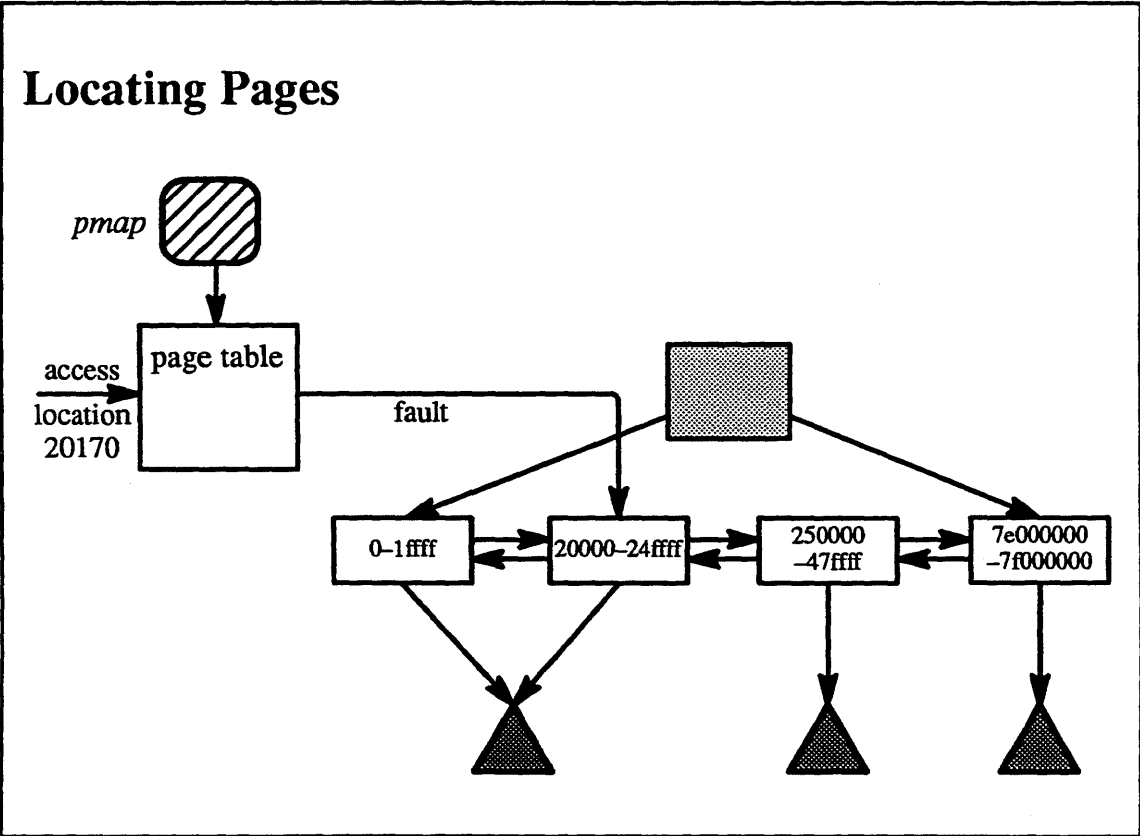
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-14. VM Components



4-14.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: Locating Pages

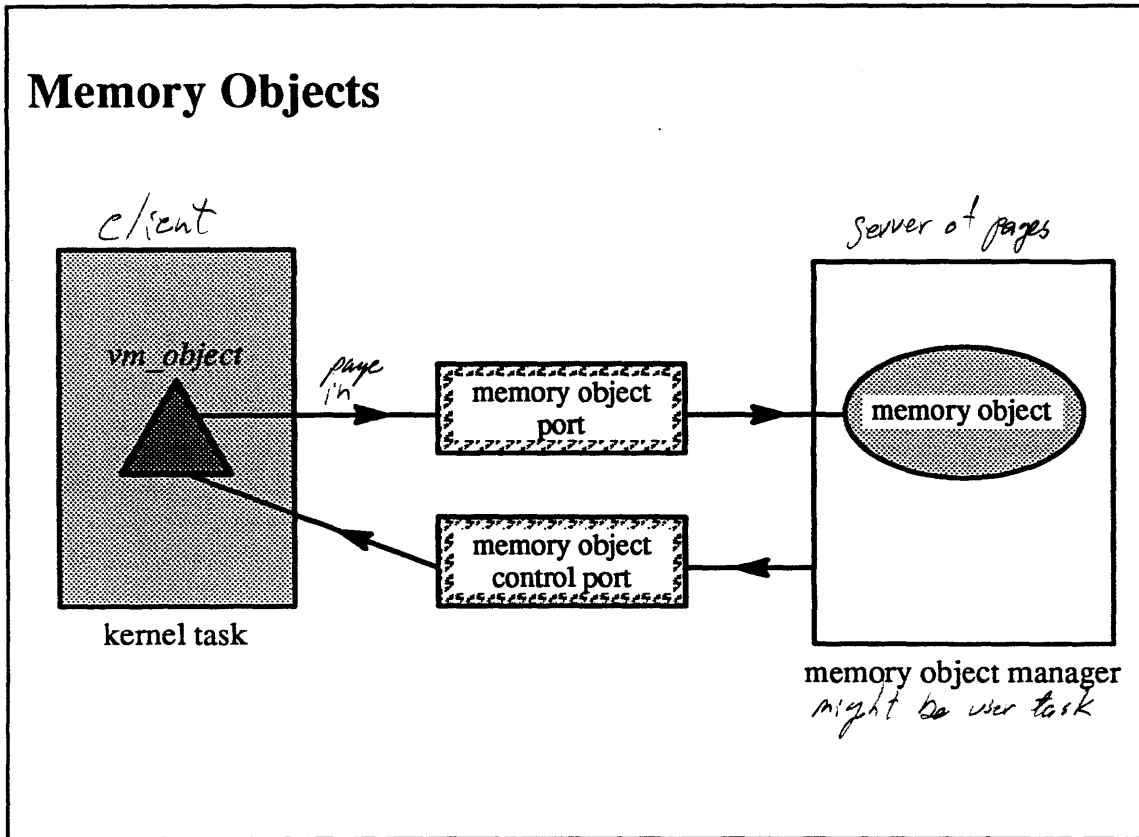
A page fault occurs when a page is referenced that is not mapped by the hardware. The page-fault handler must determine if this is a legitimate reference; if so, it must allocate primary storage for the page and put data into the real memory.

1. The page-fault handler first scans the list of *vm_map_entries* 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*.

Module 4 — Virtual Memory

4-15. Memory Objects

Page In/Out



4-15.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: 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 *paggers*) 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

Module 4 — Virtual Memory

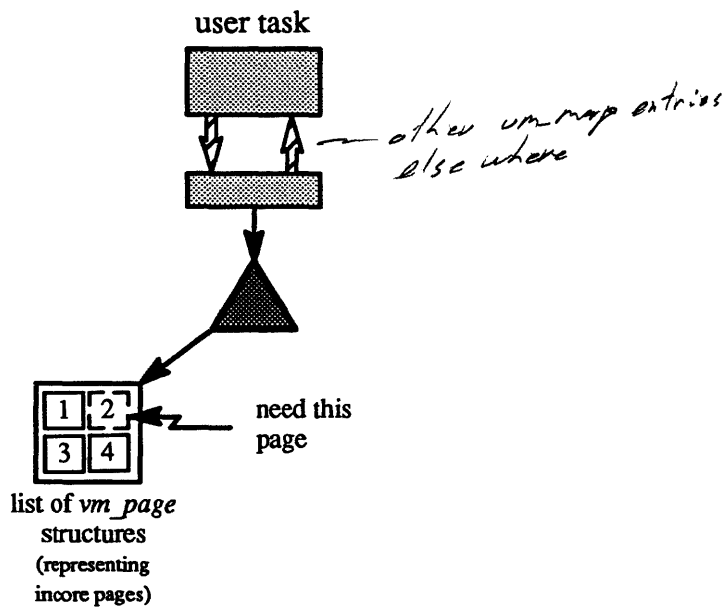
Student Notes: Memory Object Management: Interfaces

- 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) *look ahead*
 - 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

Module 4 — Virtual Memory

4-17. Memory Objects

Pagein, part 1



4-17.

© 1990, 1991 Open Software Foundation

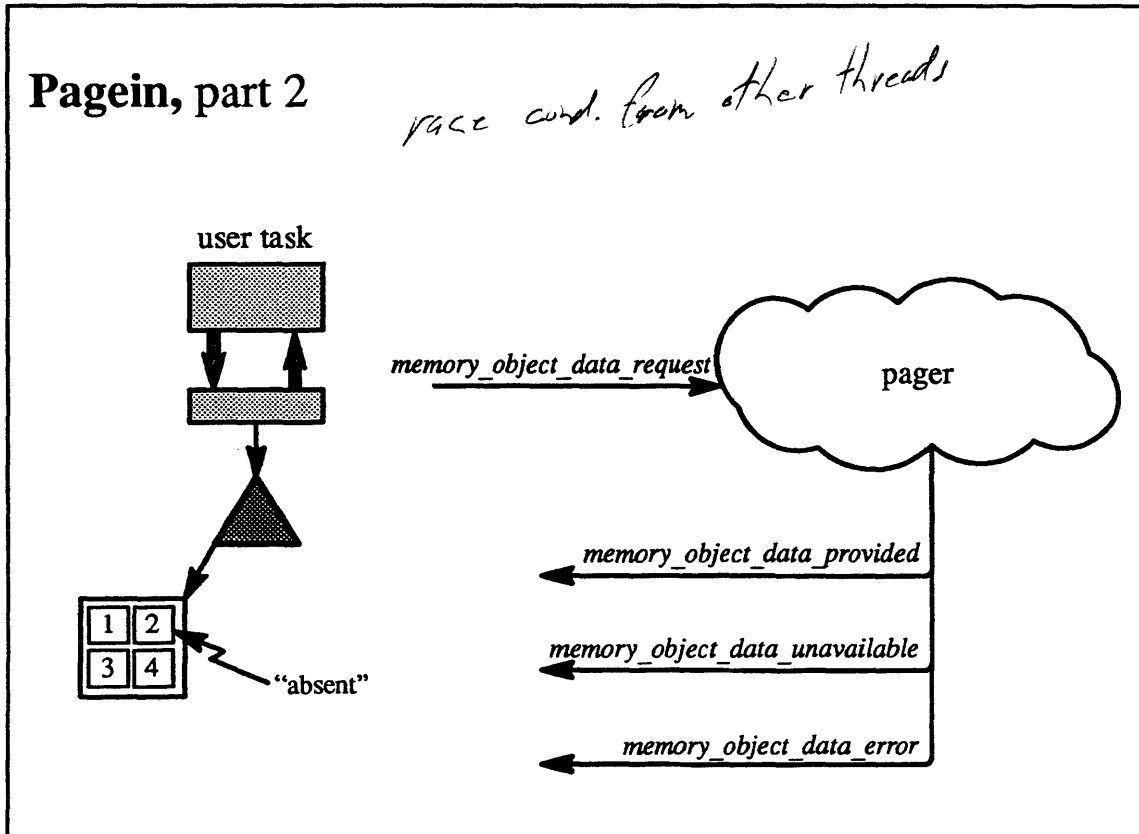
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-18. Memory Objects



4-18.

© 1990, 1991 Open Software Foundation

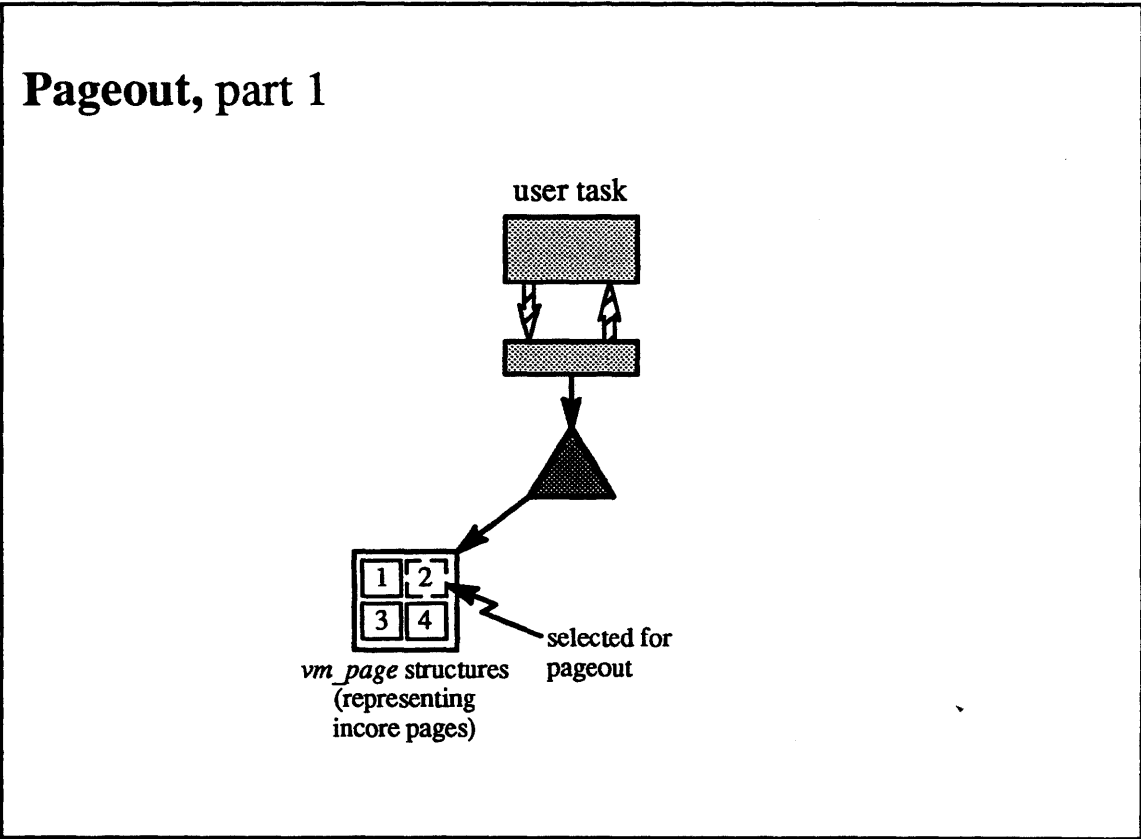
Module 4 — Virtual Memory

Student Notes: 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



Module 4 — Virtual Memory

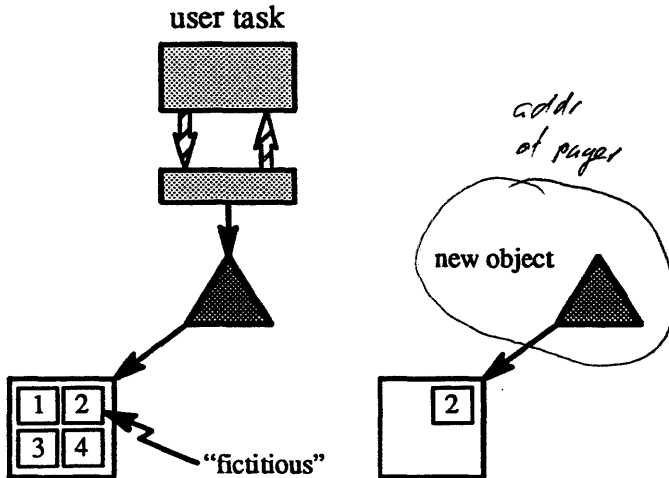
Student Notes: Pageout, part 1

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.

Module 4 — Virtual Memory

4-20. Memory Objects

Pageout, part 2



4-20.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: Pageout, part 2

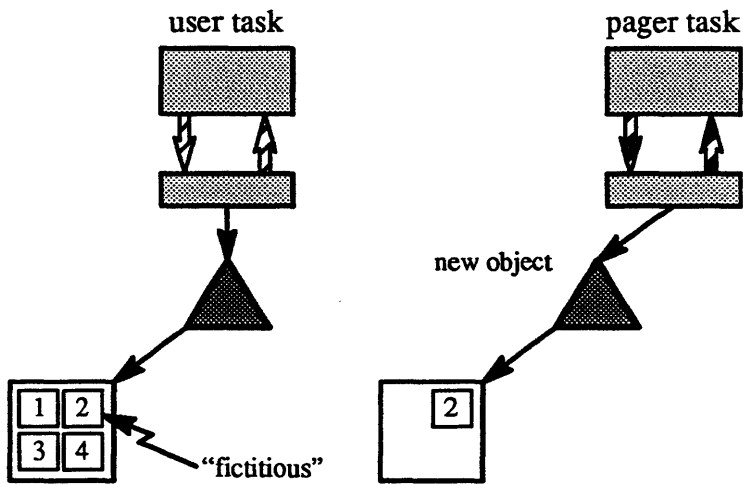
The pageout daemon then:

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*

Module 4 — Virtual Memory

4-21. Memory Objects

Pageout, part 3



4-21.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: 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 *pageins* with *pageouts*: 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*.

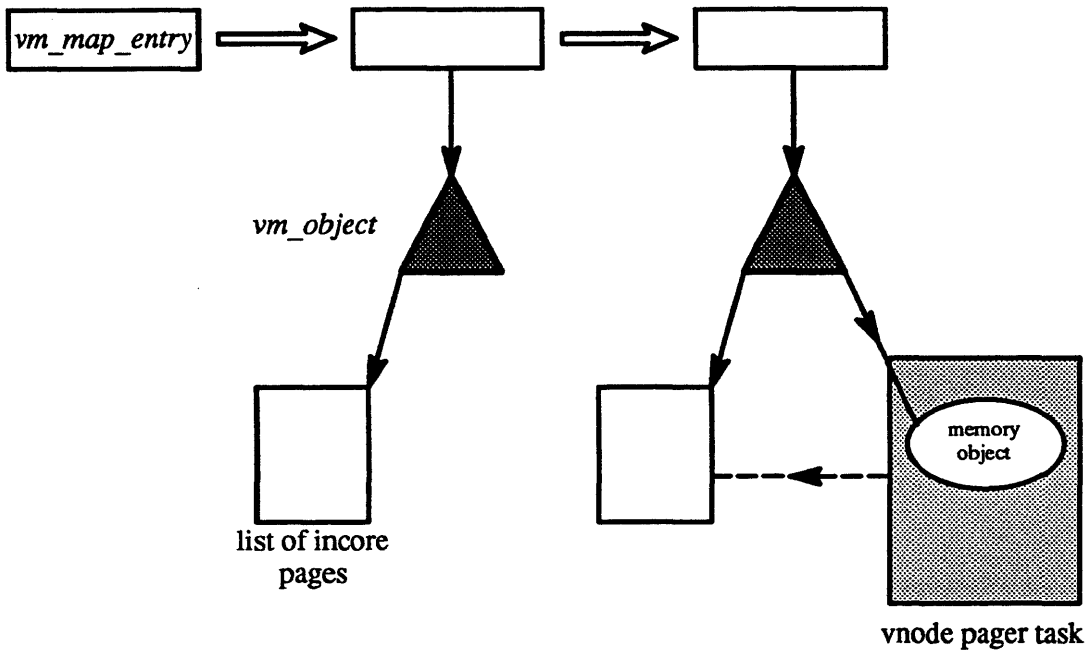
The interface to the *vnode pager* is identical to that of other pagers. However, the *pageout* daemon is assured that the *vnode pager* will always complete a *pageout*. Thus the page being paged out is not returned to the domain of the *pageout* daemon, but instead is “wired,” assuring that the *pageout* 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.

Module 4 — Virtual Memory

4-22. Memory Objects

Lazy Evaluation of Object Creation



4-22.

© 1990, 1991 Open Software Foundation

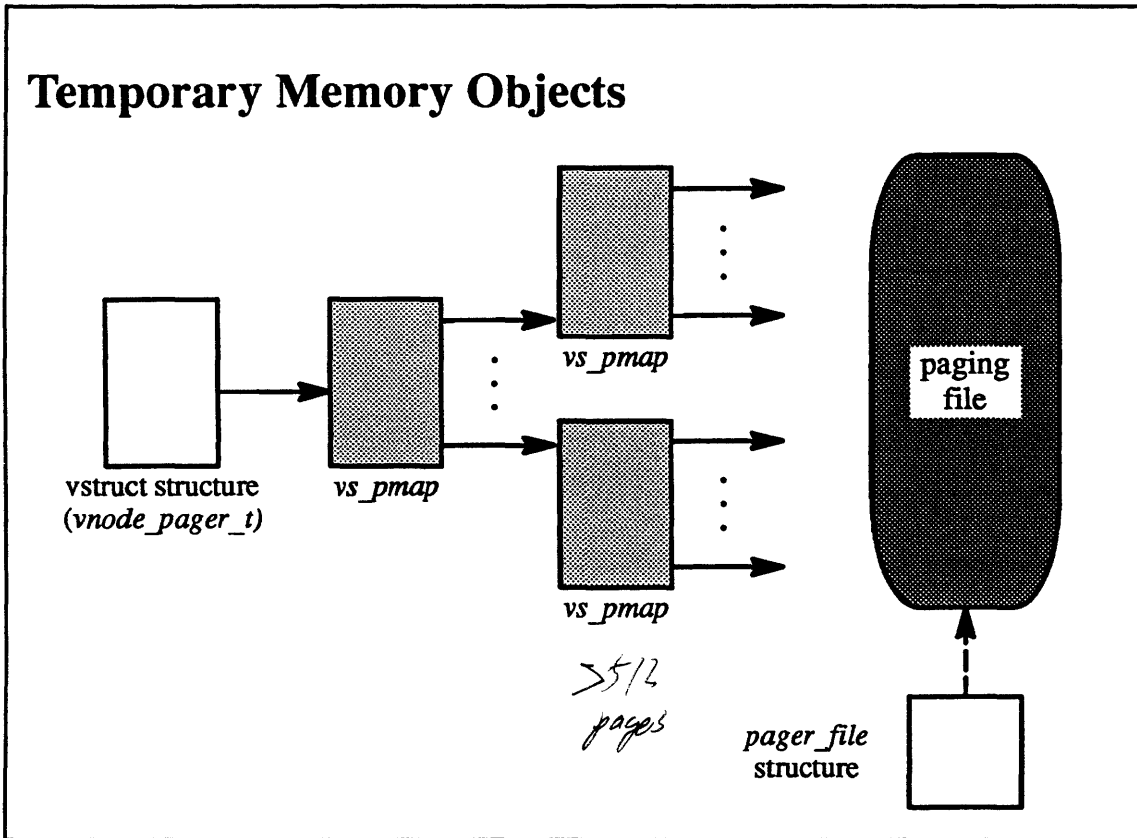
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-23. Memory Objects



4-23.

© 1990, 1991 Open Software Foundation

*file
or raw device*

Module 4 — Virtual Memory

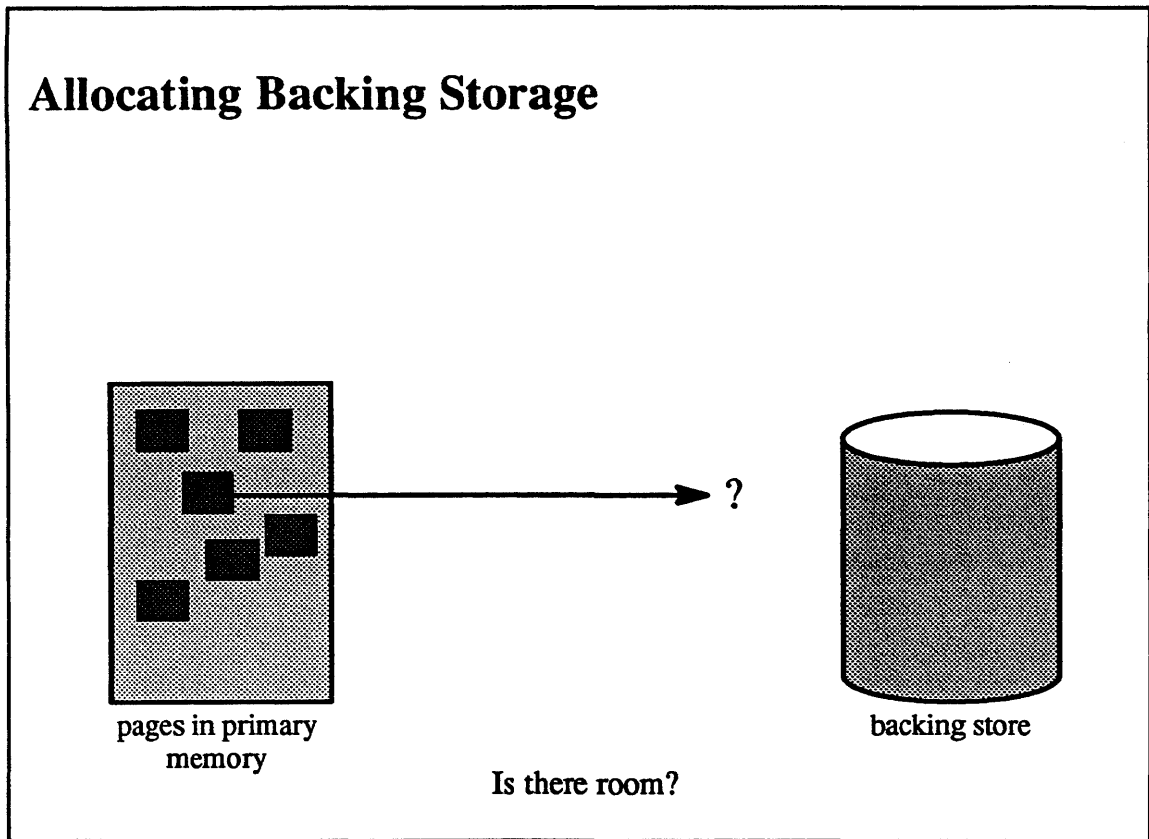
Student Notes: Temporary Memory Objects

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_pmaps* (and space in the paging files) are allocated only when necessary to back up a page.

4-24. Memory Objects



Module 4 — Virtual Memory

Student Notes: Allocating Backing Storage

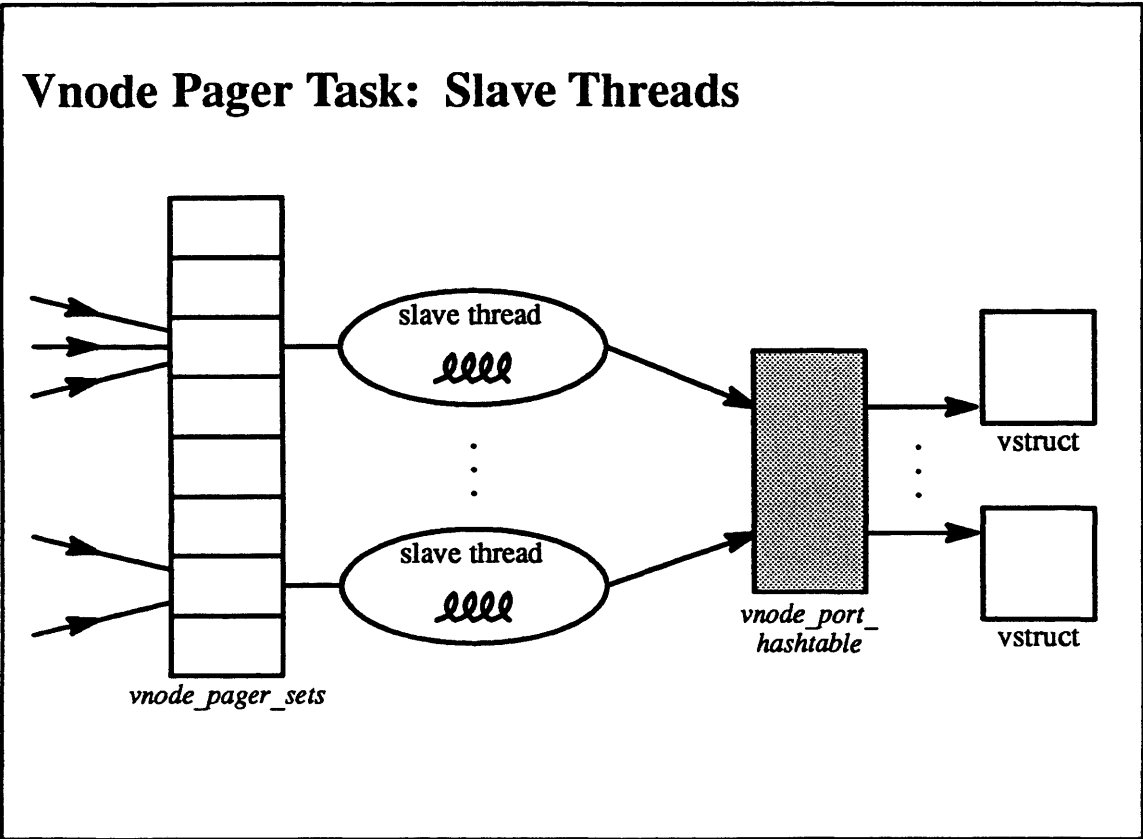
OSF/1 takes a liberal approach to the allocation of the backing store: backing store is allocated only when necessary, i.e., when a page must be written out. This approach differs from the conservative approach used in earlier versions of UNIX, in which backing store and virtual memory are allocated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

Module 4 — Virtual Memory

4-25. Memory Objects



Module 4 — Virtual Memory

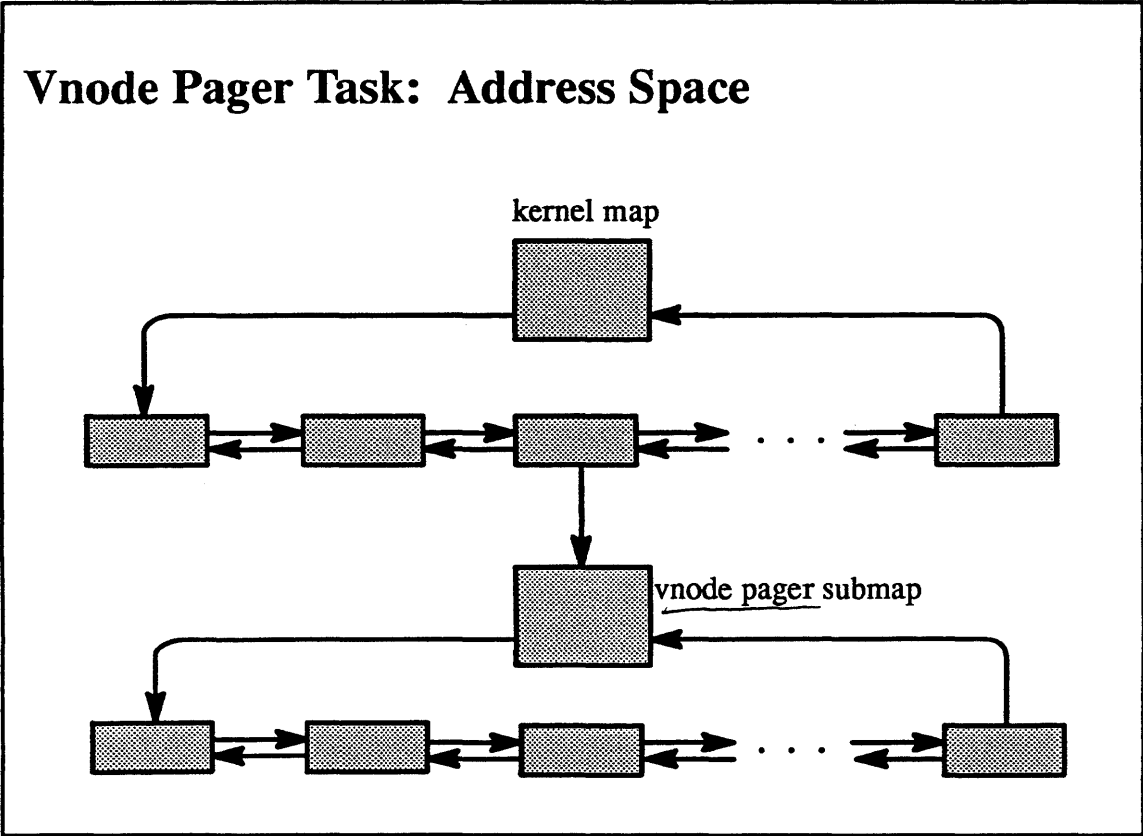
Student Notes: Vnode Pager Task: Slave Threads

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 choosing 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*.

Module 4 — Virtual Memory

4-26. Memory Objects



4-26.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: Vnode Pager Task: Address Space

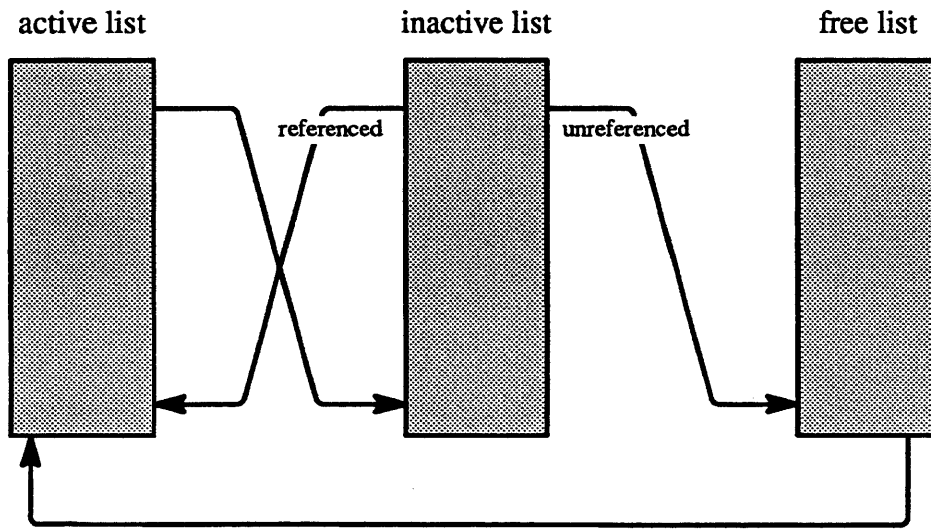
The vnode pager is implemented as a very special task. It has a *task* structure, a *u_task* structure, and contains threads, but its address space is the kernel address space. It has access to all of the kernel address space, though its private data structures are segregated within a special submap.

Module 4 — Virtual Memory

4-27. Memory Objects

Page Replacement

fifo order



Module 4 — Virtual Memory

Student Notes: 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

Swapping

Swapout

- *unwire* the kernel stack

Swapin

threads

- *fetch* and *wire* the kernel stack

Module 4 — Virtual Memory

Student Notes: Swapping

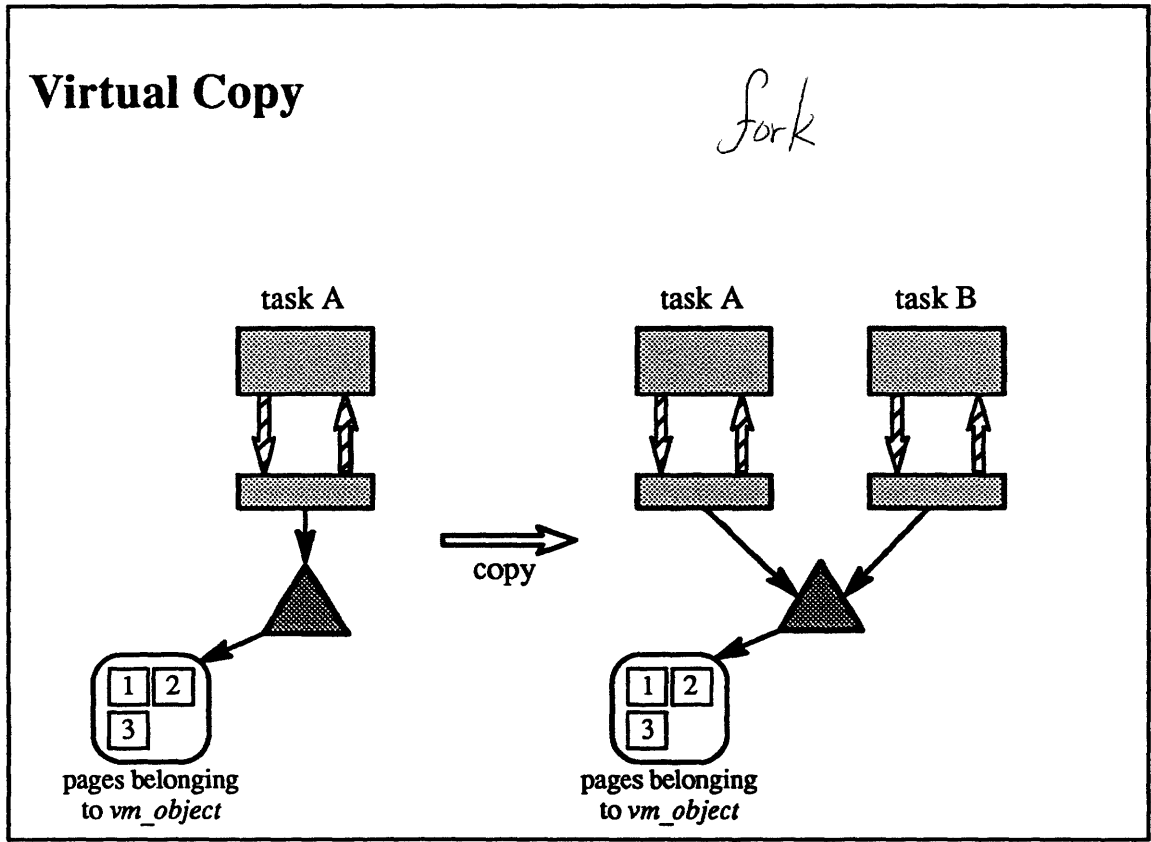
Swapping is handled by two kernel threads: a *swapout thread* and a *swapin thread*. The pageout daemon wakes up the swapout thread in response to memory shortages (but no more than once a minute). The swapout thread scans the list of all threads and swaps out those nonrunnable, interruptible threads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

Module 4 — Virtual Memory

4-29. Copying and Sharing



4-29.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

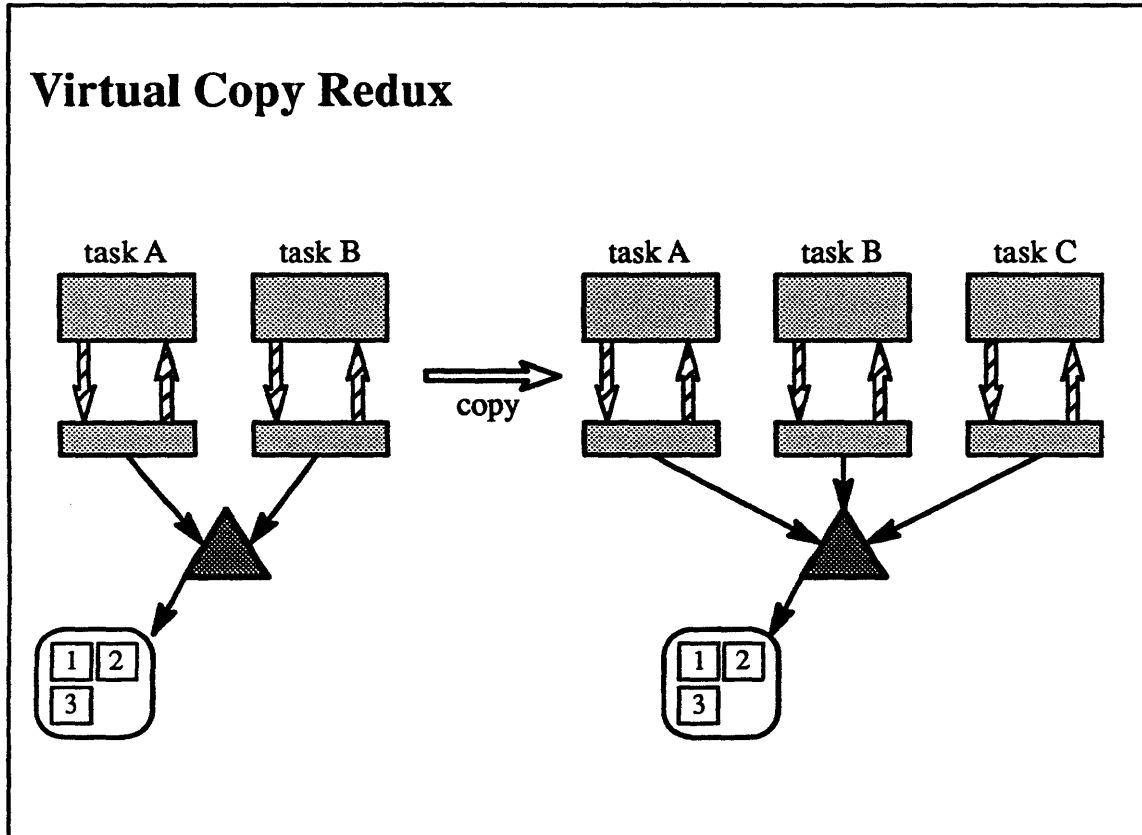
Student Notes: Virtual Copy

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.

Module 4 — Virtual Memory

4-30. Copying and Sharing



4-30.

© 1990, 1991 Open Software Foundation

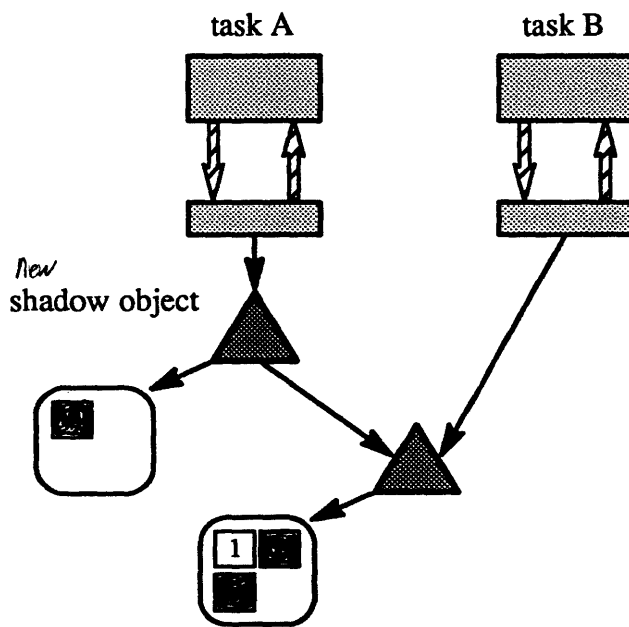
Module 4 — Virtual Memory

Student Notes: Virtual Copy Redux

Suppose that task B from page 4-58 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



Module 4 — Virtual Memory

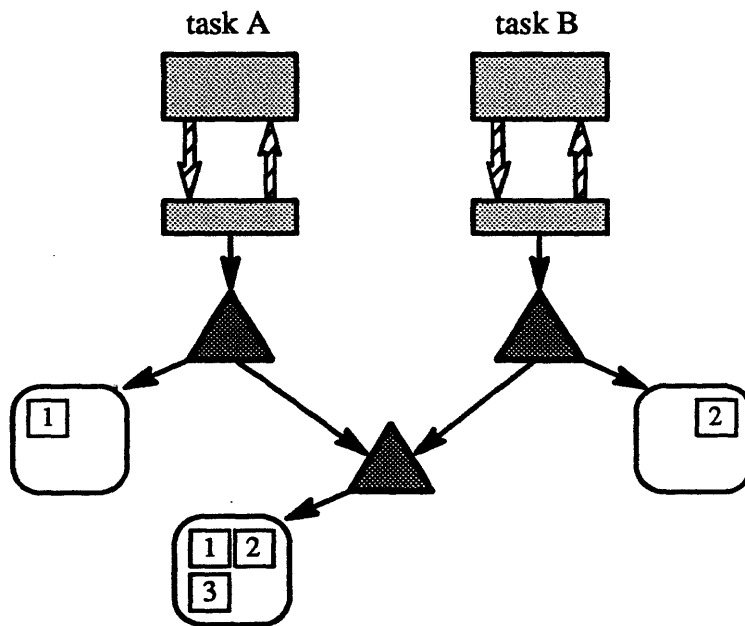
Student Notes: Virtual Copy and Modified Pages, part 1

This picture shows the situation of page 4-58 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



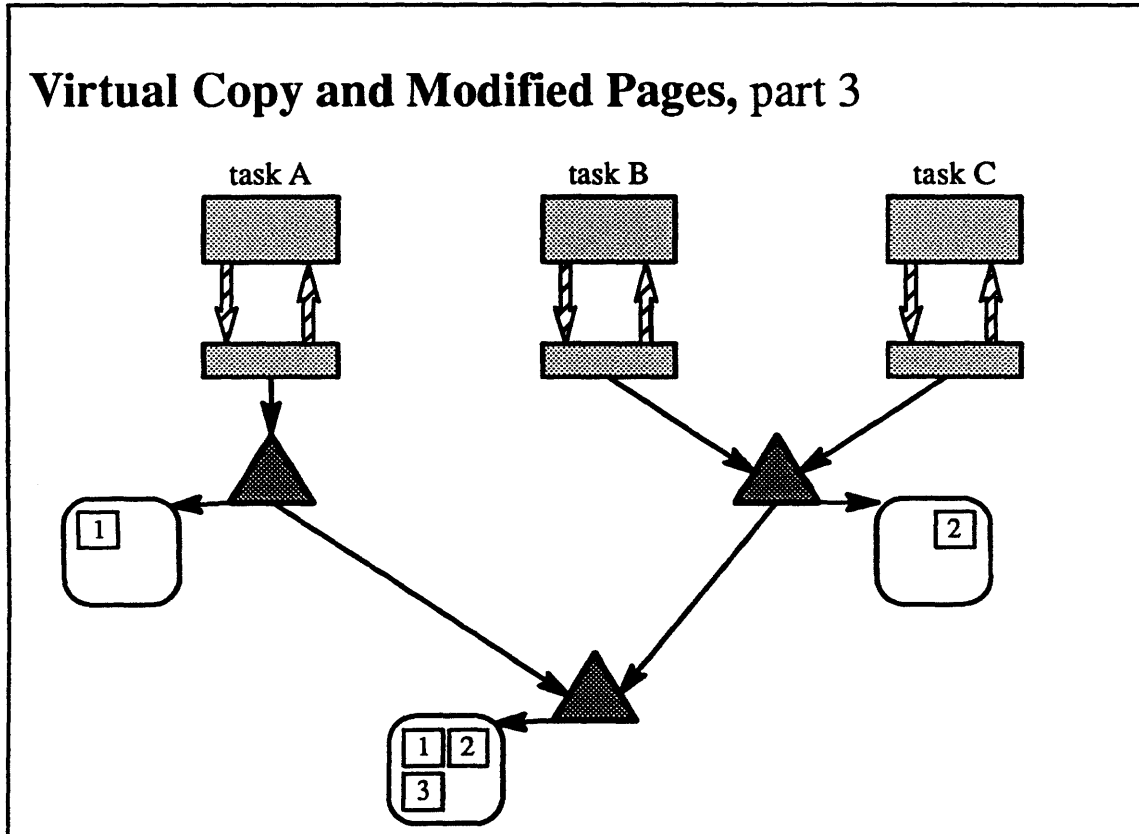
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-33. Copying and Sharing



4-33.

© 1990, 1991 Open Software Foundation

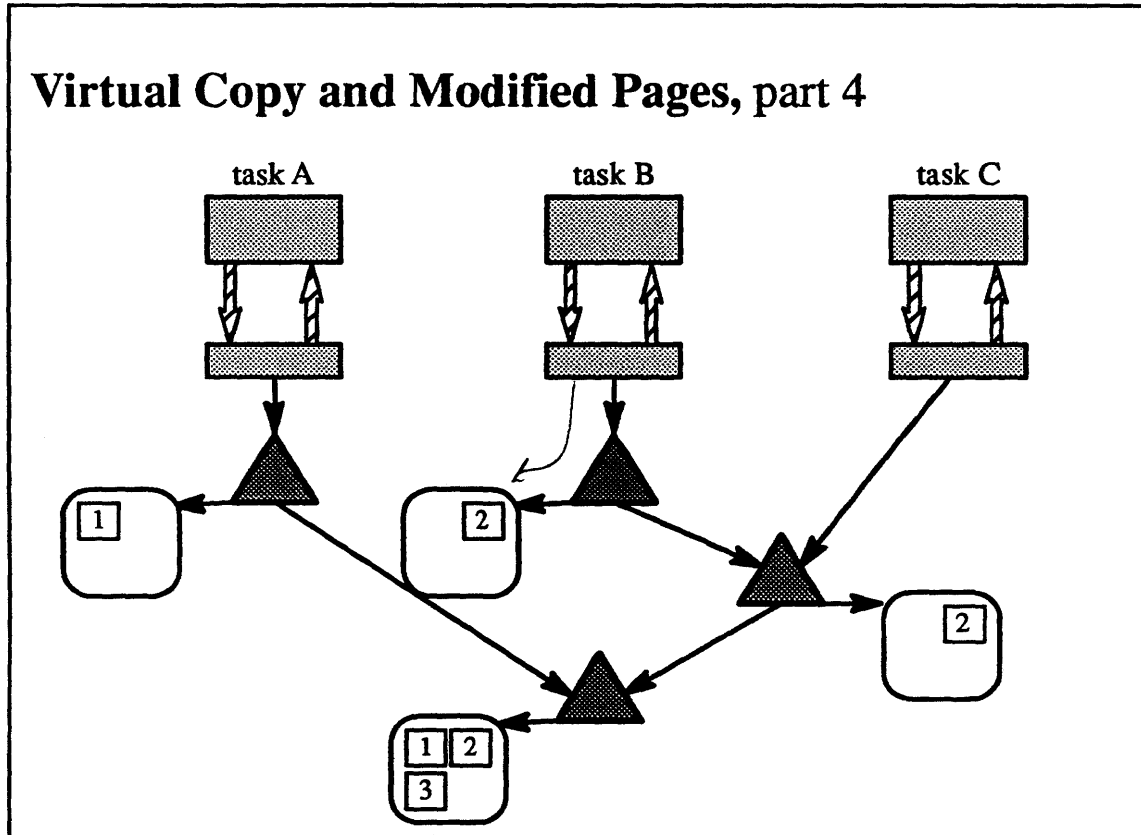
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-34. Copying and Sharing



4-34.

© 1990, 1991 Open Software Foundation

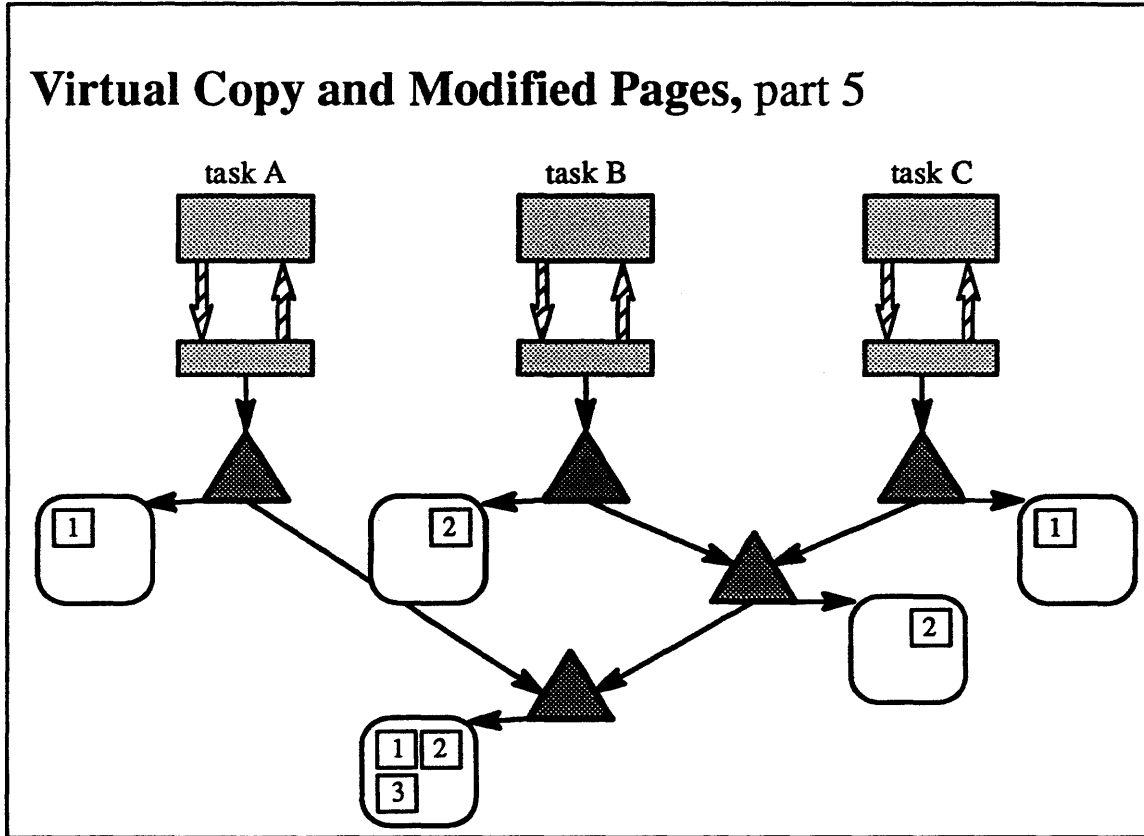
Module 4 — Virtual Memory

Student Notes: 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



4-35.

© 1990, 1991 Open Software Foundation

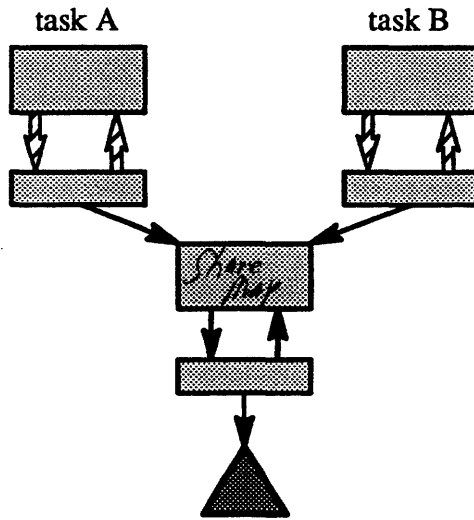
Module 4 — Virtual Memory

Student Notes: 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.

4-36. Copying and Sharing

Sharing



Module 4 — Virtual Memory

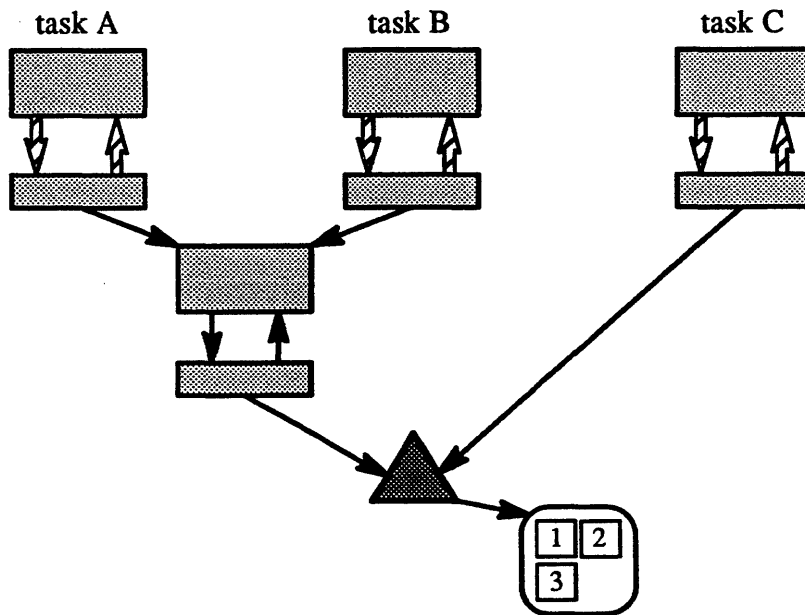
Student Notes: Sharing

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_entries*, each of which points to a *vm_object*. The *vm_map_entries* of the tasks sharing this memory point to the share map (not to *vm_objects*).

Module 4 — Virtual Memory

4-37. Copying and Sharing

Share Then Copy, part 1



4-37.

© 1990, 1991 Open Software Foundation

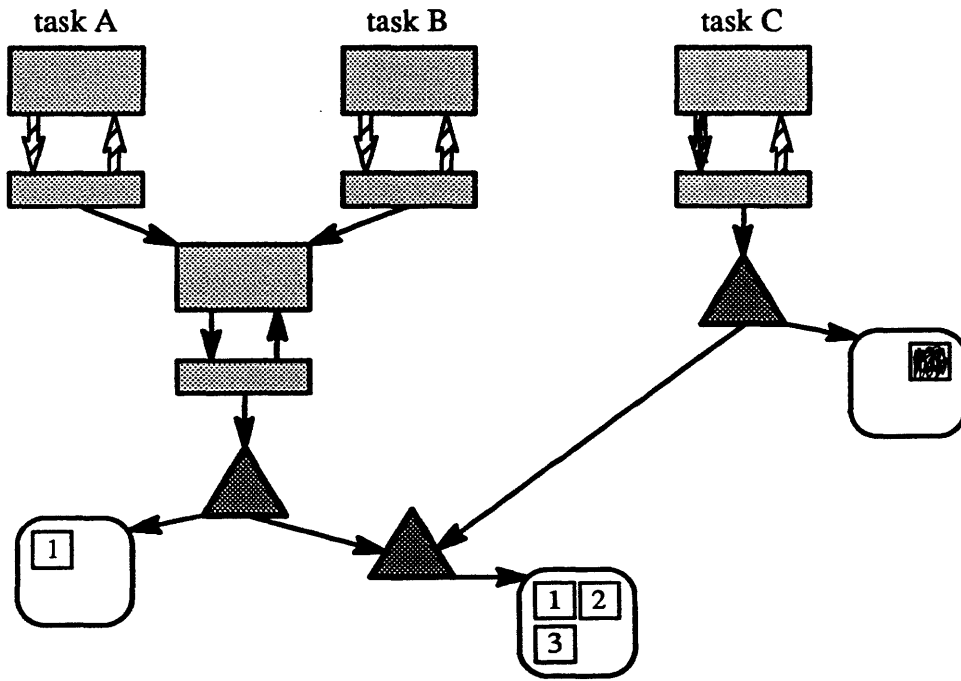
Module 4 — Virtual Memory

Student Notes: 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



Module 4 — Virtual Memory

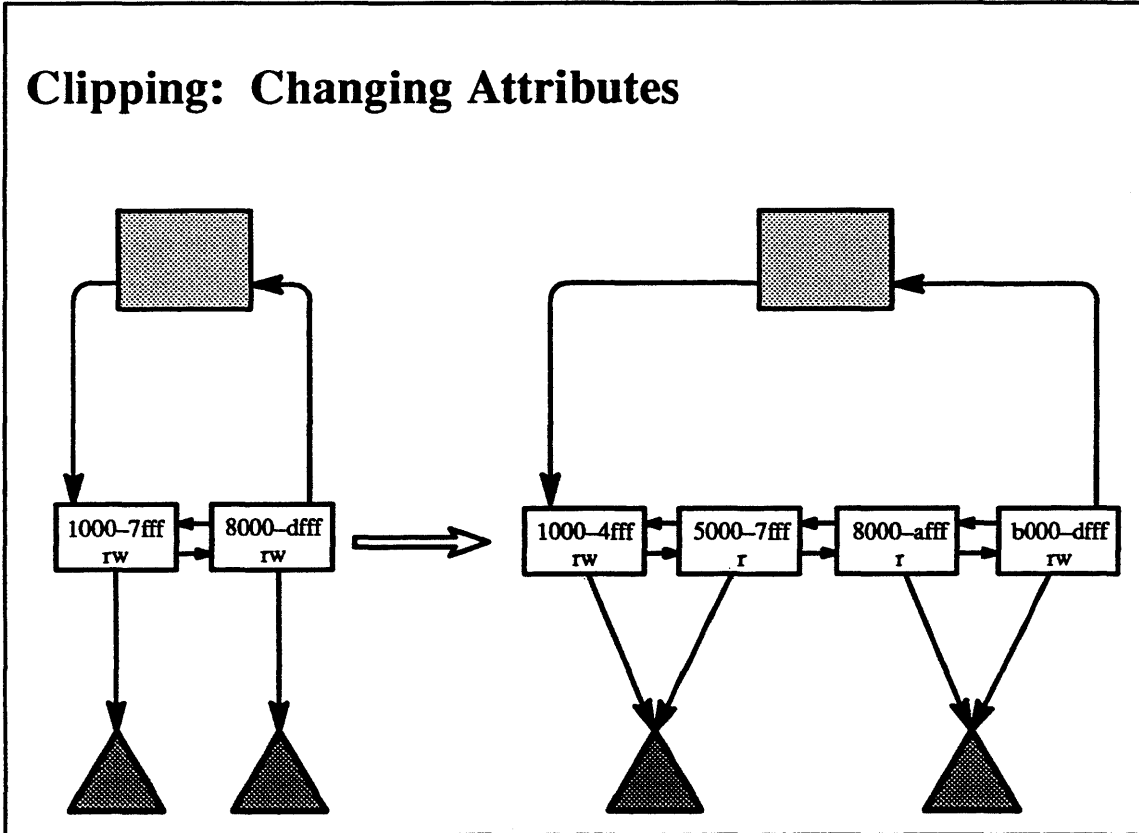
Student Notes: Share Then Copy, part 2

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_entries* 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_entries* 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



Module 4 — Virtual Memory

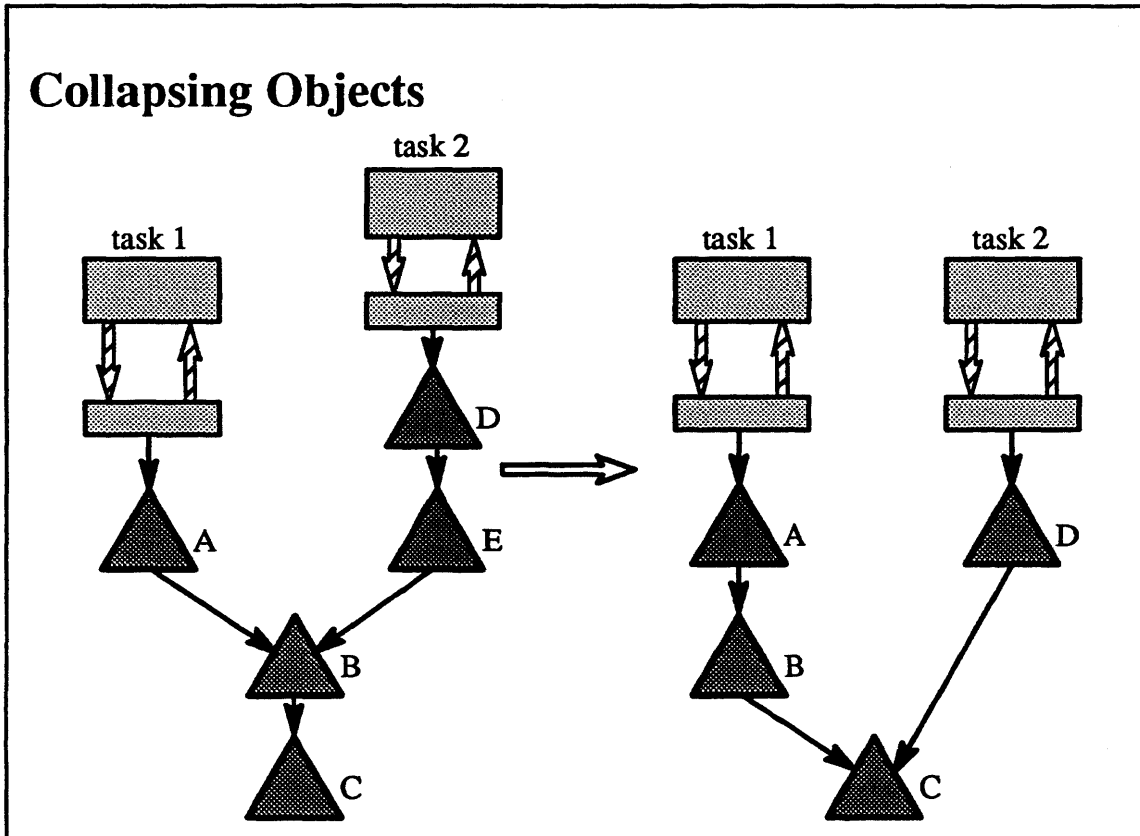
Student Notes: Clipping: Changing Attributes

Programmers do not see the organization of the address space imposed by the *vm_map_entries*. 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_entries* 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_entries*, each of which must be split in two to allow a read-write portion and a read-only portion.

4-40. Copying and Sharing



4-40.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: 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.

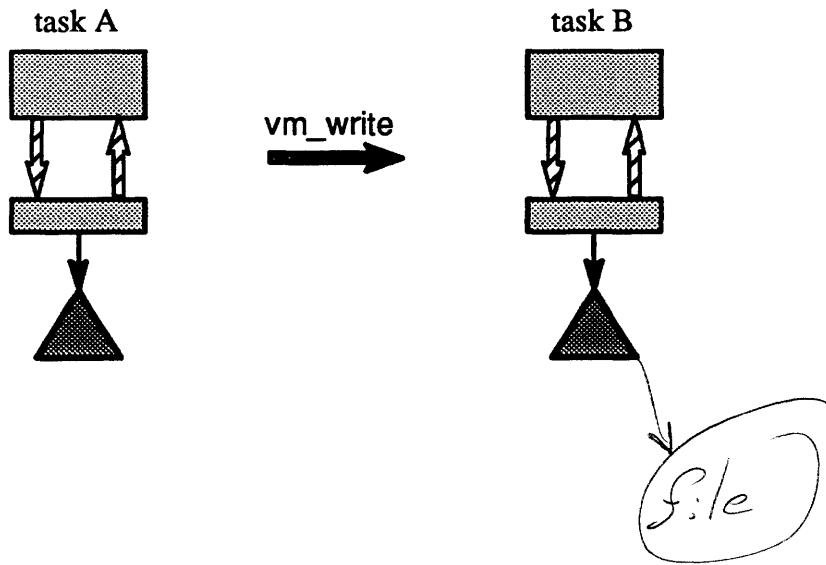
The first rule is that if a *vm_object* is pointed to by only a single *vm_object* via a shadow link, then it is not necessary to have both objects: they may be combined into a single object.

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



Module 4 — Virtual Memory

Student Notes: 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

Module 4 — Virtual Memory

Student Notes: Optimizing the Virtual Copy Operation

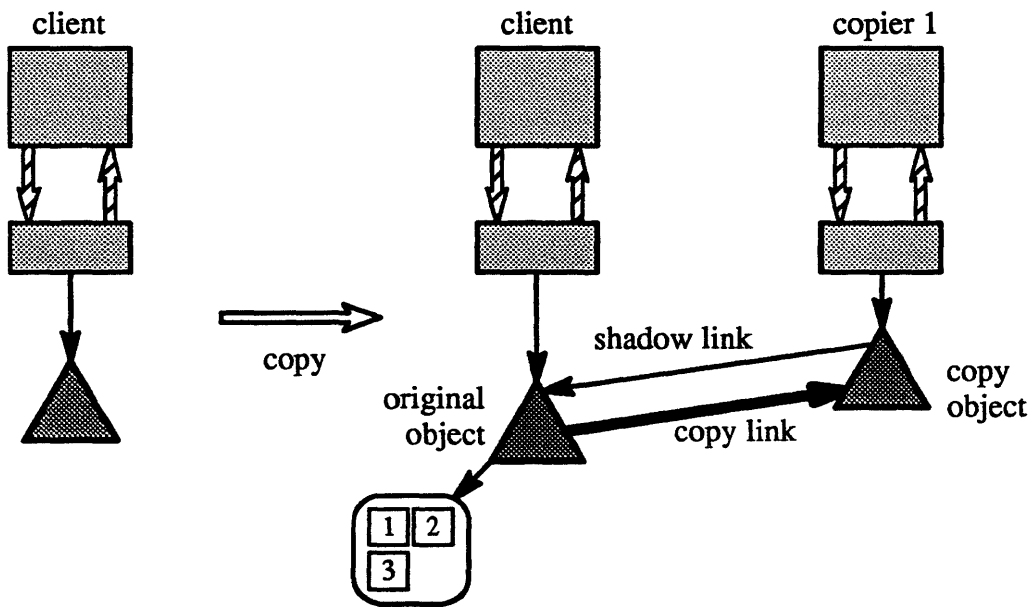
Suppose now that we are making a virtual copy of a portion of an address space into which a permanent object has been mapped; the virtual copy is a temporary object, not a permanent object.

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



Module 4 — Virtual Memory

Student Notes: Virtual Copy from Permanent Objects: COPY_DELAY, part 1

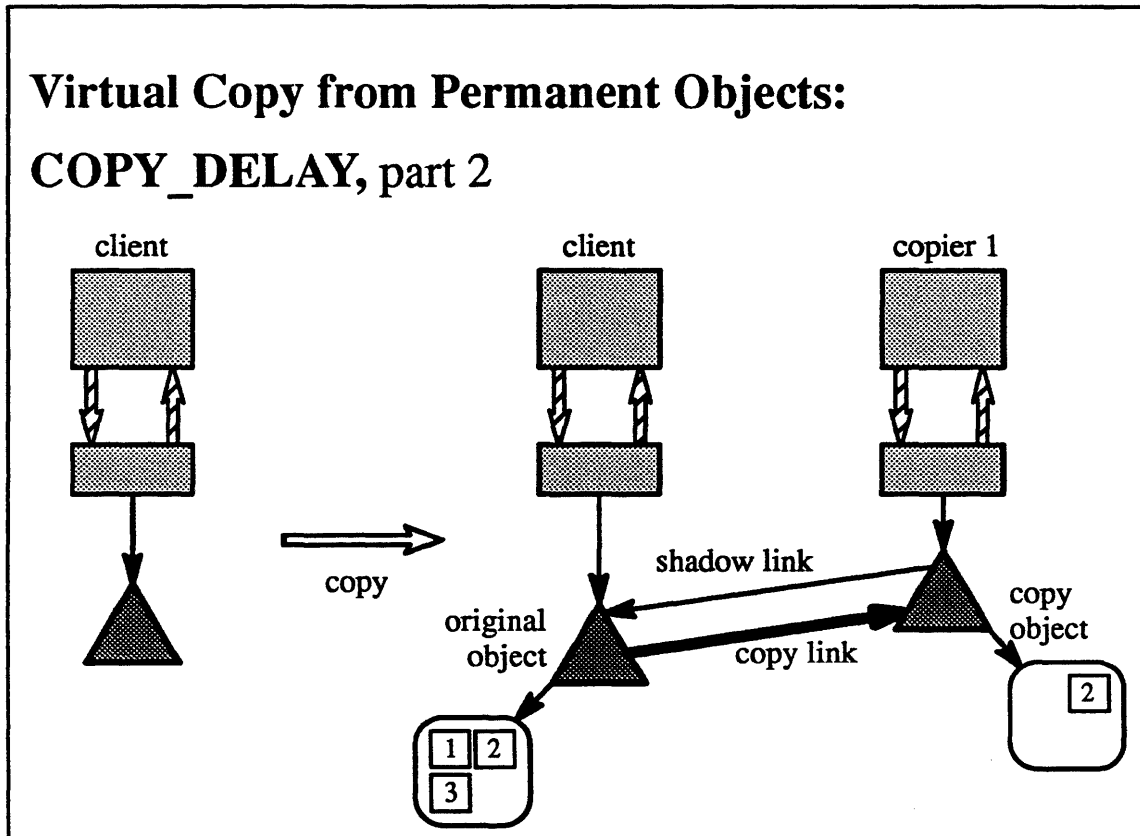
In the simplest case, the memory object manager and all of its clients are on the same machine. Thus the kernel is immediately aware of any change made to the object. The major concern here is to make certain that all changes clients make to the object are reflected in the object itself. The representation 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



Module 4 — Virtual Memory

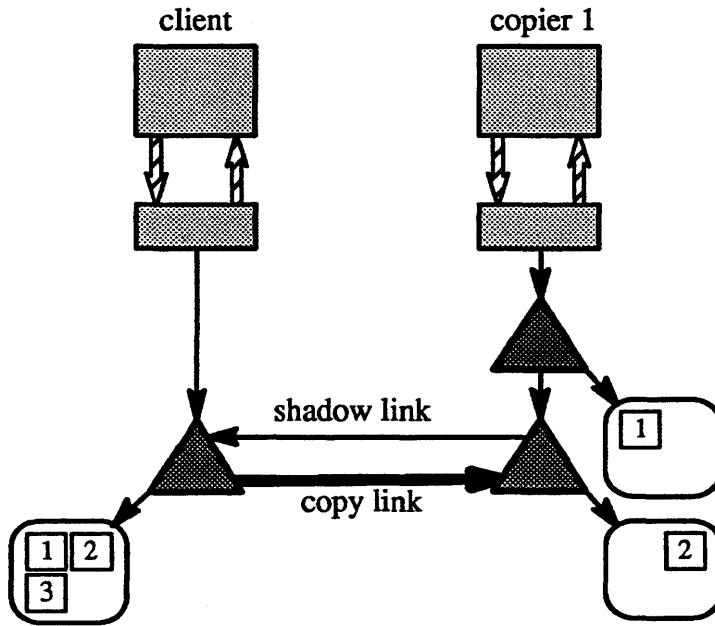
Student Notes: Virtual Copy from Permanent Objects: COPY_DELAY, part 2

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



4-45.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

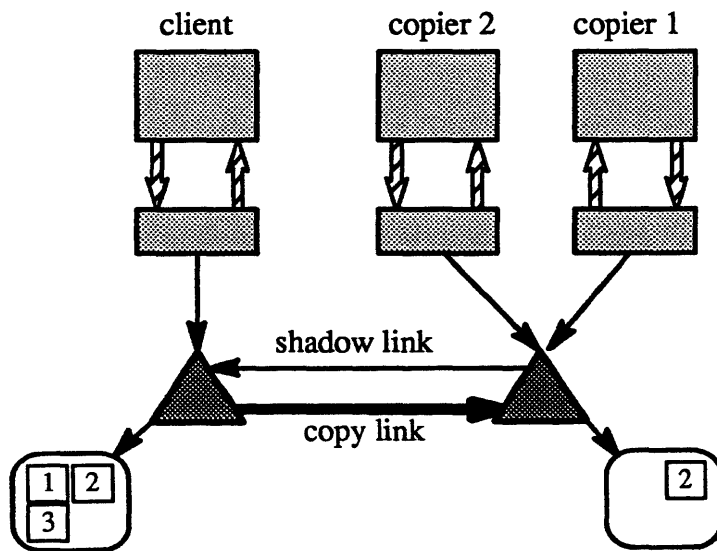
Student Notes: Virtual Copy from Permanent Objects: COPY_DELAY, part 3

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



Module 4 — Virtual Memory

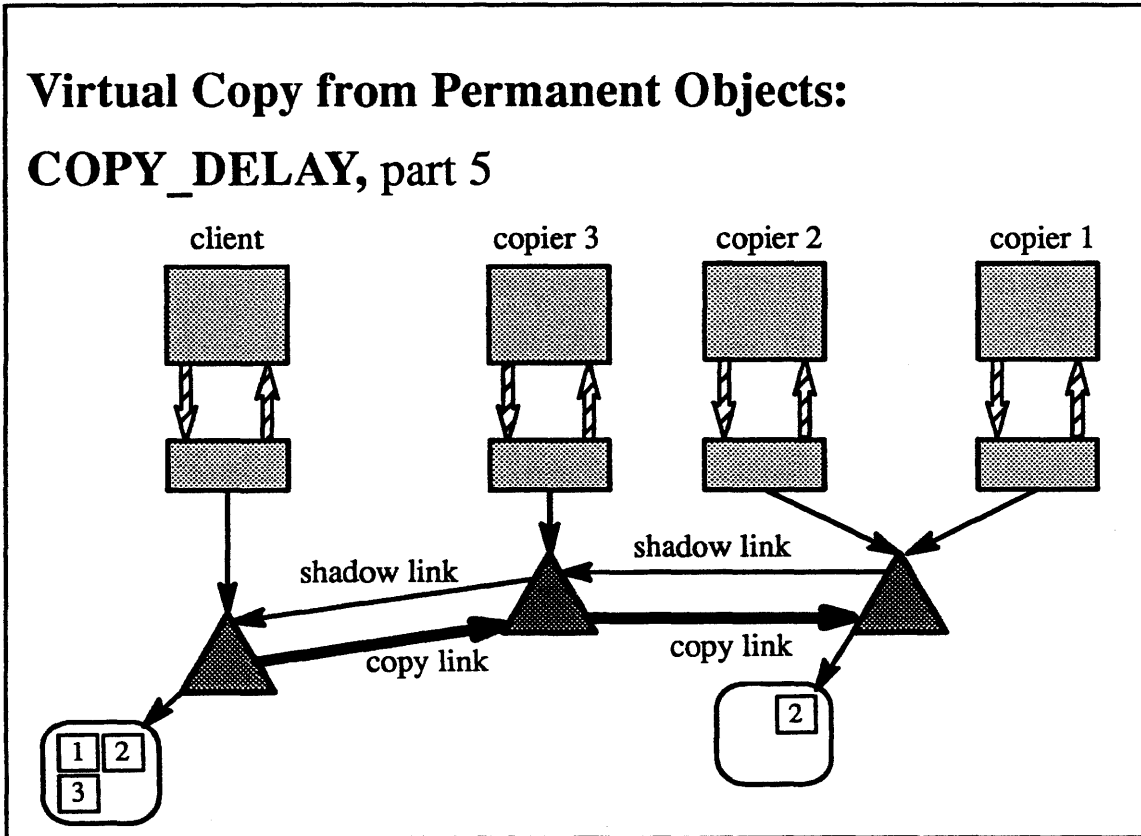
Student Notes: Virtual Copy from Permanent Objects: COPY_DELAY, part 4

The copier of page 4-88 has executed a fork system call, creating copier 2.

4-47. Copying and Sharing

Virtual Copy from Permanent Objects:

COPY_DELAY, part 5



Module 4 — Virtual Memory

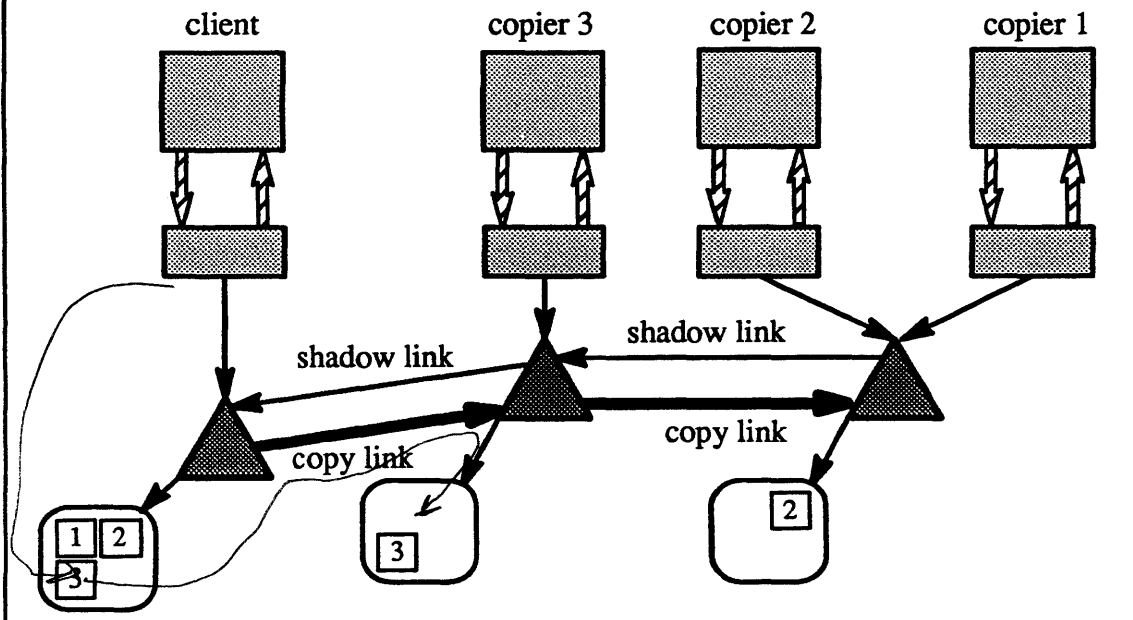
Student Notes: Virtual Copy from Permanent Objects: COPY_DELAY, part 5

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



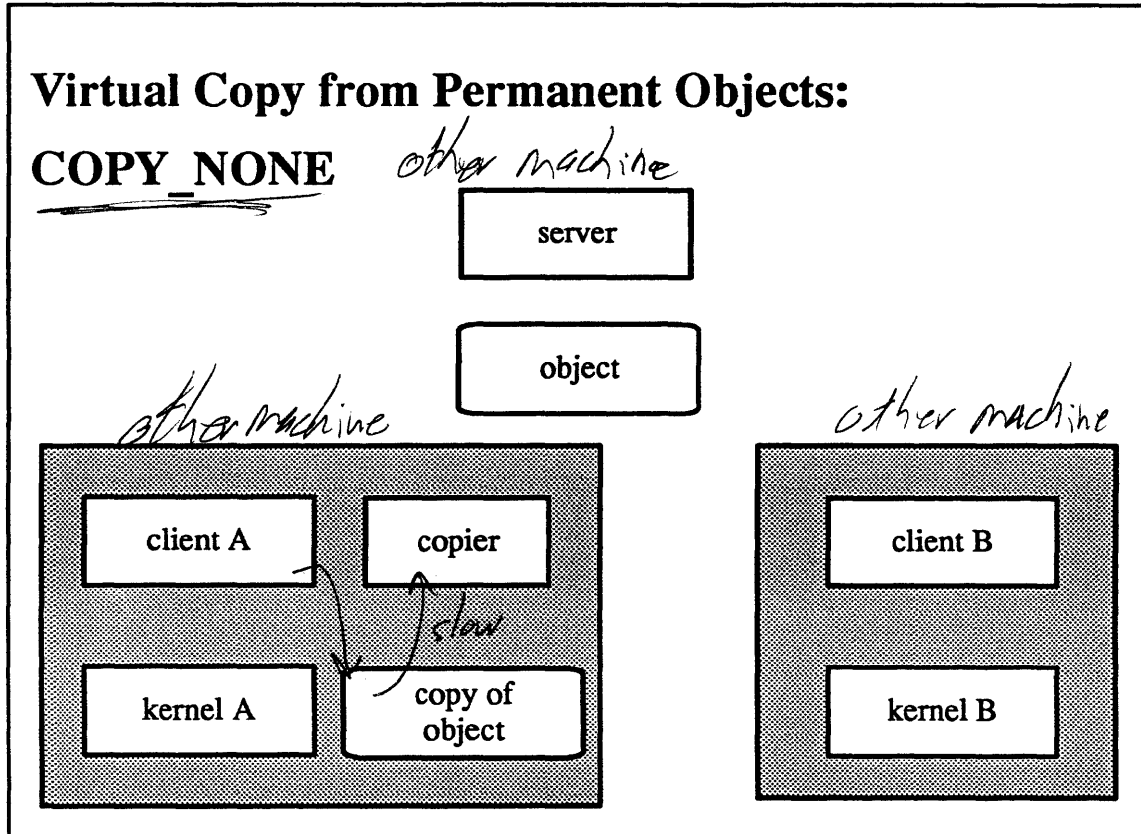
Module 4 — Virtual Memory

Student Notes: Virtual Copy from Permanent Objects: COPY_DELAY, part 6

The client now modifies page 3. But, before this is allowed to happen, a copy of the page is propagated across the copy link to copier 3's copy object, assuring copier 3 of a correct view. Since the shadow link of copier 1's and copier 2's copy object points to copier 3's copy object, their view is maintained as well.

Module 4 — Virtual Memory

4-49. Copying and Sharing



4-49.

© 1990, 1991 Open Software Foundation

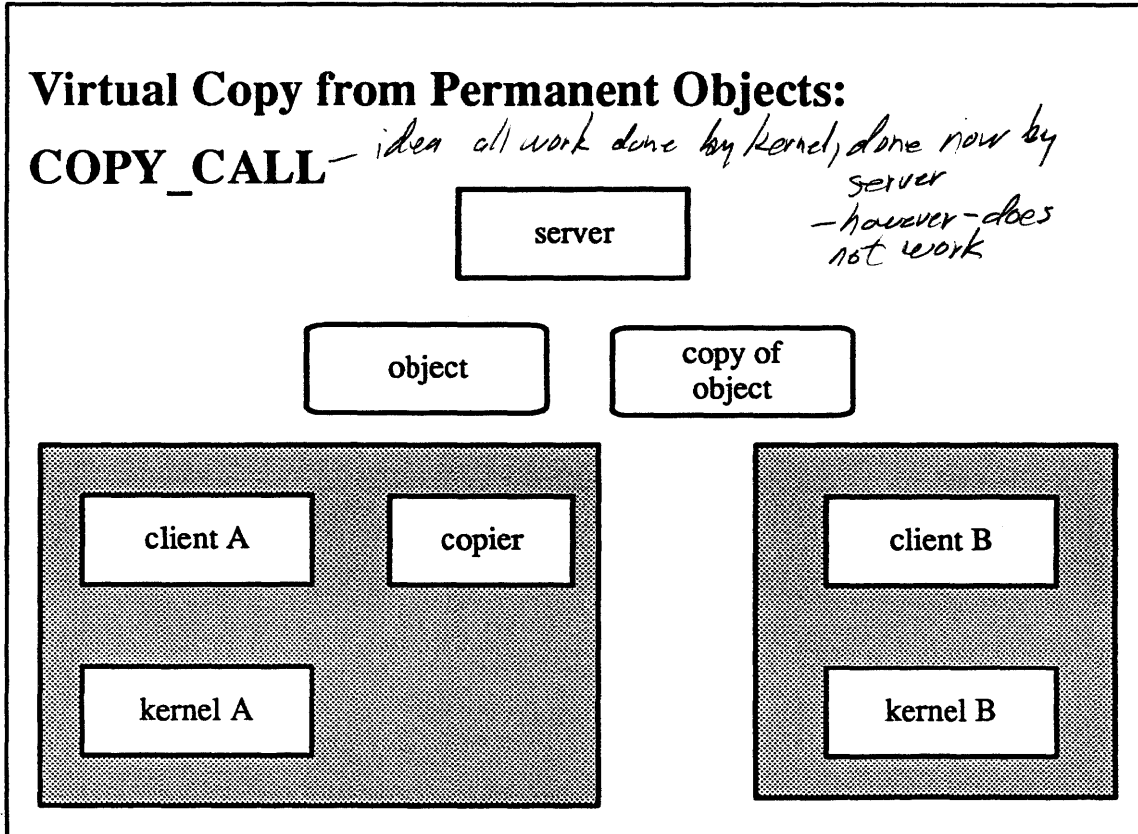
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-50. Copying and Sharing



4-50.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

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

*Page tables - like VAX
no page table - ~~to~~ translation cache - MIPS
386
RS6000 - inverted page tables*

Module 4 — Virtual Memory

Student Notes: Pmaps *machine dependent issues*

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*)

Module 4 — Virtual Memory

Student Notes: 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

Module 4 — Virtual Memory

Student Notes: 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* → *page out*
 - remove a page from all maps and indicate whether the page has been modified

Module 4 — Virtual Memory

Student Notes: Operations on Pmaps: Physical Pages

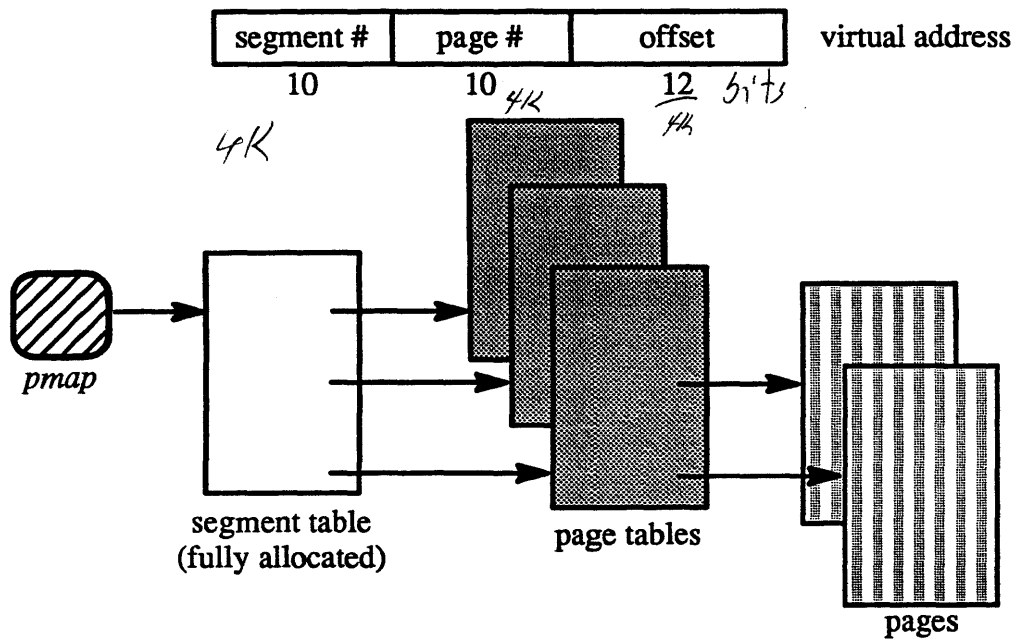
This set of operations affects a physical page and all of the *pmaps* in which it appears. A *pmap_copy_on_write* message would be called as part of a virtual copy operation to implement copy-on-write semantics. It makes certain that write permission is not allowed for this page in all of the maps in 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 *pmaps* but *pmap_remove_all* must check to see if the page has been modified via any of these *pmaps*. If it has, the modification is indicated by setting a bit in a global array.

Module 4 — Virtual Memory

4-55. The Pmap Module

Forward-Mapped Segmented-Paged Architecture



4-55.

i386

© 1990, 1991 Open Software Foundation

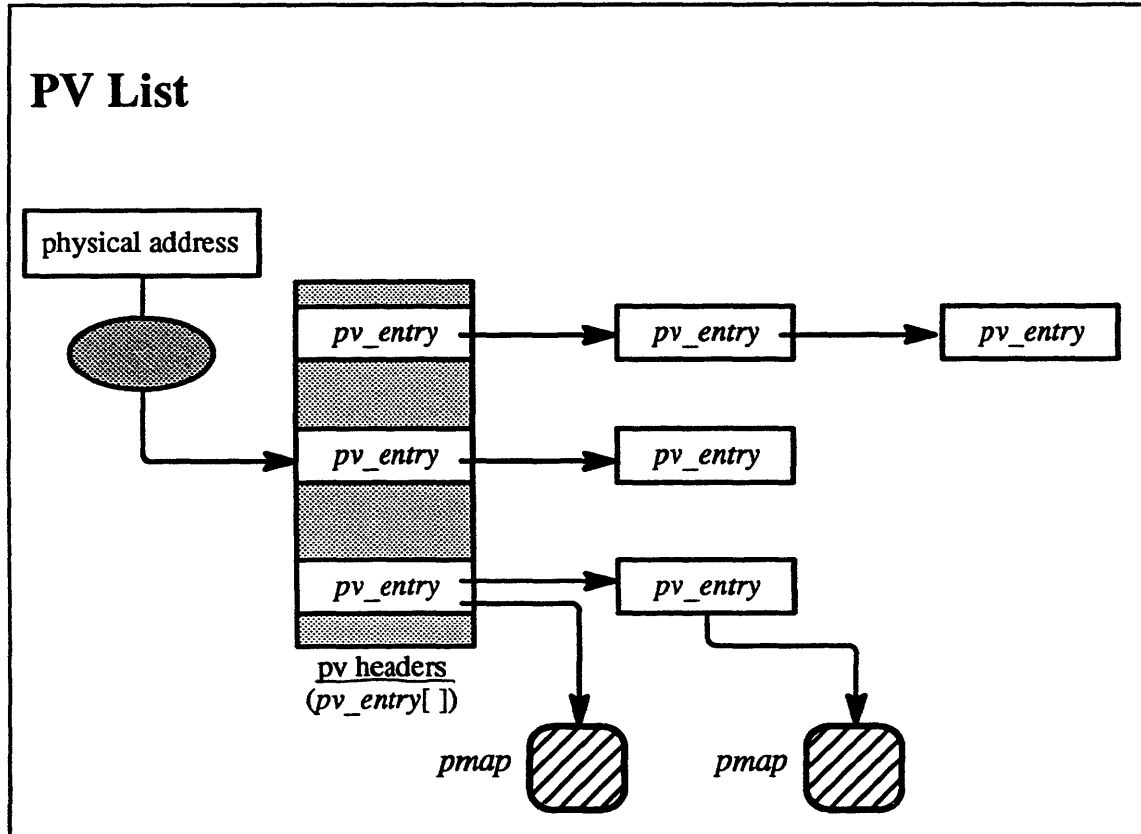
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-56. The Pmap Module



4-56.

© 1990, 1991 Open Software Foundation

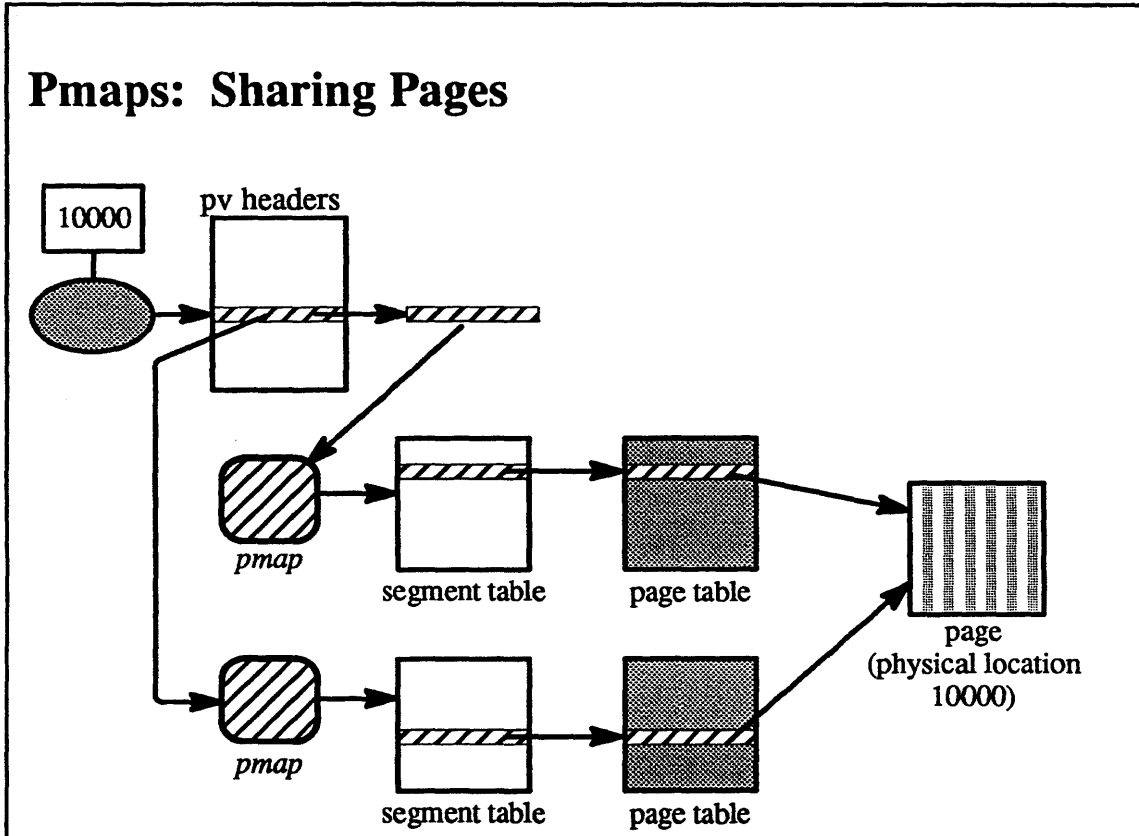
Module 4 — Virtual Memory

Student Notes: 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_entries*. 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.

Module 4 — Virtual Memory

4-57. The Pmap Module



4-57.

© 1990, 1991 Open Software Foundation

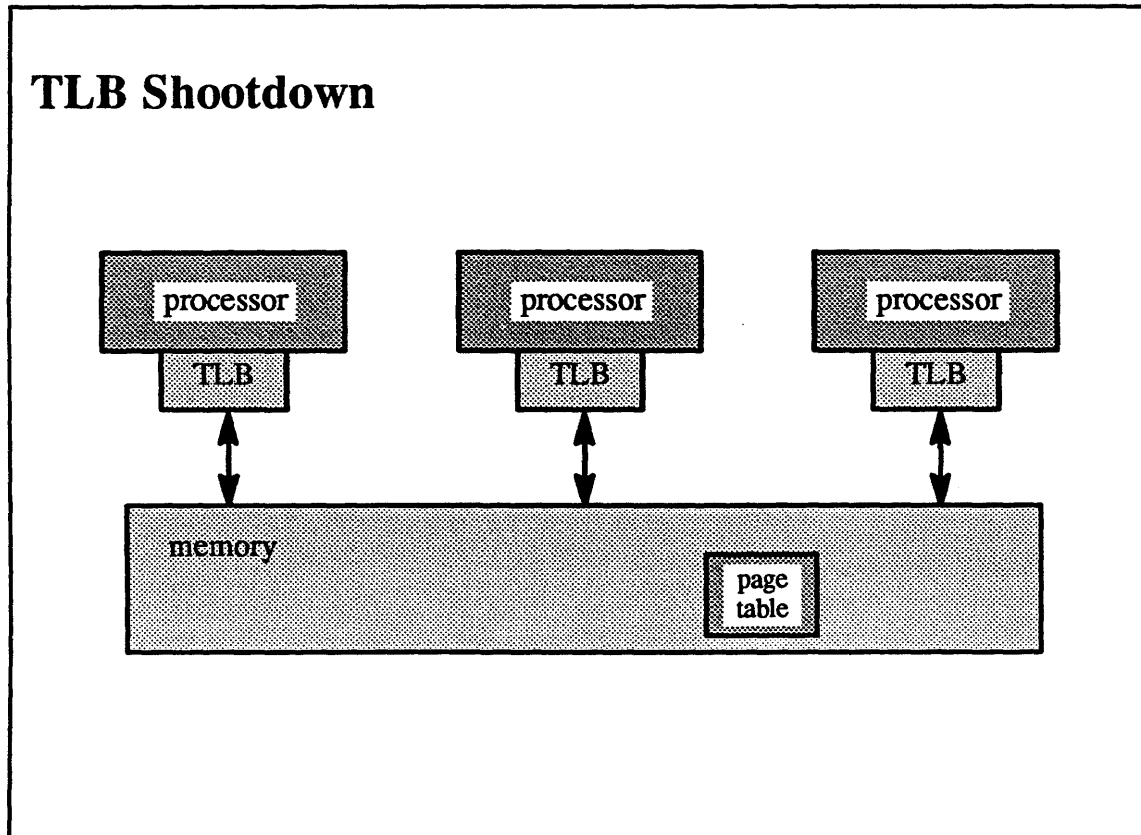
Module 4 — Virtual Memory

Student Notes: 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.

Module 4 — Virtual Memory

4-58. The Pmap Module



4-58.

© 1990, 1991 Open Software Foundation

Module 4 — Virtual Memory

Student Notes: TLB Shutdown

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 shutdown algorithm can be found in Black, 1989.

4-59. The Pmap Module

TLB Shutdown Algorithm

Initiator:

lock *pmap*

send interrupts to all processors using
the *pmap*

spin on all-processor bit vector, waiting
for others to acknowledge

invalidate TLB

update translation map

unlock *pmap*

Responders:

clear bit in vector

wait (spin) for *pmap* unlock

invalidate TLB

return from interrupt

Module 4 — Virtual Memory

Student Notes: TLB Shutdown 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_entries* 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.

Module 4 — Virtual Memory

- 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 4 — Virtual Memory