Sun Education
Not to be copied

SunOS Internals

March 9, 1990

Revision C.2

Credits and Trademarks

Sun Workstation is a registered trademark of Sun Microsystems, Inc.

SunStation, Sun Microsystems, SunCore, SunWindows, SunView, DVMA and the combination of Sun with a numeric suffix are registered trademarks of Sun Microsystems, Inc.

UNIX, UNIX/32V, UNIX System III and UNIX System V are trademarks of AT&T Bell Laboratories.

Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc., and may also be a trademark of various telephone companies around the world. Sun will be revising future versions of software and documentation to remove references to Yellow Pages.

Copyright 1989 by Sun Microsystems.
Printed 1989.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical or otherwise, without prior explicit written permission from Sun Microsystems.

March 9, 1990
Revision C.2

Table of Contents

Module 1 – Overview

The Sun OS operating system **1–6**

syscall() **1–10**

Traps and Interrupts **1–12**

User Mode **1–14**

Top Half of Kernel **1–16**

Bottom Half of the Kernel **1–18**

spl() **1–20**

Processor Status Word **1–22**

hardclock( ) **1–26**

hardclock( ) **1–28**

softclock() **1–30**

Table of Contents(cont.)

Module 2 – Memory Management

Definitions 2-4

Definitions 2-5

General Memory Information 2-8

MMU translations 2-10

MMU Translation for Sun-3s 2-12

MMU Translation for Sun-4s 2-14

Page Table Entries 2-16

Cache 2-18

Cache Coherency 2-24

Reasons for implementing 4.0 VM 2-26

The layers of the VM system 2-32

mmap(2) 2-36

mmap(2) 2-37

msync(2) 2-38

munmap(2) 2-40

AS Structure 2-46

Seg Structure 2-48

Seg_ops Structure 2-50

Segvn_data Structure 2-54

Table of Contents(cont.)

The Vnode Segment Driver **2-58**

Anonymous Memory **2-60**

The anon and anoninfo Structures **2-62**

The anon_map structure **2-64**

mapping structures **2-66**

The Seg_map Segment Driver **2-74**

The Read and Write Routines **2-76**

The seg_dev segment driver **2-78**

The seg_kmem driver **2-80**

The Page Structure **2-82**

The HAT Layer **2-88**

Page Fault Handling **2-90**

Paging **2-94**

Paging In **2-96**

Page Replacement **2-98**

The Clock Algorithm **2-100**

The swapout algorithm **2-102**

The swapout algorithm **2-103**

The swapout algorithm(cont.) **2-104**

The swapin algorithm **2-106**

The swapin algorithm **2-107**

Table of Contents(cont.)

Module 3 – Process Management

The Process State 3-4
The Proc Structure 3-10
Process Address Space 3-22
The User Structure 3-24
Process Scheduling 3-26
Queues 3-28
Priorities 3-30
Scheduling 3-32
Context Switches 3-34
sleep() 3-36
wakeup() 3-38
setpri() 3-40
schedcpu() 3-42
Computing CPU Usage 3-44
Creation of Processes 3-46
fork1() 3-48
Before Fork 3-58
After Fork 3-60
execve() 3-62
getxfile() 3-72
relvm() 3-76
exit() 3-78
wait() 3-86
Signals 3-88
Signal Routines 3-94
sigvec() 3-96
psignal(p,sig) 3-100
psig() 3-106

Table of Contents(cont.)

Module 4 – Shared Libraries

Shared Library Implementation Goals **4-4**

SunOS Approach **4-6**

New Link Editor **4-8**

Position Independent Code **4-10**

Compiler Changes **4-12**

Assembler Changes **4-14**

Link Editor Changes **4-16**

The link_dynamic structure **4-20**

a.out execution **4-22**

Interposition **4-28**

Construction of a Shared C Library **4-30**

Table of Contents(cont.)

Module 5 – The File System

The Fast File System 5-4

The Vnode Interface 5-6

The vfs structure 5-8

vfsops 5-10

vnode structure 5-12

vnodeops structure 5-14

Opening a Vnode 5-18

The System Open File Table 5-20

Directories 5-22

inode structure 5-24

inode structure 5-25

The Superblock 5-30

cg structure 5-34

csum structure 5-38

The dirlook() Routine 5-40

Global Placement Policies 5-44

blkpref ( ) 5-46

Determining ipref 5-52

ialloc ( ) 5-54

hashalloc 5-56

ialloccg() 5-58

alloc 5-60

alloccg 5-64

alloccgblk 5-66

mapsearch() 5-68

fragment table 5-69

Table of Contents(cont.)

Module 6 – Device Management

Introduction **6-2**
Types of Files **6-4**
Types of Devices **6-8**
SunOS Bus Structures **6-10**
DVMA **6-12**
Character vs. Block Device Drivers **6-14**
Device Driver Routines **6-16**
Character Driver Top Half Interface **6-18**
Block Driver Top Half Interface **6-20**
Opening a Vnode (device) **6-22**
Virtual Memory System Map **6-27**
mmap( ) system call **6-28**
as_unmap( ) **6-30**
as_map( ) **6-32**
The Read and Write Routines **6-34**
Scatter Read and Gather Write **6-36**
read( ), readv( ) **6-38**
write( ), writev( ) **6-42**
segmap_pagecreate( ) **6-44**
segmap_release( ) **6-48**
ufs_putpage( ) **6-50**
hat_unload( ) **6-52**
pagefault( ) **6-54**
ufs_getapage( ) **6-56**
hat_memload( ) **6-58**
hat_pteload( ) **6-60**

Table of Contents(cont.)

Module 7 – Interprocess Communication

Interprocess Communication – Sockets 7-1

Server/Client Model 7-2

Sockets (BSD) 7-4

socket( ) 7-6

Socket Structure 7-10

Mbuf Structure 7-14

bind( ) 7-20

listen( ) 7-26

accept( ) 7-30

connect( ) 7-32

sonewconn( ) 7-34

send( ) 7-40

recv( ) 7-46

Table of Contents(cont.)

Module 8 –System Startup

Bootstrap **8-4**

Vmunix **8-12**

Appendix A –Labs and Questions

Overview Questions **A-2**

Memory Questions **A-7**

Lab 1 Memory **A-11**

VM questions **A-13**

Process Scheduling Questions **A-17**

Process Creation Questions **A-26**

Signal Questions **A-31**

Lab 2 Process Management part a **A-33**

Lab 2 Process Management part b **A-34**

Lab 2 Process Management part c **A-35**

Lab 2 Process Management part d process creation **A-38**

Lab 2 Process Management part e Signals **A-39**

File Questions **A-40**

Lab 3 File System **A-48**

I/O Management Questions **A-50**

Table of Contents(cont.)

Appendix B – Header Files

file.h **B-2**

fs.h **B-5**

hat.h **B-15**

inode.h **B-17**

mbuf.h **B-25**

mmu.h **B-31**

mount.h **B-36**

page.h **B-38**

pcb.h for Sun3s **B-41**

pcb.h for Sun4s **B-43**

proc.h **B-45**

protosw.h **B-47**

seg_map.h **B-52**

seg_vn.h **B-54**

socket.h **B-56**

socketvar.h **B-61**

ucred.h **B-65**

User.h **B-66**

vfs.h **B-71**

vm_hat.h (Sun-3 version)
machine specific **B-75**

vnode.h **B-81**

Table of Contents(cont.)

Appendix C

Using kadb **C-2**

powers of two **C-8**

Module 1

Overview

Objectives: Upon completion of this module, the student will be able to:

- explain what distinguishes the SunOS from other Unix operating systems
- list the major components of the operating system and explain how they are related
- give at least three examples of how the kernel is entered from the user mode
- explain the different characteristics of the upper and lower half of the kernel
- list the major functions of the clock algorithm and explain why it is broken into two parts.

Evaluation: quiz

Reference information: Course Manual

Introduction

What will be covered in this class:

- Memory Management
- Process Management
- Signals
- The File System
- Device Management
- Inter Process Communications
- Shared Libraries
- System Startup

Introduction

Memory management is at the core of the changes that took place at 4.0. It separates Sun from other implementations of 4.3BSD and System V. It will be discussed at some length, in the next module, and most of the other modules will relate back to that one.

The process management module will discuss how processes share the CPU, it will also cover the sleep, wakeup mechanism, and process creation and termination.

Signals have been modeled after the hardware interrupt mechanism. The signal module will discuss the various signals, how they are delivered, how they are caught or masked, and other aspects of signal management.

The file system uses the fast file system, developed for 4.2 BSD. The module discusses the data structures and algorithms involved.

The device management module will discuss in detail how the VM system relates to I/O management of reads and writes, and how peripheral devices in general interface with the kernel via character or block device drivers.

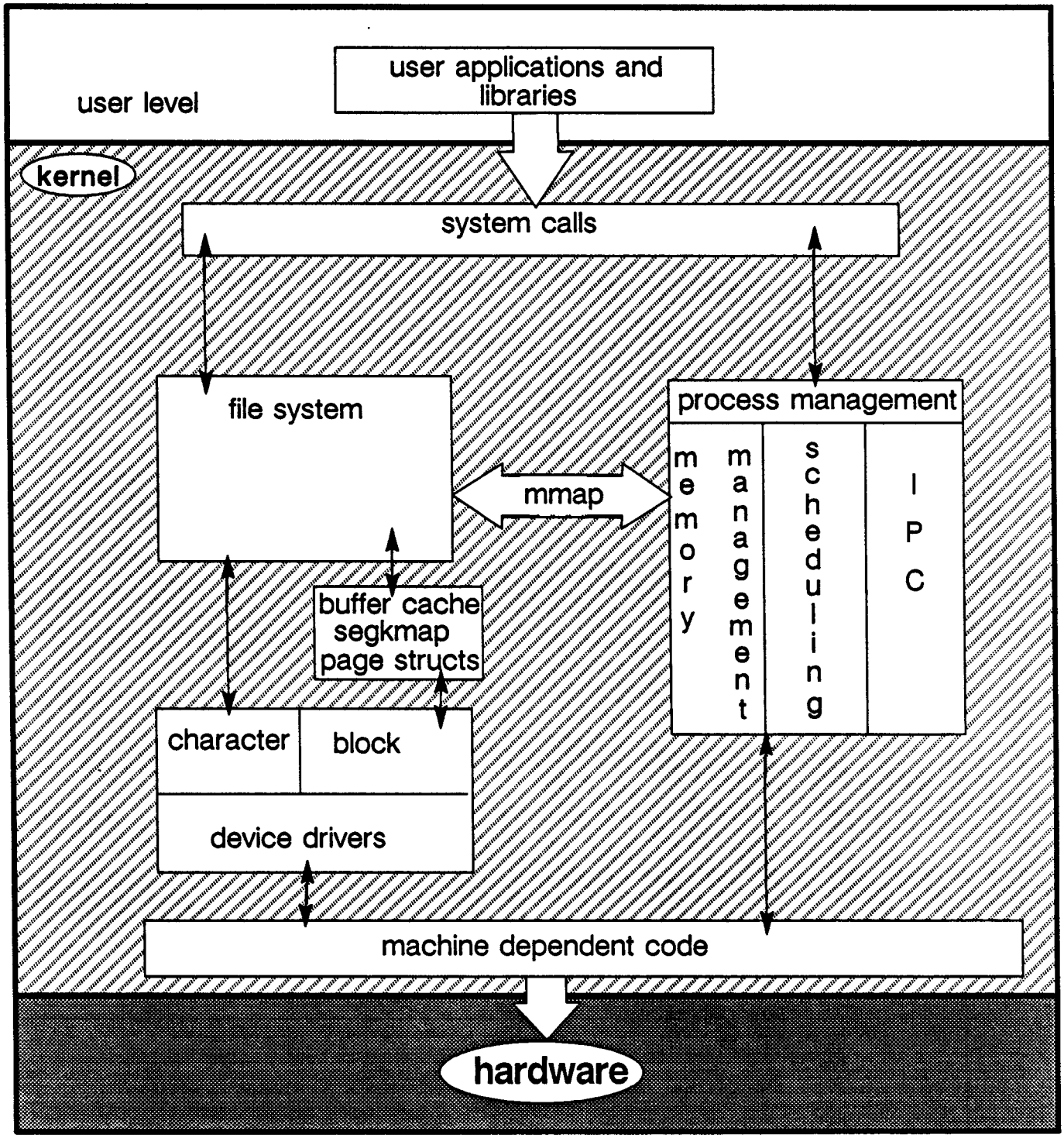
The interprocess communications section will explain the inner workings of the socket interface.

Shared libraries will be related to the VM system's ability to share objects. The steps and structures used to dynamically link routines during program execution will be discussed.

System startup will be a brief summary of how the system is initialized from power up until the first user can login.

This module will cover some topics that either fall outside the topics defined in the afore mentioned modules, or are related to more than one topic.

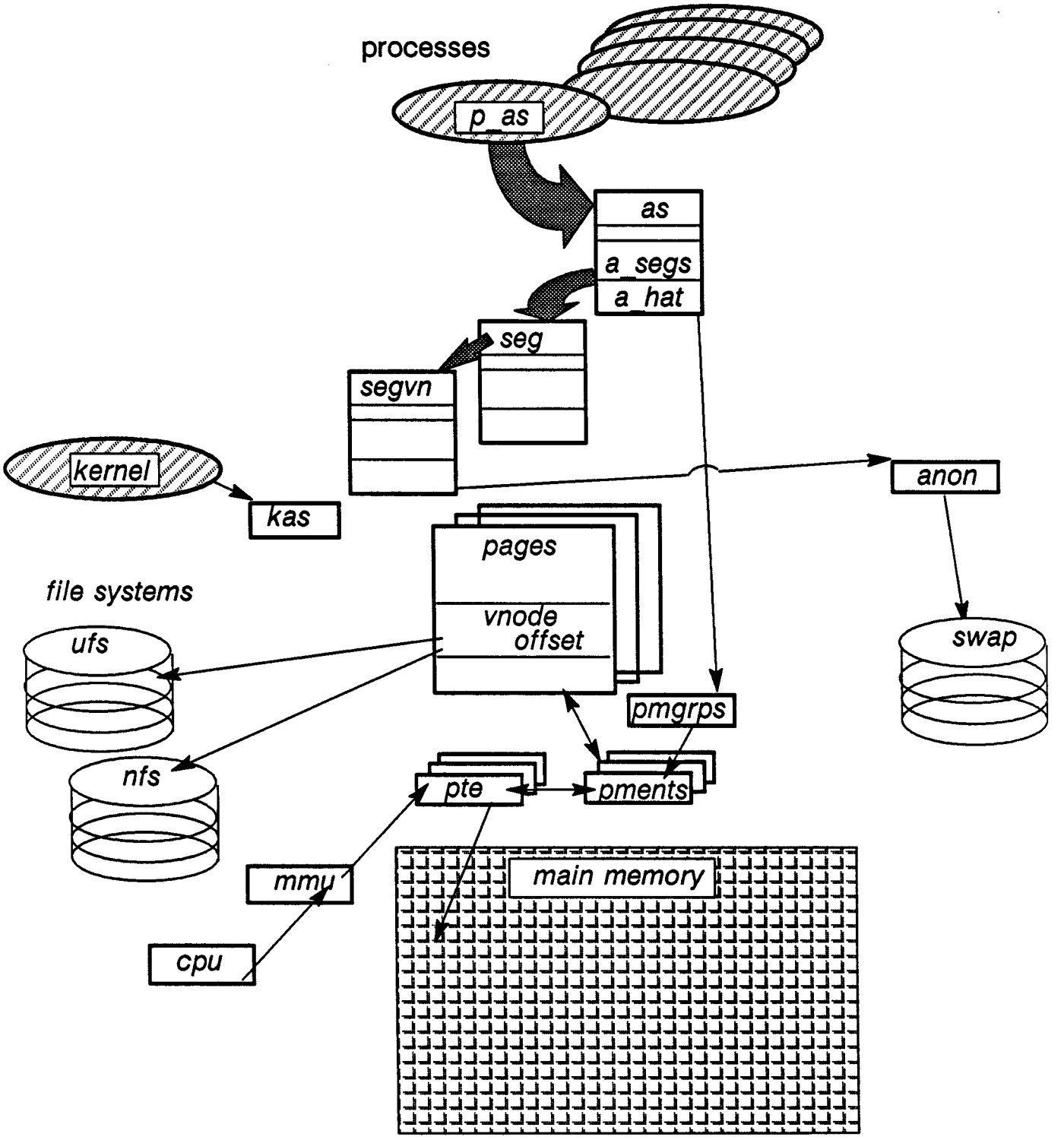
The Unix operating system



The Unix operating system

This diagram represents the view most users associate with the Unix operating system.

The Sun OS operating system



The Sun OS operating system

The thing that distinguishes the SunOS from other operating systems is the role of the virtual memory (VM) system.

This section will be discussed in depth in the next module, and many other modules will refer back to these concepts.

This diagram is only meant to serve as an introduction of some of the structures, and their relationships in the operating system. For the sake of clarity there have been many simplifications (and omissions) of structures that will be addressed in greater depth in the appropriate modules.

The Kernel

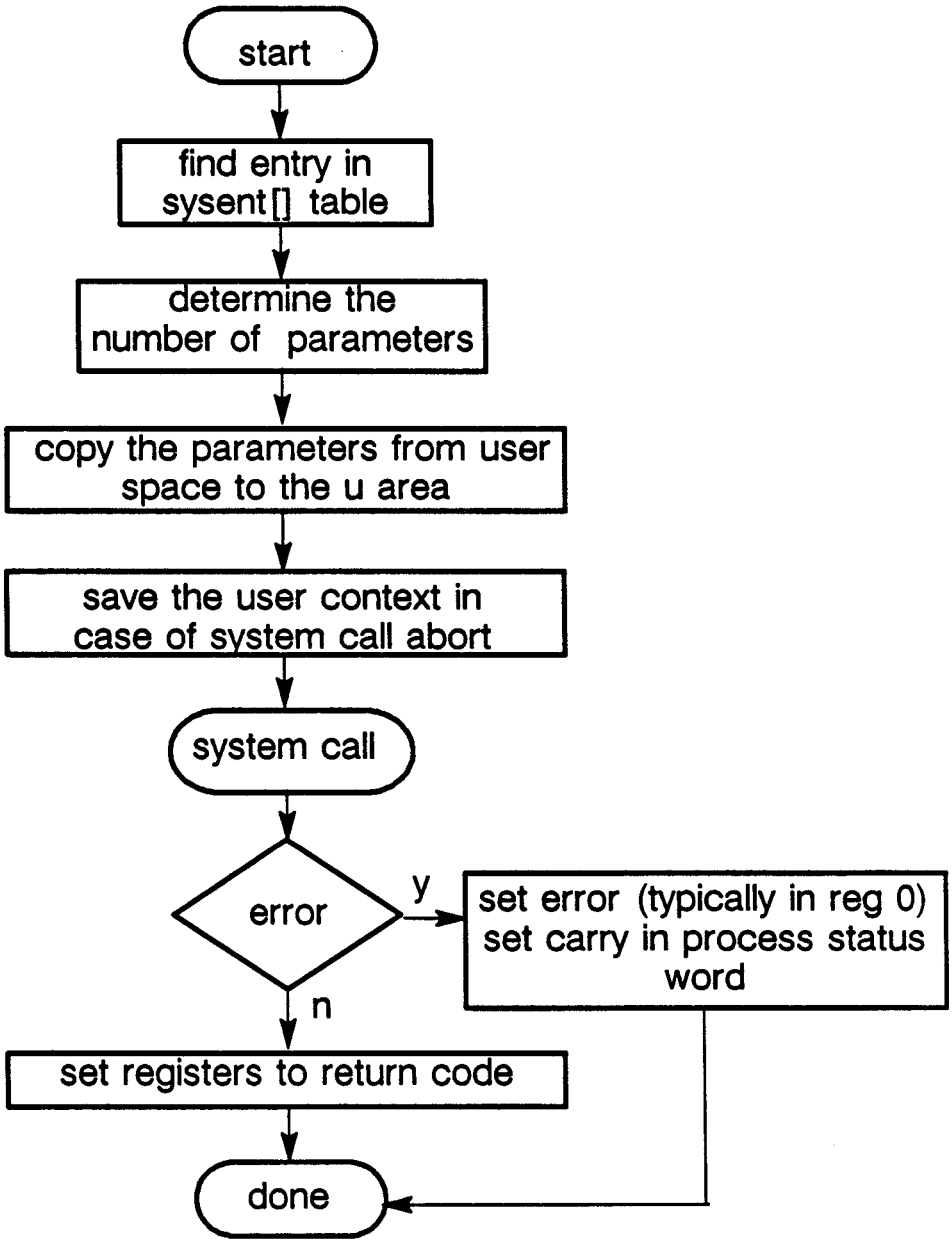
The kernel can be viewed as a resource manager. The resources to be allotted to various processes are:

- The CPU
- Main Memory (possibly cache)
- Swap Space
- The File System
- Network Facilities
- Process Management Facilities
- Clock Functions

The Kernel

The clock function may be thought of as a means of distributing these resources rather than an actual resource.

syscall()


start
find entry in sysent[] table
determine the number of parameters
copy the parameters from user space to the u area
save the user context in case of system call abort
system call
error
y
set error (typically in reg 0) set carry in process status word
n
set registers to return code
done

syscall()

Processes running in the SunOS operating system are divided into kernel and user level code.

A user requests a resource from the kernel via a system call.

A system call generates a trap which changes the mode of execution to kernel mode. The kernel call will have to complete its operation, (successfully or not) before the user application will be run again.

System calls are synchronous to the user application, they are handled by the *syscall()* routine. The arguments passed to the *syscall()* routine for Sun–3s are *code,* which is an index into the *sysent[]* array (an array of system call entry points), and a pointer to a register structure. On Sun–4s only a pointer to a register structure is passed, and the index information is taken from the register.

Arguments are copied from the user's address space, to the u structure, which is readily accessible to the kernel.

To set up for a system call abort, the *syscall()* routine calls *syscall_setjmp()*. On Sun–3s this is a special *setjmp()* that only saves a subset of the registers (for speed).

An unsuccessful system call will return a –1, and an error code is placed in the global variable *errno* (taken from *u.u_error*). This will occur if one of the routines implementing the system call encounters an error. It may also occur if the kernel blocks for some resource, and while sleeping is delivered a signal which causes the system call to abort.

There are special considerations for a process to back up a program counter to the instruction that initiated the system call, they are:

- parameters passed to the system call should not be modified
- partial handling of the call must be taken into account.

Traps and Interrupts

- The kernel may also be entered as the result of other types of traps, or an interrupt.
- Interrupts occur asynchronous to the executing process, typically due to a controller that needs to be serviced.
- Also the clock generates interrupts at regular intervals to handle timer driven events.
- An exception is an unexpected event caused by the currently executing process.

Traps and Interrupts

The system call, as explained earlier, is actually just a type of system trap.

Traps may be initiated by software or hardware. An example of a software initiated trap is when the scheduling routine detects a process that has a better software priority than the currently running process. The *aston()* routine is called, and the system will perform a context switch as soon as possible.

An example of a hardware initiated trap would be a page fault, which is detected by the memory management unit *(mmu)* hardware. This causes the page fault handling routines to make the page available, and restart the process.

Examples of exceptions are: generating an illegal address, executing a privileged instruction, or a divide by zero.

Interrupts are handled in a manner similar to the system call mechanism. The controller would supply a vector interrupt address when requesting service from the cpu and this will point to an interrupt handler.

The exception or interrupt handler may back up the program counter to re–execute the instruction after handling a fault or an exception.

User Mode

In User Mode:

- The process runs in a context.
- The process may be pre–empted by a higher priority process.
- It uses the user stack, the stack segment, in the user address space.
- It can only access what has been mapped to its user address space, such as text, data, stack and mmaped files.
- It cannot sleep in User mode.
- The process will store its per process information in a user area and process structure.

User Mode

A user's context is defined by its general cpu registers, which include pointers to the user's address space, such as the stack, data, and text segments. The user's context is further defined by per process data structures defined in the process management section.

Top Half of Kernel

The kernel is divided into a top half and bottom half.

The top half of the kernel:

- Is entered due to a trap such as a system call.
- Runs in the executing user's context.
- Runs in the kernel stack, which is part of the user area.
- It can call sleep to wait for a system resource.
- It cannot be pre–empted
- The top half of the kernel may be interrupted.

Top Half of Kernel

The upper half of the kernel just runs in the context of the current user. It has access to kernel address space *(kas)*, and the address space of the currently running process *(as)*.

The kernel stack is fixed in size, and is embedded in the user structure, which is accessible from the kernel mode.

While executing in the kernel mode the process is not pre-emptive. This is one reason most Unix systems cannot do true real time processing.

The upper half of the kernel can be interrupted by the bottom half of the kernel, by any interrupt handler that has a higher interrupt priority.

Bottom Half of the Kernel

The bottom half of the kernel

- Has no context.
- Runs as a result of a hardware interrupt, asynchronous to the running process.
- Runs on a system interrupt stack in kernel address space.
- Interrupts are handled by device drivers, or kernel interrupt routines (ex. clock).
- Cannot block (call sleep).
- For I/O, the bottom of the kernel works in tandem with the top half of the kernel.

Bottom Half of the Kernel

The bottom half is driven by the occurrence of hardware events such as the completion of a disk I/O, or a clock interrupt. It is typically not related to the currently executing process. Therefore it must not modify the kernel stack for that process.

The bottom half of the kernel executes in the context of the user that happened to be running when the interrupt took place. If the bottom half of the kernel ever called *sleep()* it would put that process to sleep, and that process probably has nothing to do with the interrupt.

The top half may request an operation to be carried out by the bottom half. Then the top half sleeps, upon completion of the operation, the bottom half requests the top half be scheduled to run.

spl()

example interrupt levels for Sun–3



higher

7	nonmaskable interrupt realtime cpu clock
6	serial communications
5	*hardclock()*
4	video vert. int. graphics, ALM
3	tape, ethernet
2	xylogics and SCSI Disks
1	*softclock()*
0	all interrupts enabled

lower



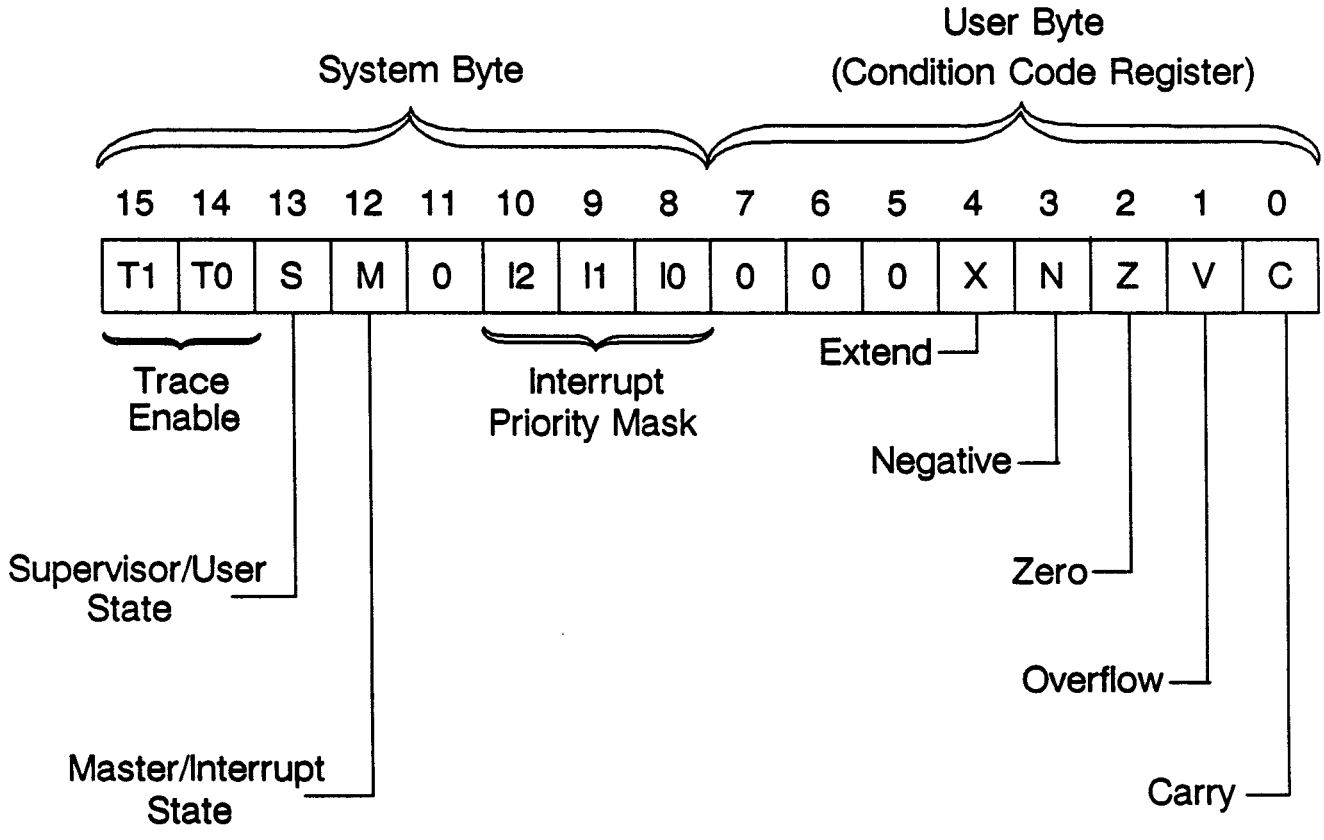
spl()

Since interrupts occur asynchronous to the executing process it is sometimes necessary to raise the priority of an executing process while manipulating certain structures.

This is done with a call to the assembler routine *spl()* (set priority level). By raising the priority of the currently executing process, interrupt driven processes are masked, so they do not interrupt the process during critical sections of code.

The safest way to set the priority level is to save the current priority level, do the crucial processing (such as manipulating a system queue), then to return the priority to its original level. Sometimes in the kernel certain assumptions are made about what *spl* level the process was running at prior to raising it, and it is just restored to that level.

Processor Status Word



Processor Status Word

Each time a trap or an interrupt occurs, the executing process has to have its machine state saved.

This typically includes the CPU's general register set, and the processor status long word.

The Clock

- The clock routine is divided into two halves.
- The clock routine is driven by interrupt signals from an on board clock chip.
- The hard clock runs at interrupt priority 5 and will block all lower interrupt levels.

The Clock

The clock algorithms *hardclock()* and *softclock*() are entered due to an interrupt signal driven by a clock chip. This raises the interrupt signal at regular intervals, for example 100 times per second for a SUN–4 and 50 times a second for a SUN–3.

Each time the clock routine interrupts is referred to as a tick.

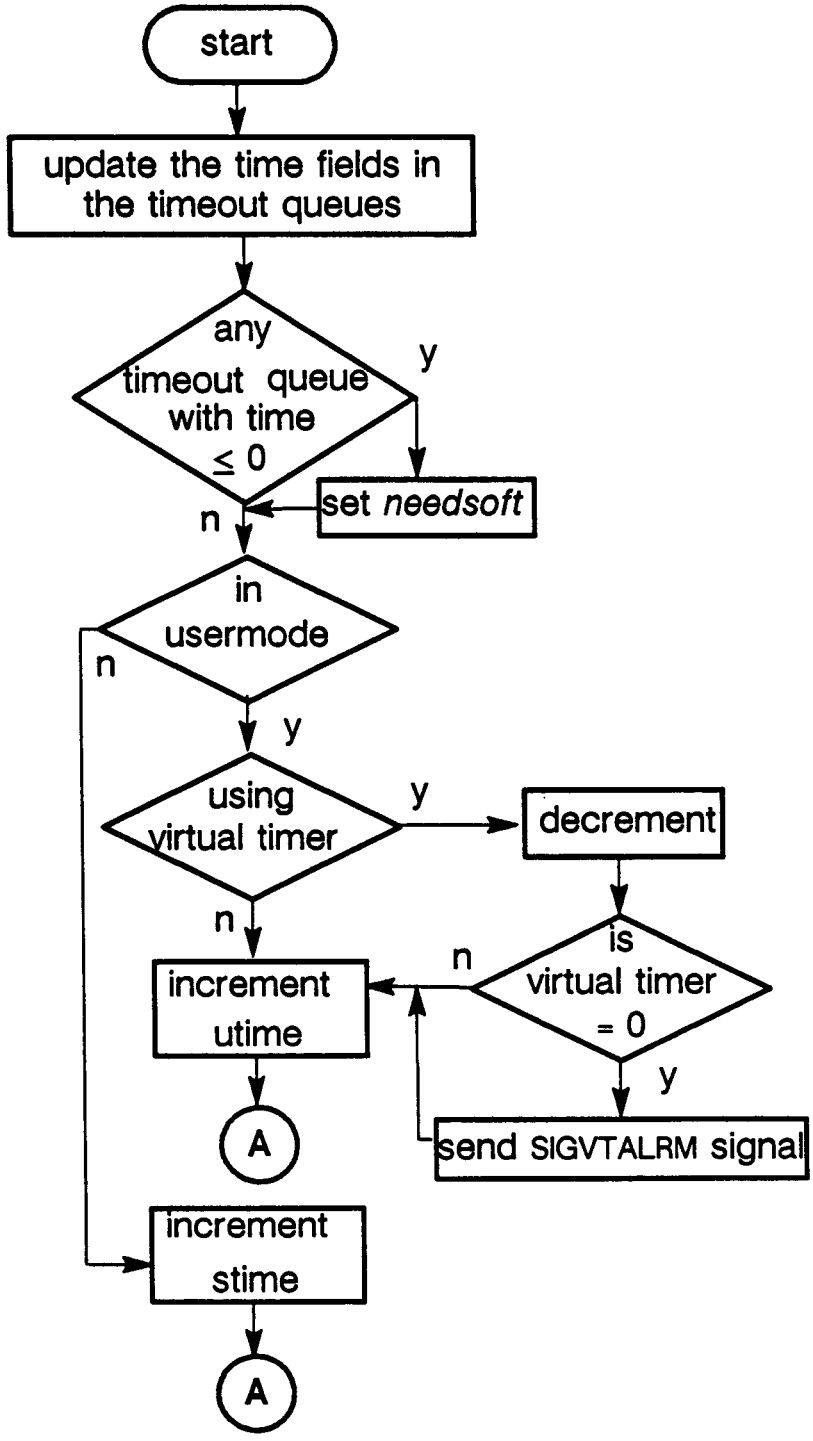
The tasks performed by the clock routine are split into two functions. The *hardclock()* routine runs every tick, it has to finish its processing in less than 0.01 seconds or it will miss the next clock interrupt.

Also since the clock runs at such a high priority it will block other crucial activity until it is finished.

The hard clock routine handles only the most important time related activities.

Other time related activities are handled by a software interrupt driven process that runs at a lower priority, called the *softclock()*.

hardclock( )


start
update the time fields in the timeout queues
any timeout queue with time ≤ 0
y
set needsoft
n
in usermode
n
y
using virtual timer
y
decrement
n
is virtual timer = 0
n
y
increment utime
A
send SIGVTALRM signal
increment stime
A

hardclock( )

The hardclock performs the following functions:

If there is any event in the timeout queues that the time field has expired (time $\leq$ 0), set *needsoft.*

If the process has exceeded a cpu resource limit, send SIGXCPU signal.

Collect status information, such as was the CPU executing in user mode, executing in kernel mode, or idle. Before incrementing the timers it checks the global variable *noproc,* to see if a process is running.

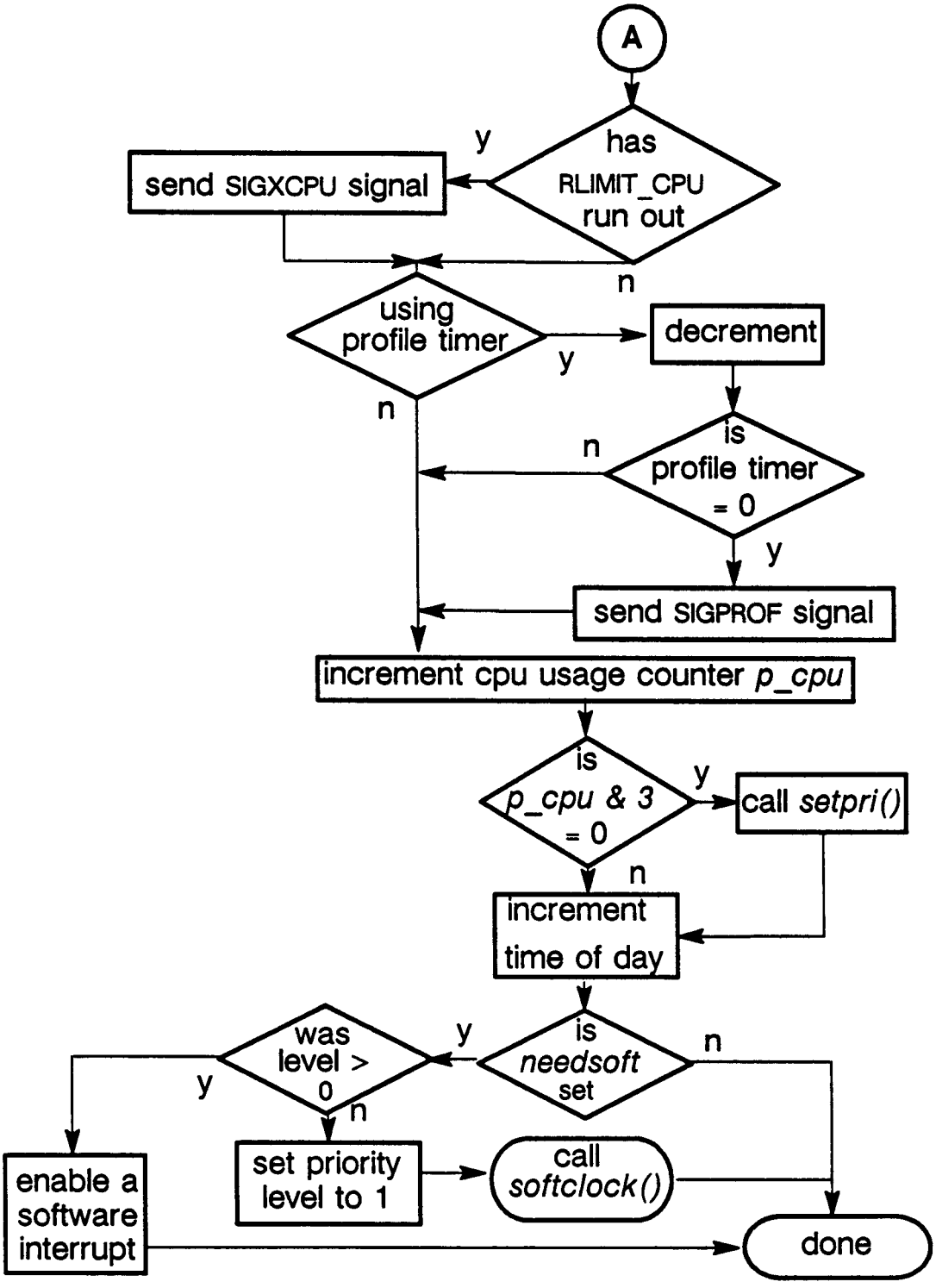
Increment the current processes cpu_timer, if 4 ticks have been accumulated, call *setpri(),* to recalculate its priority.

Check if the process is using a virtual or profiling timer decrement the timer and if 0, signal the process.

Increment time of day.

If needsoft is set, and if before the hardclock interrupt the system was at level 0, set the hardware priority to level 1 and call *softclock().* If needsoft is set and we are returning to an interrupt priority level greater than 0, then just enable the priority 1 (software) interrupt so it can take place later.

hardclock( )


A
has RLIMIT_CPU run out
y
send SIGXCPU signal
n
using profile timer
y
decrement
n
is profile timer = 0
n
y
send SIGPROF signal
increment cpu usage counter p_cpu
is p_cpu & 3 = 0
y
call setpri()
n
increment time of day
is needsoft set
y
n
was level > 0
y
n
enable a software interrupt
set priority level to 1
call softclock()
done

hardclock( )

The hardclock performs the following functions:

If there is any event in the timeout queues that the time field has expired (time $\leq$ 0), set *needsoft.*

If the process has exceeded a cpu resource limit, send SIGXCPU signal.

Collect status information, such as was the CPU executing in user mode, executing in kernel mode, or idle. Before incrementing the timers it checks the global variable *noproc,* to see if a process is running.

Increment the current processes cpu_timer, if 4 ticks have been accumulated, call *setpri(),* to recalculate its priority.

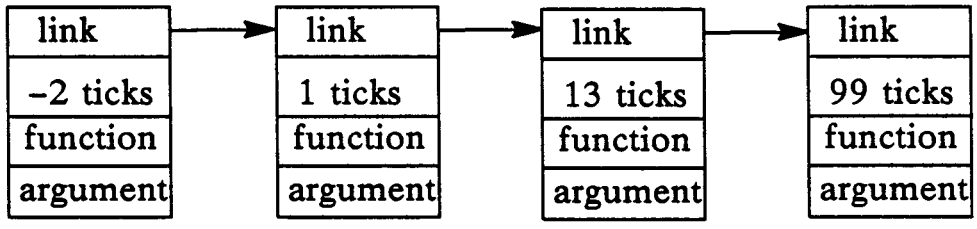
Check if the process is using a virtual or profiling timer decrement the timer and if 0, signal the process.

Increment time of day.

If needsoft is set, and before the hardclock interrupt the system was at level 0, then call *softclock().* If needsoft was set and we are returning to an interrupt priority level greater than 0, then just enable the priority 1 (software) interrupt so it can take place later.

softclock()

timeout queues



softclock() takes care of timed events such as

- watchdog timers
- system event timers
- network transmission timers

softclock()

The *softclock()* runs at a lower priority than the *hardclock()* (spl level 1). It takes care of the less critical timing functions so less time is spent in *hardclock()*.

The *softclock()* routine works off a timeout queue that keeps track of a function to execute, and the number of clock ticks before the function is to be called.

The event is placed on the queue such that the number of ticks in its queue is calculated to be the number of ticks after the previous event has taken place that this function should be called. In this way the *softclock()* routine only has to update the first element of the queue rather than every time in each queue.

If the number of ticks is negative, (as in the first queue in the diagram) it represents how many ticks have passed since the function should have been called. It is possible that the *hardclock()* will execute again before all the events with time ≤ 0 in the timeout queues can be handled.

An argument is available so that the same function can be scheduled by more than one process without conflict.

Module 2
Memory Management

Objectives: Upon completion of this module, the student will be able to:

- List the motivations for the virtual memory rewrite at 4.0.
- List the layers of the Sun–OS 4.0 VM system and define the role of each layer.
- List the steps and data structures involved in virtual to physical address translation.
- In the lab locate the page structure, the segment table, a page table entry, using adb and be able to break out the bit fields to ascertain information about a given page.

Evaluation:

Lab 1.

Reference information:

Sun–3 Architecture: A Sun Technical Report (part number FF121–01/30K)

Introduction

In this module we will cover:

- Sun's implementation of a new VM system
- The motivation for the new VM
- The routines and structures involved
- Labs

Introduction

At SunOS 4.0 the virtual memory subsytem underwent extensive changes.

In this chapter we will discuss the motivation for these changes followed by a general discussion of the concepts involved.

Also we will go into some detail about the routines and structures involved in the implementation of the new Virtual Memory system .

In the lab we will be using *abd* to examine some of the relevant memory management structures and to trace the virtual to physical address translation process.

Definitions

Virtual Memory – allows us to access more data than can be placed in main memory at any given instant. It gives us a window to look at memory such that the user does not have to be concerned with where the memory object is physically located. Virtual memory is all the files you can map into a process address space (all mountable file systems).

Address Space – a window to virtual memory. Your process or logical address space can be thought of as the range of machine addresses your process can generate while in execution. The contents of that address space are defined by the mappings onto objects in virtual memory (files).

Copy–On–Write – when a page is to be modified make a separate copy of the page and modify the copy instead of the original.

Definitions

Virtual Memory – Traditionally virtual memory consisted of main memory and swap space (a paging area). With the advent of virtual memory systems such as Sun's 4.0 where pages do not necessarily have to be backed by swap space (in other words we page directly to the file system) the concept of virtual memory has been expanded to include any random access device. So it consists of main memory , swap space, local and remote file systems. Note this does not imply that all objects residing on any random access device are currently in virtual memory (mapped to a virtual address).

Address Space – The notion of the address space being a simple vector of pages means that although the traditional concepts of the stack, text and data segments still exist, the user process is not confined to use his address space in this manner. In other words using the *mmap(2)* system call a process can put a text or data page anywhere within his defined address space. Also it is possible to have more than one text or data segment per process.

Copy–On–Write – when more than one process shares an object in virtual memory the changes made to the objects can be made visible to all the processes (shared), or just visible to the process making the modification (private). If the access is to be private, then the first write access to a page is intercepted. Instead of modifying the object a copy to a private page is done, and the copy is modified. The rest of the object can be continued to be shared.

Definitions (cont.)

Page – a page is a contiguous collection of bytes varying in length according to the supporting hardware. For the purposes of this class unless otherwise stated a page can be taken as 8 kilobytes (8 kb = 8,192 bytes).

Segment – a segment is a collection of pages. We will see at least three definitions of the term segment in this class.

Context – several processes can be sharing the CPUs resources at any given time. Each process has its own virtual address space, its own definition of the general registers, its own text, data, and stack etc. This is a process's context.

Definitions (cont.)

Segment Definitions

1. As used by the memory management unit (MMU) a segment is a group of contiguous pages referred to by an entry in the segment table during address translation. See the *hat* (hardware address translation) and MMU (memory management unit) sections for more details about the virtual to physical translation tables.

2. To a programmer a segment (ex. text, data, or stack) is a collection of not necessarily contiguous pages of variable length associated with a given process.

3. In the 4.0 virtual memory (VM) system a segment refers to a group of pages treated in a like manner by a segment driver (see 4.0 VM section).

General Memory Information

Why do we use virtual memory?

- Size Limitations
  - Multi–tasking Environment
  - Shared Memory Objects
- Transparency

General Memory Information

Size Limitations – put simply we can't fit everything we want into main memory at the same time. Some programs have extremely large working sets (memory requirements). Also there are a number of processes running concurrently on most workstations even with a single user. These processes will eventually exhaust main memory and still need more space.

Transparency – each individual user process does not (and should not) have to be concerned with the management of the hardware and kernel structures that control physical memory resources.

MMU translations

How is a virtual address translated to a physical address?

- Memory is divided into segments and the segments are subdivided into pages.

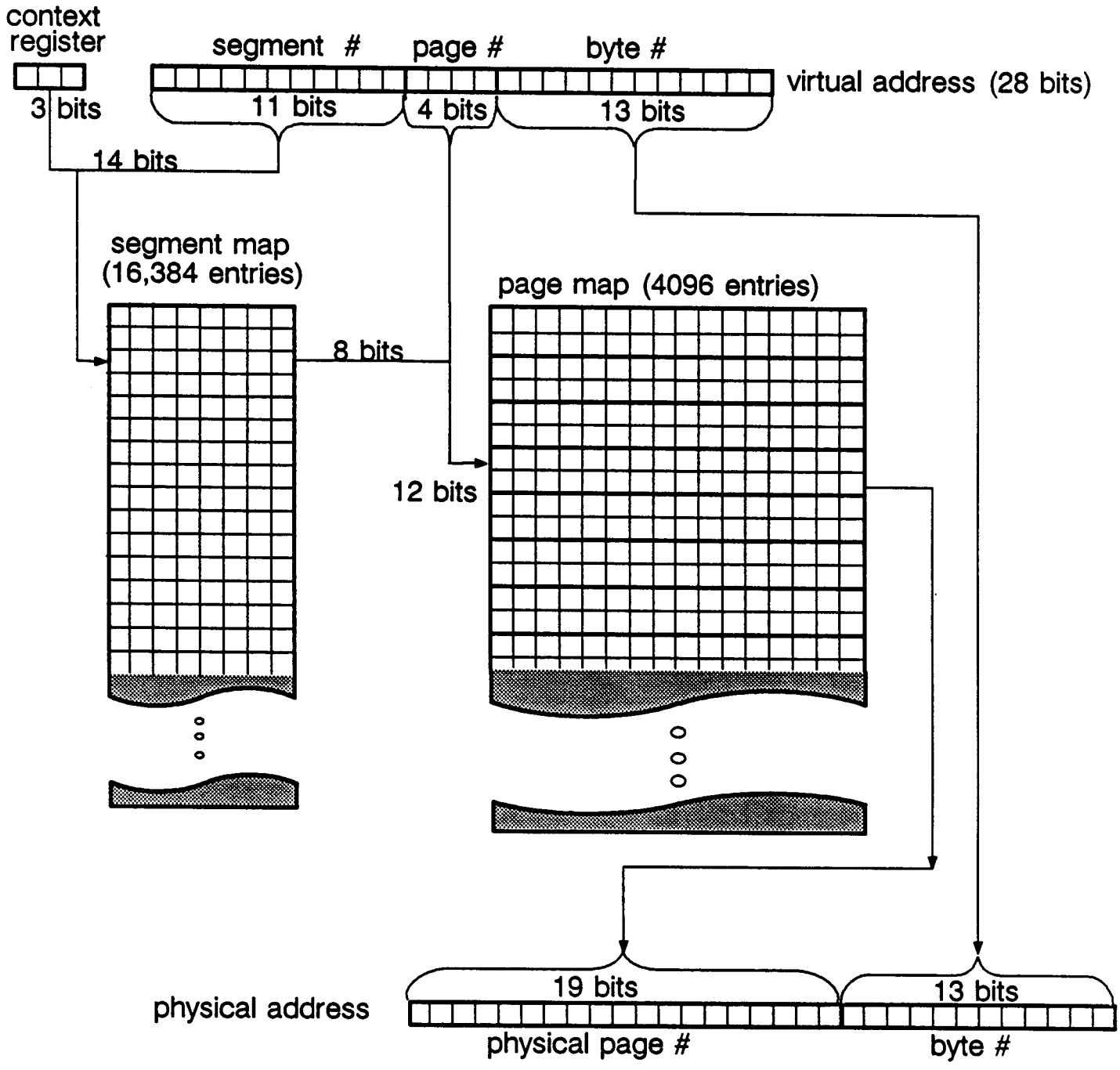
- The segments are addressed in a segment table, each entry points to a group of page table entries.

- A page table entry contains information about one physical page such as its physical location, or whether its been modified.

- A cache is used to speed up the address translation so that multiple accesses to main memory are not necessary.

MMU translations

MMU Translation for Sun-3s



MMU Translation for Sun–3s

First the MMU determines which process is making the translation by examining the context register. On the Sun–3 there are 8 contexts.

Next bits 27 through 17 from the virtual address are used to index into a portion of memory called a segment.

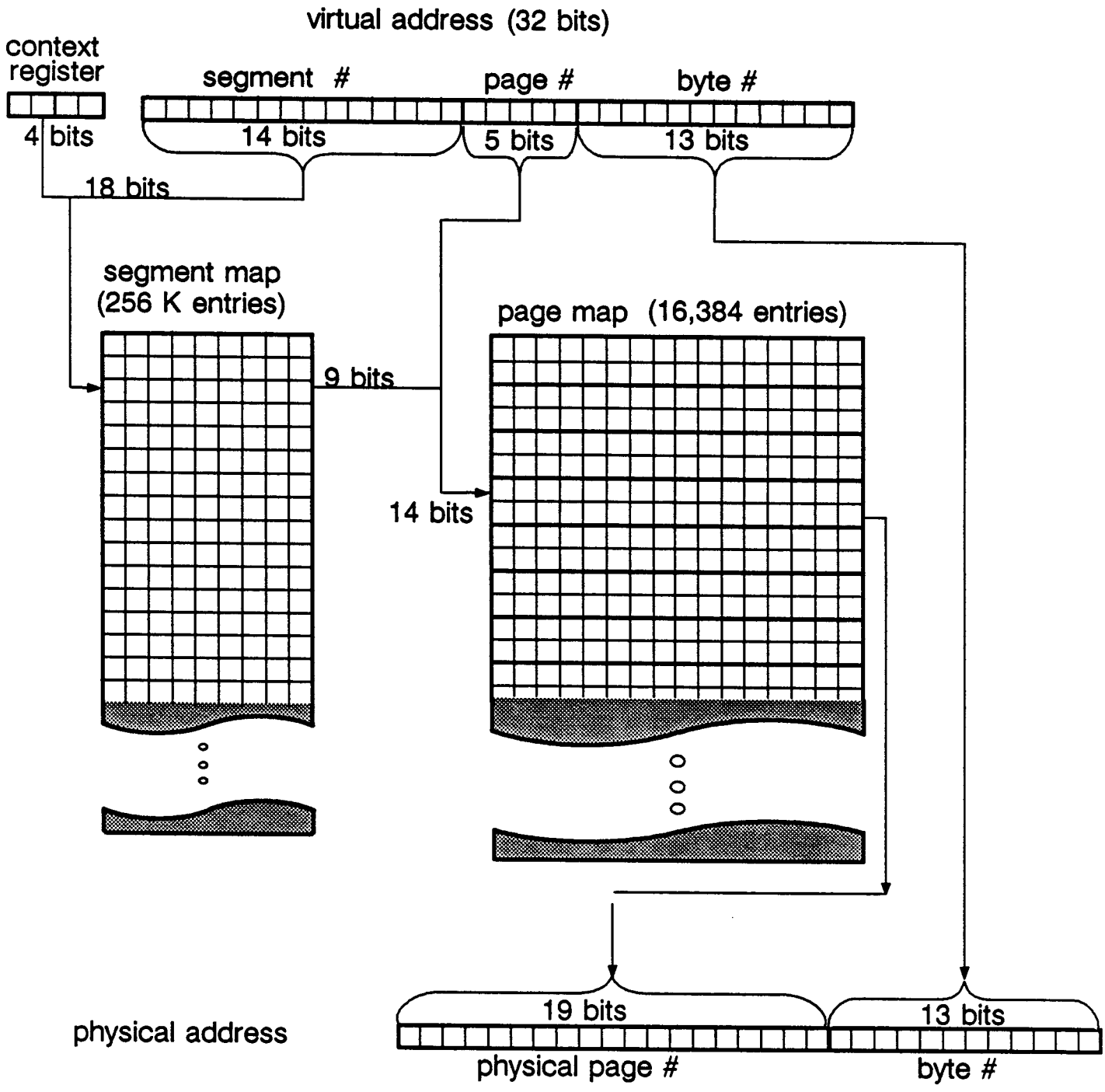
8 bits from the segment table entry points to a group of 16 pages referred to as a Page Map Entry Group.

Bits 16 through 13 from the virtual address are used to index into the page map group to a particular page map entry.

The Page Map Entry tells us whether the entry is resident and what type of physical memory is being used to store the object such as main memory or swap space. It also indicates what physical page the virtual address is on.

Bits 12 through 0 give the byte offset within the page.

MMU Translation for Sun–4s



MMU Translation for Sun–4s

First the MMU determines which process is making the translation by examining the context register. On the Sun–4 there are 16 contexts.

Next bits 31 through 18 from the virtual address are used to index into a portion of memory called a segment.

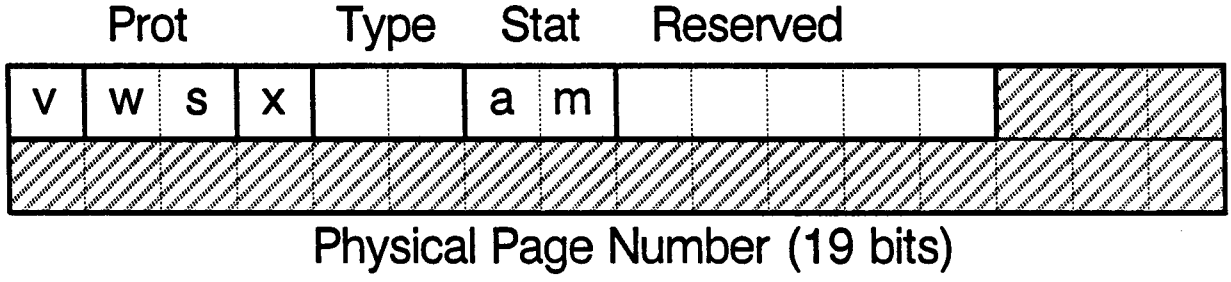
9 bits from the segment table entry points to a group of 32 pages referred to as a Page Map Entry Group.

Bits 17 through 13 from the virtual address are used to index into the page map group to a particular page map entry.

The Page Map Entry contains physical page frame number.

Bits 12 through 0 give the byte offset within the page.

Page Table Entries



V – the valid bit – indicates that this entry is valid (resident) and also enables read and execute rights to this page.

prot

W – the write bit – allows write access to this page.

S – the supervisor bit – this bit means that access to this page is restricted to the kernel mode.

X – the don't cache bit – this bit is used by the kernel to keep a page from being placed in cache because of possible aliasing problems.

Page Table Entries

stat

a – access bit – set when page is referenced.

m – modify bit – indicates page has been changed.

type

0 – type 0 – for main memory ranges from 0x00000000 to the maximum size according to how much main memory is on the system.

1 – type 1 – for on–board I/O devices such as Ethernet, SCC's, TOD, and the PROMS. The address space allocated for type 1 memory is from 0xF0000000 to 0xEF000000.

2 – type 2 – for 16 data bit VME devices. The physical address may use 16, 24, or 32 bits on the physical address bus.

For type 2 VME devices with 32 bit addresses the address range is from 0x00000000 to 0xFFFFFFFF. The 24 address bit VME is taken from the top of the 32 address bit range and goes from 0x00000000 to 0xFFFFFF. The 16 address bit VME is taken from the top of the 24 address bit range and goes from 0x00000000 to 0xFFFF.

3 – type 3 – for 32 bit VME devices. The range is from 0x00000000 to 0xFFFFFFFF.

Physical Page Number – the 19 high order bits of the physical address.

Cache

Cache is a high speed memory buffer between the CPU and the MMU.

Cache is not used as the exclusive memory on the system because:

- **Cache is expensive**
- **Cache takes up more physical space**
- **Cache consumes more power**
- **There is a diminishing point of returns**

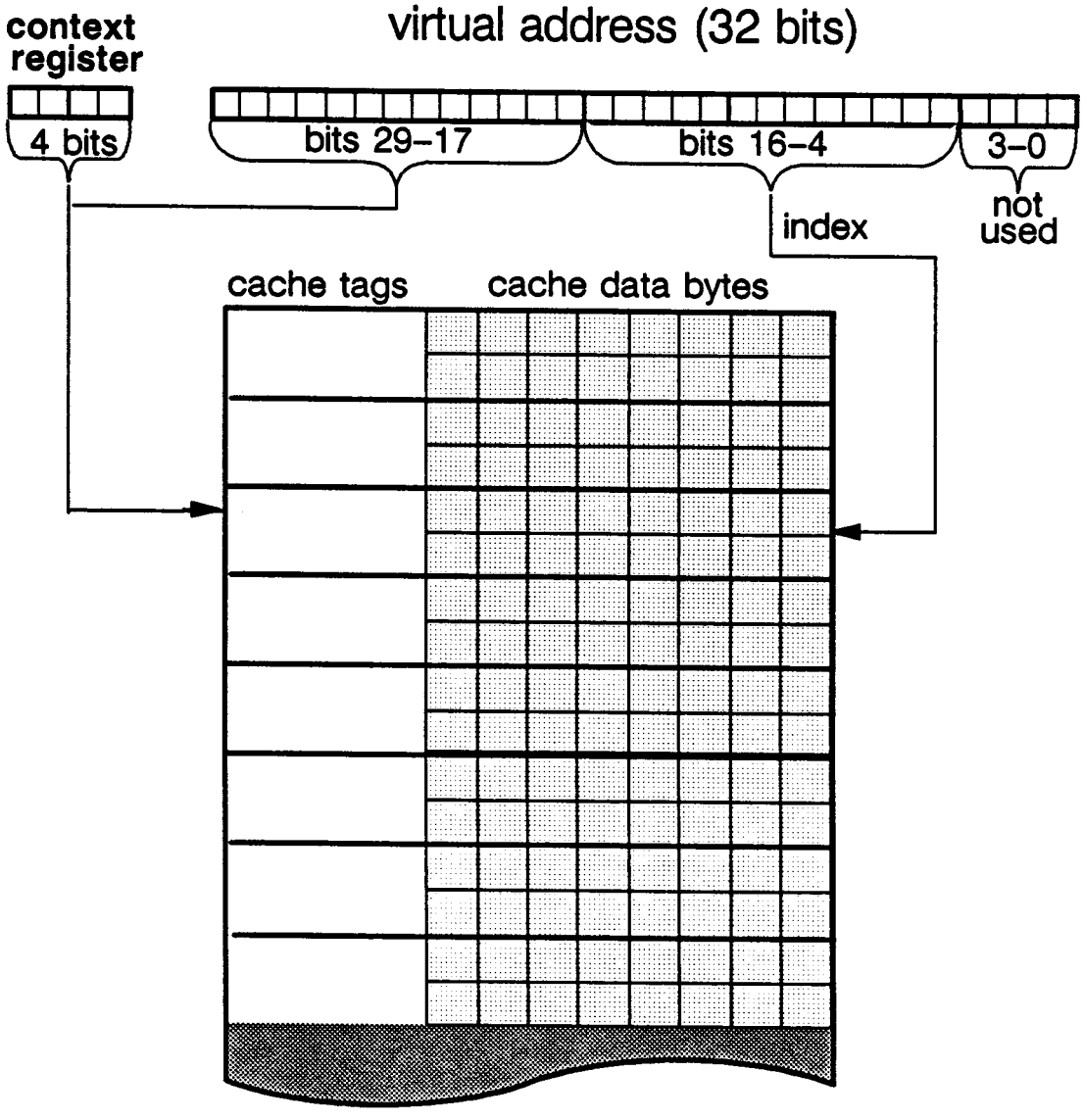
Cache

Cache uses a different type of circuitry than main memory. The type of chip is a Static Random Access Memory (SRAM) type. The type of chip used on main memory is Dynamic Random Access Memory (DRAM). The access time for SRAM chips is in the order of 10–50 nano seconds whereas the access time for DRAM chips is in the order of hundreds of nano seconds.

The internal nature of the SRAM chip (non capacitive) means it is much faster. The density (amount of data that can be stored on the chip) of SRAM is much less. The amount of current drawn by the chip is greater than that necessary to power DRAM chips.

Diminishing rate of returns means that by increasing the size of cache we don't get a corresponding increase in the cache hit rate. For example on a 64KB cache the cache hit rate (the percentage of times we find the target data in cache) may be in the 90 – 95% range. On a 128KB cache the hit rate may increase to only the 95 –98% range. Of course this is very application dependent. In general the cache hit rate will be influenced by the size of cache, the amount of data read into cache on each memory read (on Sun–3 and Sun–4s each cache block is 16 bytes), and the locality of reference.

Cache (cont.)


context register
4 bits
virtual address (32 bits)
bits 29–17
bits 16–4
3–0
index
not used
cache tags
cache data bytes

Cache (cont.)

This diagram illustrates the cache on Sun–4/200's. The data portion of the cache is 128KB in size. Since the cache fits between the CPU and the MMU it is indexed and identified by virtual addresses, supplied by the CPU or DVMA.

The cache tags serve to identify which process and virtual address range the data came from. When a cache entry is originally filled bits 17–29 of the virtual address and the context number is placed in the cache tag. The cache tag also contains a valid bit for the data and tag in the block, a modified bit if the block is dirty, and the write allowed and supervisor access bits from the MMU entry.

Since the cache is 128KB long and each entry or block consists of 16 bytes there are a total of 8,192 cache entries. Therefore 13 bits (bits 4–16 of the virtual address) are needed to index into the appropriate cache entry.

Bits 4–16 are used to index to a cache entry and then the context number and bits 17–29 are compared to the contents of the cache tag. If they match, it is a cache hit and the data belongs to that process and that virtual address range. Otherwise it is a cache miss and the data has to be retrieved from memory or backing store.

Cache (cont.)

Sun uses two types of cache.

- Write Through Cache – always initiates a write to main memory at the same time it writes to cache.

- Write Back Cache – writes only to cache and then back to main memory only when:

  - A cache write is about to overwrite a modified cache entry that hasn't been saved yet.

  - A context switch is to take place. Then all references to user area pages are flushed.

  - A context switch is to take place and the context is being freed (probably for use by another processs).

Cache (cont.)

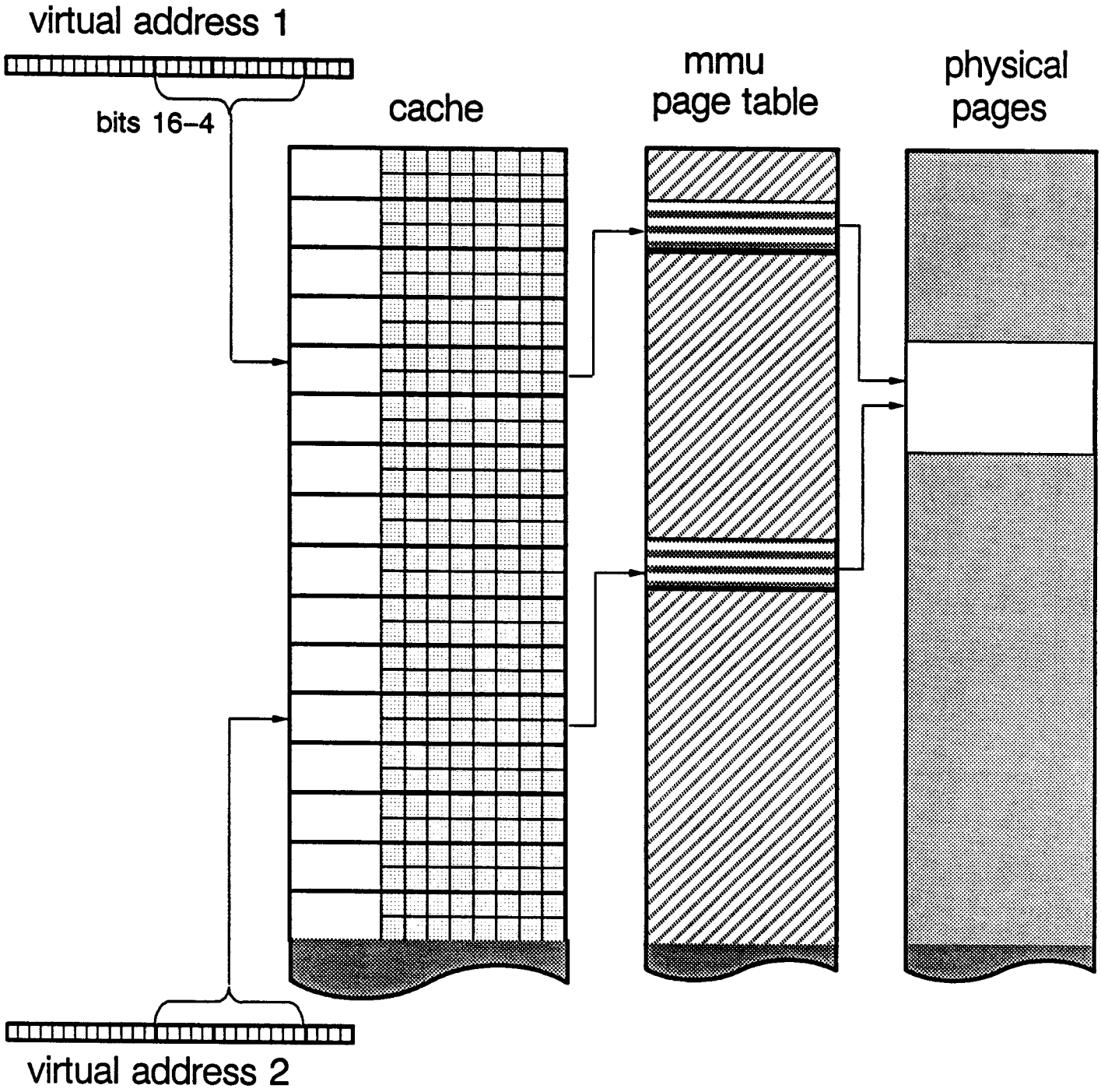
The advantage to a write through cache is that it ensures that the cache contents are consistent with that of main memory. The disadvantage is before one write can take place we may have to wait for the previous write to complete. Also some of the writes may not have needed to go to main memory if the data in cache changed more than once.

The advantage to a write back cache is that we don't have to wait for previous cache writes to update memory before we write to cache. These types of caches are used on the Sun–3/200 and Sun–4/200 series machines. The disadvantage to the write back cache is that either the cache or the kernel has to make sure the contents of cache is consistent with main memory.

If the virtual address bits that are used to index into cache are the same in two different virtual addresses then one cache entry may overwrite another. If the old entry has been modified it must be flushed to main memory.

If a context switch or swap out is about to take place all cache entries for that process (as identified in the context bits from the cache tag) have to be flushed to main memory.

Cache Coherency



Cache Coherency

It is possible for more than one process or virtual address to map to the same physical page. If the bits that index into cache are not the same then there will be more than one cache entry that corresponds to a given physical address. It therefore can be possible for one of the cache entries to be modified without modifying the contents of the other cache entry. The entry that is flushed out last will end up being the contents of main memory. This can lead to unpredictable results for the process that was using the other cache entry. This is known as cache aliasing and the kernel (or device driver) is responsible for ensuring cache coherency. This can be achieved by using the *x (don't cache)* bit in the page table entry, or forcing alignment so the virtual addresses will always use the same cache entry.

Reasons for implementing 4.0 VM

- Traditionally UNIX provided little support for sharing memory and no support for a file mapping facility.
- Sun wanted to provide a platform for shared libraries
- To be able to support a variety of interfaces including our own implementation of 4.2 BSD and AT&T System V Unix.
- Sun's previous implementation (which was modeled towards the VAX hardware architechture) was too constraining.
- The amount of work to revamp the old VM to meet the above goals was prohibitive.
- There were concerns for the long term maintainability of the old VM system.

Reasons for implementing 4.0 VM

File mapping has been partially implemented in our 4.2 BSD derived Unix, it is unsupported in AT&T's System V Unix.

If the right changes were made to the kernel many of the goals could be implemented as libraries (for example the support for shared libraries).

There has been no movement in the computer market in general towards a memory management scheme that encompassed all the afore mentioned goals.

Also the new VM was developed with an eye for future enhancements such as lightweight processes, multiprocessor environments, as well as advances in hardware technologies.

Compatibility Constraints

The New VM system:

- had to be binary compatible
- did not have to be source compatible
- should provide flexibility in the use of the user address space
- should give the system control of some of the details concerning user address space

Compatibility Constraints

Current executable objects *(a.out)* files still had to run. The partially implemented *mmap(2)* system call still had to function.

Source level incompatibility means the source may have to be changed the next time it is compiled.

A flexible use of user address space makes it possible to use software abstractions that make use of large sparsely populated address space. This is how segments are utilized in many software architectures.

On systems that implement a virtual cache the system should be able to control virtual address assignment in order to enhance performance. Also page level protections are necessary to implement an efficient copy–on–write mechanism.

Features of the 4.0 VM System

- As of 4.0 the virtual memory system operates on pages rather than segments in order to accommodate efficient copy on write facilities.

- A process's address space is mapped onto objects in virtual memory.

- Each process in virtual memory is backed by some physical storage (i.e. disk or main memory).

- A page of a process's address space can only map to one physical object .

- A physical page in any given object may map to a multitude of process address space pages.

Features of the 4.0 VM System

By using a page oriented system, a separate copy of a page for a process is only necessary for the page being modified and only at the time it is modified. Of course this is only necessary if the page is mapped private rather than shared. Other systems require whole segments (groups of like pages such as stack or data) to be copied and allocated swap space for a process at execution startup time.

- Ordered by physical page number
- Hashed by page identity
- Manages Cache for virtual memory

The page structure keeps track of pages mapped from file system objects to virtual addresses.

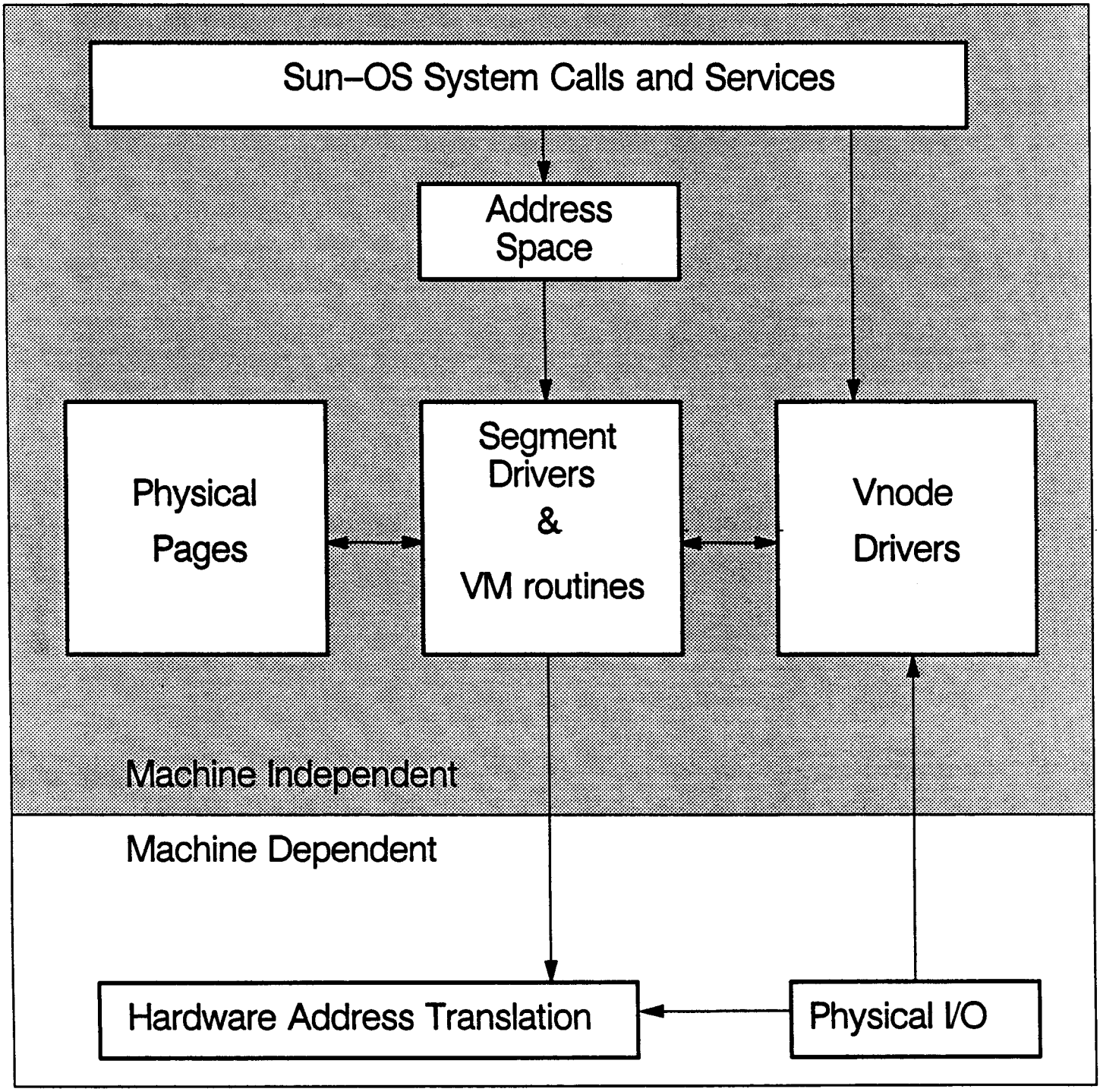
The VM system and object managers use this structure to manage main memory as a cache to anything accessed in virtual memory.

The page identity is determined from the vnode (**p_vnode*) and offset (*p_offset*) fields in the page structure.

A hash lookup on the vnode, offset pair is used to locate the page by name.

All pages for a vnode are on a doubly-linked list. The forward pointer is p_vpnext and the reverse pointer is p_vpprev.

The layers of the VM system



The layers of the VM system

Unix System Calls and Services – the user interface to the kernel's virtual memory management structures and procedures. The predominant system call is *mmap* (the read and write calls are implemented through an internal version of mmap).

Address Space – state information about a process's address space is maintained through a linked list of contiguous address mappings arranged by virtual address.

Segment Drivers – the address space mappings are termed segments and operations on segments are implemented through a set of segment drivers. Memory is allocated to a given device driver according to the use of that particular page of memory (ex. kernel pages, vnode pages).

Vnode Drivers – the vnode object managers are responsible for creating and initializing a requested page which may include interfacing with the physical I/O layer to obtain the data.

Physical I/O – is the machine dependent layer which manages backing store memory according to the physical characteristics of the peripheral device.

Hardware Address Translation – (hat) layer is the machine dependent code that manages the hardware translations to pages in the machine's MMU.

Sun–OS System Calls and Services

The interface to the VM system from the system call layer primarily concerns these calls:

- *mmap(2)*
- *msync(2)*
- *munmap(2)*
- *mprotect(2)*
- *read(2)*
- *write(2)*
- *fork(2)* and *vfork(2)*

Sun–OS System Calls and Services

The system call layer's principle interface to the VM system as of Sun–OS 4.0 is the *mmap(2)* system call. The following section reviews the *mmap(2), the msync(2),* and the *munmap(2)* calls and their arguments.

The nature of the *fork(2)* call changed as a result of changes made to VM system and these changes are discussed in the Process Management Module.

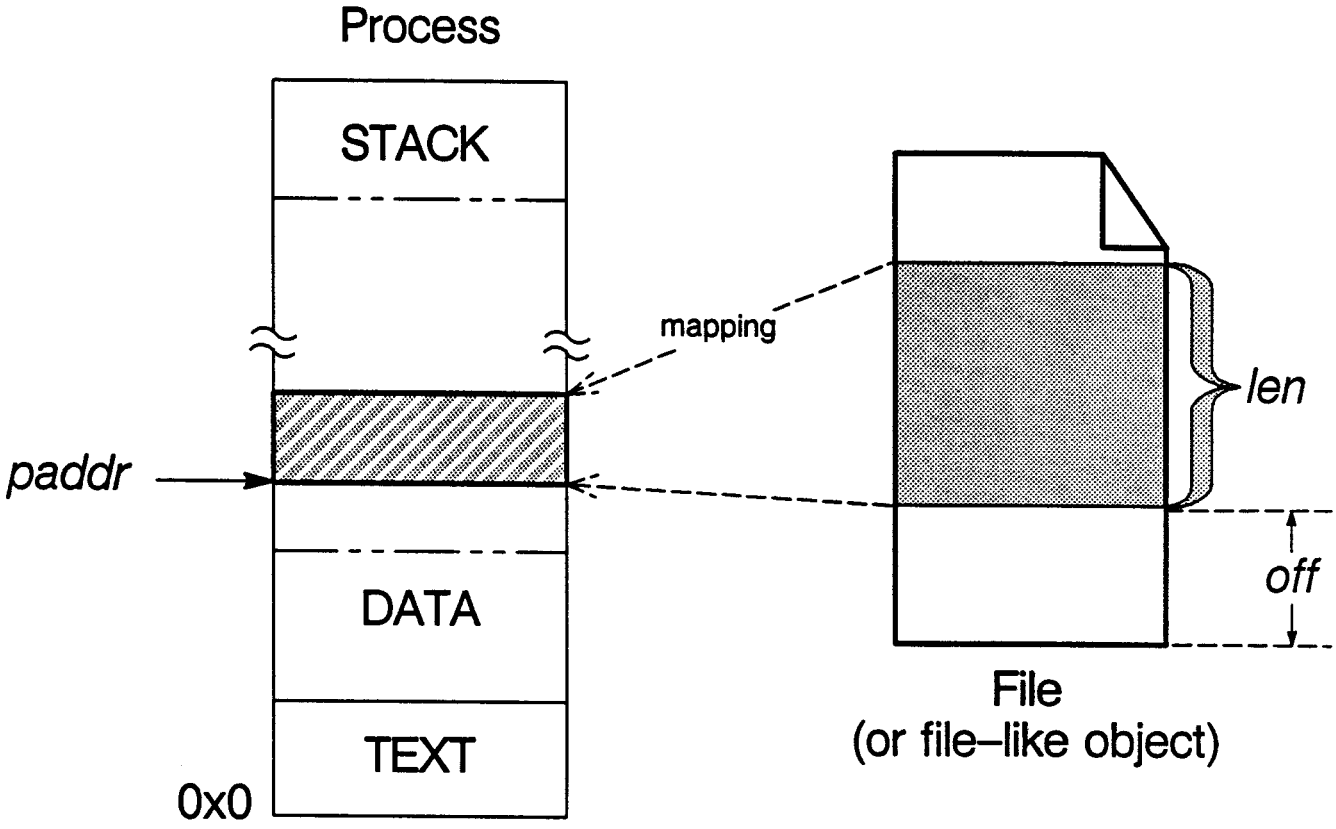
The *read(2)* and *write(2)* system calls are now implemented with an internal version of the *mmap* call and is discussed in the Vnode Segment Driver section in this module.

mmap(2)

- Function for mapping pages of memory – *mmap(2)*:

```
#include <sys/types.h>
#include <sys/mman.h>

paddr = mmap(addr, len, prot, flags, fd, off)
caddr_t addr, paddr;
int     len, prot, flags, fd;
off_t   off;
```



mmap(2)

mmap() establishes a mapping between the process's address space at an address *paddr* for *len* bytes to the memory object represented by *fd* at *off* for *len* bytes. *off* must fall on a page boundary (*getpagesize(2)*). Note that in 4.0 you don't need to have allocated any memory at position *addr*, although it is advised that you do so. In 3.5 you MUST have allocated space for *mmap()* to map to. If successful, the return value of *mmap()* will be *paddr*. If *addr* is 0, the system will choose an appropriate address *paddr*. In any event only the return value of *mmap()* should be used.

Memory mapping can be useful if, say, you have an array of structures in a file that you would like to access via pointers without having to use *read* and *write*. Or, you may have an array of structures in memory and out on disk, and you would like to be able to use one function to do all the work. The mapped object will only be loaded into memory on an as–needed basis.

The argument *prot* determines whether read, execute, write, or some combination of accesses are permitted to the mapped pages. Legal options include (from <*sys/mman.h*>):

```
#define  PROT_READ       0x4    /* page can be read */
#define  PROT_WRITE      0x2    /* page can be written */
#define  PROT_EXECUTE    0x1    /* page can be executed */
#define  PROT_NONE       0x0    /* page cannot be accessed */
```

The argument *flags* tells a little bit more about the handling of the mapped pages, such as whether they can be shared among processes, each process has a private copy, and so on. Note that *MAP_PRIVATE* actually implies copy–on–write, so each page is shared until an explicit write reference is made to it:

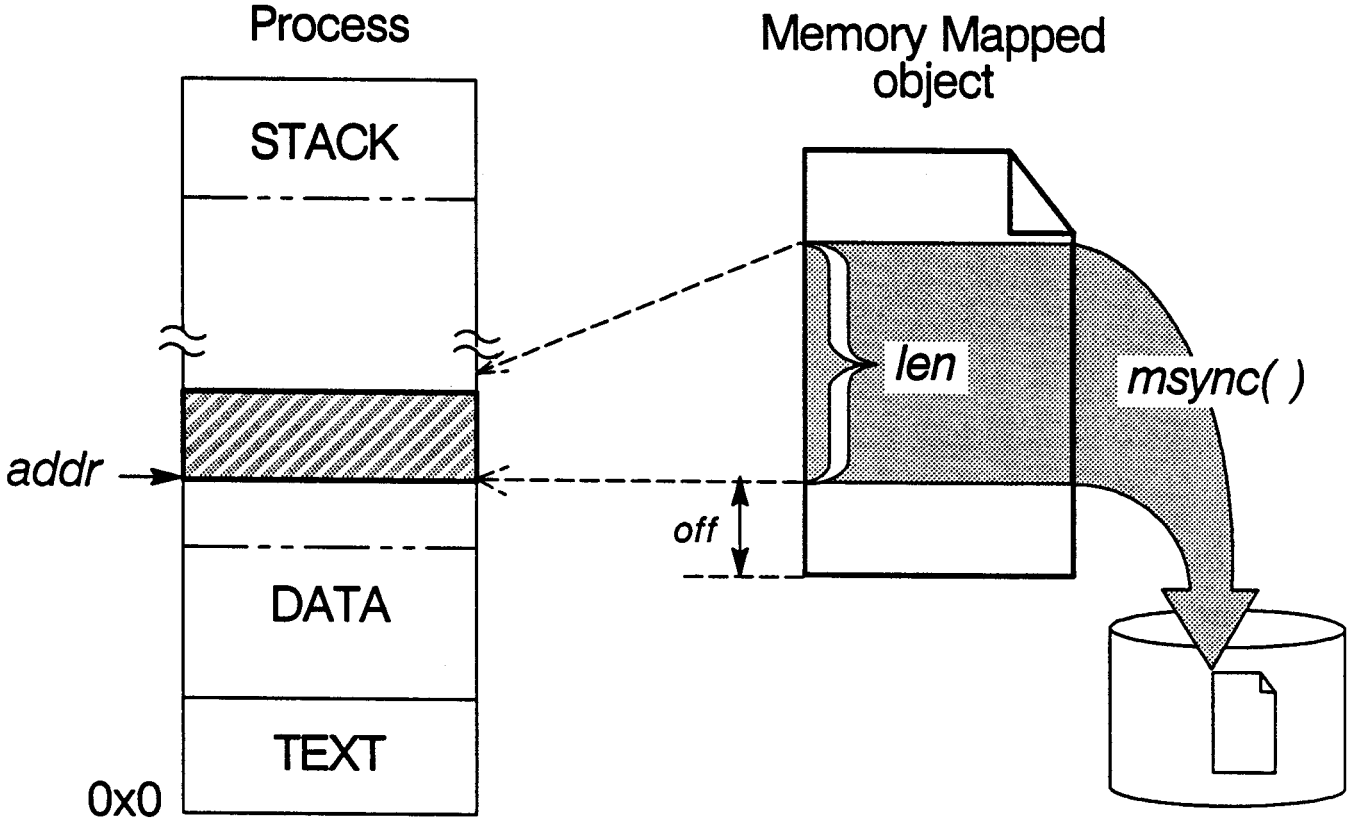
```
#define  MAP_SHARED      1      /* share changes */
#define  MAP_PRIVATE     2      /* changes are private */
```

msync(2)

- Synchronize memory with physical storage – *msync(2)*:

```
#include <sys/types.h>
#include <sys/mman.h>

int msync(addr, len, flags)
caddr_t addr;
int len, flags;
```



msync(2)

The system call *msync()* is used by programs or applications which require that a given mapped memory object to be maintained in some "known" state.

msync() is used to write out to disk (permanent storage locations) all modified copies of memory pages associated with a memory mapped object. The range of pages to be written to disk is the span *addr* to *(addr + len)*.

The following flags (found in <*sys/mman.h*>) are used to control the behavior of *msync()*:

MS_ASYNC . . . . . return from *msync()* call immediately
MS_INVALIDATE . . . . . flag in–memory mapping as being invalid

Normally, *msync()* will block until all I/O operations have been completed by the system. By using the *MS_ASYNC* flag, *msync()* will return immediately once all I/O operations have been *scheduled*.

msync() also allows cached copies of in–memory data to be flagged invalid by using the *MS_INVALIDATE* flag. This forces the system to retrieve data from permanent storage upon the next reference to the object (read operation).

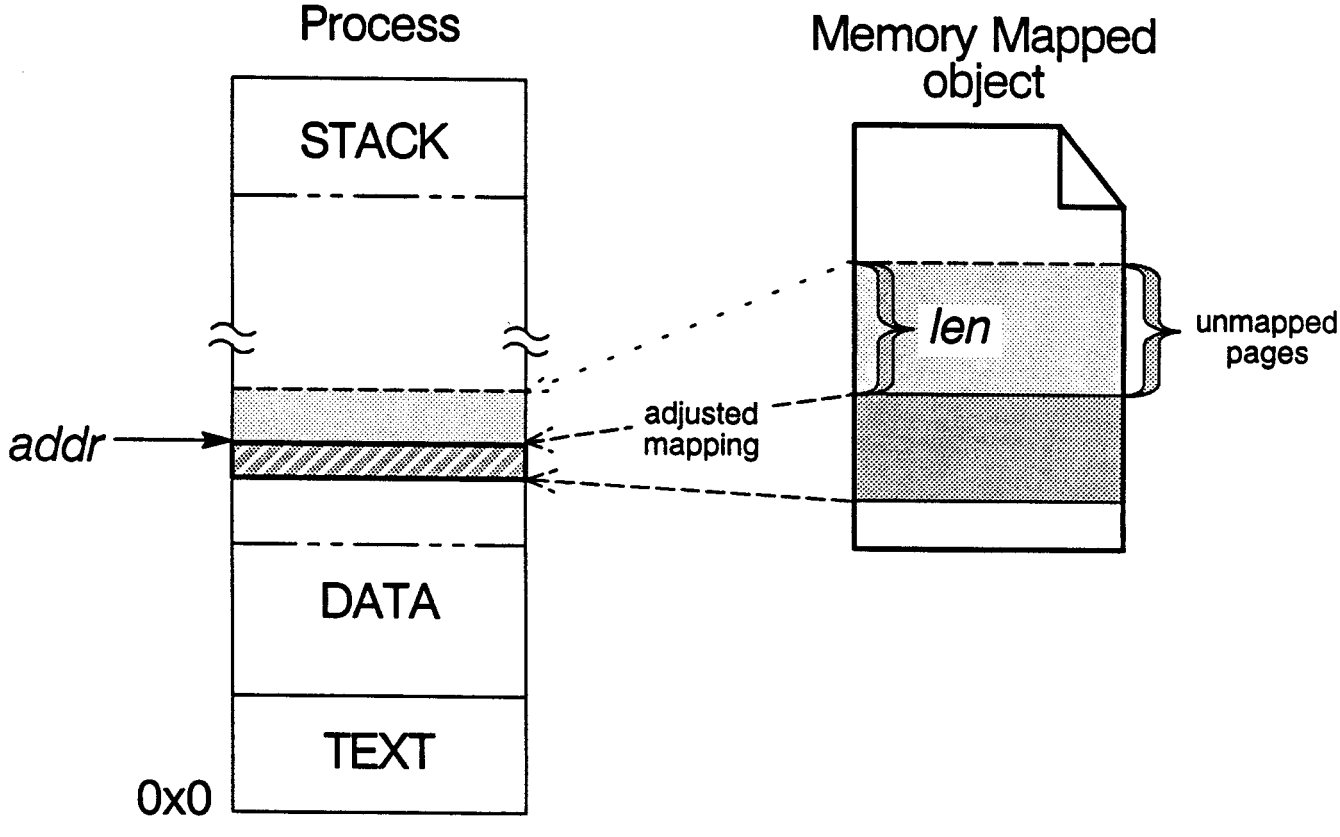
msync() returns a zero upon a successful completion; otherwise a –1.

munmap(2)

- Unmap pages of memory – *munmap(2)*:

```
#include <sys/types.h>
#include <sys/mman.h>

int munmap(addr, len)
caddr_t addr;
int  len;
```



munmap(2)

The system call *munmap()* unmaps pages of a memory mapped object. After the pages specified by the range *addr* to (*addr* + *len*) have been unmapped, any further reference to those pages will cause a Segment Violation signal (*SIGSEGV*) to be sent to the process making the reference, unless those pages are considered part of the process's *stack* or *data* segments.

The parameter *addr* must be a multiple of the system memory page size (*getpagesize(2)*).

The range of memory specified by *addr* to (*addr* + *len*) must fall inside the valid address space for the calling process, and must *also* be a multiple of the system page size.

Upon a successful return, the value of *munmap()* is zero; otherwise a –1.

The Address Space

The procedural interface to the address space falls into two categories. The first class performs operations that manipulate the entire address space such as:

- Space Allocation (*as_alloc()*)
- Space Destruction (*as_free()*)
- Space Duplication (*as_dup()*)
- Swap Out (*as_swapout()*)

The Address Space

On Sun–3s and Sun–4s the kernel is mapped into the top of the users address space. On Sun–2s the kernel lives in its own address space. A process views its address space as a simple vector of pages.

The Address Space (cont.)

The second class performs operations that manipulate a given range of virtual addresses such as:

- Fault Processing (*as_fault()*)
- Verifying Protections (*as_checkprot()*) (*as_setprot()*)
- Synchronizing the Address Space with its Physical Backing (*as_sync()*)
- Mapping and Unmapping Objects (*as_map()*) and (*as_unmap()*)
- Retrieving Attributes of Mapped Objects (*as_memory()*)

The Address Space (cont.)

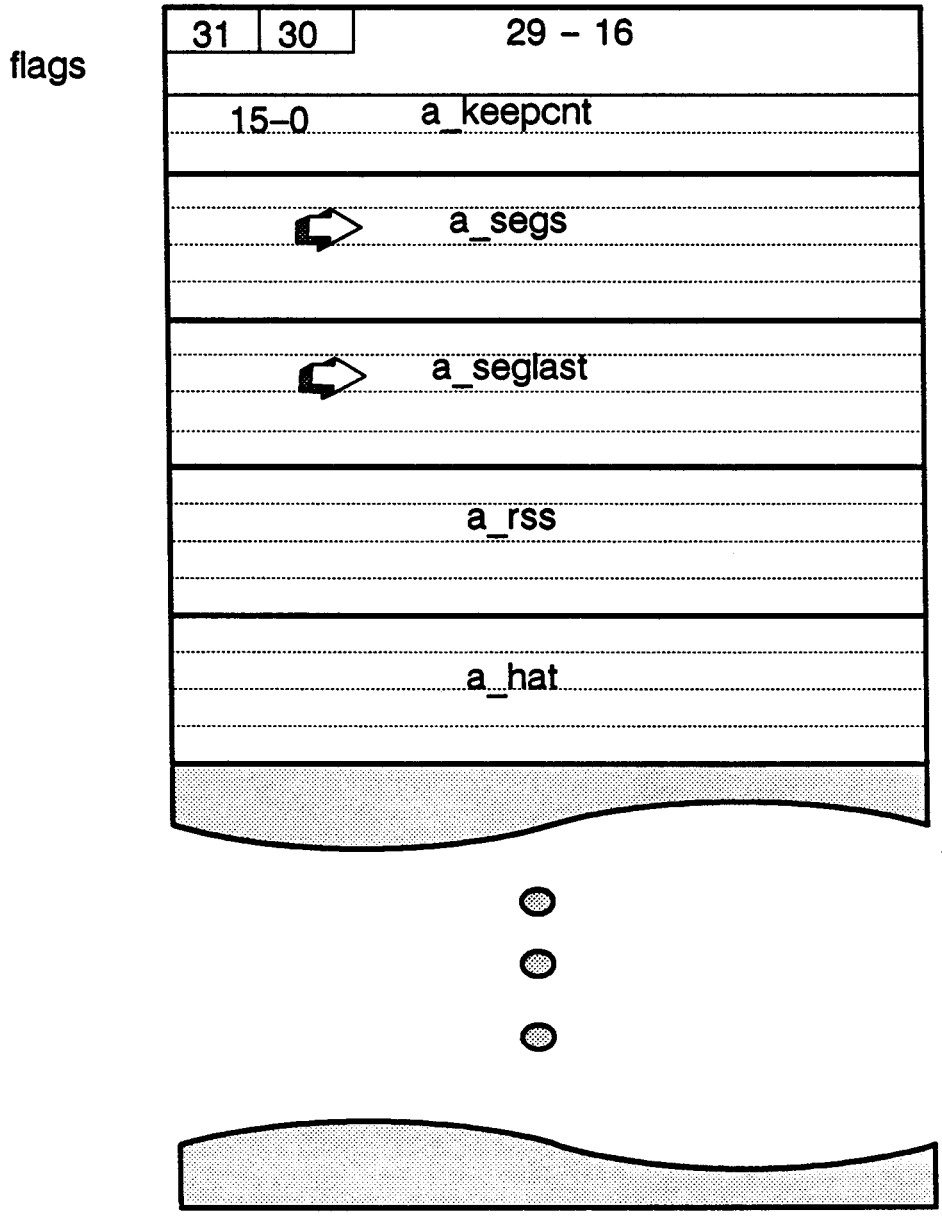
A process's address space is maintained and manipulated through the *as* structure (see next page).

An address space consists of a doubly linked list of segments sorted by virtual address and pointed to by the *as structure (a_segs).*

SunOS maintains an *as* structure to manage the kernel's address space. This as structure is shared by all SunOS processes when operating in the kernel mode.

Each process has an *as* structure to manage its user address space when operating in user mode. Just as every Unix process shares the address space of the kernel when executing in kernel mode, several processes can share a user's address space. These multiple levels of control on a given address space (sharing the *as* structure) are the basis for lightweight processes.

as Structure



as Structure

flags –

bit 31 – a_lock –as structure is locked for use.
bit 30 – a_want – an indication that a process needs to use this *as structure* when available.

a_keepcnt – keeps track of the number of processes sharing this address space.

a_segs – pointer to the linked list of segments that make up this address space.

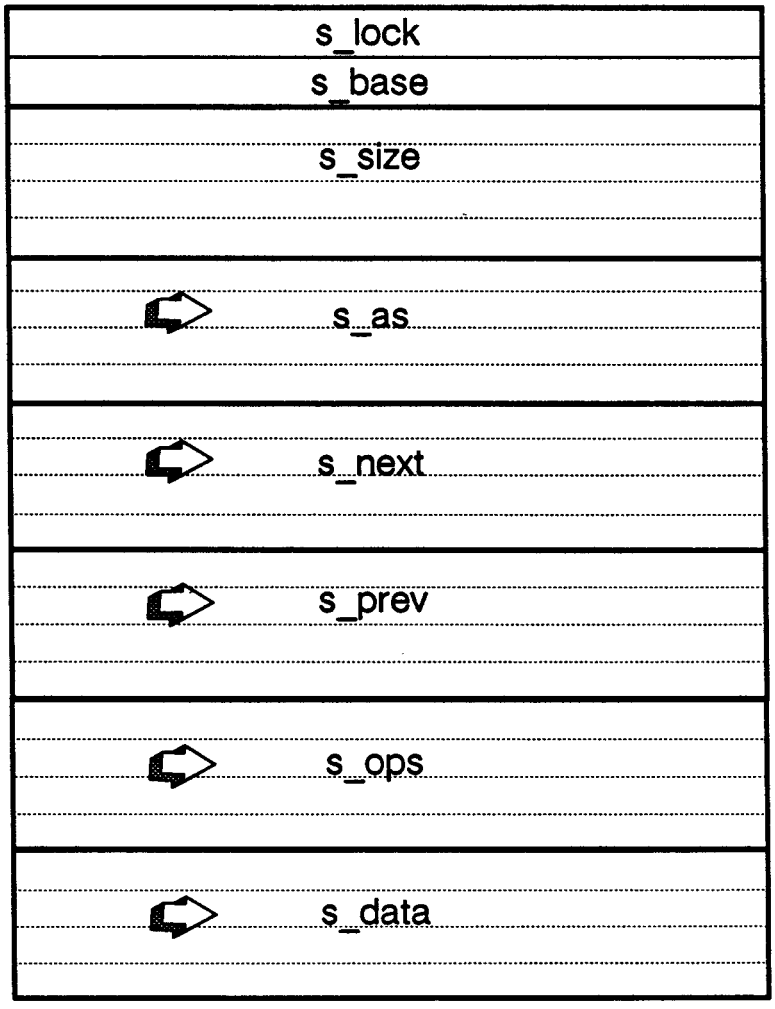
a_seglast – to speed access time during a virtual address translation a 'hint' *(a_seglast)* is used to point to the last segment found each time a virtual address is translated. This is used as the starting point for the segment search.

a_rss – the current copy of the resident set size (in pages).

a_hat – the hardware address translation structure for this address space is used by the hat layer (machine dependent) portion of the VM system to describe the MMU mappings. The *hat structure* is not accessed by the machine independent layers of the VM system.

Seg Structure

A process's address space contains a set of segments. A segment is divided into public and private data. The *seg* structure contains the public information.



Seg Structure

s_lock – lock to prevent races.

s_base – the base virtual address.

s_size – the size of the segment in bytes.

s_as – pointer to the *as* (address space) that contains this segment.

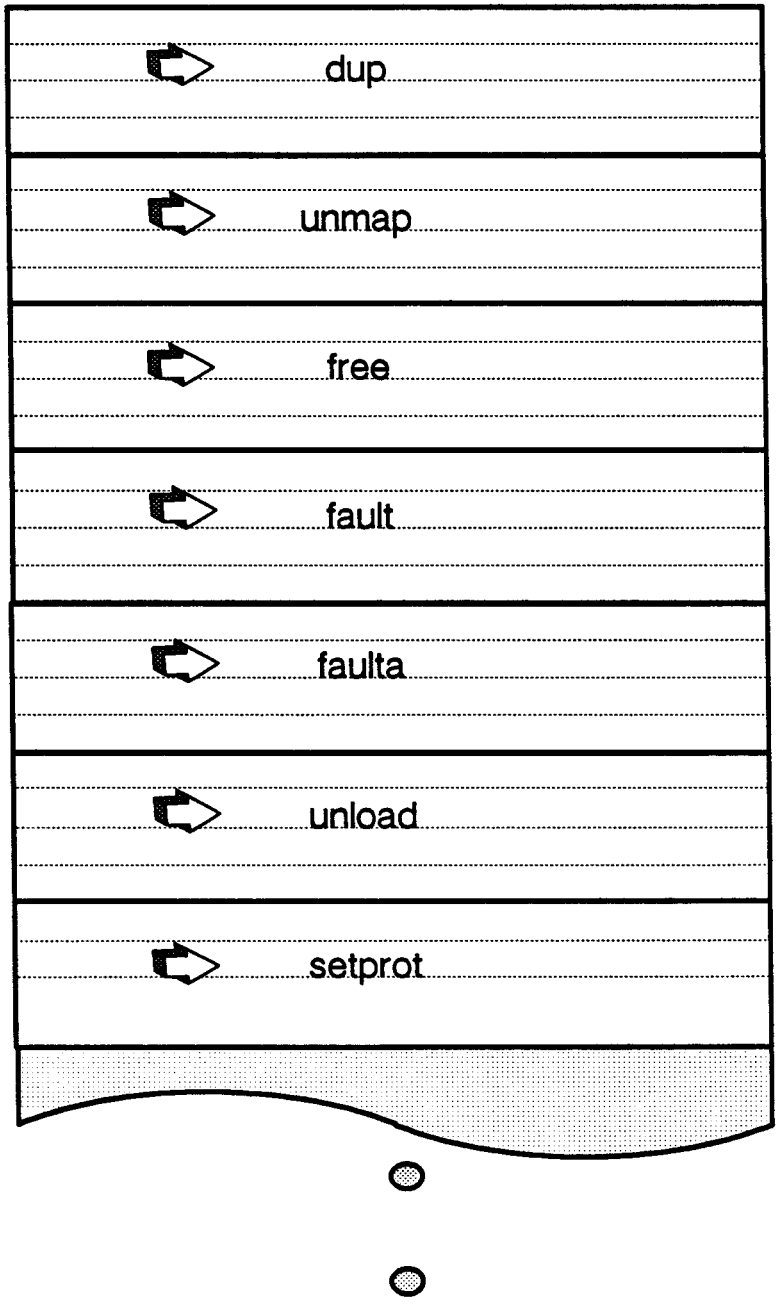
s_next – a pointer to the next segment in this address space.

s_prev – a pointer to the previous segment in this address space.

s_ops – pointer to a seg_ops structure which contains a set of routines to manipulate pages grouped as a segment. The next few pages briefly describe the routine's function and the parameters passed to these routines.

s_data – pointer to the private data structure.

Seg_ops Structure



Seg_ops Structure

dup – pointer to the routine that duplicates a segment; the parameters passed this routine are *seg*, a pointer to the source segment's seg structure, and *newsegp,* a pointer to the address of where the new segment's seg structure should go.

unmap – a pointer to the routine that unmaps a portion of a segment. The parameters passed are *seg, addr*, and *len*. *unmap* unmaps *len* bytes starting at address *addr* in segment *seg*. It may have to split seg into two segments if the middle is unmapped.

free – a pointer to the routine to free a segment. It receives seg, a pointer to the segment to free.

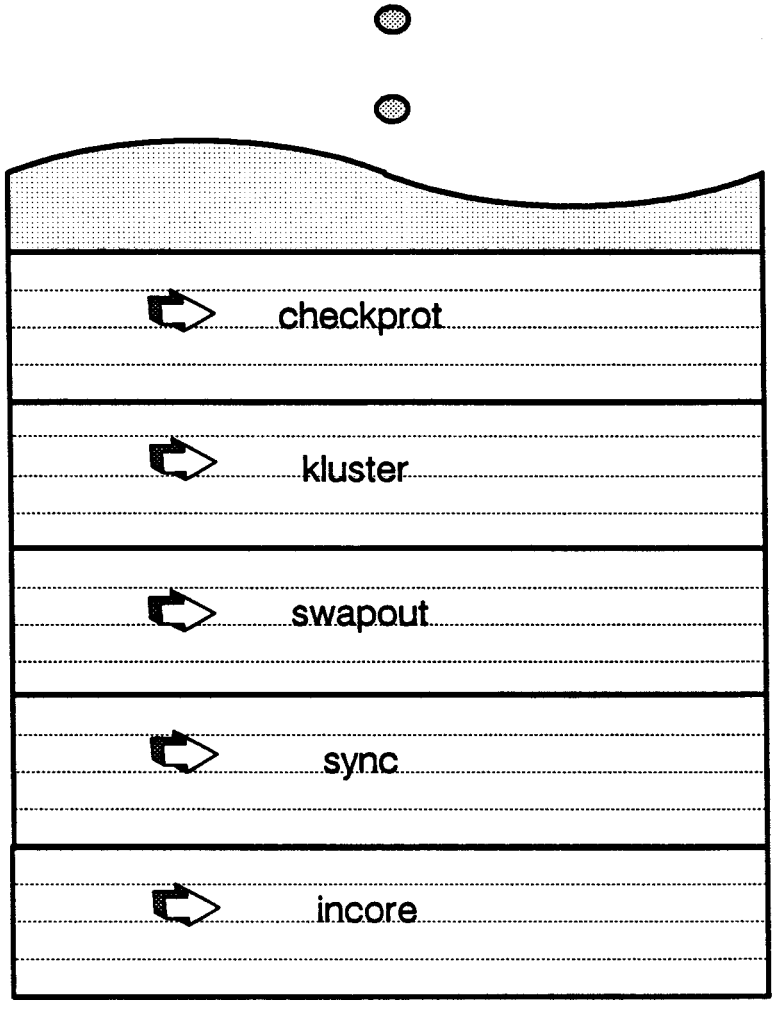
fault – a pointer to the routine to handle a page fault. *fault* receives *seg, addr, len, type,* and *rw*. It faults in pages starting at address *addr*, for *len* bytes. *type* indicates that a protection error occurred (set for copy–on–write), or that the pages should be locked/unlocked (F_SOFTLOCK, F_SOFTUNLOCK). *rw* indicates whether the pages will be read (SREAD) or written (SWRITE), from the point of view of memory.

faulta – a pointer to the routine to handle an asynchronous page fault (either to pagein or pageout). It receives *seg*, a pointer the the segment, and *addr,* the address to fault.

unload – a pointer to the routine to unload pages based on whether they've been referenced or modified. unload receives the segment address (*seg*), the address to unload (*addr*), and the referenced (*ref*) and modified (*mod*) bits.

setprot – a pointer to the routine to change protections on pages in the segment. It receives the address of the segment, *seg*, the address to start at (*addr*), the number of bytes to change (*len*), and the new protections (*prot*).

Seg_ops Structure (cont.)



Seg Structure (cont.)

checkprot – checkprot receives the segment (seg), the address (addr), the number of bytes (len) starting from that address, and the protection (prot) that you want to verify. It verifies that all the pages from addr to addr+len have protection prot.

kluster – this will take the segment (*seg*), at address (*addr*), and a *delta*. It will determine if it makes sense to do kluster/read ahead to *addr* + *delta* based on the mapping for *addr*.

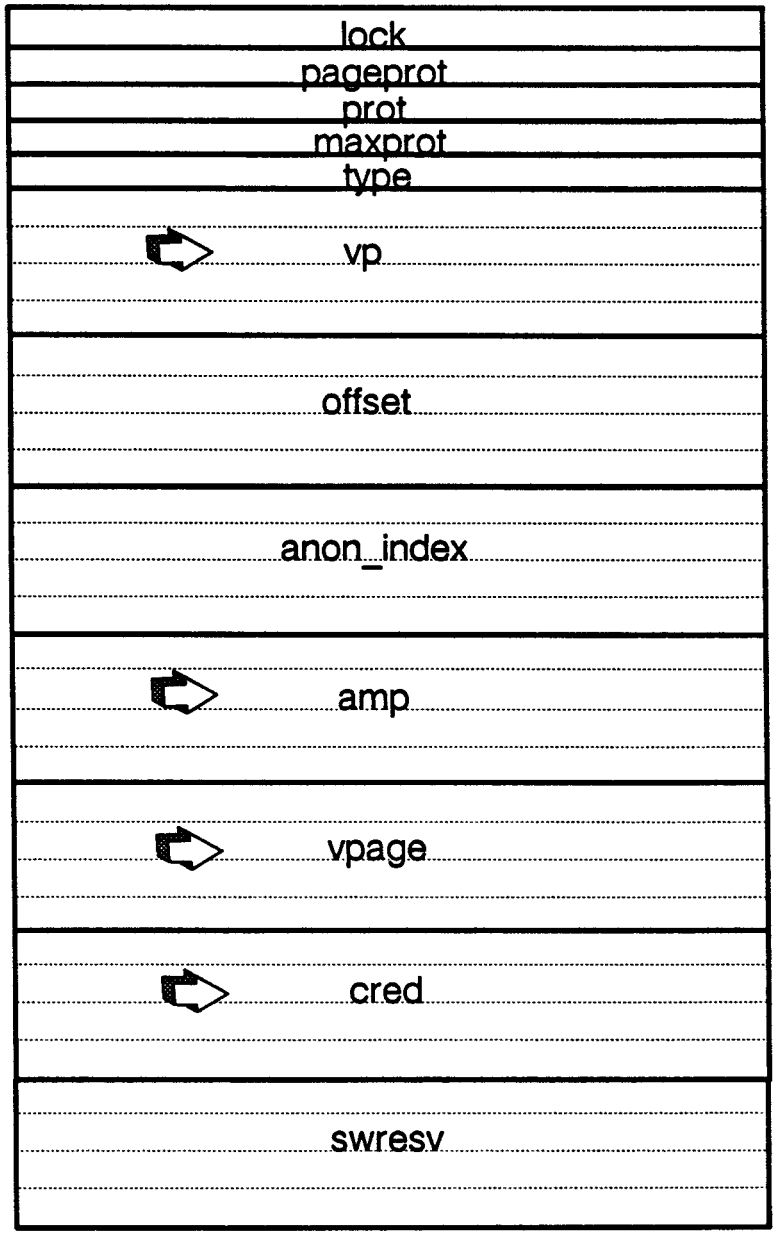
swapout – this routine only receives the address of the segment, *seg*. It must swap the pages of *seg* out to secondary storage.

sync – receives *seg, addr, len* and *flags*. *sync* will make sure that the bytes addr to *addr* + *len* in memory are synchronized with the actual object, such as the disk blocks they represent. *flags* corresponding to the flags given to the *msync* system call (i.e., whether to do the operation asynchronously, invalidate the data in memory upon return, etc.).

incore – receives the segment (*seg*), the address (*addr*) and the number of bytes (*len*), and an array of boolean result flags, one for each page (*vec*). It will determine if the corresponding pages we have mapped are actually in memory (in–core), or still on backing store.

Segvn_data Structure

The per segment mapping private data is managed by the seg_vn driver via the *segvn_data* structure.



Segvn_data Structure

lock – structure lock for exclusive access.

pageprot – set if page protections are present.

prot – current segment prot if page protections are 0.

maxprot – maximum segment protections (determined by open())

type – type of sharing (shared or private).

vp – pointer to vnode that segment mapping is to.

offset – starting offset of vnode for mapping.

anon_index – starting index into anon_map anon array

amp – pointer to anon share structure, if needed.

vpage – per–page information, if needed.

cred – pointer to ucred structure.

swresv – swap space reserved for this segment.

Types of Segment Drivers

- seg_vn – used to map regular files and anonymous memory.
- seg_map – transient mappings used during fault resolution on kernel page faults.
- seg_dev – mappings used for special files for devices (ex. frame buffers).
- seg_kmem – miscellaneous mappings for the kernel.

Types of Segment Drivers

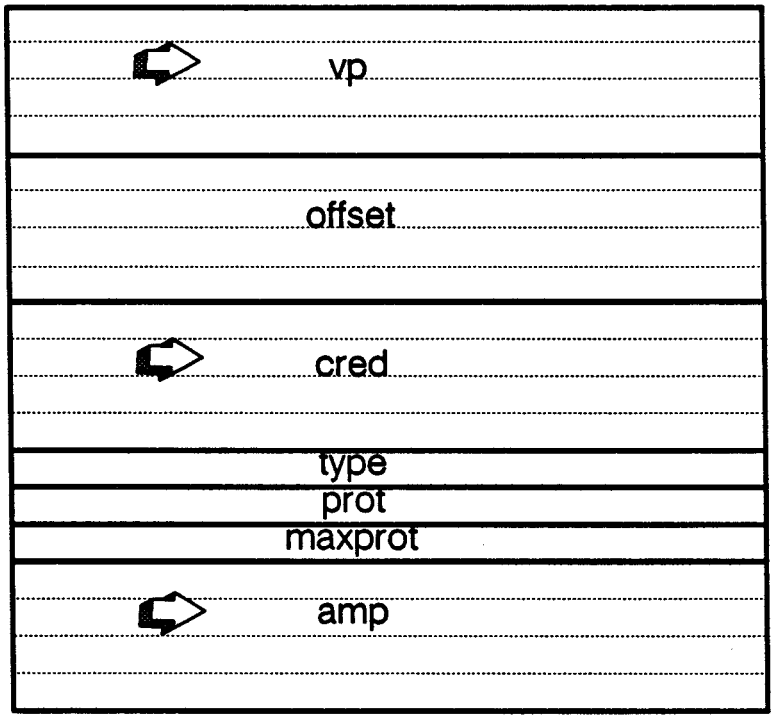
Segments are managed by a set of segment drivers according to the type of pages contained within the segment.

The segment drivers are free to coalesce adjacent segments of similar type pages. This speeds up the segment search for the address space *(as)* routines.

The segment drivers call the vnode object managers when a requested page results in a fault.

The Vnode Segment Driver

segvn_crargs structure



The Vnode Segment Driver

The vnode segment driver (*seg_vn*) is the most heavily used on the system. The *segvn_crargs* structure is passed to the *segvn_create* routine to create a vnode segment.

vp – a pointer to the vnode structure that defines the vnode from which this segment is being mapped.

offset – the starting offset within the vnode where the mapping for this segment began.

cred – a pointer to the credential structure for the user creating the segment. The credential structure contains real and effective user and group ids among other things (see ucred.h in appendix B).

type – the type of mapping (shared or private) requested for this segment.

prot – the protections set by *mmap(2)* on this segment (not to exceed maxprot).

maxprot – the maximum protections set on this segment based on permissions granted when the file was open. This applies to a segment with a shared mapping, as denoted by the *type* field.

amp – a pointer to the *anon_map* structure. This will be a null pointer if the segment is not anonymous memory.

Anonymous Memory

- The vnode segment driver also serves to manage anonymous memory.

- Anonymous memory is used for:

  copy–on–write

  uninitialized data and stack

- The *anon* structure serves as a substitute for a name (vnode, offset) reference to these pages.

- Each page of anonymous memory will either be in swap or main memory, and every page will have an *anon* structure associated with it.

Anonymous Memory

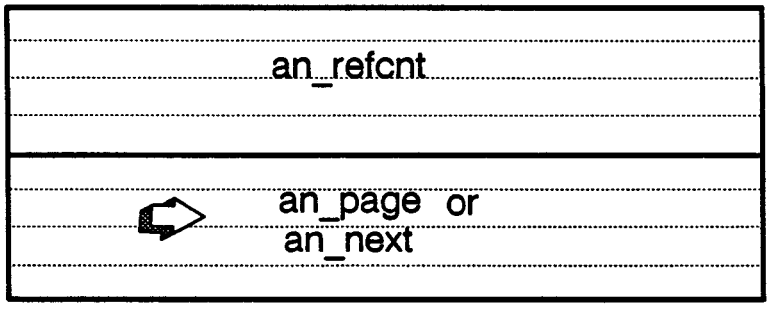
Pages that are created as a result of copy–on–write operations or the initial fault to uninitialized data and stack pages are treated as anonymous memory.

Anonymous memory is not identified in the file system name space or may not be associated with a physical device such as a disk. Since it is not associated with a vnode, offset pair (the page identifier) we use the *anon* structure and the *anon_map* structures (depicted on the following pages) to manage this type of memory.

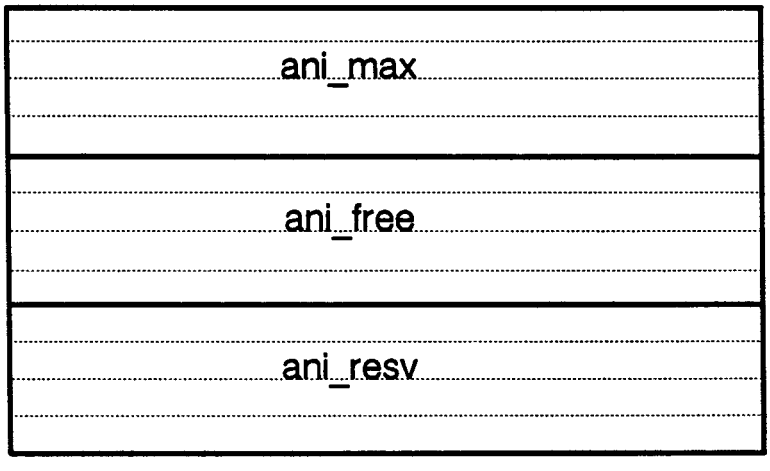
The anon structures are mainly used to keep track of when the last reference to a copy–on–write page goes away so the vnode segment driver can release the page.

The anon and anoninfo Structures

anon



anoninfo



The anon and anoninfo Structures

The anon Structure

an_refcnt – a reference count to track multiple references to an anonymous page such as the result of a *fork(2)* system call.

union

> **an_page** – a pointer to the page, a level of indirection is used to anonymous pages since the pages may not be backed by any swap space. Therefore these pages cannot be referenced by an offset in some storage device as named objects are by the vnode, offset pair. The *an_page* is just a hint to where the current page probably is since the page may be moved underneath us.
>
> **an_next** – If the page is on the free list *an_next* points to the next anon structure in the free list.

The anoninfo Structure

ani_max – maximum number of anon pages available.

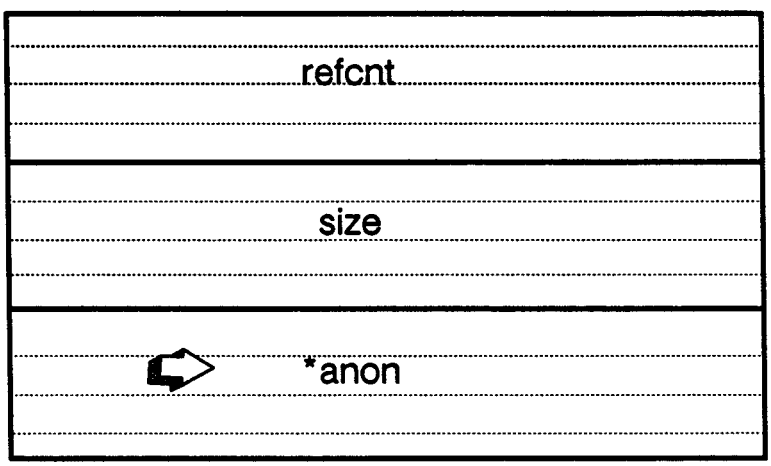
ani_free – number of anon pages currently free.

ani_resv – number of anon pages reserved.

The anon_map structure

The *anon_map* structure is used by the seg_vn driver to treat a cluster of anonymous pages as a unit.

The anon_map Structure



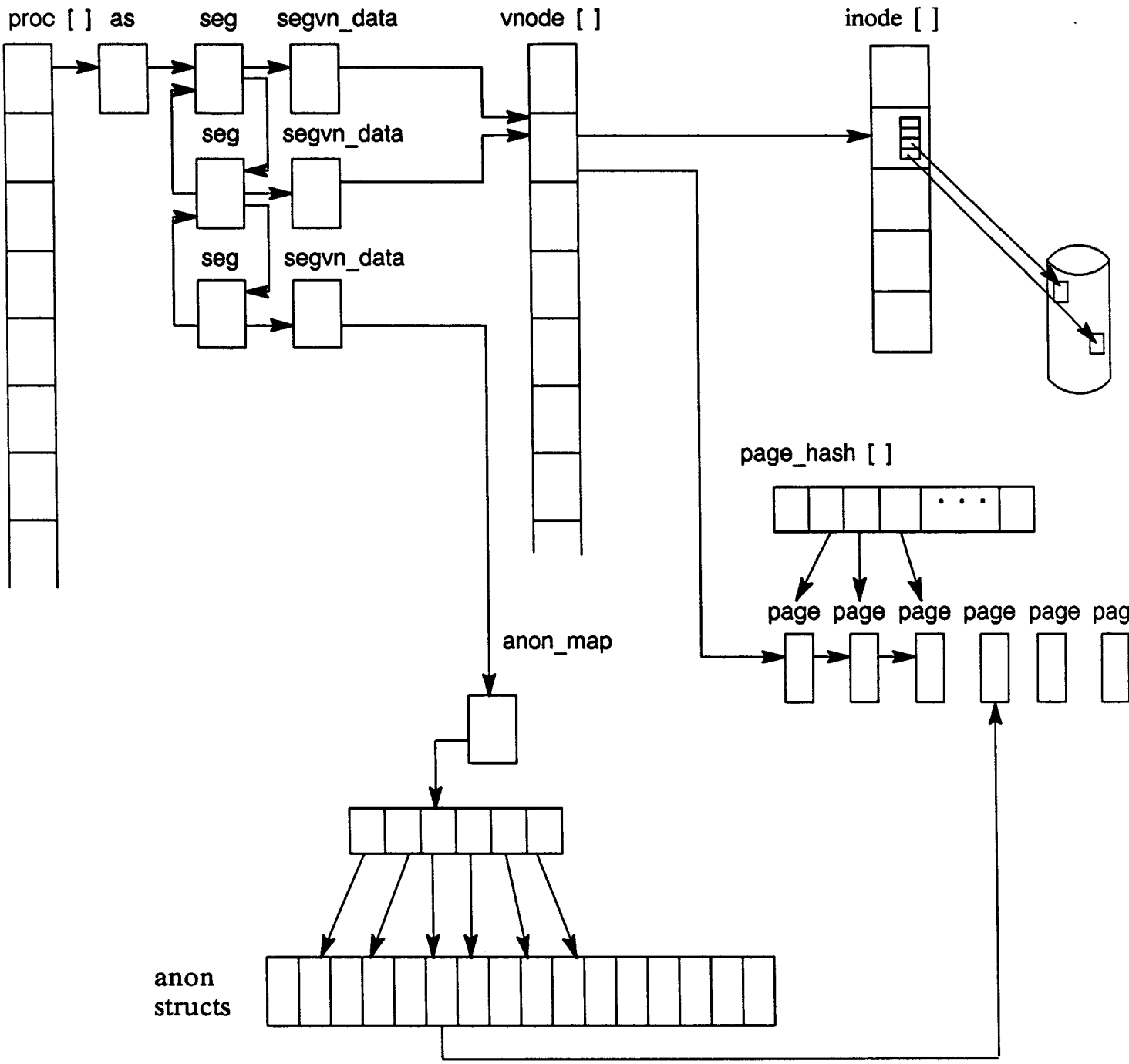
The anon_map structure

refcnt – a reference count of *segvn_data* structures that point to this *anon_map* structure. When *anon_dup()* copies one array to another (*struct anon **anon* to another *struct anon **anon()*) it increments the *refcnt.*

size – since more than one segment can share an *anon_map* structure, an unmap operation to pages with a reference count of one (or with a private mapping) could free up a range of anon slots in the anon array. This fragmentation makes it necessary to keep the *size* bytes mapped by the array so we can free everything when the reference count goes to zero.

***anon** – pointer to an array of anon pointers. Shows up in *anon_map* in *seg_vn .h* as ***anon.* See appendix B for the actual include files.

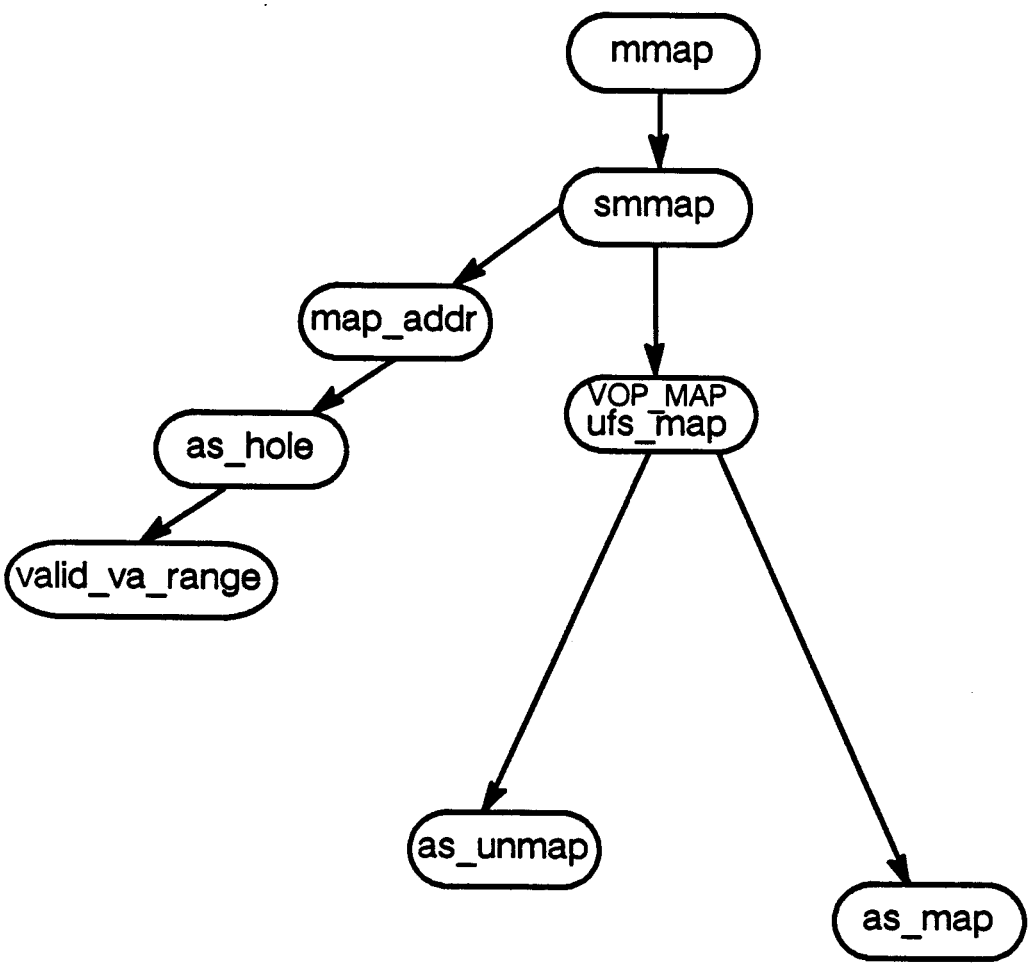
mapping structures



mapping structures

This diagram shows how the physical data is mapped to a user's address space.

mmap( )


mmap
smmap
map_addr
as_hole
valid_va_range
VOP_MAP
ufs_map
as_unmap
as_map

mmap( )

The *mmap( )* system call is probably the most basic system call in the SunOS as mapping is the focal point of the Virtual Memory System(VM). The *mmap( )* system call calls the internal version of the routine *smmap( )* which first checks that the correct flags are passed and that either MAP_SHARED or MAP_PRIVATE is specified. If this is a "new" *mmap( )* call, it will call *map_addr( )*, a machine dependent routine to determine at what address the object (file) will be mapped to.

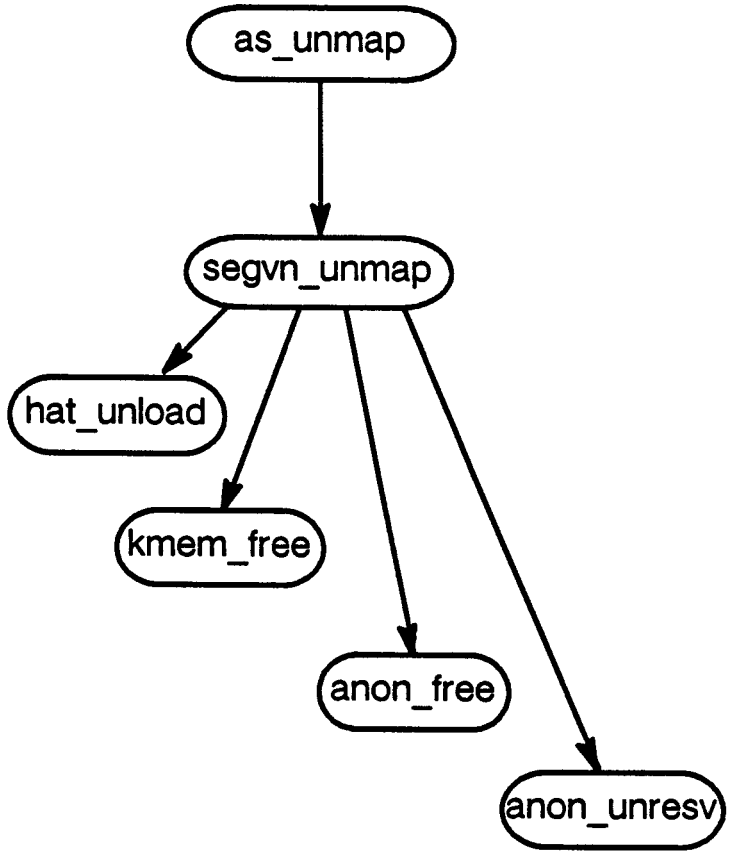
map_addr( ) makes sure that the addresses are shm aligned and will make adjustments to make it so. It calls *as_hole( )* to search for a large enough hole starting from below the stack limit which in turn calls *valid_va_range( )* to check the validity of the address range passed.

If the user requests an address to be used, *smmap( )* will do page alignment checks and boundary checks before calling *getvnodefp( )* to get the file pointer corresponding to the descriptor. The file[] entry holds the vnode pointer in its private data member. If the vnode is not cacheable, the map request is refused as this violates the don't cache attribute. *smmap( )* will also check that the proper protections are set before calling the vnode map routine which for UFS is the *ufs_map( )* routine.

The *ufs_map( )* does not do much except check that the offset and the length are not negative and that the vnode is for a regular file. It also initializes a segvn_crargs structure that will be passed to the *as_map( )* routine. It calls *as_unmap( )* to remove existing mappings on the address range before calling *as_map( )* to map in the file.

as_unmap( ) will search the address space of the process for the segments that fall within the address and size that are given and will call the segment driver unmap routine for each segment that is found. The *segvn_unmap( )* routine calls *hat_unload( )* to unload any hardware translations in the address range taken out.

as_unmap( )


as_unmap
segvn_unmap
hat_unload
kmem_free
anon_free
anon_unresv

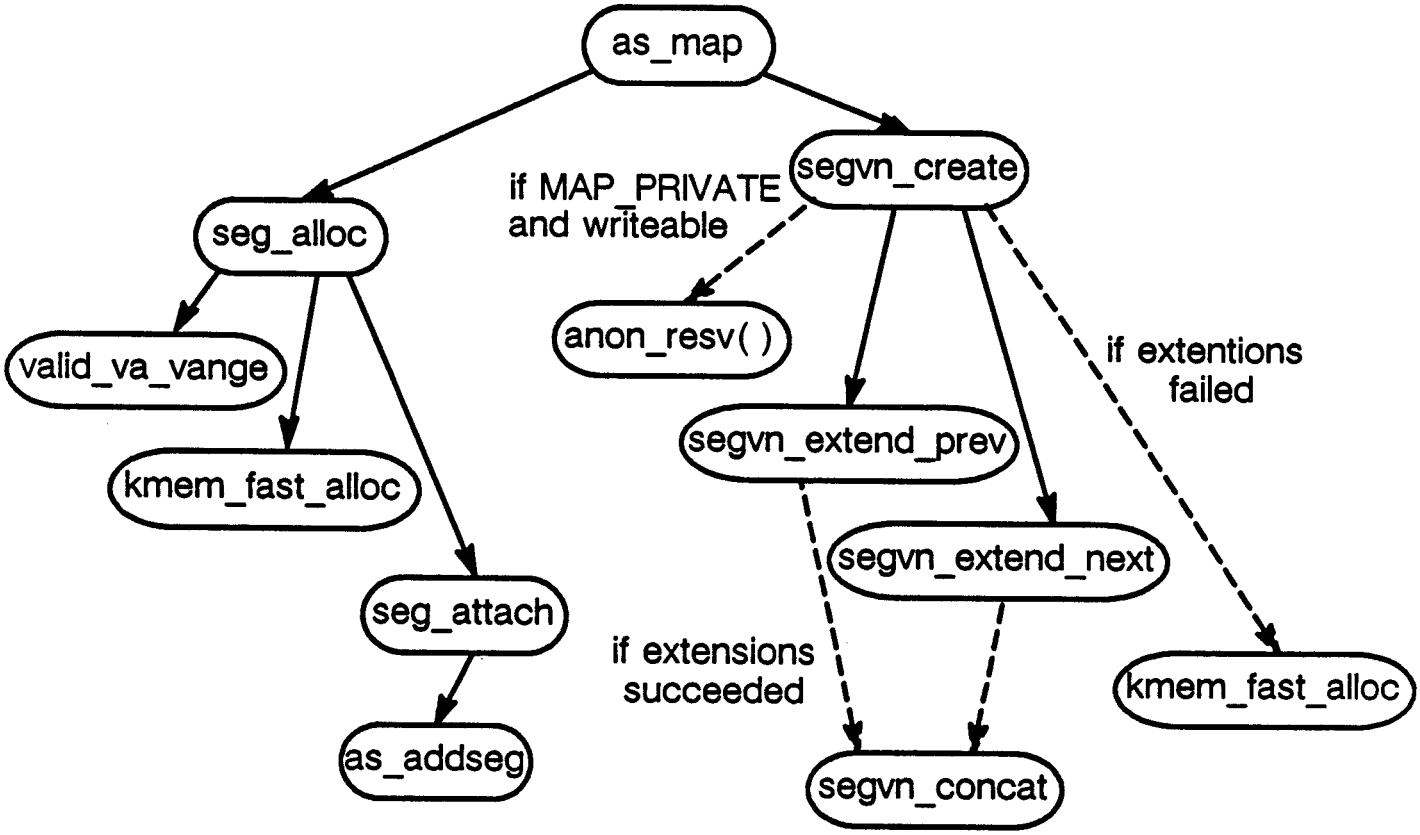
as_unmap( )

as_unmap( ) loops through all the segments in the address range and for each valid segment within the address range, calls the segment driver unmap routine which is segment dependent. For the vnode segment, this will be the *segvn_unmap( )* which will unmap mappings within this segment.

segvn_unmap( ) calls *hat_unload( )* to unload all hardware translations for the address range specified in the parameters. If the entire segment is being unmapped, we can let go of the whole segment by calling *seg_free( )*. If it is not, *segvn_unmap( )* will check if the address range being unmapped is at the beginning of the segment and it calls *kmem_free( )* to free vpage structures if there are any allocated. If the segvn_data structure has a non-null pointer to an anon_map structure and the reference count on the structure is still one or if the segment is mapped MAP_PRIVATE, then we have to free the now unused parts of the anon_map array by calling *anon_free( )*. If anonymous pages were reserved earlier, then we give up this reservation by calling *anon_unresv( )*. *segvn_unmap( )* will then update the segment structure and return success to *as_unmap( )*.

If the address range to be unmapped is at the end of the segment, then the above procedure will be done on that particular portion of the segment and it means the segment will be shorter than it was before. The slightly more complicated case, is when the address range being unmapped is in the middle of the segment in which case the segment will actually be split into two with a new segment being allocated for the high end.

as_map( )


as_map
seg_alloc
valid_va_vange
kmem_fast_alloc
seg_attach
as_addseg
if MAP_PRIVATE
and writeable
segvn_create
anon_resv( )
segvn_extend_prev
segvn_extend_next
if extentions
failed
if extensions
succeeded
kmem_fast_alloc
segvn_concat

as_map( )

The *as_map( )* routine calls *seg_alloc( )* which in turn calls the *valid_va_range( )* routine to determine if the address plus the size may contain a mappable range of addresses of a minimum size. *seg_alloc( )* then calls *kmem_fast_alloc( )* to allocate the segment structure for the address range. It then calls *seg_attach( )* which turns around and calls the *as_addseg( )* routine. This routine adds the segment into its proper place in the segments list in the process address space as. After these routines return successfully, *as_map( )* calls *segvn_create( )*.

segvn_create( ) checks that the type MAP_PRIVATE and MAP_SHARED are not specified at the same time, and that if a pointer to an anon structure is given, it is illegal to also specify a vnode pointer or MAP_PRIVATE. If MAP_PRIVATE is specified with write protections, this is a case that might need private or anonymous pages, so call *anon_resv( )* to reserve storage in the swap area. If there is more than one segment in the address space, and the segment is adjacent virtually, *segvn_create( )* will try to concatenate it to these segments. Concatenation is not done if an explicit anon_map structure is supplied or if this is a segment for the kernel's address space. *segvn_create( )* calls *segvn_extend_prev( )* to see if the new segment can be concatenated to a previous one. If this succeeds, it calls *segvn_concat( )* to concatenate the resulting segment to the next one. If *segvn_extend_prev( )* fails, then it will just try to concatenate the new segment to the next one by calling *segvn_extend_next( )*. If it fails in all of these, then it means that the new segment should remain on its own. Finally *segvn_create( )* calls *kmem_fast_alloc( )* to allocate a segvn_data structure for this new segment and returns success to *as_map( )*.

The Seg_map Segment Driver

- *seg_map* is used for kernel transient mappings.
- *seg_map* manages the kernel address space as a large sparsely populated range of virtual addresses.
- *seg_map* supports only shared mappings.
- *seg_map* provides fast map and unmap operations.
- *seg_map* implements the vnode *read(2)* and *write(2)* routines via *segmap_getmap()* and *segmap_release()*.

The Seg_map Segment Driver

The *seg_map* driver manages the kernel address space as a virtual cache. When a virtual address is referenced within a *seg_map* segment it is likely the vnode, offset (page identity) is already loaded in the MMU and therefore no page fault will occur.

The fast mapping and unmapping operations performed for system read and write use the *segmap_getmap()* and *segmap_release()* routines under the segment driver. *segmap_getmap()* returns a virtual address within the *seg_map* segment for a given vnode, offset. So it performs a similar function to *bread()* in traditional Unix. *segmap_release* releases the mappings and if necessary writing back dirty pages for a given virtual address. So *segmap_release* performs the functions of both *brelse()* and *bdwrite()* in traditional Unix.

The Read and Write Routines

The *read()/write()* routines operate by:

- Mapping the file to/from the kernel address space
- Copying the data from disk to the physical pages pointed to by the mapping

Since *read(2)* and *write(2)* use mapping you get a coherent access to the file no matter how you access it.

The Read and Write Routines

The *read(2)* and *write(2)* calls are still compatible with the older versions from the user perspective.

The seg_map driver performs the *read(2)* and *write(2)* calls using an internal version of *mmap* with a *type* of shared mapping.

For example when you do a read or write:

1. A portion of the kernel's address space is mapped to the file you're accessing.
2. Another portion of the kernel's address space is mapped to your allocated user buffer space.
3. The memory to memory copy is performed using the *copyin* and *copyout* routines.
4. The mappings are then removed.

   Since it is using mapping, a file that is accessed and modified using the *write(2)* call and then accessed by *mmap(2)*, the modifications will be visible.

The seg_dev segment driver

The *seg_dev* segment driver operates on objects that are controlled by the character device driver interface and support an mmap interface such as:

- frame buffers
- physical memory
- kernel virtual memory
- Multibus memory
- VMEbus memory
- SBus memory

The seg_dev segment driver

This driver currently only supports shared mappings. The most common use of the *seg_dev* driver is for frame buffers.

This driver is fairly simple because many of the operations for other drivers, such as swap out, don't make any sense for device driver segments.

In order to handle a fault the *seg_dev* driver calls a function to get the request information from the device driver and then supplies the information to the *hat* layer. The *hat* layer then can load a translation to the requested physical page.

The seg_kmem driver

- The seg_kmem driver handles kernel text, data, bss and dynamically allocated memory space.
- Currently it handles MMU locked pages.
- There is a set of *seg_kmem* routines that emulate a *mapin()* interface for kernel and driver compatibility.

The seg_kmem driver

The kernel has its own address space which is shared by all processes while executing in kernel mode. The *seg_kmem* driver serves as a catch all for miscellaneous mappings to the kernel's *as.*

This driver also handles other machine specific portions of the kernel's *as* such as the Direct Virtual Memory Access (DVMA) pages.

The Page Structure

- Ordered by physical page number
- Hashed by page identity
- Manages Cache for virtual memory

The Page Structure

The page structure keeps track of pages mapped from file system objects to virtual addresses.

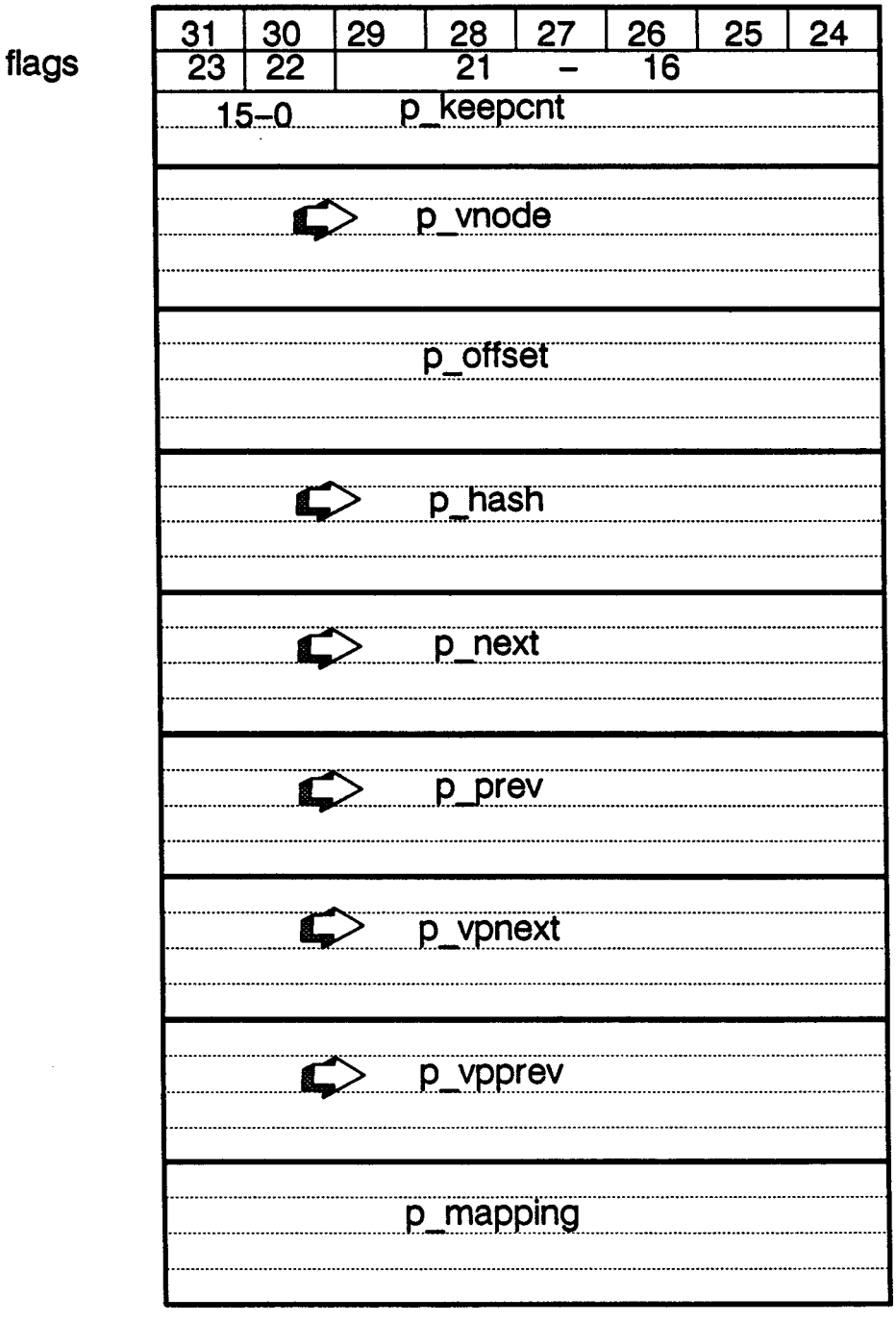
The VM system routines and the object managers use this structure to manage main memory as a cache to anything accessed in virtual memory.

The page identity is determined from the vnode (**p_vnode*) and offset (*p_offset*) fields in the page structure.

A hash lookup on the vnode, offset pair is used to locate the page by name.

All pages for a vnode are on a doubly-linked list. The forward pointer is p_vpnext and the reverse pointer is p_vpprev.

The Page Structure



The Page Structure

flags –

bit 31 – p_lock – indicates this page is locked in memory for the purpose of name manipulation.

bit 30 – p_want – indicates that some process wants and intends to use this page.

bit 29 – p_free – this page is on the free list (available for use) see *p_next* and *p_prev* pointers in this structure.

bit 28 – p_intrans – this page is transition in or out of main mem ory, see *p_next* and *p_prev* pointers in this structure.

bit 27 – p_gone – this page has been released by owning proc ess.

bit 26 – p_mod – this page has been modified while in memory.

bit 25 – p_ref – this page has been used recently, this is used by the paging (swapping) algorithm.

bit 24 – p_pagein – page is in the process of being brought into memory; data has not been initialized.

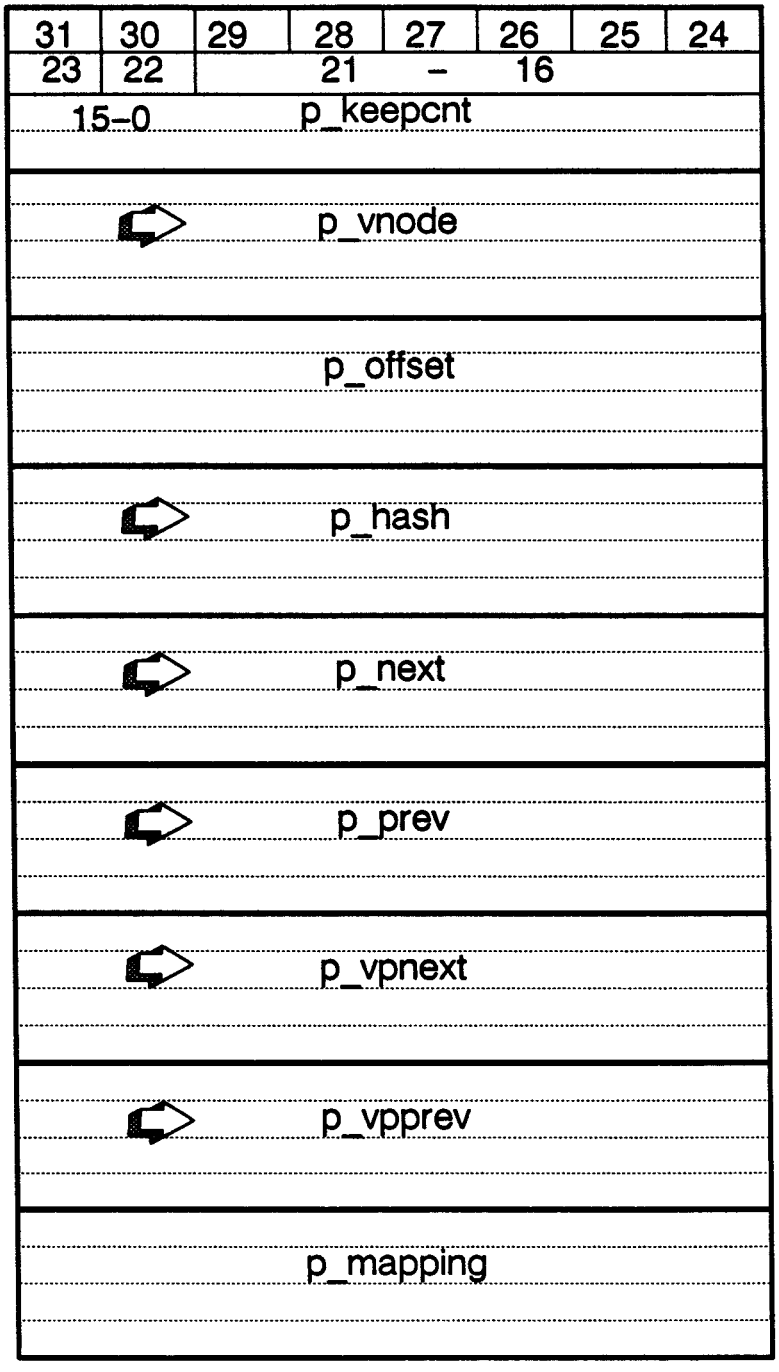
bit 23 – p_nc – this page is not cacheable because of possible problems with cache coherency.

bit 22 – p_age – this page was brought in as a result of a read ahead and has not proven its usefulness.

bits 21–16 – p_nio – the number of outstanding I/O requests needed for this page.

The Page Structure (cont.)

flags



The Page Structure (cont.)

p_keepcnt – The number of hardware translations on this page.

p_vnode – A pointer to the vnode that this page belongs.

p_offset – Page aligned byte offset into the vnode for this page.

p_hash – Pointer to hash index chain from vnode and offset.

p_next – A pointer to the next page in either the free list or the in transition list according to the *p_intrans* and *p_free* flags.

p_prev – A pointer to the previous page in either the free list or the in transition list according to the *p_intrans* and *p_free* flags.

p_vpnext – A pointer to the next page in the vnode list of pages that belong to the object described by this vnode.

p_vpprev – A pointer to the previous page in the vnode list of pages that belong to the object described by this vnode.

p_mapping – Specific information for the hardware address translation (*hat*) layer.

The HAT Layer

- The *hat* layer serves as the interface between the machine independent layers and the machine dependent layer of the VM system.

- There are presently several different versions of the *hat* layer provided for different architectures

- The *hat* layer provides the higher levels of the VM with a layer of isolation from the MMU.

- The *hat* layer handles the different types of caches present on the various architecture offered by Sun.

The HAT Layer

The machine independent layers of the VM system that interface with the *hat* layer are the different segment drivers, and the VM routines. The machine dependent layer refers to the MMU (memory management unit). The MMU may be packaged on board the CPU Processor chip (for example the Motorola 68030) or it may be packaged separately (such as the 68020 or the Sun–4 sparc architechture).

Only the *hat* layer is able to read the current mappings. The higher levels can only determine if a mapping exists. If the translation to a physical page is not locked the *hat* layer is able to remove a mapping on behalf of the operating system.

The MMUs in the Sun–2s, Sun–3s, and Sun–4s all use fixed length segment and page tables, and a fixed number of contexts.

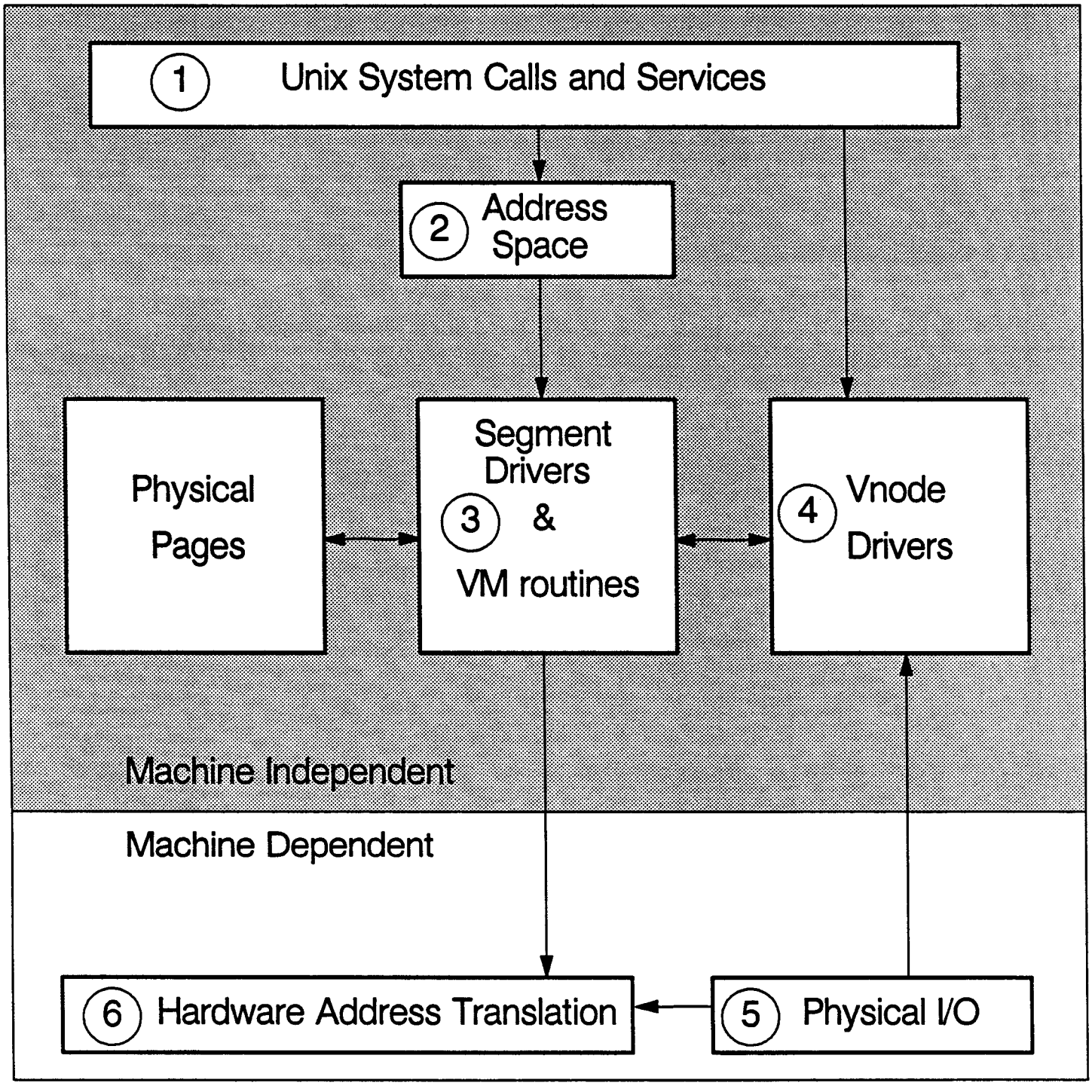
The differences in the architectures are as follows:

The Sun–2 has the kernel mapped into a separate context while the Sun–3s and Sun–4s have the kernel address space mapped into each user's context.

The Sun–2 supports up to 16 megabytes of virtual address space.
The Sun–3 supports up to 256 megabytes of virtual address space.
The Sun–4 supports up to 4 gigabytes of virtual address space.
(An example of MMU translations was given earlier in this module.)

Some of the Sun–3s and Sun–4s use a virtual address cache (cache was discussed earlier in this module).

Page Fault Handling



Page Fault Handling

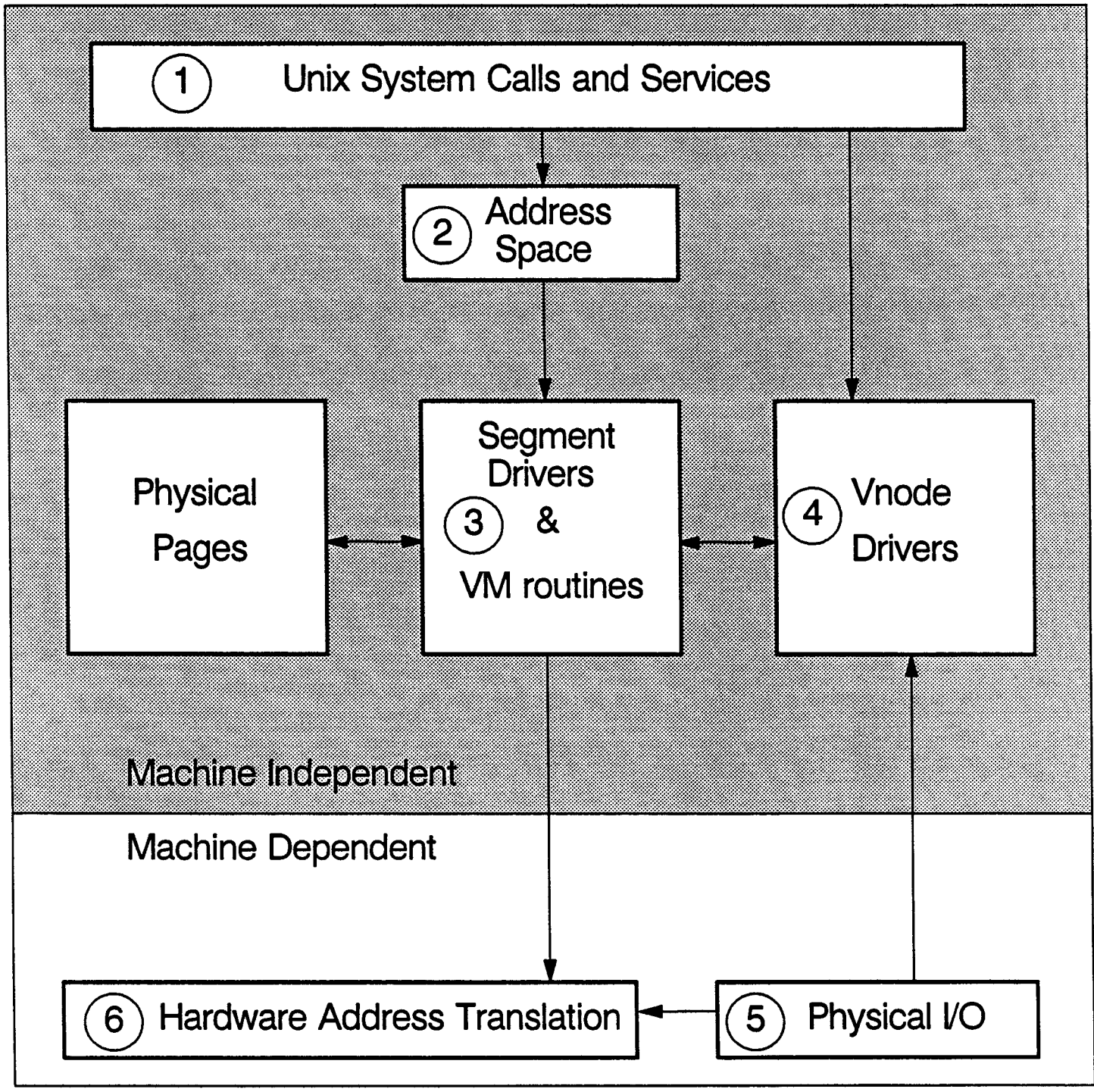
1. First a user process asks for something that causes a page fault. This could could be a hardware or software induced fault.

   Hardware faults could be the result of an invalid translation (i.e. –the page not being resident in main memory). This would be indicated by the valid bit in the page table entry. The fault could also be generated by insufficient protection access.

   Software faults are generated by a locked page.

2. In the *as* layer the fault handling routine is called with a virtual address range, the fault type , and the type of access being requested. The *as* layer fault handler performs a segment lookup and calls the appropriate segment driver.

Page Fault Handling (cont.)



Page Fault Handling (cont.)

3. In most cases it's the vnode segment driver that is called. The vnode segment driver takes the following actions to resolve the fault:

   It verifies the page protections.

   It creates an *anon_map* structure if the page being referenced is considered part of anonymous memory

   If it's a page that has to be loaded from some backing device such as disk, it calls the appropriate object manager (such as the vnode driver, go to step 4). Also the hat layer would be called to load a translation to the page retrieved by the vnode driver (step 5).

   If necessary, perform a copy–on–write to get a new page, then call the hat layer to load a translation to the writable page

4. The vnode object manager is responsible for the creation and initialization of pages that have to be retrieved from disk.

5. The physical I/O layer will handle the addressing of the appropriate physical data block on disk.

6. The hat layer will handle the mapping of the virtual address to its physical page in main memory and load the translation information into the *hat* structure contained within the *as* structure.

Paging

In general all disk I/O centers around the VM system's role as a virtual cache manager.

- **page in** – All page in requests come as a result of the cache manager's need to fill its cache(main memory).

- **page out** – can occur for three reasons:

1. A process exits or is swapped out.

2. A process or the system synchronizes the data to backing store.

3. There is insufficient memory resources to do a requested page in.

Paging (cont.)

page in – when the MMU finds a page of virtual memory is not resident, the VM has to bring the page into main memory.

On processes that have a hardware cache the virtual address is used to index into the cache entry. If the entry is not in cache the virtual address is passed to the MMU for translation to a physical address. If the MMU finds that the page is no longer resident, a disk I/O is generally required.

The page in and page out activities center around the page daemon, the routines in *vm_page.c,* and the *page* structure.

The *mmap(2)* system call does very little work other than creating the mappings for a potential access. The real work occurs when the system page faults because the mapping exists but the data doesn't. The data is moved in only as the pages are accessed. All incoming disk I/O occurs as result of these page faults.

page out – the page out process is entered as the result of an *fsync,* an *msync(2)* or the *page* daemon needs to reclaim a page for current activity. Also page out occurs if a process exits or is moved to swap.

The page out strategy would ideally like to page out pages that will not be needed in the near future. Since the system can't predict that, it uses a least recently used (LRU) algorithm to determine which pages should be paged out.

Paging In

- There is no specific *pagein* routine, there is a set of routines that govern pagein policies and manage main memory as a buffer cache.

- All of main memory can act as the buffer cache.

- A page is searched for by its page identity to see if it still has a page structure (*pagefind()*).

- If it has a page structure, but its *p_free* bit is set, the *page_reclaim* routine, attempts to reclaim it .

- The *p_mapping* field gives the mapping information to access the page table entry.

Paging In

Before initiating an I/O on a page request from a file system access, the VM routines *(vm_page.c)* check to see if the page is in memory. The *vnode, offset* fields are hashed to see if the page has an associated page structure. This is a quick lookup mechanism to the page structure no matter what list it is on.

If the page is in transition coming in wait for the page, and then reclaim it if it is on the free list *(p_free).*

If the *vnode* and *offset* match the requested vnode, offset, and it's on the free list, reclaim it. Upon entering the *unfree* routine raise the priority to keep page from being ripped off from the interrupt level. Then put the page on the vnode list *(p_vpnext* and *p_vpprev),* and remove it from the free list (*p_free* = 0).

Actually there are two physical lists that make up the logical free list. There is a circular cache list and a circular free list. When a page on a paged vnode list is freed it is put on the end of the logical free list (the cache list). If the page has no identity, or its age bit is set (*p_age* = 1) it is put at the beginning of the logical free list (the free list). The cache list is used for pages we want to leave in memory only if it's really necessary.

Aborting a page causes it to lose its identity.

Page Replacement

The system strategy for page replacement is as follows:

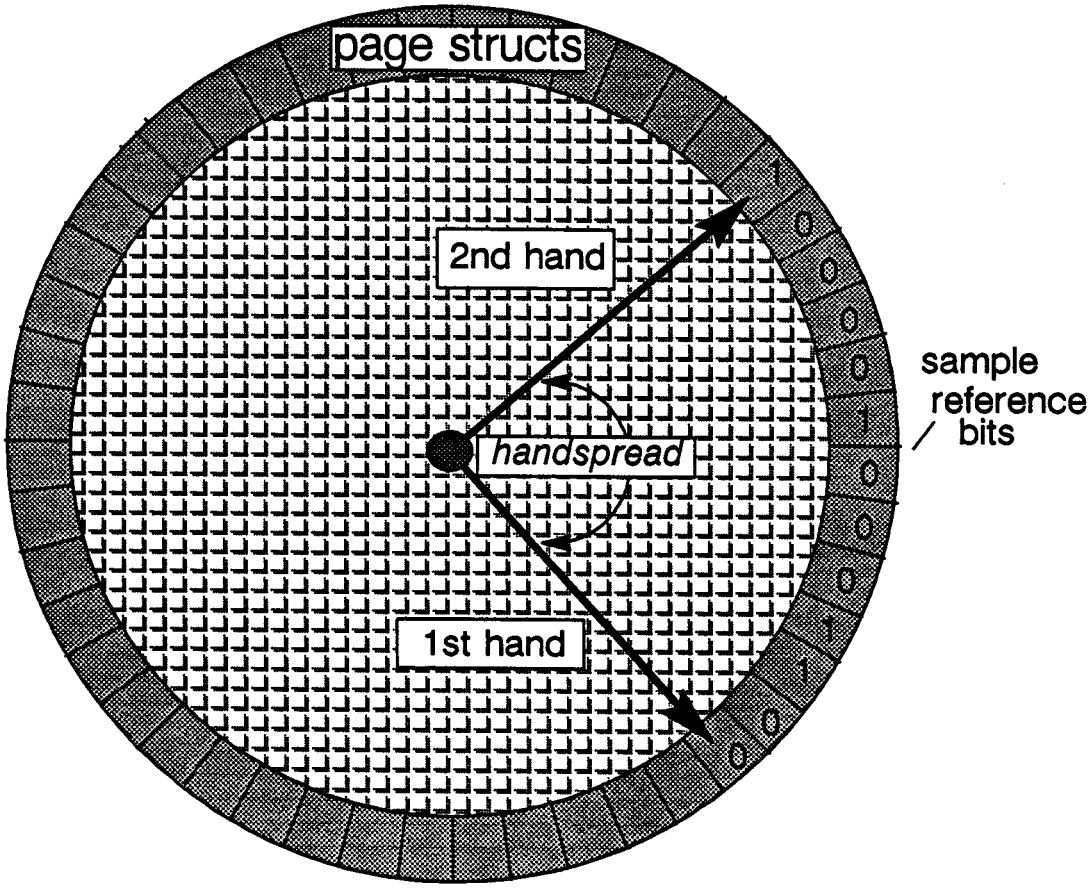
- Try to keep a minimum amount of memory free – *minfree*
- If there isn't lots of free memory *(lotsfree)* then run the *pageout* routine from the *page daemon.*
- If we still can't keep the desired amount of memory free *(desfree),* or if the free memory drops to a minimal value *(minfree),* then swap processes out completely.
- A two hand clock algorithm is used to search memory for the least recently used pages.

Page Replacement

The strategy for page replacement is to make sure we always have a minimum amount of memory available, *minfree.* To keep available memory from falling below this level, process 2, the page daemon runs whenever the number of free pages is below a given threshold, *lotsfree.* The *page daemon* runs the *pageout()* routine *(vm_pageout.c)* which scans the page structures looking for pages to page out on a least recently used basis.

The *pageout()* routine does this by using a two handed clock algorithm that views the page structures as being in a continuous loop (or circle). Remember that the page structures do not exist for kernel text and most kernel data structures so the kernel is not eligible to be paged.

The Clock Algorithm



Times to scan pageable memory at *fastscan* rate

amount of memory	time
< = 1 MB	5 sec
2 MB	10 sec
3 MB	15 sec
4 MB	20 sec
8 MB	40 sec etc.

The Clock Algorithm

In the clock algorithm the hands of the clock sweep through the page structures at the same rate, at a fixed distance apart *(handspread).* The first hand resets the reference bit in the page structure entries. The backhand will reach the page structures processed by the first hand in the order of a few seconds depending on the scanrate and the *handspread.* The backhand then checks to see if the page has been referenced in the interval. if it has not, the page goes on the free list.

The rate at which the hands sweep through the page structures varies linearly with how much memory is free. If the amount of free memory is *lotsfree* the hands will move at a minimum scan rate, *slowscan.* If the amount of free memory is near *minfree* the hands will move at the maximum rate, *fastscan.*

The *fastscan* rate is a maximum of 200 pages per second, and *slowscan* is half that. At 200 pages per second the times to sweep through memory is shown on the previous page.

Some typical values for the clock algorithm (used for Sun–3s and Sun–4s):

A typical value for *handspread* is 2MB but it is clamped to the size of main memory or 360 degrees.

A typical value for *desfree* is 100KB but it is clamped at 1/16 main memory.

A typical value for *minfree* is 32KB but it is clamped at 1/2 *desfree.*

A typical value for *lotsfree* is 256KB but it is clamped at 1/8 main memory.

The swapout algorithm

Whole processes have to be swapped out of memory when:

- the system needs memory for kernel mapping,

OR

- on the average there are two or more runnable processes,
  AND the average amount of memory (2 sample times) is less then *desfree.*
  AND (the paging rate exceeds *maxpagio*
  OR
  the amount of free memory drops below *minfree.)*

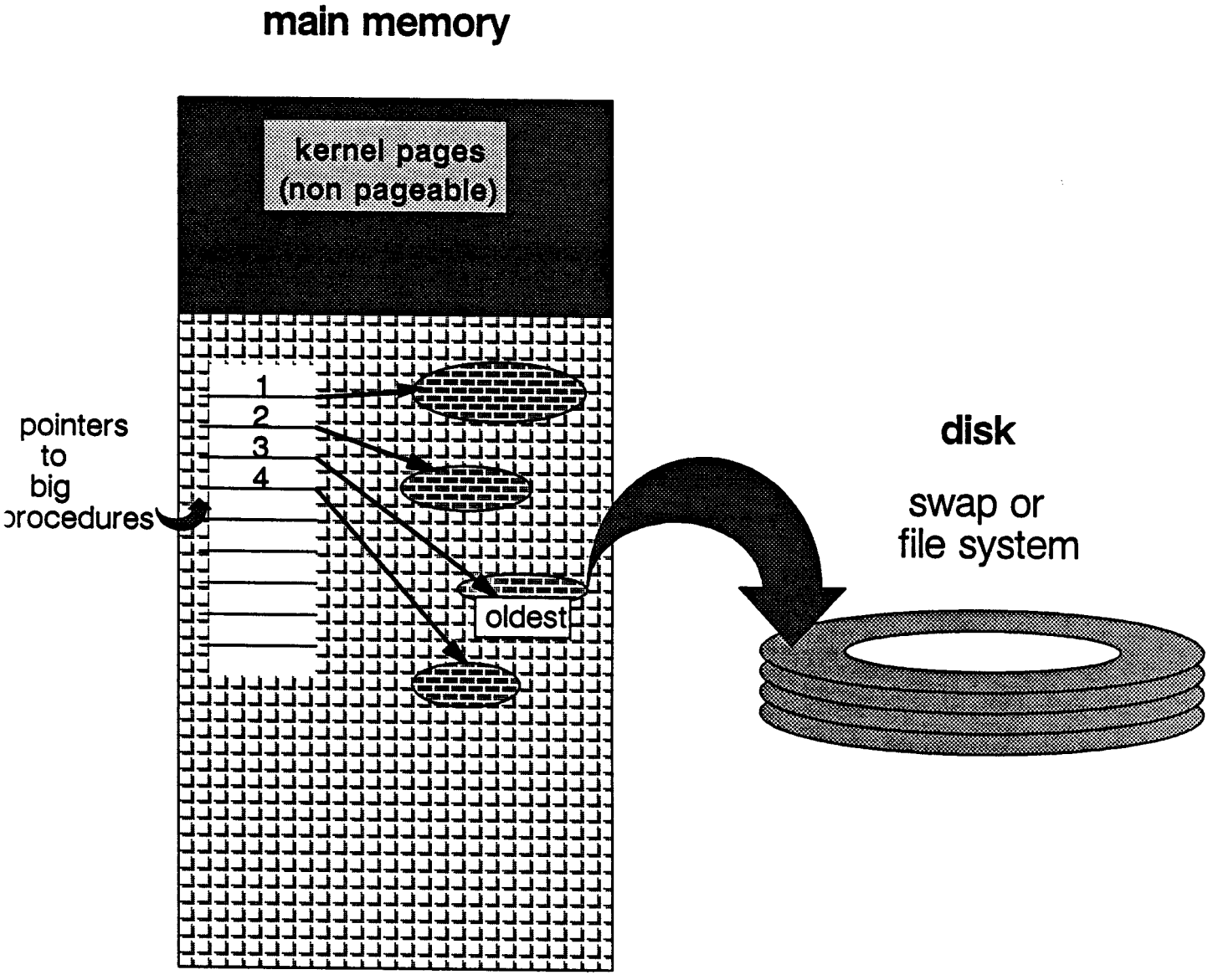
The swapout algorithm

If the *pageout()* routine can't keep up with the demand for memory the system gives up on freeing up memory one page at a time and resorts to swapping out whole processes.

maxpgio is a parameter used to guard against page thrashing. It is the maximum tolerable paging rate before processes are swapped out. It prevents too many processes from competing for memory resources.

If the amount of memory drops below *desfree* for a 5 second interval AND a 30 second interval *swapout()* is called.

The swapout algorithm(cont.)



The swapout algorithm(cont.)

The system keeps a linked list of the 10 biggest processes in terms of their memory resident set.

Kernel pages are not eligible to be swapped out.

Of the 10 biggest processes the system selects the 4 largest and swaps out the oldest, or a process that is a long time sleeper.

When a process is swapped out we keep track of the number of pages swapped out in the global statistics. This may not be the total resident set since the process may have been sharing pages with other processes. In this case not all the process's working set will be paged out.

After a process gets swapped out the system distributes the memory resources to the rest of the processes by increasing the *deficit*. The *deficit* is the difference between how much memory we would like to have available and what we actually have available. This is done so the exiting process won't create an artificial abundance of memory that will cause a process to be swapped back in prematurely.

If this does not alleviate the memory bound situation more processes may have to be swapped out.

The swapin algorithm

There are a few factors that determine which process will get swapped back in when free memory is sufficient.

- A larger process is less deserving than a smaller process.
- The longer a process has been asleep the more deserving that process is.
- The niceness factor is amplified and reduces the deservedness (nice processes come in last).

The swapin algorithm

If a process is very deserving of being brought in, the process is swapped in when half of its "swapped out" memory resident set is available.

When a process is swapped back in, only its user structure is brought in initially and the rest of the pages are faulted back in.

A process is initially swapped out on the basis of its memory resident set and it is swapped back in on the basis of the number of pages it actually freed.

Module 3

Process Management

Objectives: Upon completion of this module, the student will be able to:

- list the structures used by the system to manage processes
- identify fields in the proc structure and user structure using adb
- to identify various structures in a core dump or the live system using adb
- explain the stages of process creation and termination
- explain the philosophy and mechanism used to schedule processes
- list the signal management routines and describe their functions
- describe what the signal facility is for and how a signal is delivered

Evaluation: lab and quiz

Reference information: course guide

Introduction

A process is defined as a program in execution.

Sun and most current Unix systems assume a single thread CPU. Meaning only one process can be executing at any given instant.

Certain resources have to be available in order for a process to execute.

The system gives the illusion of concurrently running processes by sharing the resources between the active processes on the system. SunOS favors interactive processes over cpu intensive processes.

Processes are created and initiated with the *fork(2) and execve(2)* system calls.

Unscheduled hardware or software events are handled by the use of signals.

Introduction

In this module we will discuss:

The Process State

- What resources are needed for a process to execute
- The Proc Structure
- The User Structure
- The Process Address Space

Process Scheduling and Context Switching

- The State Diagram
- The *sleep* routine
- The *wakeup* routine
- The *setpri* routine
- The *schedcpu* routine
- The *roundrobin* routine
- The *swtch* routine

Process Creation and Termination

- The *fork* and *vfork* routines
- The *exec* and *wait* routine

Signals

- The various signals, and signal control masks

The Process State

In the User Mode a process requires:

- **CPU general purpose registers**
  - **Program Counter**
  - **Stack Pointer**
  - **Processor Status Register**
- **Memory Segments including:**
  - **the text segment**
  - **the data segments**
  - **the stack segments**

The Process State

The Process State

In the Kernel Mode a process requires:

- **hardware registers**
- **Program Counter**
- **System Call Arguments**
- **Stack Base Registers**
- **Kernel state for the process defined in:**
  - **the user structure**
  - **the proc structure**

The Process State

The Proc Structure

The system keeps track of processes in memory on the system by the proc structure.

The proc structure can be found on one of three lists. The *allproc* list, the *zombproc* list, or the *freeproc* list.

Active processes that are not currently executing will be on a run queue or a sleep queue, except stopped processes that are not blocked which are on neither.

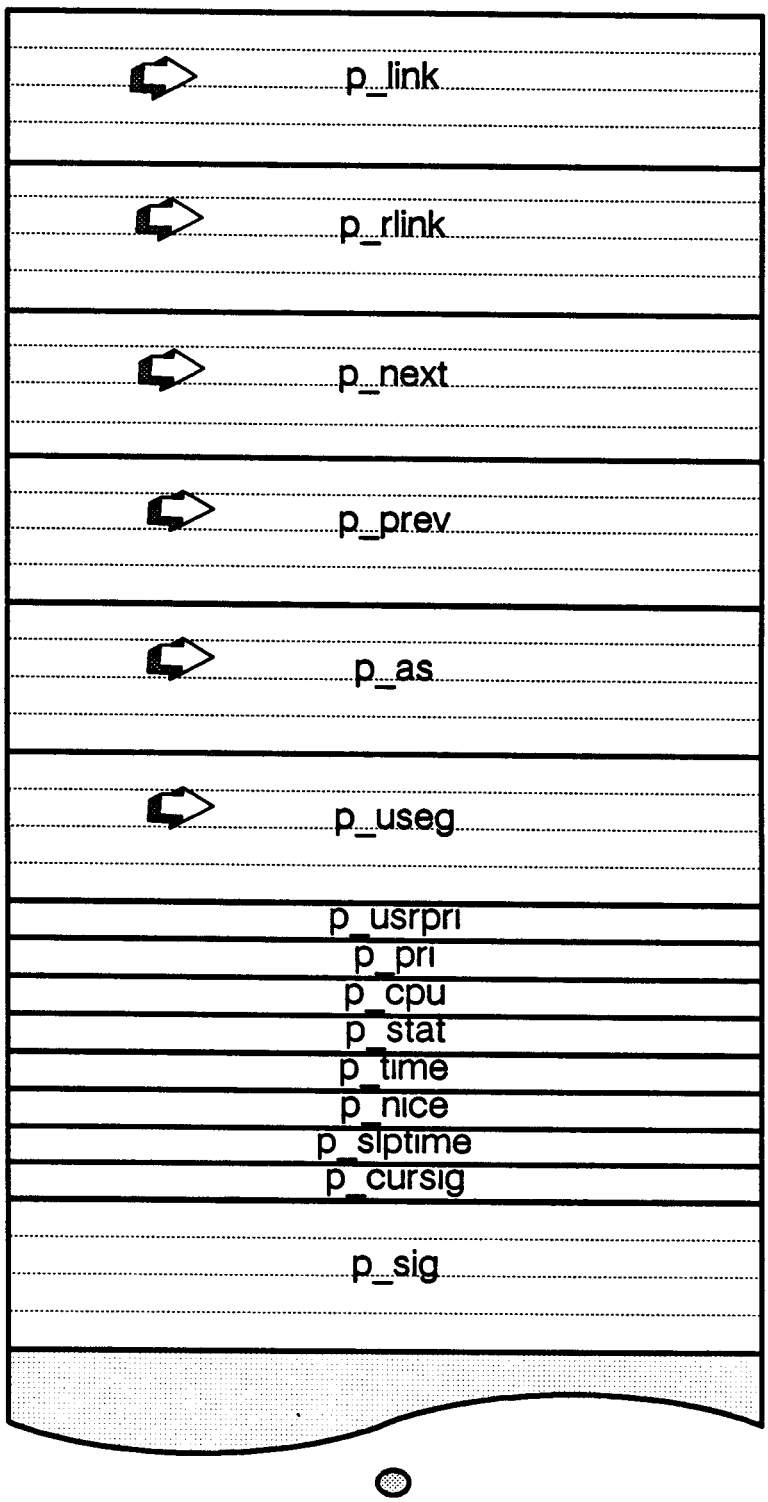
The Proc Structure

The proc structure is a memory resident structure that keeps all the information that is needed to keep track of a process whether it is currently in memory or swapped out.

If the process is in use, and not a zombie process its proc structure will be on the *allproc* list. If the process is in the *SZOMB* state its proc structure will be on the *zombproc* list.

If the process table entry is not currently in use the corresponding proc structure will be on the freeproc list.

The Proc Structure



The Proc Structure

p_link – forward pointer in linked list of running or sleeping processes.

p_rlink – backward pointer in linked list of running or sleeping processes.

p_next – forward pointer in linked list of zombie, free, and allocated processes.

p_prev – backward pointer in linked list of zombie, free, and allocated processes.

p_as – pointer to process address space structure.

p_useg – mapping information for user structure which may have gone to secondary storage.

p_usrpri – priority of process while running in user mode.

p_pri – current priority of process while running in kernel or user mode.

p_cpu – decayed estimation of cpu usage used by *schedcpu.*

p_stat – see status bits at end of structure.

p_time – the number of seconds this process has been resident in memory.

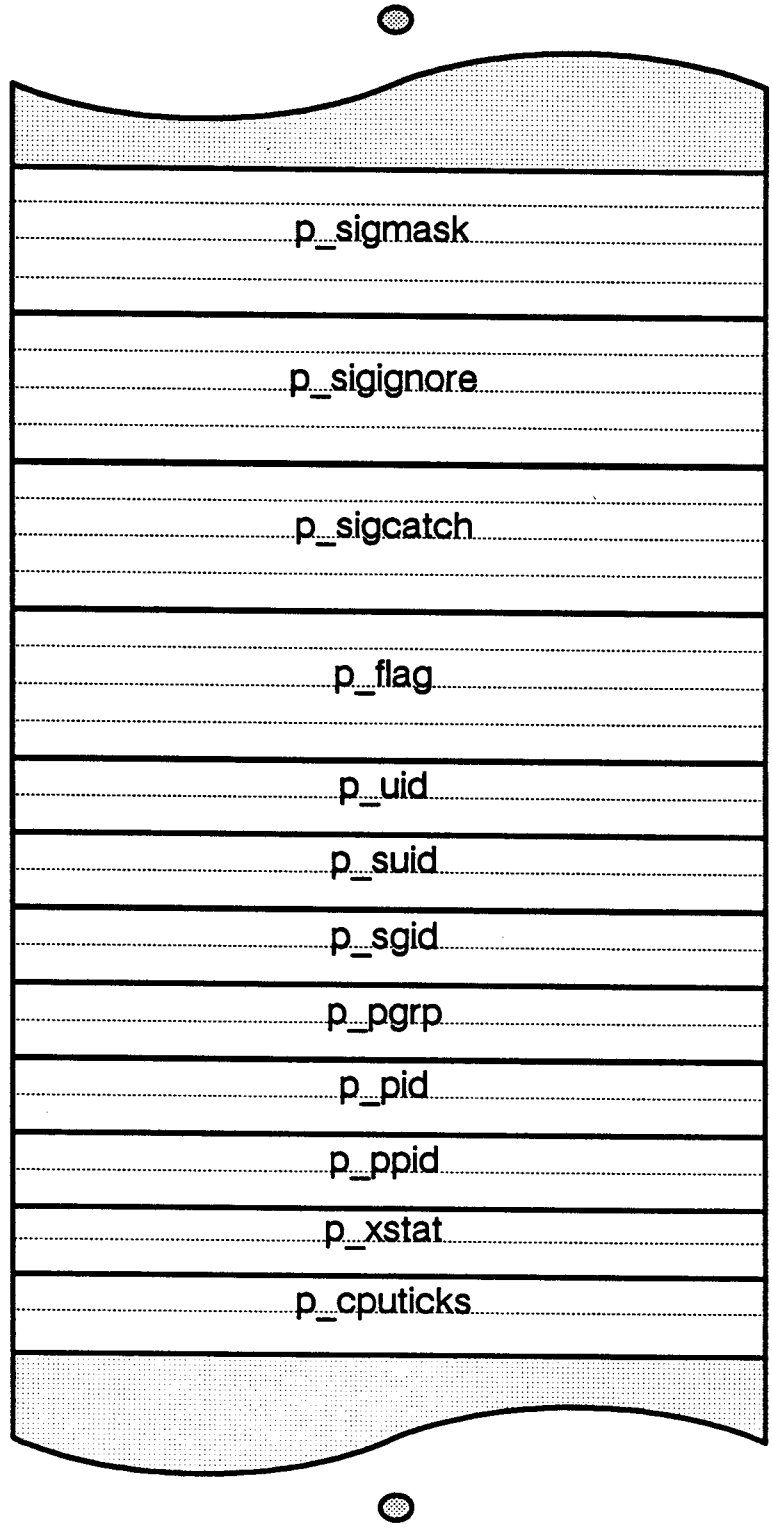
p_nice – factor to shift *p_usrpri* ranges between –20 and +20.

p_slptime – number of seconds this process has been sleeping.

p_cursig – signal currently being delivered.

p_sig – signals pending for this process.

The Proc Structure



The Proc Structure (cont.)

p_sigmask – signals this process is masking.

p_sigignore – signals this process will not take any action on.

p_sigcatch – signals that this process has a signal handler.

p_flag – see pflag bits at end of this structure.

p_uid – user id for tty signals.

p_suid – effective user id.

p_sgid – effective group id.

p_pgrp – id of process group leader.

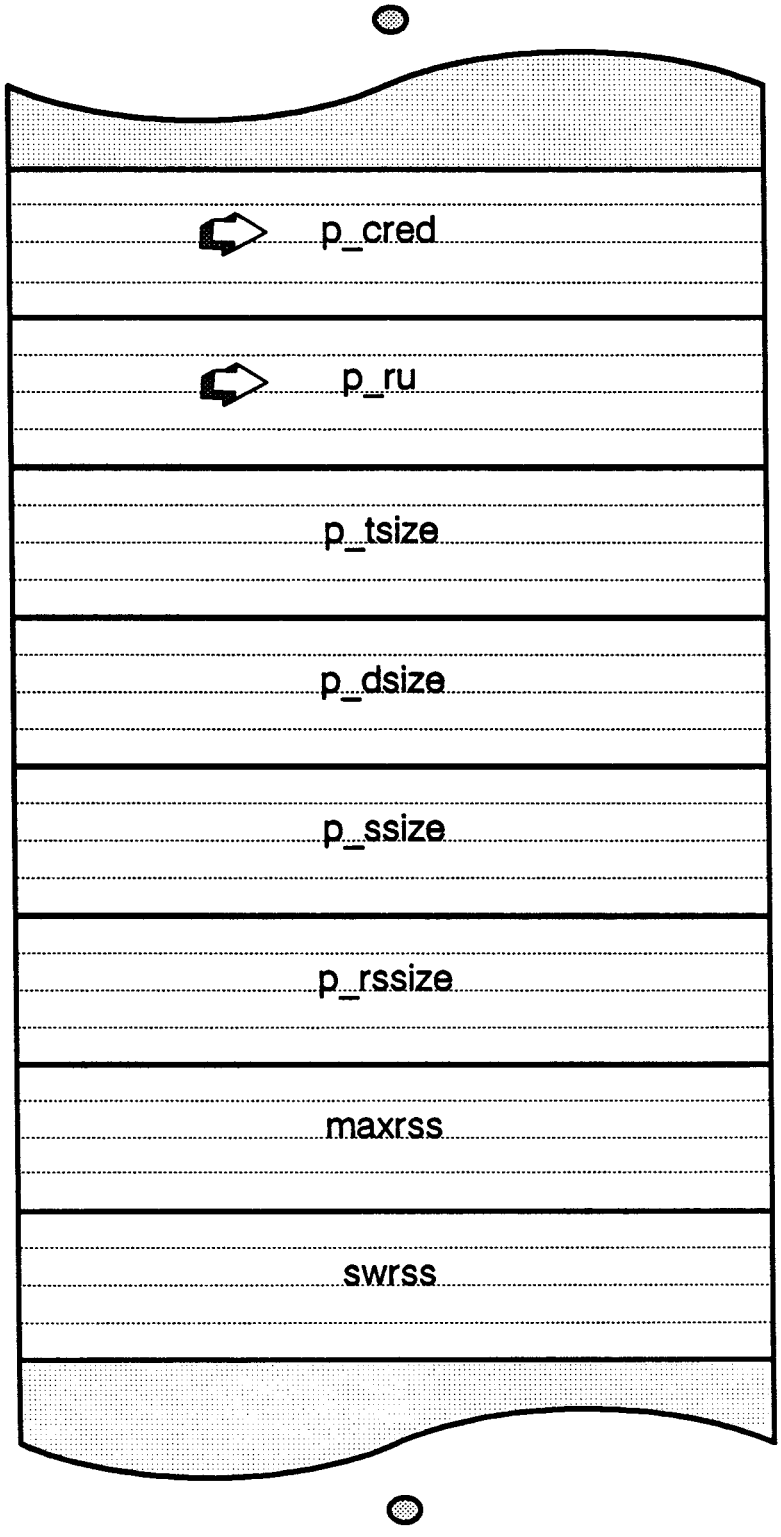
p_pid – unique process identifier. PIDs are used to identify processes by the Kernel and other processes.

p_ppid – process id of parent process.

p_xstat – exit status for *wait.*

p_cputicks – number of clock ticks while in execution.

The Proc Structure (cont.)



The Proc Structure (cont.)

p_cred – pointer to process credential structure (see Appendix B).

p_ru – pointer to rusage structure.

p_tsize – number of pages in text segment.

p_dsize – number of pages in data segment.

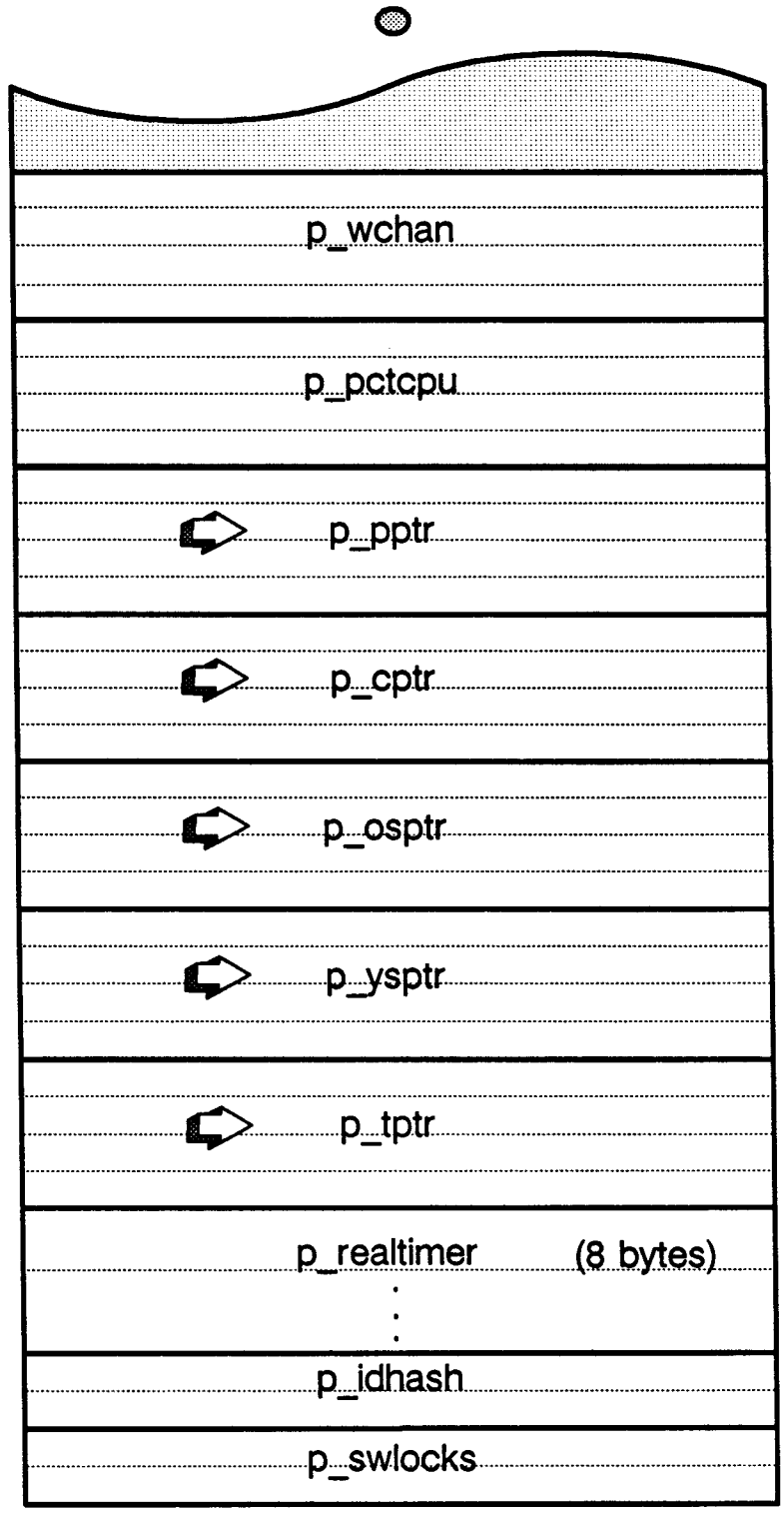
p_ssize – number of pages in stack segment.

p_rssize – current number of pages resident in main memory for this process.

maxrss – maximum resident set size (used in page replacement algorithm).

swrss – resident set size before last swap (used by *swapin*).

The Proc Structure (cont.)



The Proc Structure (cont.)

p_wchan – wait channel address that a process sleeps on.

p_pctcpu – percentage of cpu utilization by this process.

p_pptr – pointer to proc structure of parent process.

p_cptr – pointer to proc structure of youngest living child process.

p_osptr – pointer to proc structure of older sibling process.

p_ysptr – pointer to proc structure of younger sibling process.

p_tptr – pointer to proc structure of process tracing this process.

p_realtimer – timer structure (8 bytes) for absolute timer (*alarm(2)).*

p_idhash – used by routines to locate a proc struc by its pid, quickly.

p_swlocks – number of swap vnode locks held by this process.

The Proc Structure (cont.)

p_stat



```
#define SSLEEP      1
#define SWAIT       2
#define SRUN        3
#define SIDL        4
#define SZOMB       5
#define SSTOP       6
```

p_flag

				27	26	25	24
23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0

The Proc Structure (cont.)

state codes

SSLEEP = 1 process is on sleep queue awaiting an event.

SWAIT = 2 process status never reaped by parent, abandoned state.

SRUN = 3 process is runnable.

SIDL = 4 process is in intermediate state in process creation.

SZOMB = 5 process has exited, waiting for parent to pick up status.

SSTOP = 6 process has been stopped.

flag codes

SLOAD bit 0 – process is loaded in main memory.

SSYS bit 1 – process is swapper or pager process.

SLOCK bit 2 – process is being swapped out.

SSWAP bit 3 – save area flag.

STRC bit 4 – process is being traced.

SWTED bit 5 – parent has been told that this process stopped.

SULOCK bit 6 – user settable lock in core.

SPAGE bit 7 – process in page wait state (not used).

SKEEP bit 8 – flag to prevent swap out.

SOMASK bit 9 – restore old mask after taking signal.

SWEXIT bit 10 – process is exiting.

The Proc Structure (cont.)

p_stat



```
#define SSLEEP      1
#define SWAIT       2
#define SRUN        3
#define SIDL        4
#define SZOMB       5
#define SSTOP       6
```

p_flag

				27	26	25	24
23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0

The Proc Structure (cont.)

flag codes

SPHYSIO bit 11 – process doing physical i/o (bio.c).

SVFORK bit 12 – process resulted from vfork().

SVFDONE bit 13 – vfork flag used in *fork.c.*

SNOVM bit 14 – no vm (virtual memory), parent in a vfork().

SPAGI bit 15 – initialize pages from data space on demand, from inode.

SSEQL bit 16 – user warned of sequential vm behavior.

SUANOM bit 17 – user warned of random vm behavior.

STIMO bit 18 – timing out during sleep.

SPGLDR bit 19 – process is session process group leader.

STRACNG bit 20 – process is tracing another process.

SOWEUPC bit 21 – owe process an *addupc()* call at next ast.

SSEL bit 22 – selecting; wakeup/waiting danger.

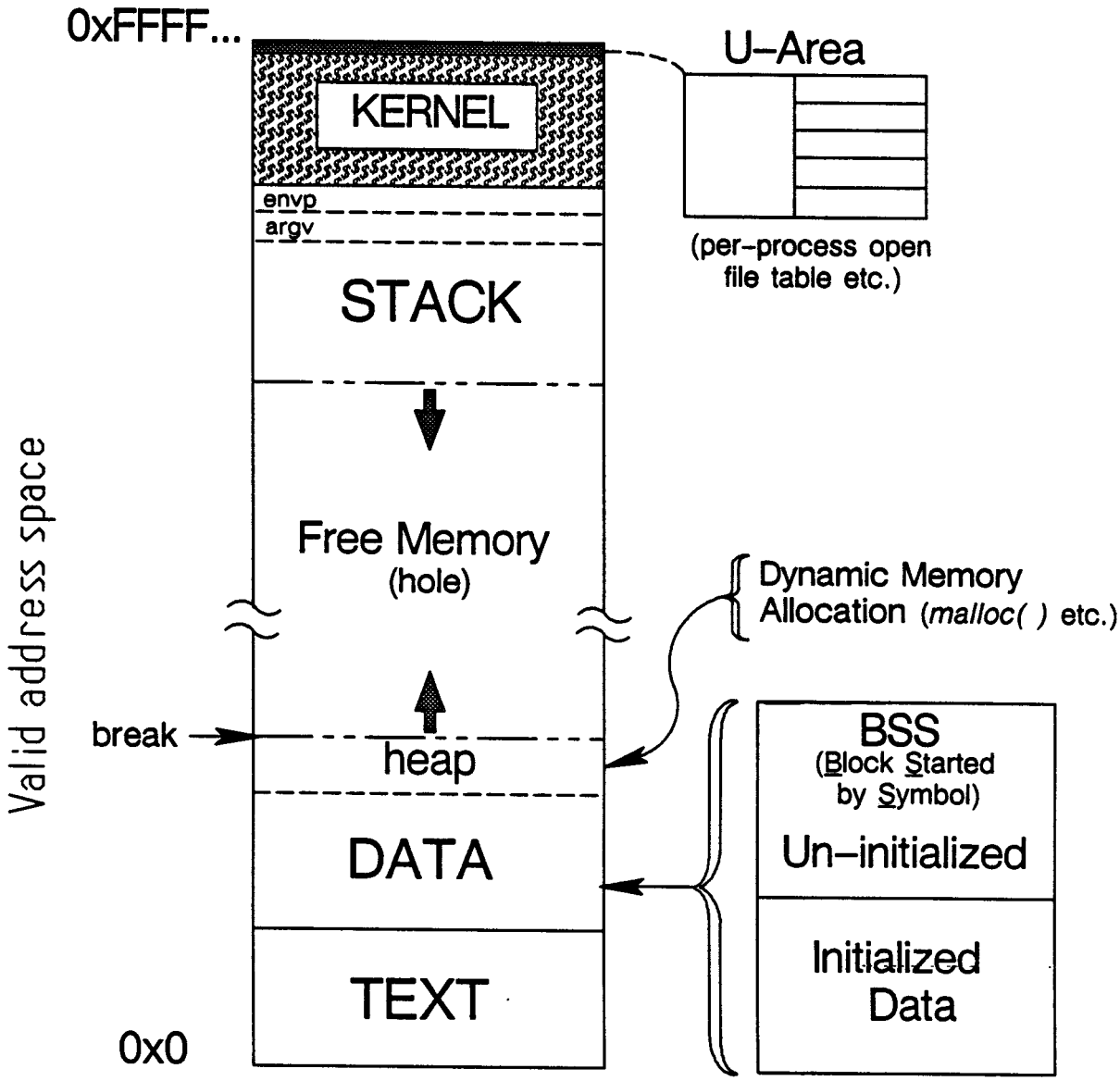
SLOGIN bit 23 – a login process (legit child of init).

SFAVORD bit 25 – favored treatment in swapout and pageout.

SLKDONE bit 26 – record-locking has been done.

STRCSYS bit 27 – tracing system calls.

Process Address Space



Process Address Space

A user's address space is actually just a vector of pages. Although the concept of the address space being broken into text, data, and stack segments is still used as a software abstraction, by compiler and kernel code.

The User Structure

Among other things the user structure contains:

- the kernel stack
- the pcb (process control block) structure
- a pointer to the proc structure (*u_procp)*
- a copy of the system call parameters, results, and error or special actions referenced while in the kernel
- process ids and protections
- sizes of segments for memory management
- signal management information
- descriptor management information and structures
- usage and performance statistics

The User Structure

The user structure is about 2300 bytes not including a fixed size kernel stack. The kernel stack on a Sun3 is 0x1800 bytes, and on a Sun4 0x3000 bytes. The include file for the entire user structure is in Appendix B.

The kernel stack is used by code that executes as the result of a trap or a system call. Interrupts run on a separate stack.

The kernel stack grows from the top of the user structure down. A redzone is set below the user structure as a safe guard against kernel stack overflow. The red zone is a page that is marked read only so a fault will occur.

The *pcb* (process control block) structure varies for different machine architectures. An example of the Sun-3 and Sun-4 *pcb*s are in Appendix B.

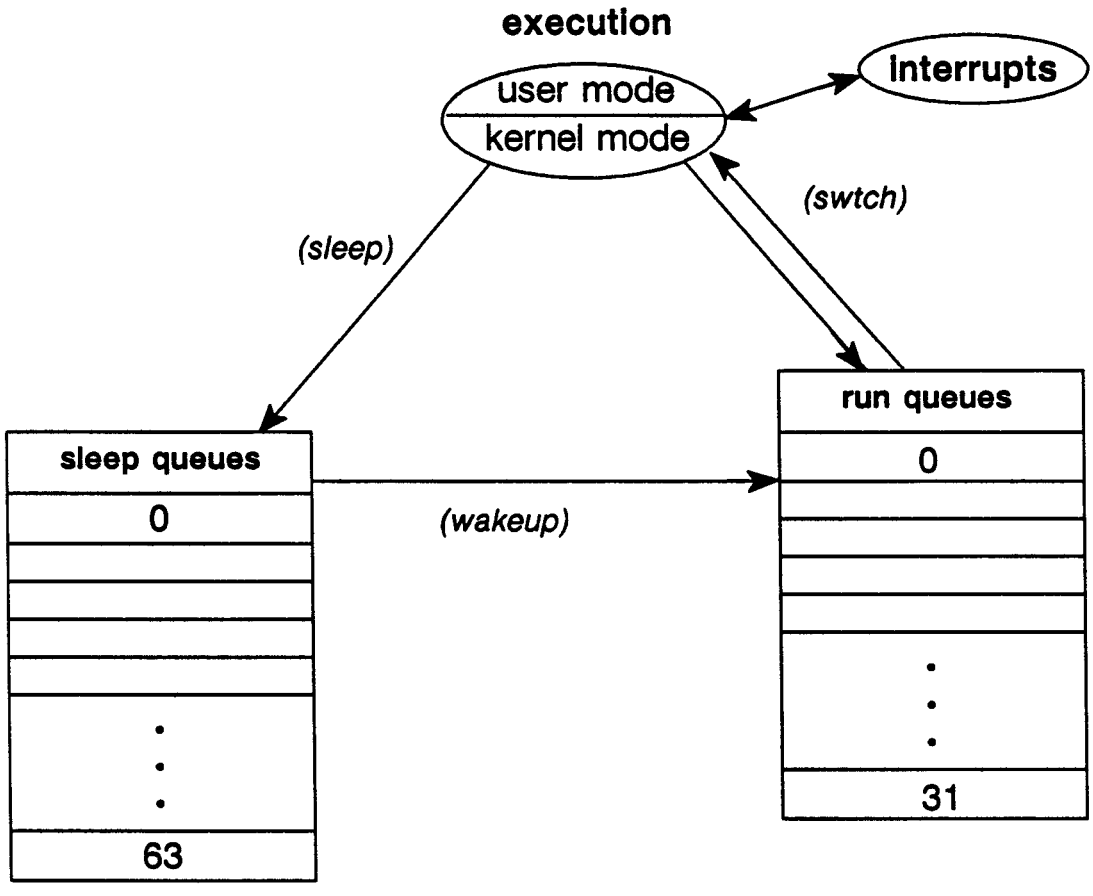
The process ids and protections are referenced through *u_cred* which points to the credential structure (Appendix B).

The size of the text, data, stack segments are kept for memory management.

For signal management we keep track of signals to be blocked, to take on the sigstack, and that interrupt syscalls.

The file descriptors for up to 64 files are kept in the usage structure (*u_ofile[]*).

Process Scheduling



Process Scheduling

Basically a process can either be running or waiting to run.

If the process just needs the CPU and nothing else it will be on one of the 32 run queues.

If a process is waiting for some system resource it will be threaded on one of the system sleep queues. The sleep queue it waits on depends on the channel address specified when it goes to sleep.

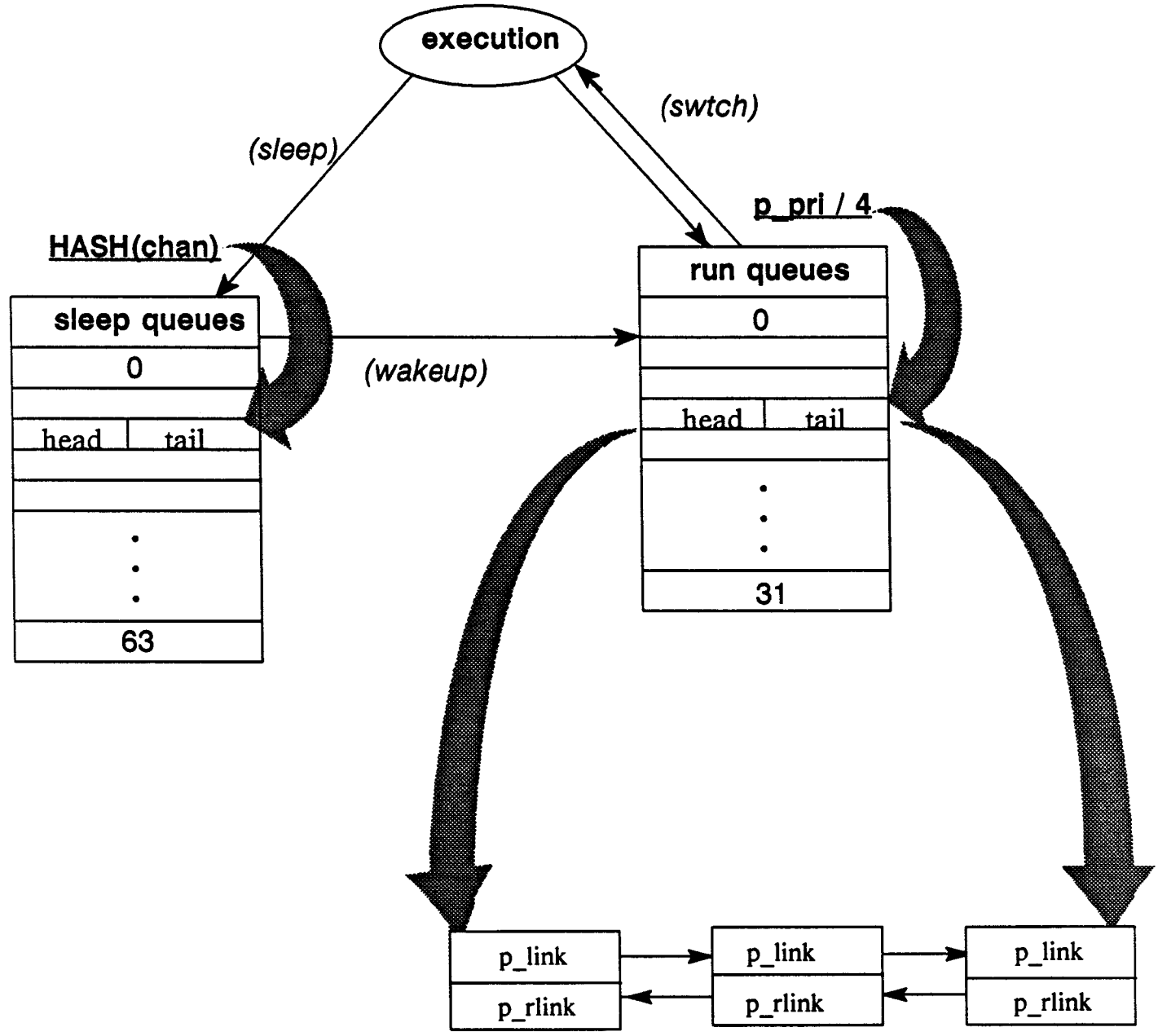
When the event that the process is waiting on takes place then a routine called *wakeup* places all the processes waiting at the channel address into execution. The process that has the highest scheduling priority will usually grab the resource. The other processes will be placed back on the wait channel by the *sleep* routine, if the resource is not available when they execute.

Each process is given a finite amount of time, called a time slice or quanta, to use the CPU.

At the end of this time slice the process must relinquish control of the CPU to the next highest priority process that is ready to run.

The act of a process giving up its execution state to wait for some other system resource (other than the cpu), or because it has completed is called a voluntary context switch.

Queues


execution
(sleep)
(swtch)
p_pri / 4
HASH(chan)
run queues
0
sleep queues
0
(wakeup)
head
tail
head
tail
63
31
p_link
p_rlink
p_link
p_rlink
p_link
p_rlink

Queues

The queues are implemented through a set of doubly linked lists, using the *p_link* and the *p_rlink* of the proc structure.

The sleep queues are hashed into by the wait channel address.

Which run queue a process is placed in is determined by dividing the current process priority by 4, this is done in *setrq()*.

Priorities

run

0	PSWP	1	2	3
1	4	5	6	7
2	8	9	PINOD	11
3	12	13	14	15
4	16	17	18	19
5	PRIBIO	21	22	23
6	24	PZERO	26	27
7	28	29	PWAIT	31
8	32	33	34	PLOCK
9	36	37	38	39
10	PSLEP	41	42	43
11	44	45	46	47
12	48	49	PUSER	51
13	52	53	54	55
14	56	57	58	59
15	60	61	62	63
16	64	65	66	67
17	68	69	70	71
18	72	73	74	75
19	76	77	78	79
20	80	81	82	83
21	84	85	86	87
22	88	89	90	91
23	92	93	94	95
24	96	97	98	99
25	100	101	102	103
26	104	105	106	107
27	108	109	110	111
28	112	113	114	115
29	116	117	118	119
30	120	121	122	123
31	124	125	126	127

Priorities

Priorities range from 0 to 127 with 0 being the highest priority and 127 the lowest.

Any Kernel process with a priority of PZERO (25) + 1 to PUSER may be awakened to receive a signal.

User priorities range from PUSER (50) to 127. Kernel processes range from 0 to PUSER–1.

Any Kernel process with a priority less than or equal to PZERO can NOT be awakened by any signals posted to the process, so any operations executed by a kernel process in this priority range is guaranteed to be atomic. That is it will finish its current operation without a context switch.

The priorities are frequently shifted according to cpu usage, *p_cpu*. Every clock tick the *hardclock()* routine increments *p_cpu* for the executing process.

Other priorities defined in *param.h* are:

PSWP 0 – the priority while moving a process to or from swap space.

PINOD 10 – the priority while waiting for file control information.

PRIBIO 20 – the priority while waiting for a disk block.

PWAIT 30 – the base priority while waiting for a resource.

PLOCK 35 – the base priority while waiting for a resource lock.

PSLEP 40 – the base priority while waiting for an event.

Scheduling

There are several routines involved in scheduling a process to run and determining its priority *(p_pri)*.

roundrobin – forces a switch between processes running on the same run queue level every 100 milliseconds.

setpri – sets a user priority.

schedcpu – recalculates process priorities once per second.

sleep – give up the processor, wait on a sleep queue until a *wakeup* call is issued.

wakeup – wakeup all processes at a specified wait channel address.

unsleep – remove a process from its sleep queue.

updatepri – recalculate the process priority if it has been sleeping more than a second.

Scheduling

roundrobin() is the routine that forces an involuntary context switch when a process has exhausted its timeslice. The timeslice is set to .1 seconds, after a process has been executing for that long *roundrobin()* is called to cause the next highest priority process in the run queues to be executed. It threads the currently executing process at the end of its appropriate run queue.

unsleep() is called with the address of a proc structure to wake up one process off a sleep queue. This is done when a signal occurs for a process that is sleeping with a priority number greater than *PZERO.*

updatepri() is called by *wakeup()* when it is determined that the process has been sleeping for more than one second. The process cpu utilization is updated using the following equation:

$$p_cpu = \left[\frac{2 * load}{2 * load + 1} \right]^{p_slptime} * p_cpu$$

Context Switches

If a process leaves execution to wait for some resource other than the cpu, it is called an voluntary context switch.

If a process gives up its execution state because it exhausted its time slice or because a higher priority process pre–empts it, this is called an involuntary context switch.

The low level routines involved in context switching are:

- swtch – searches run queues and selects a process to put in execution.
- setrq – puts processes into the appropriate run queues
- remrq – removes processes from the queues
- resume – sets up the process for execution

Context Switches

The majority of context switches are voluntary.

If it is an involuntary context switch the process will be returned to its appropriate run queue according to its scheduling priority.

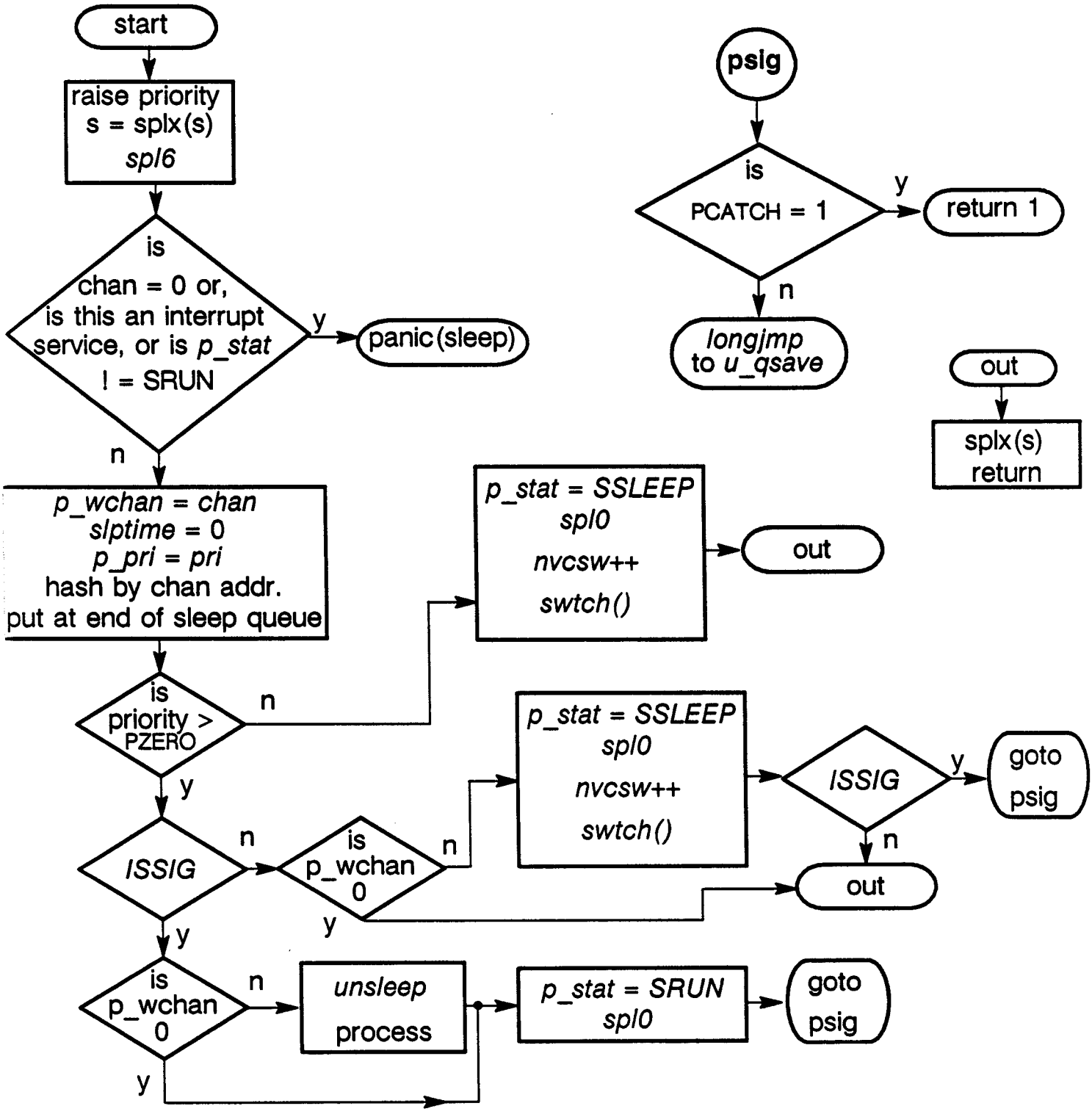
The *swtch()* routine uses a bitmap, *whichqs,* to locate the highest priority non empty queue. It removes the process from the run queues by calling *remrq()*, and calls *resume()* to set the process running.

The *setrq()* routine simply divides the process priority *p_pri* by 4 to select a run queue and threads the process at the end of that queue. So a process with a priority of 51 goes on the same run queue as a process with a priority of 48.

The *remrq()* routine removes the first process from the head of the queue.

The *resume()* routine initiates the system's general registers from the *pcb,* and sets the process into execution (called by *swtch()*).

sleep()


start
raise priority
s = splx(s)
spl6
is
chan = 0 or,
is this an interrupt
service, or is p_stat
! = SRUN
y
panic(sleep)
n
p_wchan = chan
slptime = 0
p_pri = pri
hash by chan addr.
put at end of sleep queue
is
priority >
PZERO
n
p_stat = SSLEEP
spl0
nvcsw++
swtch()
out
y
ISSIG
n
is
p_wchan
0
n
p_stat = SSLEEP
spl0
nvcsw++
swtch()
ISSIG
y
goto
psig
n
out
y
y
is
p_wchan
0
n
unsleep
process
p_stat = SRUN
spl0
goto
psig
y
psig
is
PCATCH = 1
y
return 1
n
longjmp
to u_qsave
out
splx(s)
return

sleep()

The *sleep()* routine is called with the channel address, *chan,* and the priority, *pri,* it is to have when it is set running again.

If *chan* is 0, or this is an interrupt routine, or if this process was not flagged as *SRUN*, something is seriously wrong, panic!

Set the wait channel address to *chan, p_pri* to *pri,* and the sleeptime to zero in the *proc* structure.

Hash the channel address *chan,* to select one of the system sleep queues to put this process on. Put it on the end of that sleep queue.

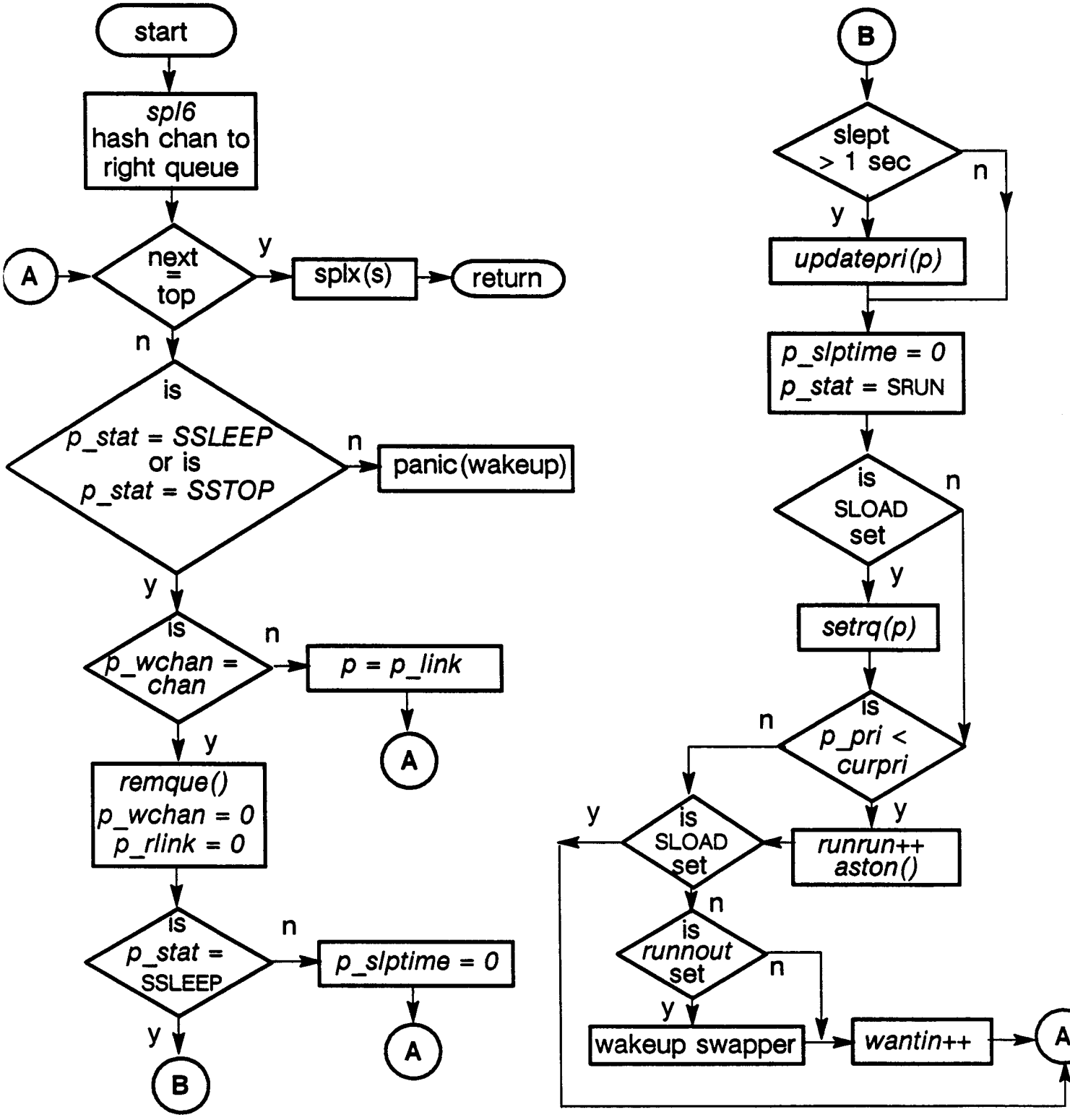
If the process priority is less than or equal to *PZERO* set the status to *SSLEEP,* lower the priority to enable interrupts, increment the number of voluntary context switches in the user's *ru* structure, and run somebody else.

If the process priority is greater than *PZERO*, so that the process is receiving signals, check if a signal has arrived.

> If a signal has arrived wakeup the process with the *unsleep* routine. Set the status to *SRUN,* and go to *psig.*
>
> If it was not a signal, make sure somebody didn't wake it during *ISSIG.*
>
> > If it's already awake, then go to out.
> >
> > If its *p_wchan* is still nonzero, set the status to *SSLEEP,* lower the priority to enable interrupts, increment the number of voluntary context switches in the user's *ru* structure, and run somebody else.

When the process runs again after a switch, if it was a signal go to *psig.* Otherwise go to out, which just lowers the priority and returns 0.

wakeup()


start
spl6
hash chan to right queue
A
next = top
y
splx(s)
return
n
is p_stat = SSLEEP or is p_stat = SSTOP
n
panic(wakeup)
y
is p_wchan = chan
n
p = p_link
A
y
remque()
p_wchan = 0
p_rlink = 0
is p_stat = SSLEEP
n
p_slptime = 0
A
y
B
B
slept > 1 sec
n
y
updatepri(p)
p_slptime = 0
p_stat = SRUN
is SLOAD set
n
y
setrq(p)
is p_pri < curpri
n
y
is SLOAD set
y
runrun++
aston()
n
is runnout set
n
y
wakeup swapper
wantin++
A

wakeup()

The *wakeup()* routine will wakeup all the processes with the same *p_wchan* value.

Raise the priority to prevent interrupts. Hash *chan* to get to the right sleep queue.

For every process in this sleep queue:

If the process is not sleeping and not stopped serious problems, panic!

If this process is on this sleep queue but did not sleep with the *chan* that we're waking, get the next process. In other words processes with different channel addresses can hash to the same sleep queue.

If it is the same channel address, remove it from the queue, and zero out the wait channel address and the backward link in the *proc* structure.

If the status is not *SSLEEP,* set the sleep time to zero.

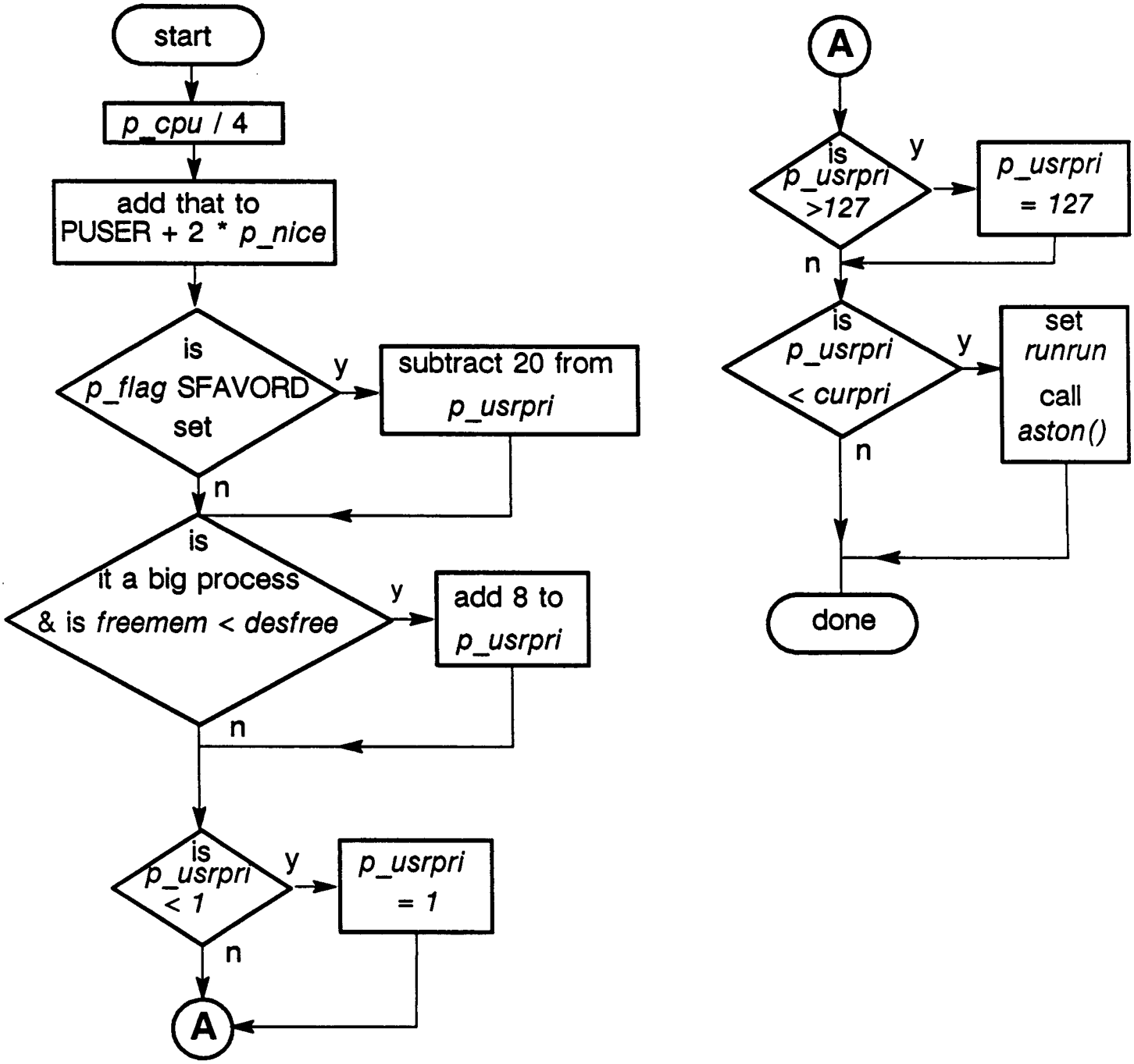
If the process has slept more than a second, call *updatepri()* to bump up the priority.

Set sleeptime to zero and the status to *SRUN.*

If a process is resident in memory, call *setrq(p)* to put it on the appropriate run queue. After putting it on a run queue, or if it wasn't resident in memory to begin with, check if it has a higher priority than the currently executing process. If so, increment *runrun* and call *aston()* to cause a context switch as soon as possible.

If the process is not resident in main memory, increment *wantin* to indicate that the process needs to be swapped in.

setpri()


start
p_cpu / 4
add that to PUSER + 2 * p_nice
is p_flag SFAVORD set
y
subtract 20 from p_usrpri
n
is it a big process & is freemem < desfree
y
add 8 to p_usrpri
n
is p_usrpri < 1
y
p_usrpri = 1
n
A
A
is p_usrpri >127
y
p_usrpri = 127
n
is p_usrpri < curpri
y
set runrun call aston()
n
done

setpri()

The *setpri()* routine is called to reassign a process priority. It is called from *schedcpu*.

For each clock tick (when *hardclock()* executes) that a process is found in execution the *p_cpu* field in the proc structure is incremented.

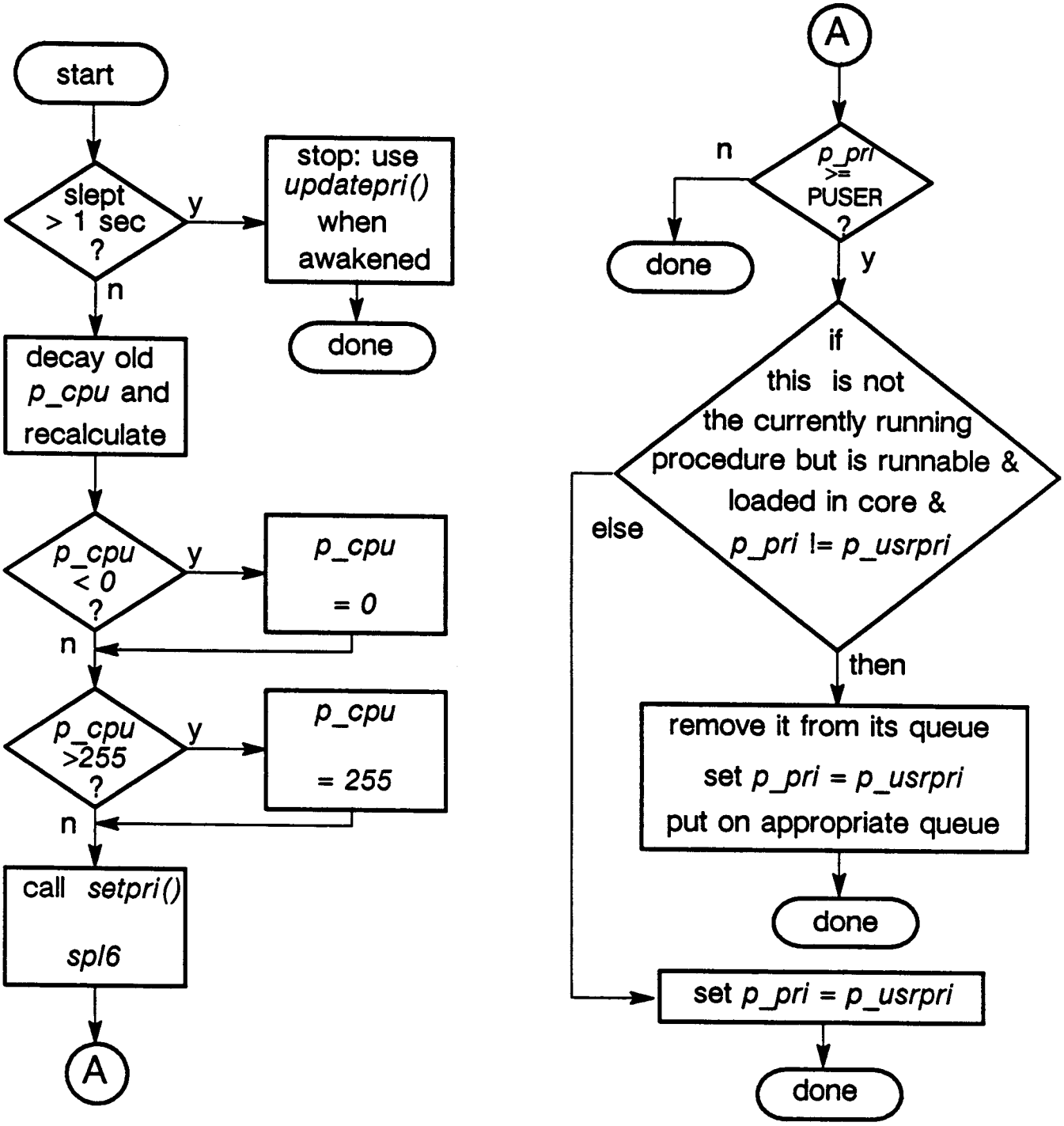
If a process has the *p_flag SFAVORD* flag set, 20 is subtracted from its priority giving more execution time.

If the process is using *maxrss* (maximum resident set size) or more pages and the system is low on memory, add 8 to its priority. Since this gives the process less execution time and it's over *maxrss*, its pages are more likely to be paged out. See the section on the page replacement algorithm.

Make sure the priority is clamped between 1 and 127.

If the resultant priority is less (ie at higher priority) than the currently executing process set the global flag *runrun*, and call *aston()* to cause a context switch as soon as possible.

schedcpu()


start
slept > 1 sec ?
y
stop: use updatepri() when awakened
done
n
decay old p_cpu and recalculate
p_cpu < 0 ?
y
p_cpu = 0
n
p_cpu >255 ?
y
p_cpu = 255
n
call setpri()
spl6
A
A
n
p_pri >= PUSER ?
done
y
if this is not the currently running procedure but is runnable & loaded in core & p_pri != p_usrpri
else
then
remove it from its queue
set p_pri = p_usrpri
put on appropriate queue
done
set p_pri = p_usrpri
done

schedcpu()

The *schedcpu()* routine runs once per second to reshuffle the priorities of user processes.

For all the allocated process structures:

If this process has been sleeping for more than a second, don't recalculate the priority until it wakes up. At that time use *updatepri()*.

Decay the old cpu usage and fold in the recent cpu usage (see next page).

Clamp the cpu usage field in the proc structure to somewhere between 0 and 255.

Call *setpri()* to calculate a new priority based on recent cpu usage. Raise the priority to prevent interrupts.

Is the new priority in the user range? If not we're done.

If this process is not executing, but its loaded, and we've changed its priority, then set the running priority to the user priority and put it on the right queue.

Otherwise just set the running priority to the user priority.

Computing CPU Usage

The cpu usage is calculated within *schedcpu* according to the following equation:

$$p_cpu = p_cpu * \frac{(2 * load)}{(2 * load + 1)} + nice$$

Computing CPU Usage

$$p_cpu = p_cpu * \frac{(2 * load)}{(2 * load + 1)} + nice$$

example if load equals 1:

$$p_cpu = p_cpu * \frac{2}{3} + nice$$

example if load equals 10:

$$p_cpu = p_cpu * \frac{20}{21} + nice$$

The values used to compute the cpu usage are actually shifted by a factor of 256 in order to avoid any floating point calculations. The reason the computations are scaled by 256 is because the *p_cpu* field in the proc structure is capped at 255 (see *schedcpu*).

Creation of Processes

- The creation of a process is accomplished using the *fork(2)* system call.

- The fork call was changed at 4.0 to copy only the mappings from the parent process to the child.

- The vfork system call still exists in which case the child borrows the parents address space until an exec or exit.

Creation of Child Processes

The *fork()* system call creates a new process by creating a duplicate of the calling process. This new process is termed the *child* process, while the calling process is termed the *parent* process.

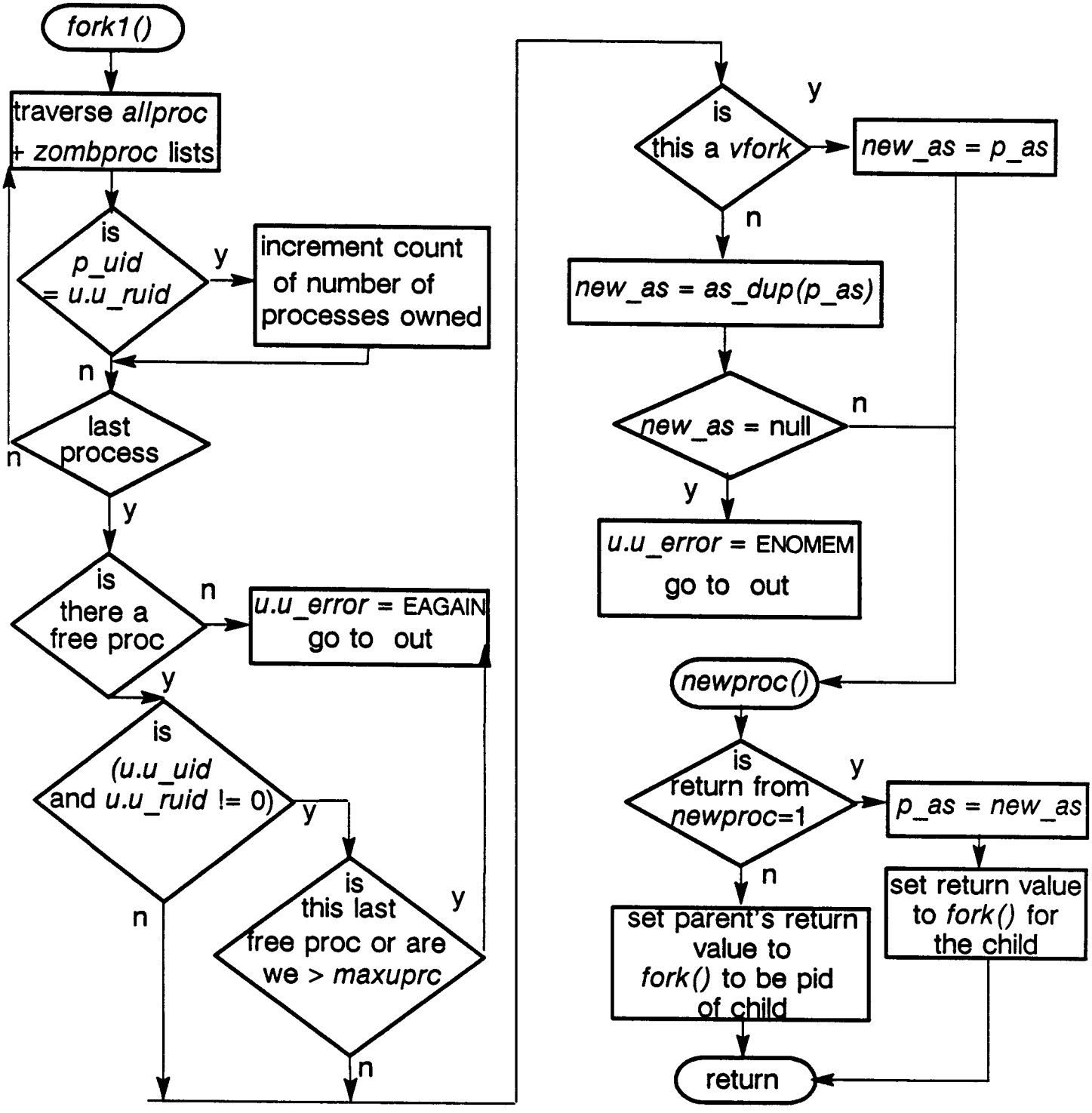
The two processes can be differentiated from each other based on the value returned by *fork()*. When the call to *fork()* returns to the parent process, the return value is the child process' PID, while the return value in the child process will be zero. As with most other system calls, if a call to *fork()* fails, the return value will be –1. Some instances where *fork()* will fail, and no child process will be created are:

- The system-imposed limit on the total number of processes under execution would be exceeded.
- The system-imposed limit on the total number of processes under execution by a single user (UID of process) would be exceeded.
- There is insufficient swap space for the new process.

By copying the mappings to the parent process rather than the contents of the address space, considerable time is saved. The address space of the child is typically overlaid by an *execve()* call immediately after the fork so the parent's address space is not needed.

The *vfork()* system call was initially removed at 4.0, but unfortunately too many routines were using the fact that the child could modify the parent's address space before returning it to the parent as a means of communicating information back to the parent process.

fork1()


fork1()
traverse allproc + zombproc lists
is p_uid = u.u_ruid
y
increment count of number of processes owned
n
last process
n
y
is there a free proc
n
u.u_error = EAGAIN go to out
y
is (u.u_uid and u.u_ruid != 0)
y
is this last free proc or are we > maxuprc
y
n
n
is this a vfork
y
new_as = p_as
n
new_as = as_dup(p_as)
new_as = null
n
y
u.u_error = ENOMEM go to out
newproc()
is return from newproc=1
y
p_as = new_as
set return value to fork() for the child
n
set parent's return value to fork() to be pid of child
return

fork1()

fork() calls *fork1()* passing a 0 for a fork and a 1 for a vfork.

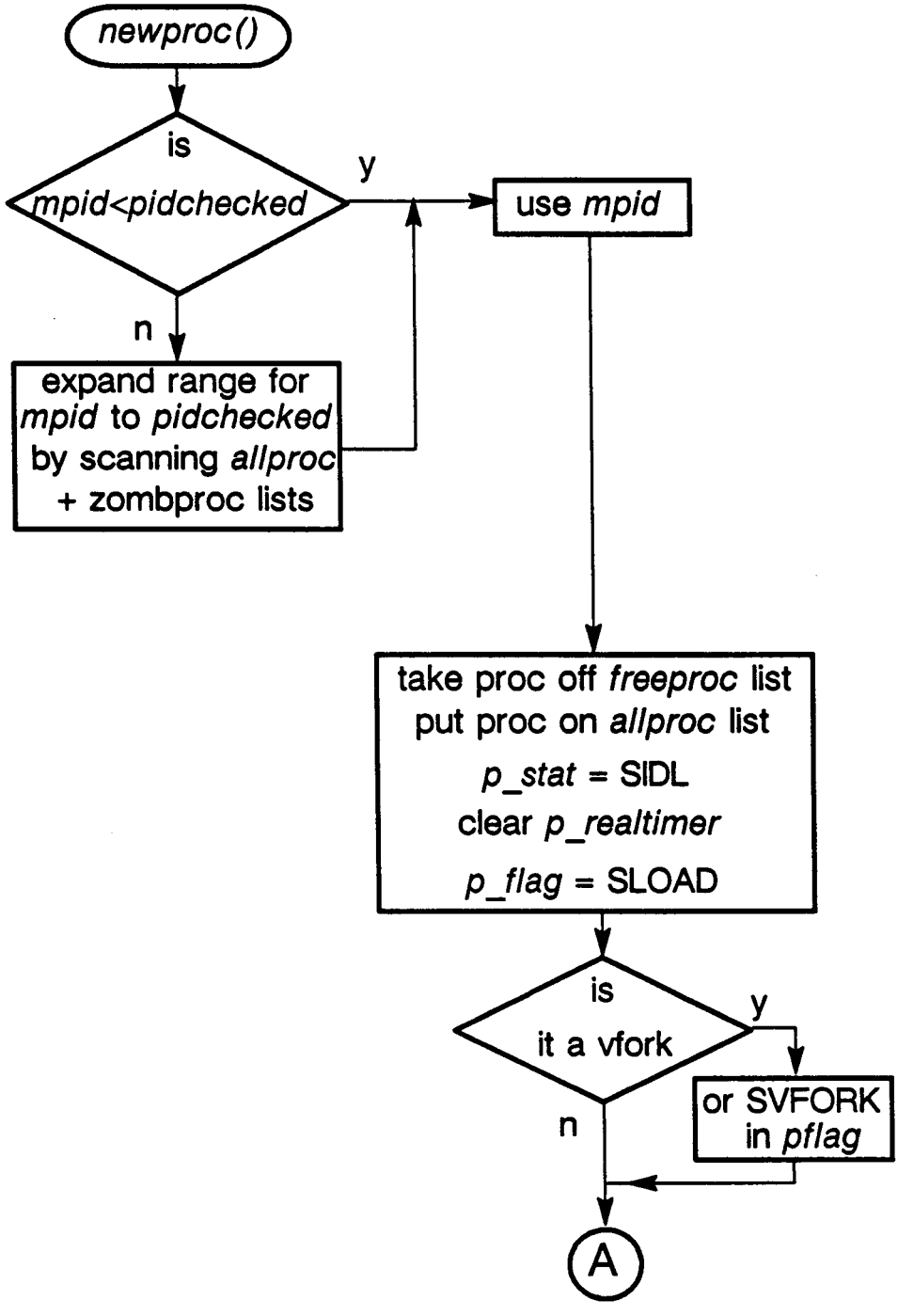
Start by counting how many user processes we have on the *allproc* list and the *zombproc* list.

If we're out of proc structures return EAGAIN error to user structure. If this is not super user (or does not have an effective *uid* of 0) and it's the last proc structure slot, or if the number of procs in use is greater than *maxuprc* then return EAGAIN.

If this is a *vfork* operation then point to the same address structure, otherwise make a duplicate of the mapping information in the *as* structure for the new process.

Call *newproc()* to initialize proc structure. When we have returned from *newproc()* check if it is the child or the parent process. If it is the child process set the *p_as* pointer to the returned pointer (from *newproc()*) and return zero to the child process. If it is the parent process set the return value to be the child's *pid*.

newproc()

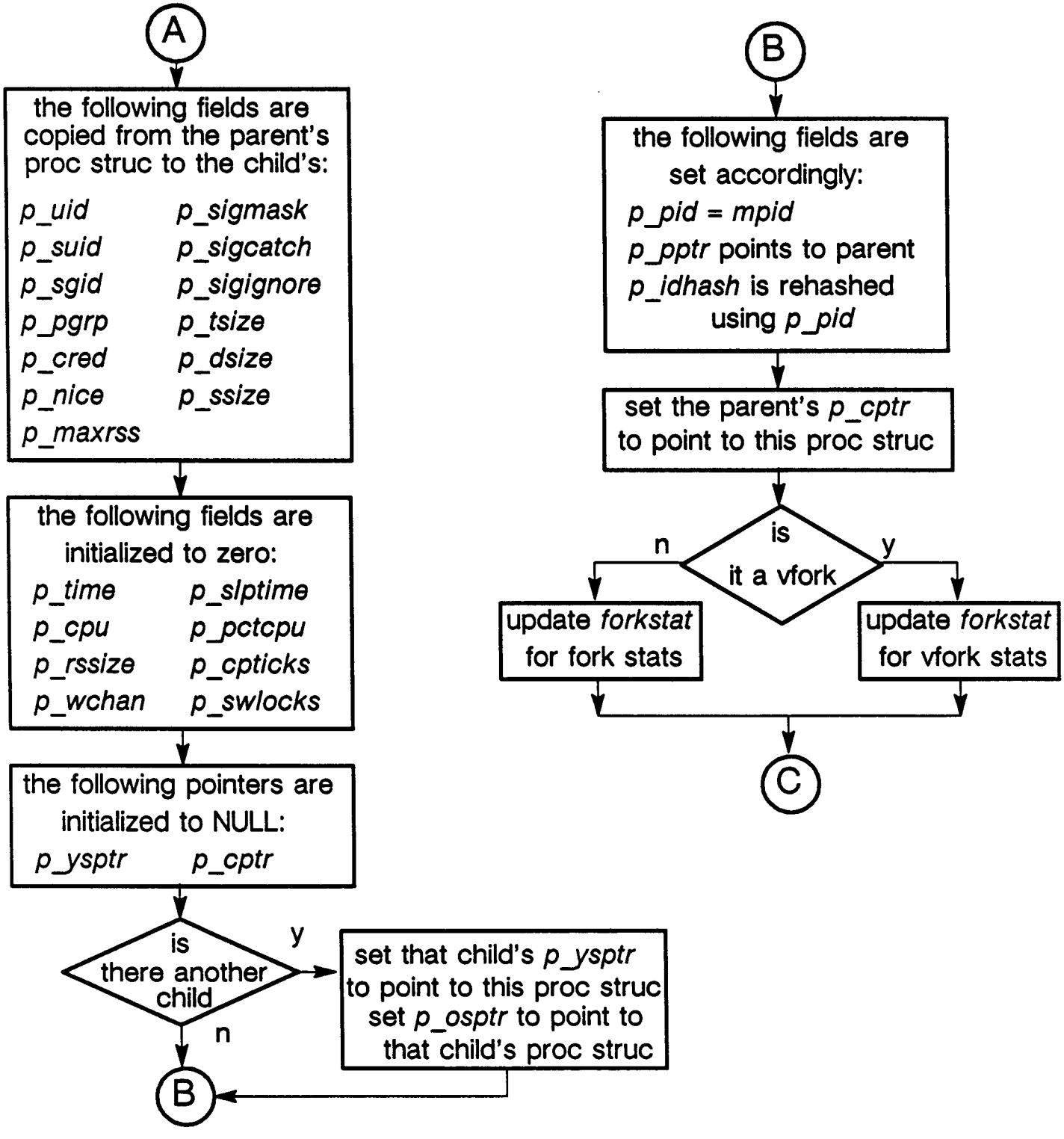

newproc()
is
mpid<pidchecked
y
use mpid
n
expand range for
mpid to pidchecked
by scanning allproc
+ zombproc lists
take proc off freeproc list
put proc on allproc list
p_stat = SIDL
clear p_realtimer
p_flag = SLOAD
is
it a vfork
y
or SVFORK
in pflag
n
A

newproc()

In *newproc()* find a unique *pid* the new process can use. We take *mpid* off the range of available *pids,* which is from *mpid* to *pidchecked.* If there is nothing available in that range (that is *mpid* ≥ *pidchecked*) then we expand the range, by searching the *allproc* list and *zombproc* lists for unused *pids. mpid* will be the *pid* used for the child process.

Take a proc structure from the head of the *freeproc* list, put it on the *allproc* list. Mark it as SIDL, clear the realtimer, and set the SLOAD flag.

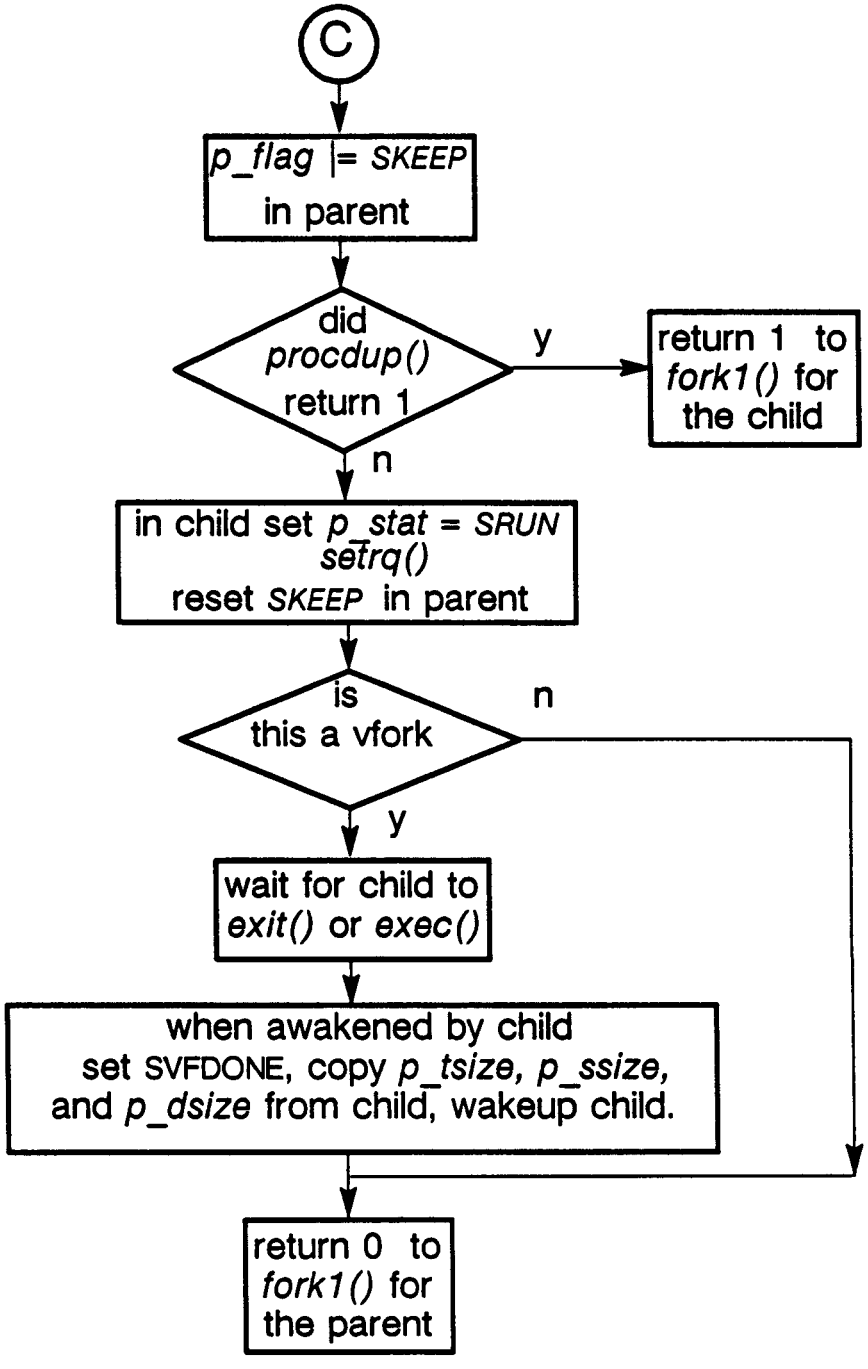
newproc() (cont.)



newproc() **(cont.)**

The *p_cptr* field points to this process's child process. If this process is the parent of more than one process, then the other 'children' are linked to the first child process through the *p_ysptr* (younger sibling pointer) and older siblings of other child processes are pointed to by *p_osptr.*

newproc() (cont.)



newproc()(cont.)

Lock parent in memory (using the *SKEEP* flag in proc structure) while initializing the user structure for the new process.

Call *procdup()* to set up the user area for the child, it will return a 1 to *newproc()* for the child process, and a zero to the parent (see next page).

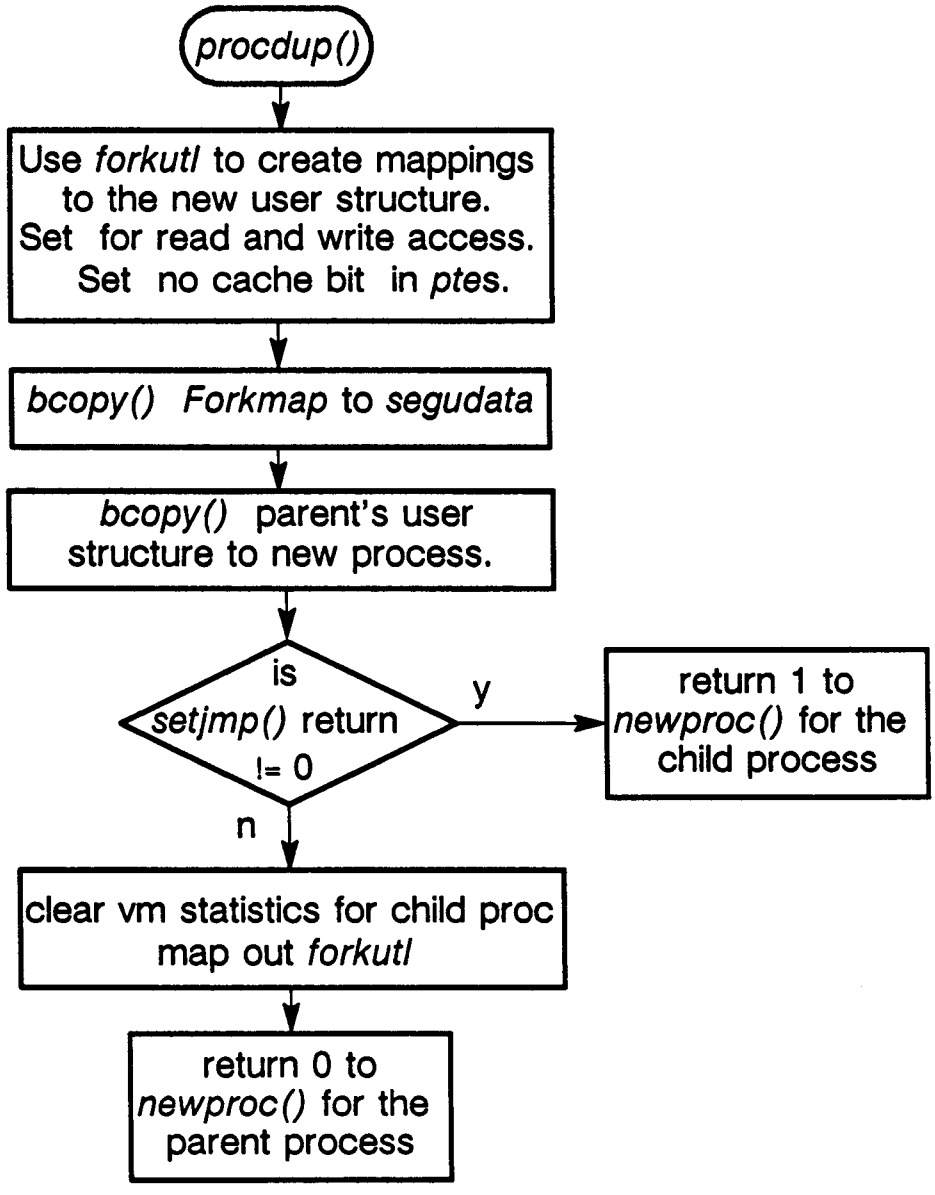
When returning to the parent set the child running.

If the child was spawned by a call to *vfork(2),* go to sleep using the address of the child's proc structure, until the child completes *(exit()),* or executes another program *(exec()).*

The parent is awakened by the child from a routine called *relvm(),* (used to release the virtual memory back to parent) which was in turn called from *exec()* or *exit().* At that time set the SVFDONE bit, copy the size fields from the child (who may have grown), and wakeup the child.

In either case (*vfork* or not), return a 0 to *fork1()* for the parent.

procdup() (cont.)



procdup() (cont.)

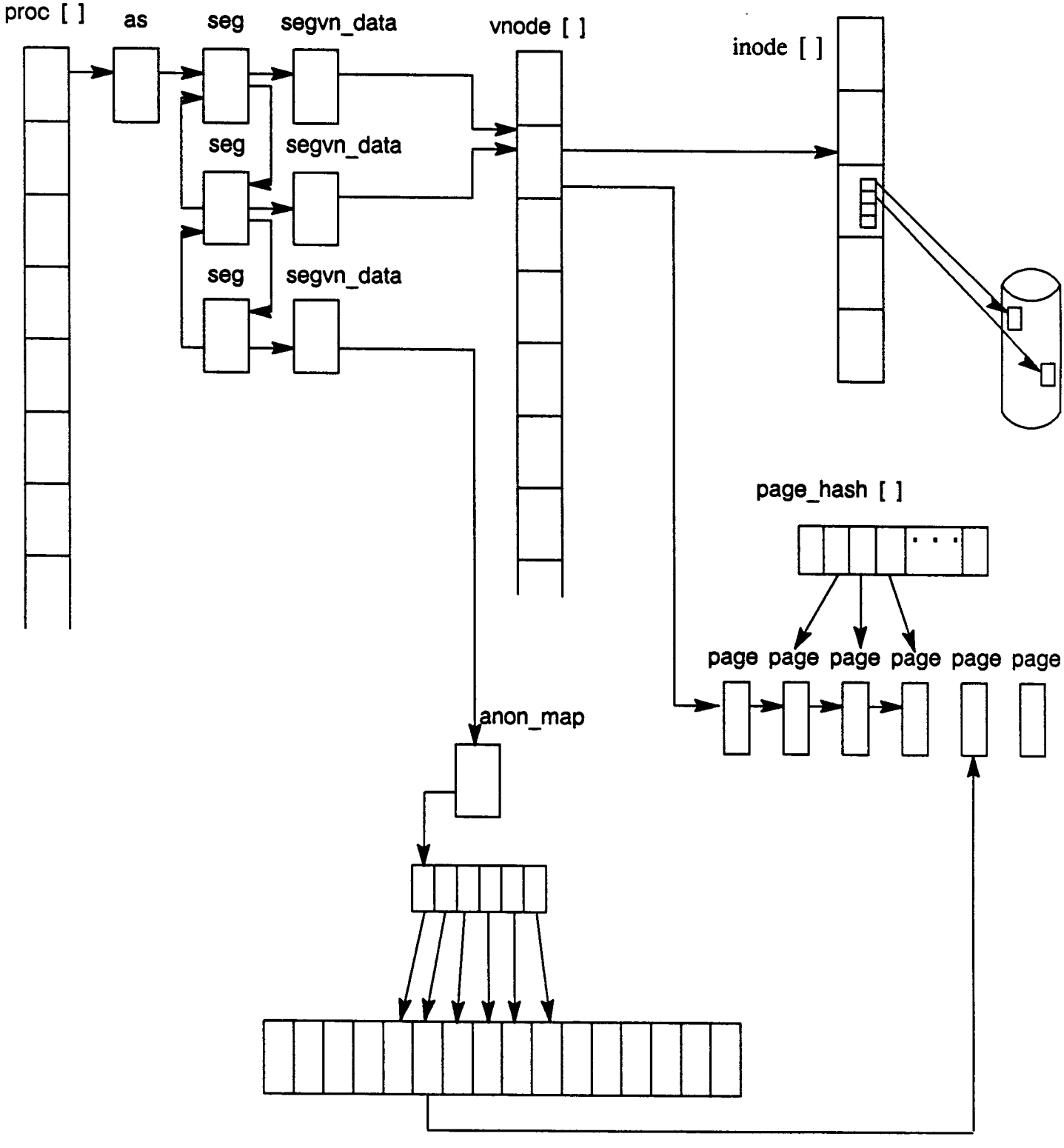
The *procdup()* uses a space in the kernel's virtual address space called *forkutl* to create a user structure. The *segkmem_mapin()* routine creates the mappings to this area and places the mappings in *Forkmap*.

The mappings are then placed in a statically allocated array of mappings to the user structures that has a one to one correspondence with the proc structures. The array of user structure mappings is *segudata*.

Now *bcopy()* the parent's user structure to the child.

Setup for the nonlocal goto, so that when the new process is resumed it will start here. For the child process it will actually be *resume()* rather than *setjmp()* that will return a nonzero value. In this case, 1 is returned to *newproc()* for the child. If not we clear out the vm statistics of the new process, call *segkmem_mapout* to release the mappings to *forkutl*, since they've been copied to *segudata.* Return 0 to *newproc()* for the parent.

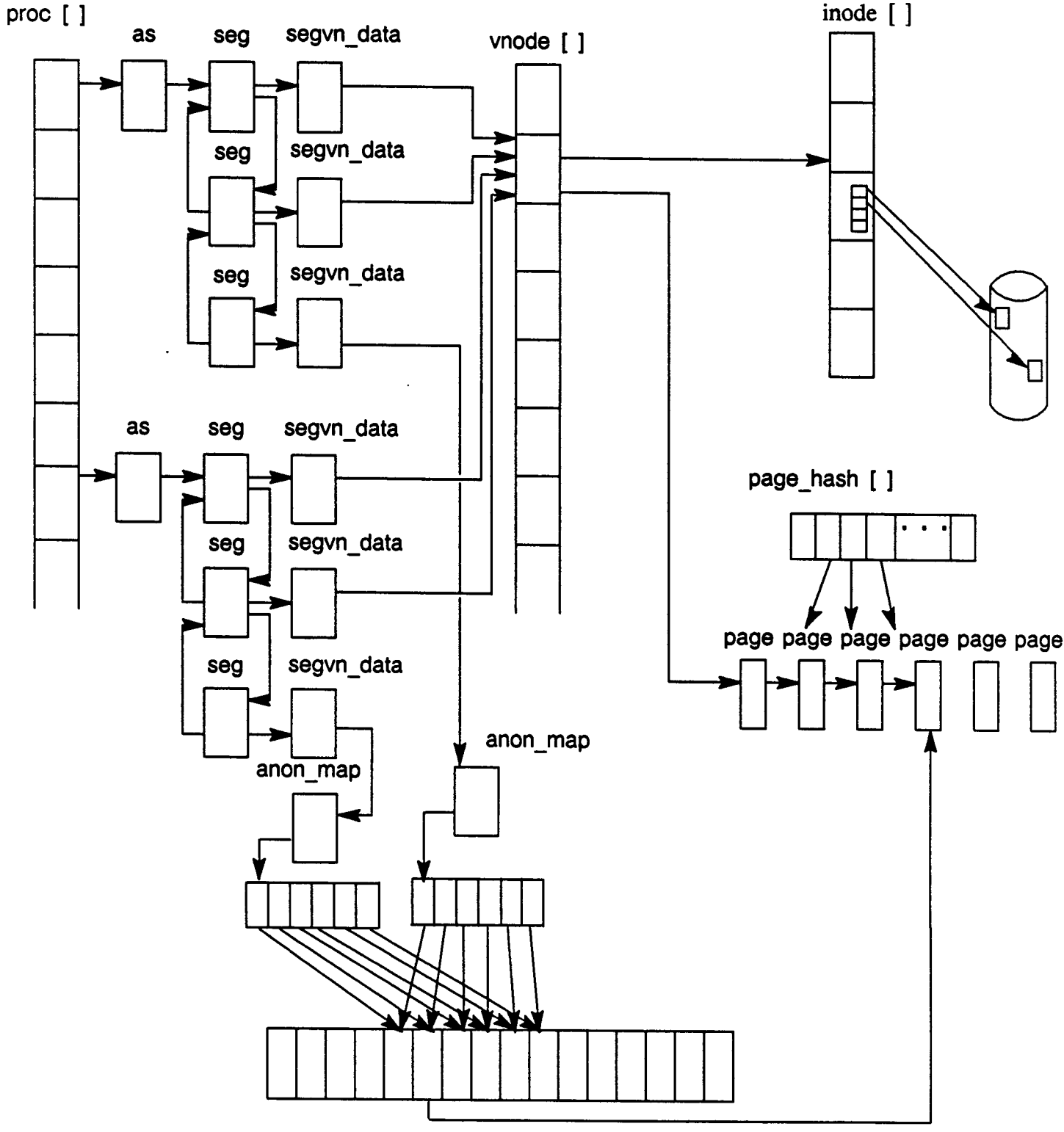
Before Fork


proc [ ]
as
seg
segvn_data
vnode [ ]
inode [ ]
seg
segvn_data
seg
segvn_data
page_hash [ ]
page
page
page
page
page
page
anon_map

Before Fork

This diagram illustrates the structures that are used to map a process's address space prior to a *fork()* system call.

After Fork


proc [ ]
as
seg
segvn_data
vnode [ ]
inode [ ]
seg
segvn_data
seg
segvn_data
as
seg
segvn_data
page_hash [ ]
seg
segvn_data
page page page page page page
seg
segvn_data
anon_map
anon_map

After Fork

After the *fork()* system call it can be seen that very little physical memory is used compared to the way fork was previously done. Only the seg, segvn_data, and anon structures need to be duplicated and the actual text, data and stack pages are shared.

execve()

- *execve()* loads a process image from an executable disk file
- *execve()* is passed arguments via the *argv* vector
- *execve()* can be passed an environment from the *envp* argument

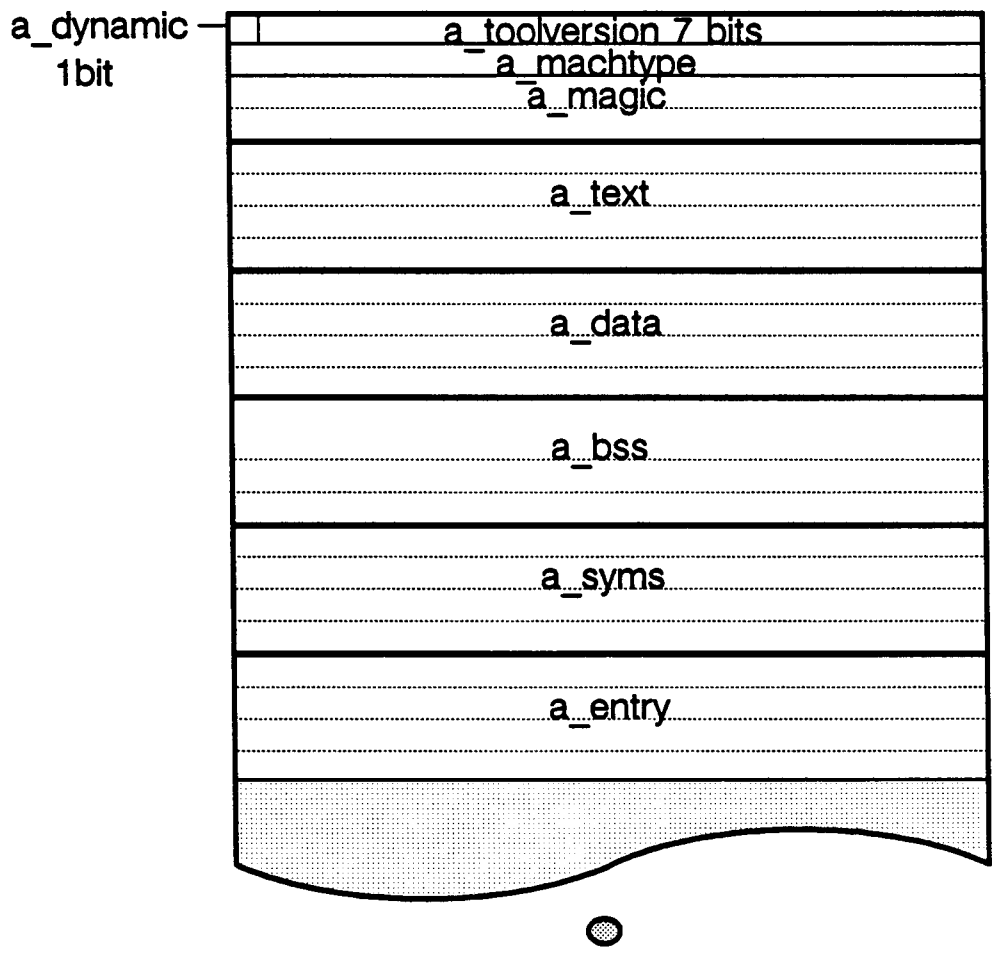
execve()

execve(name, argv, envp) transforms the calling process into a new process. The new process image is constructed (as per any process invocation) from an ordinary disk file, pointed to by the argument *name.* This file can be either an executable object file (e.g. a.out) or a shell script. The second argument is the *argv* vector for the new process which, by convention, contains at least one argument: the name of the program which has been *exec*–ed, i.e. the last component of *name.* Other arguments can be added as needed. The third argument *envp* is the *environment* passed to the new process image. This *environment* can either be the same as that of the calling process, or it may be hand–built or tailored for use by the new process.

There can be no return from a *successful* call to *execve()* as the calling process's core image is overwritten by the *exec*–ed program's image. This is the mechanism whereby all process images become active.

execve() does not conduct PATH searching to locate *name*, so *name* must either be a relative, or an absolute pathname.

exec structure



exec structure

a_dynamic – has a _DYNAMIC (means it's using shared libraries)

a_toolversion – version of toolset used to create file

TV_SUN2_SUN3 – 0

TV_SUN4 – 1

a_machtype – machine type where:

M_OLDSUN2 – 0 – old Sun–2 executable file

M_68010 – 1 – runs on M68010 or M68020

M_68020 – 2 – runs on M68020

M_SPARC – 3 – runs only on sparc

a_magic – magic number where:

OMAGIC – 0407 – old impure format

NMAGIC – 0410 – read only text

ZMAGIC – 0413 – page–in on demand text

a_text – the size of the text segment in bytes

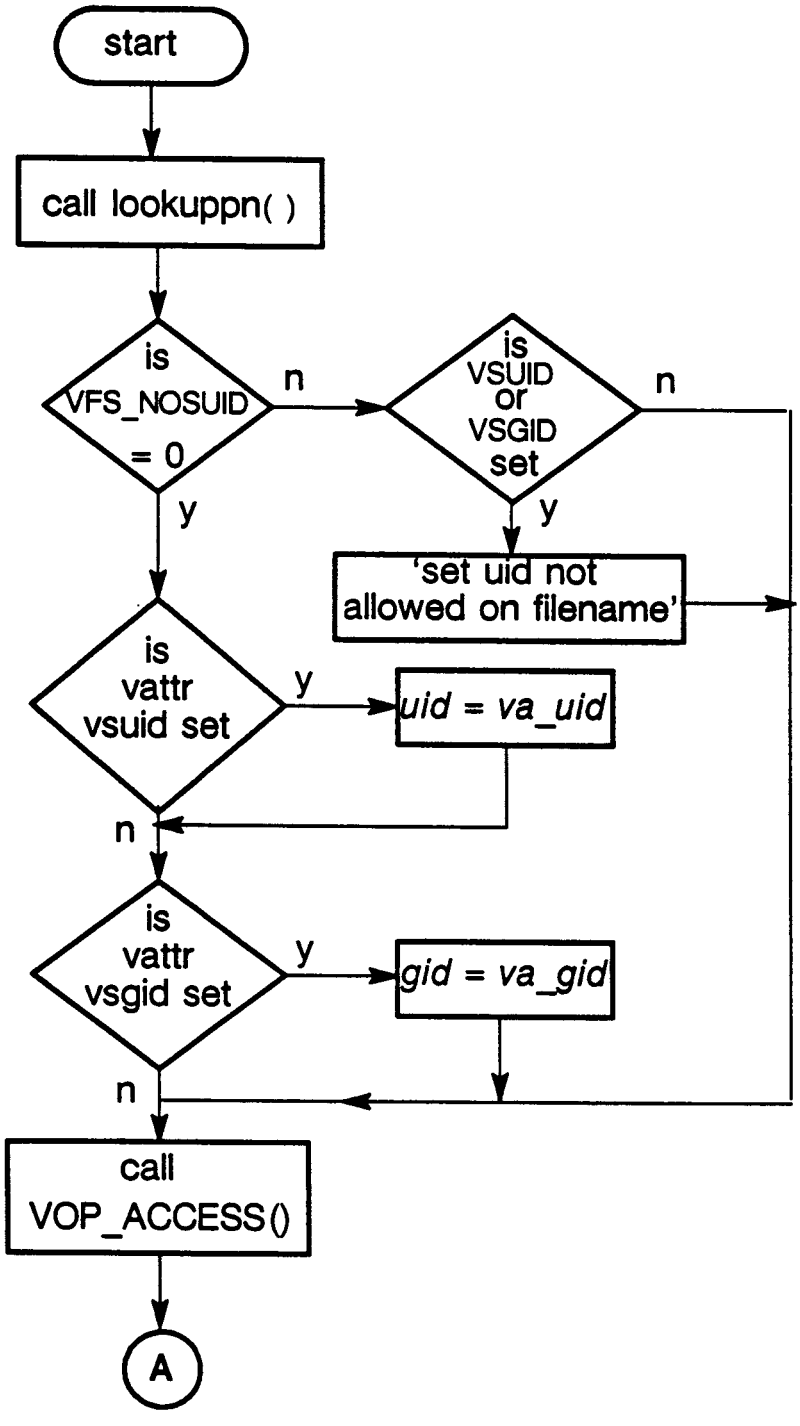
a_data – the size of the initialized data segment in bytes

a_bss – the size of the bss (uninitialized data) segment in bytes

a_syms – the size of the symbol table

a_entry – the entry point

execve()


start
call lookuppn( )
is VFS_NOSUID = 0
n
is VSUID or VSGID set
n
y
y
'set uid not allowed on filename'
is vattr vsuid set
y
uid = va_uid
n
is vattr vsgid set
y
gid = va_gid
n
call VOP_ACCESS()
A

execve()

The *exec()* routine releases the address space mappings of a process and replaces them with mappings to an executable file in the file system.

Execve() calls *lookuppn()* to look up the pathname of the executable file.

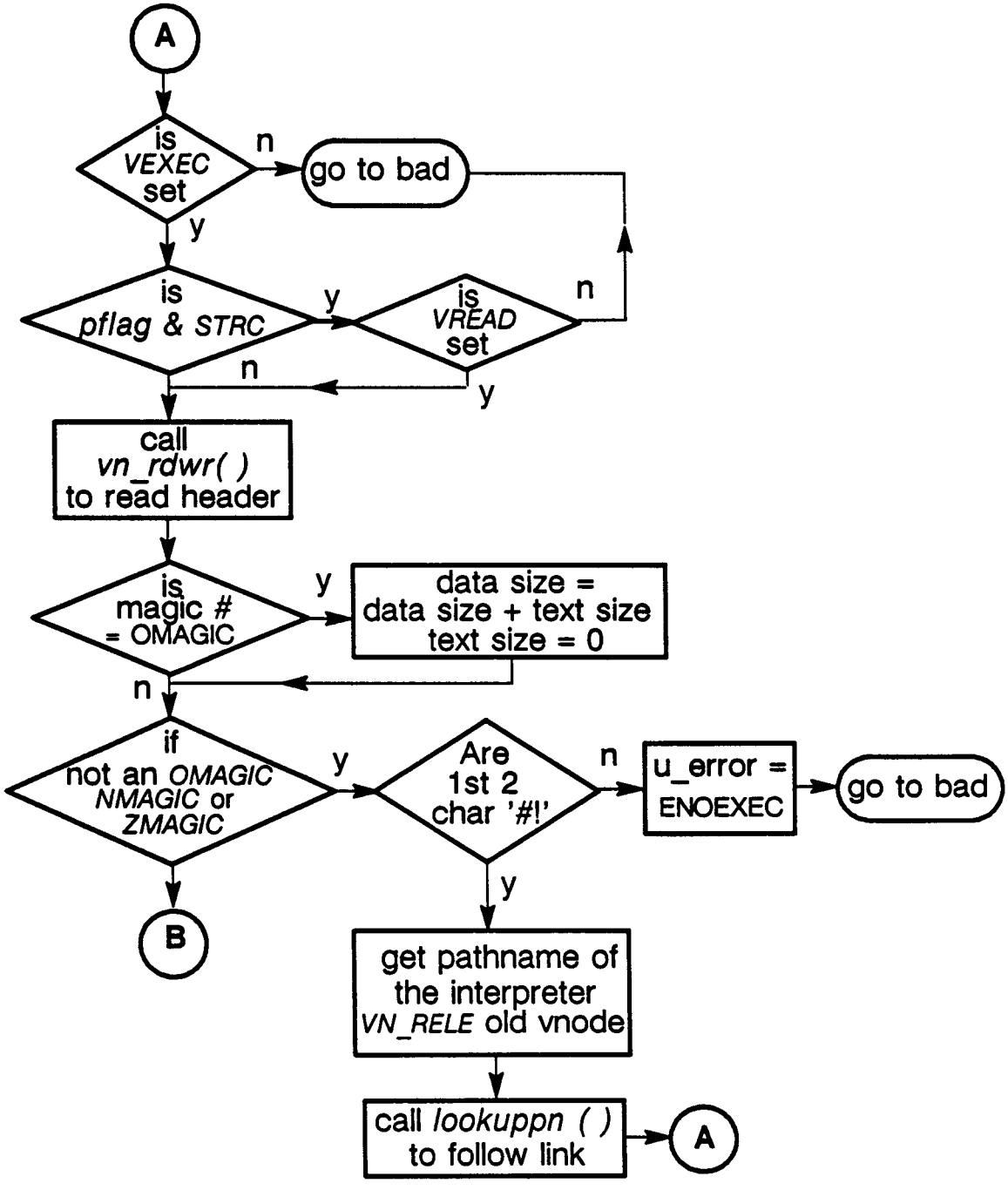
If the file system that the vnode is located in allows setting the effective user ids *(VFS_NOSUID = 0)*, then:

If the *vattr* structure has the set uid on execution mode bit set *(VSUID)*, set the effective user id to the uid in the *vattr* structure *(va_uid)*.

If the *vattr* has the group id mode flag set *(VSUID)*, set the effective group id to *va_gid*.

Check the access to the executable file by calling *VOP_ACCESS* with a pointer to the user's credential structure.

execve() (cont.)


A
is VEXEC set
n
go to bad
y
is pflag & STRC
y
is VREAD set
n
y
n
call vn_rdwr( ) to read header
is magic # = OMAGIC
y
data size = data size + text size text size = 0
n
if not an OMAGIC NMAGIC or ZMAGIC
y
Are 1st 2 char '#!'
n
u_error = ENOEXEC
go to bad
B
y
get pathname of the interpreter VN_RELE old vnode
call lookuppn ( ) to follow link
A

execve() (cont.)

If there is no execute access go to *bad.*

If the process is being traced and there is no read access go to *bad.*

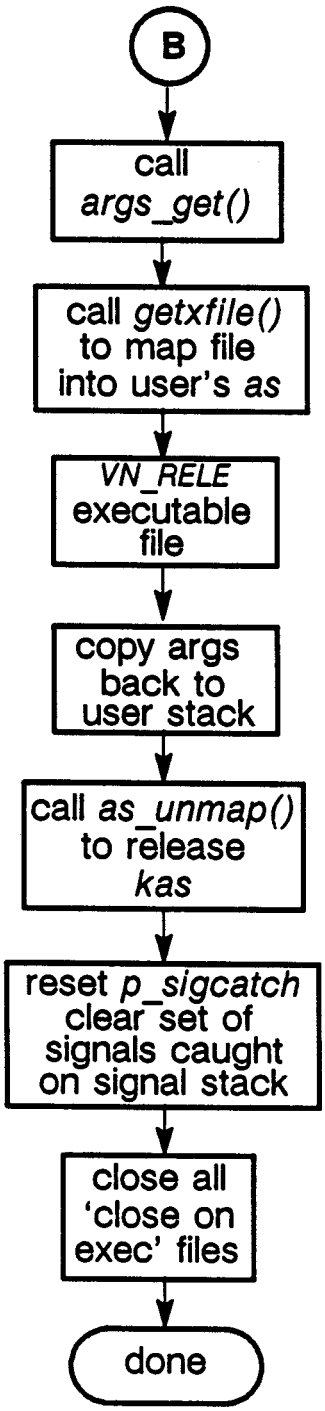
Call *vn_rdwr()* to read in the file headers.

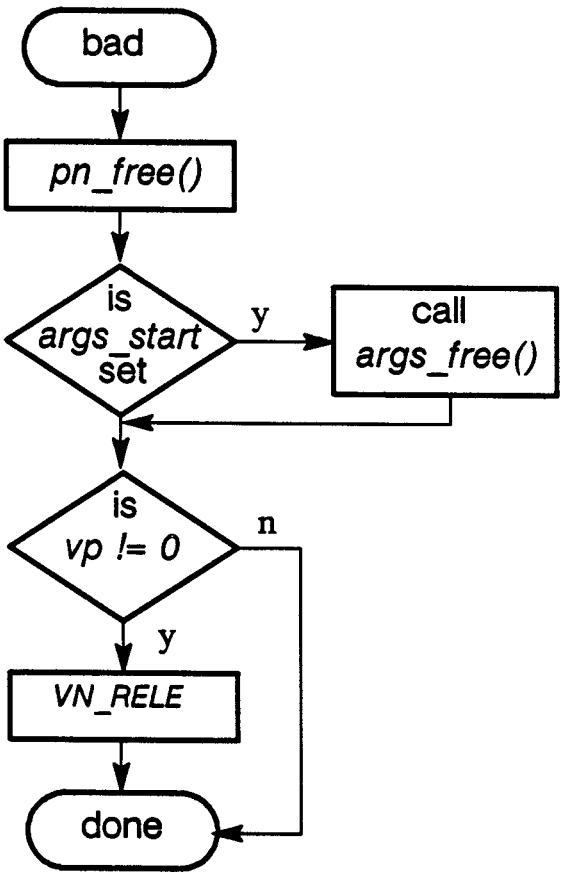
If the magic number is *0MAGIC*, it is impure code and the text has to be put in the data segment.

If there is no magic number and the first two characters are not #!, then it is not an executable *(ENOEXEC)* goto bad.

If it is a shell file get the pathname of the interpreter and call *lookuppn()* and go to *again* to get the interpreter.

execve() (cont.)


B
call
args_get()
call getxfile()
to map file
into user's as
VN_RELE
executable
file
copy args
back to
user stack
call as_unmap()
to release
kas
reset p_sigcatch
clear set of
signals caught
on signal stack
close all
'close on
exec' files
done


bad
pn_free()
is
args_start
set
y
call
args_free()
is
vp != 0
n
y
VN_RELE
done

execve()

Call *args_get()* to copy args to kernel address space *(kas)*.

Since the old process address space is not accessible after being released, the arguments to the exec call are copied to the kernel address space temporarily.

The arguments are moved to kernel address space (*kas*) in sections *(arg_hunk)* defined as 16 Kbytes.

args-get() calls *as_hole()* to find a first fit in *kas* for the arguments.

It is possible that before the space found by *as_hole()* is locked down by *as_map()* we could be put to sleep and another process could grab the same space through *as_hole()*. Therefore the *argsget_flag* is set to *ARGS_GET_IN_PROCESS* for the first process that enters; any subsequent processes will set the argsget_flag's *ARGS_GET_WAITING* bit and sleep on the address of the *argsget_flag.*

When the first process finishes mapping the *kas* any processes sleeping at *argsget_flag* are awoken.

Call *getxfile()* to read in and set up memory for the executable file.

Upon return release the vnode for the executable file.

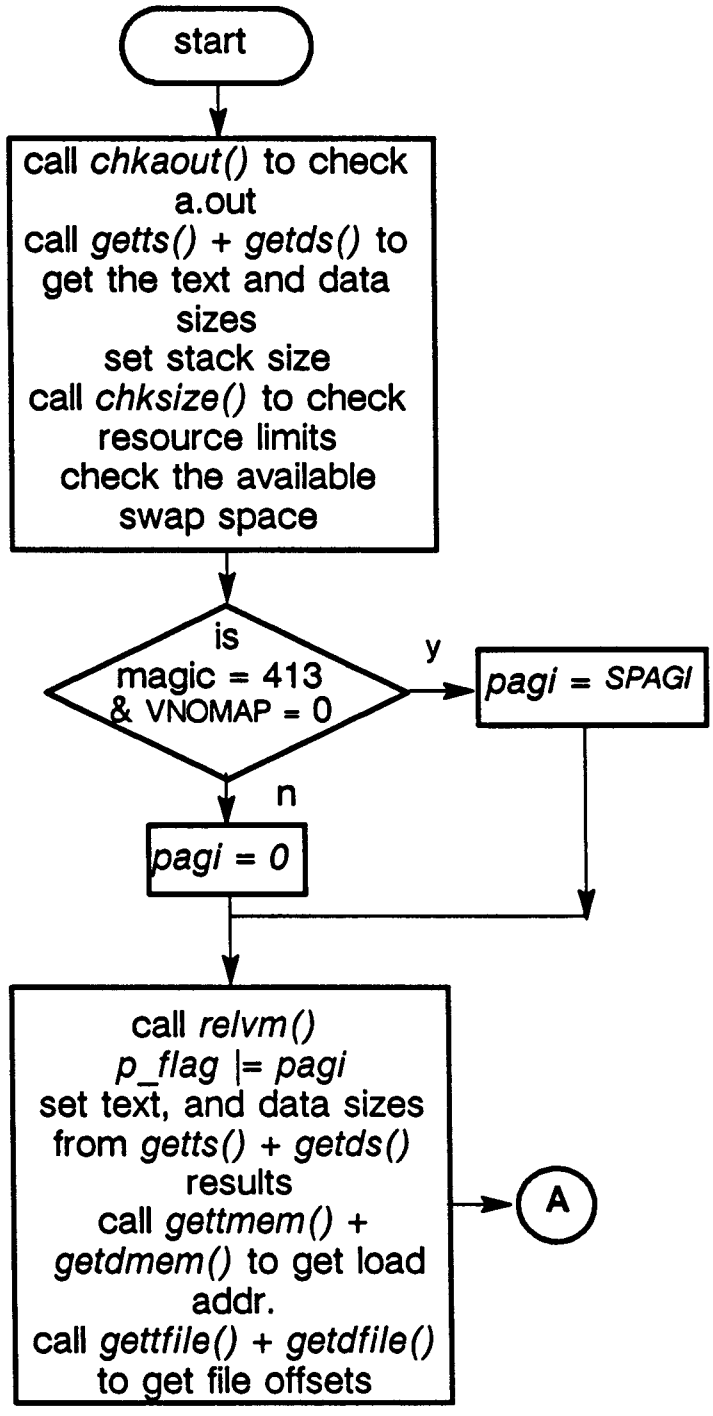
Copy the arguments to the user's stack.

Release the kernel address space.

Clear the signal catching mask and clear any signals caught on the signal stack.

Close all 'close on exec' files.

getxfile()


start
call chkaout() to check a.out
call getts() + getds() to get the text and data sizes
set stack size
call chksize() to check resource limits
check the available swap space
is magic = 413 & VNOMAP = 0
y
pagi = SPAGI
n
pagi = 0
call relvm()
p_flag |= pagi
set text, and data sizes from getts() + getds() results
call gettmem() + getdmem() to get load addr.
call gettfile() + getdfile() to get file offsets
A

getxfile()

getxfile()

Calls *chkaout()* to see if the *a.out* type is supported for the underlying architecture.

Next, it calls *getts()* and *getds()* to get the text and data sizes.

The stack size is set large enough to accommodate the arguments.

The *chksize()* routine makes sure none of the resource limits are violated by the size of the text, data, or stack segments.

The process then takes a look at the *anon_info* structure to determine if there is enough swap space to start the program. With shared libraries, there is no guarantee since the exec may succeed and then the *ld.so* may fail. Since it is not known at exec time what anonymous memory will be required for shared libraries (or even what shared libraries will be loaded at run time), there is no way to anticipate swap space requirements.

If the executable is paged on demand and the vnode can be mapped, then *pagi* = *SPAGI* (init data space on demand flag in proc.h), else *pagi* = 0.

Call *relvm()* for this process to release the old mappings for this process.

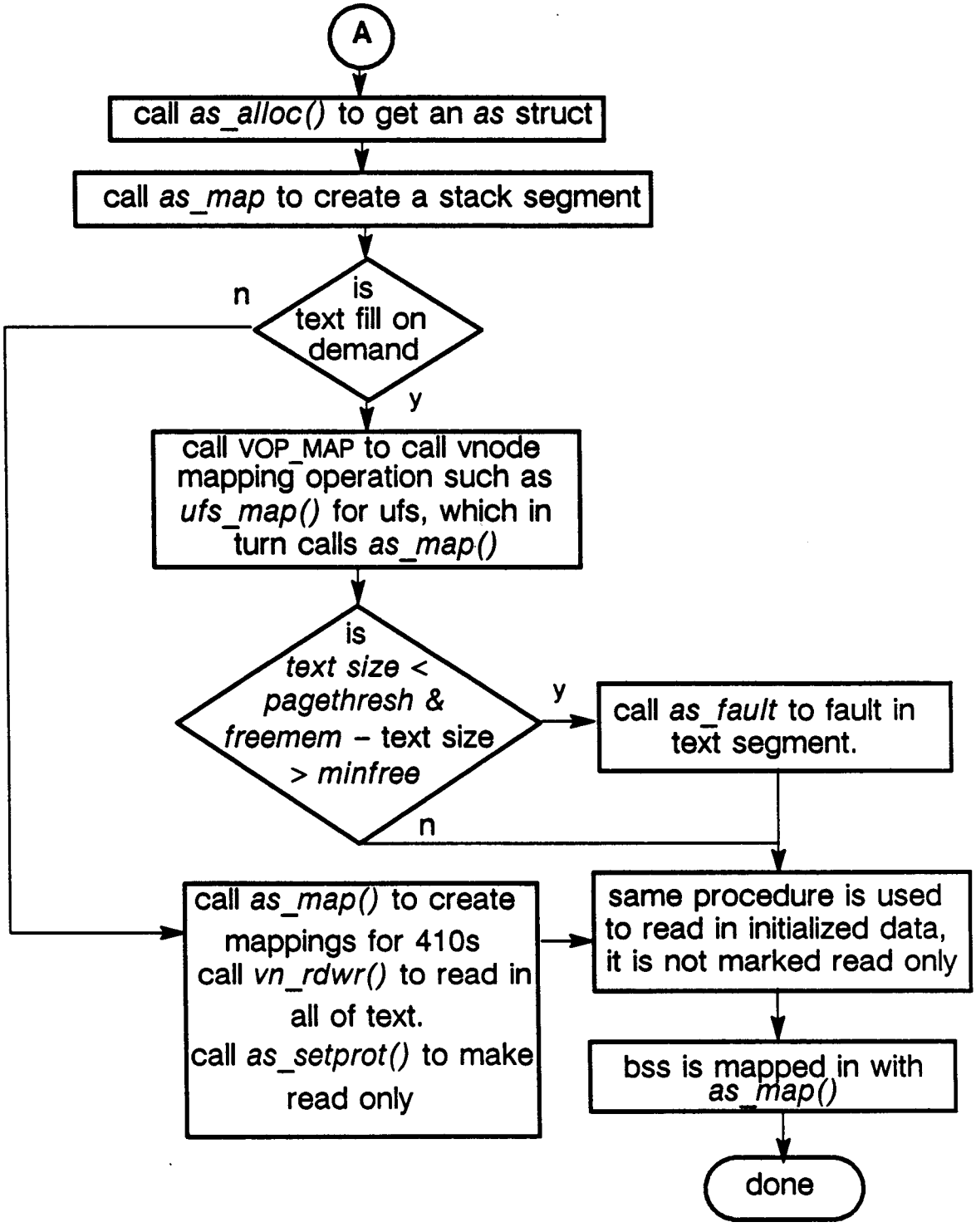
Upon return clear flags concerning paging behavior and set to *pagi*.

Set sizes for text, data and stack in *proc* structure.

Call g*ettmem()*, and *getdmem()* to get the load memory address for the text and data segments.

Call *gettfile()* and *getdfile()* to get the file byte offsets of the text and data segments.

getxfile() (cont.)



getxfile() (cont.)

Call *as_alloc()* to create a new *as* structure for the process. Then call *as_map()* to create the address space mappings for the stack segment.

If the magic number is ZMAGIC (413), the text is fill on demand:

Use the object oriented routine *VOP_MAP()* to call the specific vnode mapping operation to map in the text segment, for a ufs file this would be *ufs_map.*

If the text segment is small, then pre-fault in the entire text segment with *as_fault()*. The text is determined to be small by the *PREREAD()* routine which checks if we have room to move in the entire text segment without cutting the free memory to less than *minfree*. If the text is then less than *pagethresh* it will fault in the entire text segment all at once. The constant *pagethresh* is defined as 280 KB on Sun-3s and Sun-4s.

If the file is not fill on demand, it is NMAGIC, (410) then:

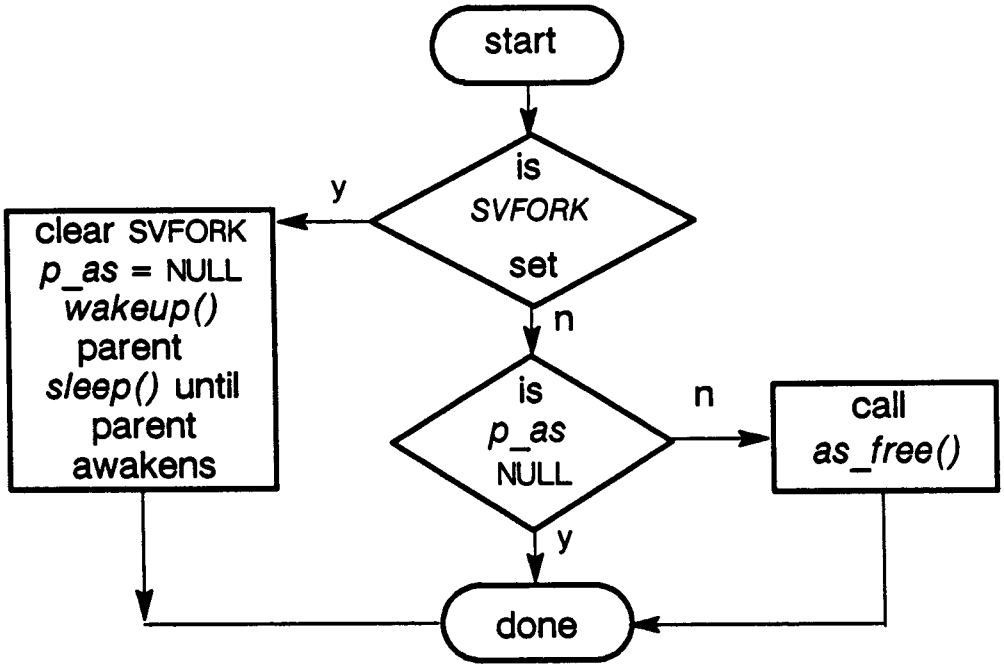
Call *vn_rdwr()* to read in entire text.

Call *as_setprot()* to make the text segment read only.

Use the same sequence, using *VOP_MAP()* and *PREREAD()* to map in the initialized data segment, only it is not marked as read only.

Since the uninitialized data can just be read in as zeros and does not have to be read from a non local file system it can be mapped with *as_map()* rather than *VOP_MAP()*.

relvm()


start
is
SVFORK
set
y
clear SVFORK
p_as = NULL
wakeup()
parent
sleep() until
parent
awakens
n
is
p_as
NULL
n
call
as_free()
y
done

relvm()

relvm():

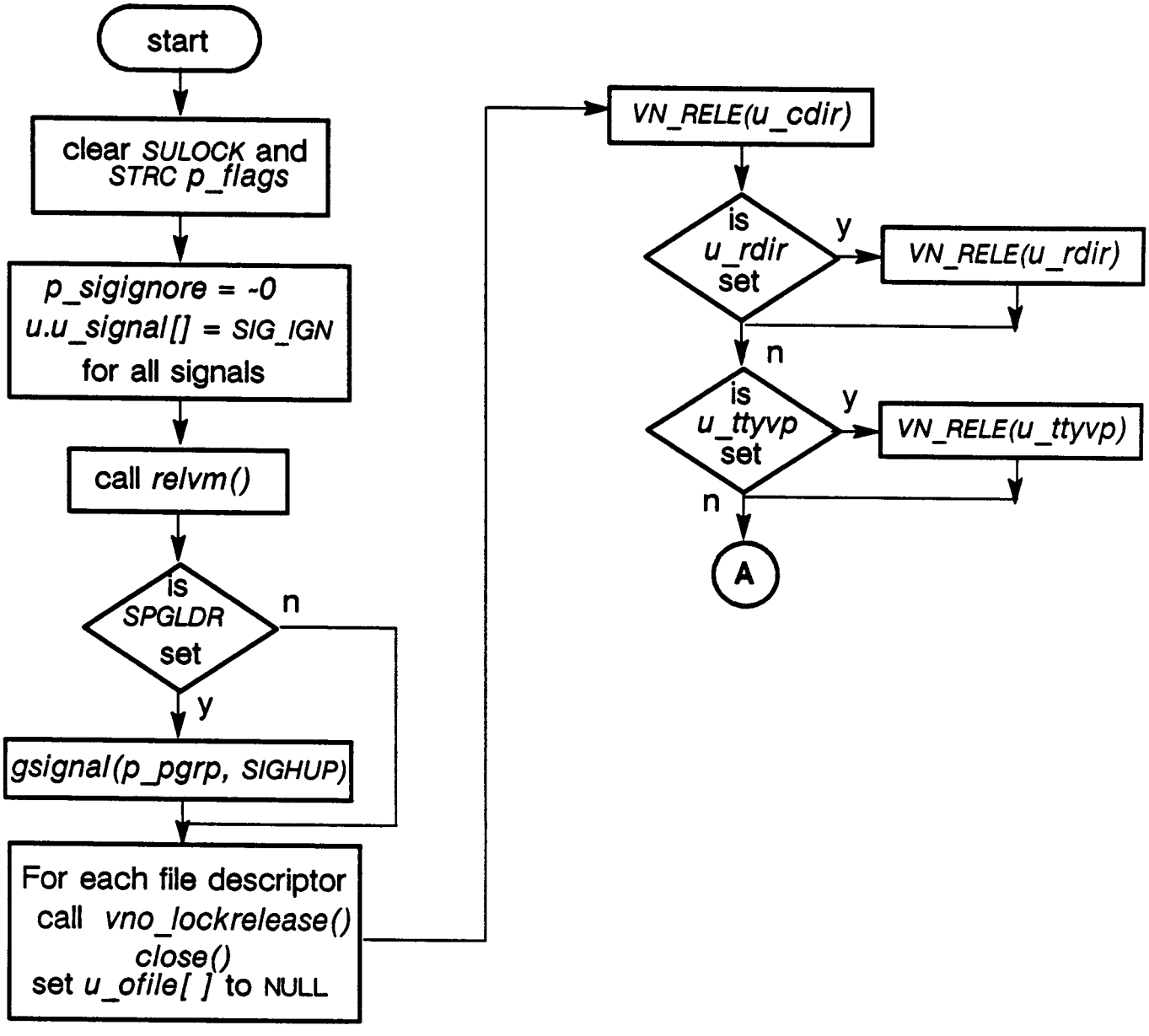
Checks if this process is a result of a vfork *(SVFORK)*; if not,

Check if *as* pointer in the proc structure is NULL;

If not call *as_free()* to release this process' virtual memory.

If it was a vfork, clear *SVFORK* flag, set *as* pointer to NULL to give the parent back its address space. Wake up the parent, and sleep until the parent takes back its *as* and sets the SVDONE flag.

exit()


start
clear SULOCK and STRC p_flags
p_sigignore = ~0
u.u_signal[] = SIG_IGN
for all signals
call relvm()
is SPGLDR set
n
y
gsignal(p_pgrp, SIGHUP)
For each file descriptor
call vno_lockrelease()
close()
set u_ofile[ ] to NULL
VN_RELE(u_cdir)
is u_rdir set
y
VN_RELE(u_rdir)
n
is u_ttyvp set
y
VN_RELE(u_ttyvp)
n
A

exit()

The *exit()* routine is used to release the resources of a process and put the process into a zombie state.

First clear the trace *(STRC)* and user lock *(SULOCK)* flags in the proc structure, and set the SWEXIT flag. Ignore all signals set *cpticks* and percentage of CPU to 0.

Call *relvm()* to release the address space of the process.

If this process is a process group leader, signal any processes in the signal group with a *SIGHUP*.

For each *fd* in the user's file descriptor table :

release any system V locks

call *close()* to close the file associated with the descriptor

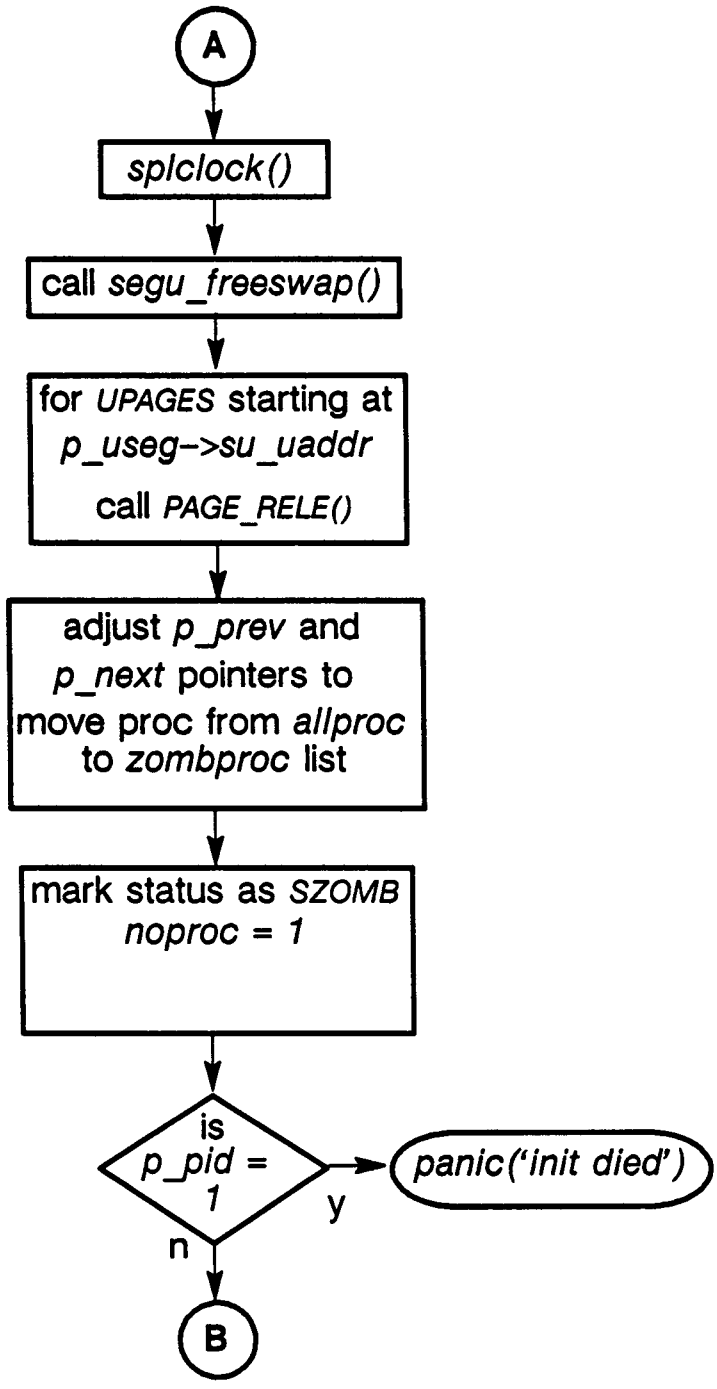
set each entry in the *u.u_ofile[]* array entry to null

Release the vnode for the user's current directory, and if the user has a root directory defined release its vnode.

If this process has a vnode for a controlling terminal release that vnode.

Remove any limit to the file size in the resource limit structure

exit()


A
splclock()
call segu_freeswap()
for UPAGES starting at p_useg->su_uaddr
call PAGE_RELE()
adjust p_prev and p_next pointers to move proc from allproc to zombproc list
mark status as SZOMB
noproc = 1
is p_pid = 1
y
panic('init died')
n
B

exit()

Raise the priority to that of the clock process so no one can allocate the memory we are trying to release from the u area, since we are still using the u area.

Call *segu_freeswap()* to free the swap space allocated for the u area.

For all pages in the u area from *p_useg->su_addr* for *UPAGES* pages:

Call *PAGE_RELE()* to put each page on the free list.

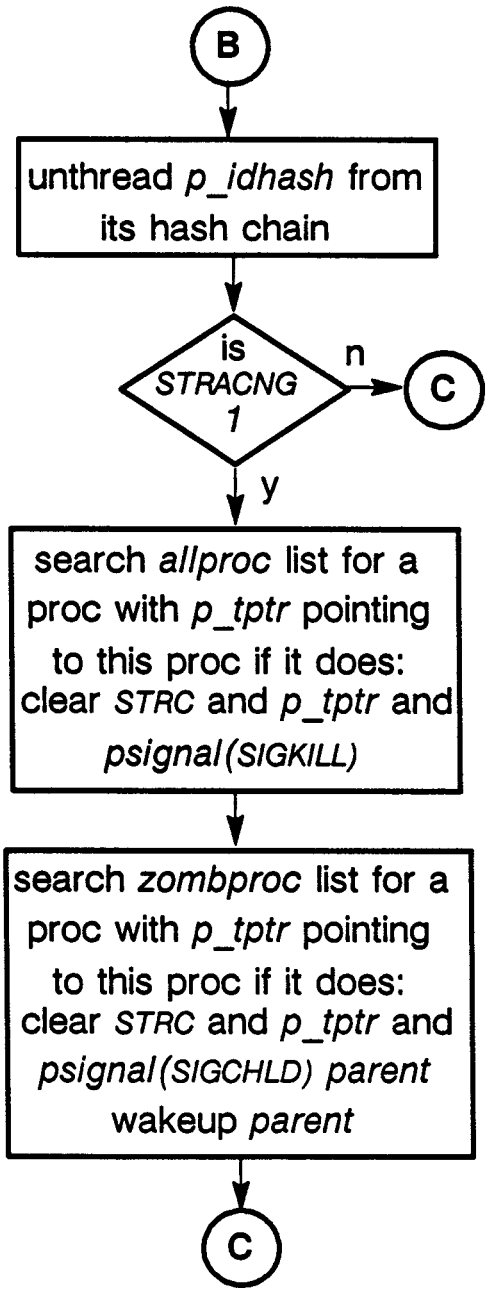
Remove the proc structure from the *allproc* list and thread it onto the *zombproc* list.

Mark the status of the process *SZOMB.*

Set *noproc* to 1 to indicate no process is currently running.

If the process exiting is init *panic('init died').*

exit() (cont.)


B
unthread p_idhash from its hash chain
is STRACNG 1
n
C
y
search allproc list for a proc with p_tptr pointing to this proc if it does: clear STRC and p_tptr and psignal(SIGKILL)
search zombproc list for a proc with p_tptr pointing to this proc if it does: clear STRC and p_tptr and psignal(SIGCHLD) parent wakeup parent
C

exit() (cont.)

Take the process off the pid hash list.

If not on the pid hash list, *panic('exit')*.

Set exit status *(p_xstat)*.

If tracing any processes:

Look though *allproc* list, for any *proc* that has *p_tpr* pointer to this *proc* struct:

set its *STRC* and *p_tpr* pointer to 0

send it a kill signal *(SIGKILL)*.

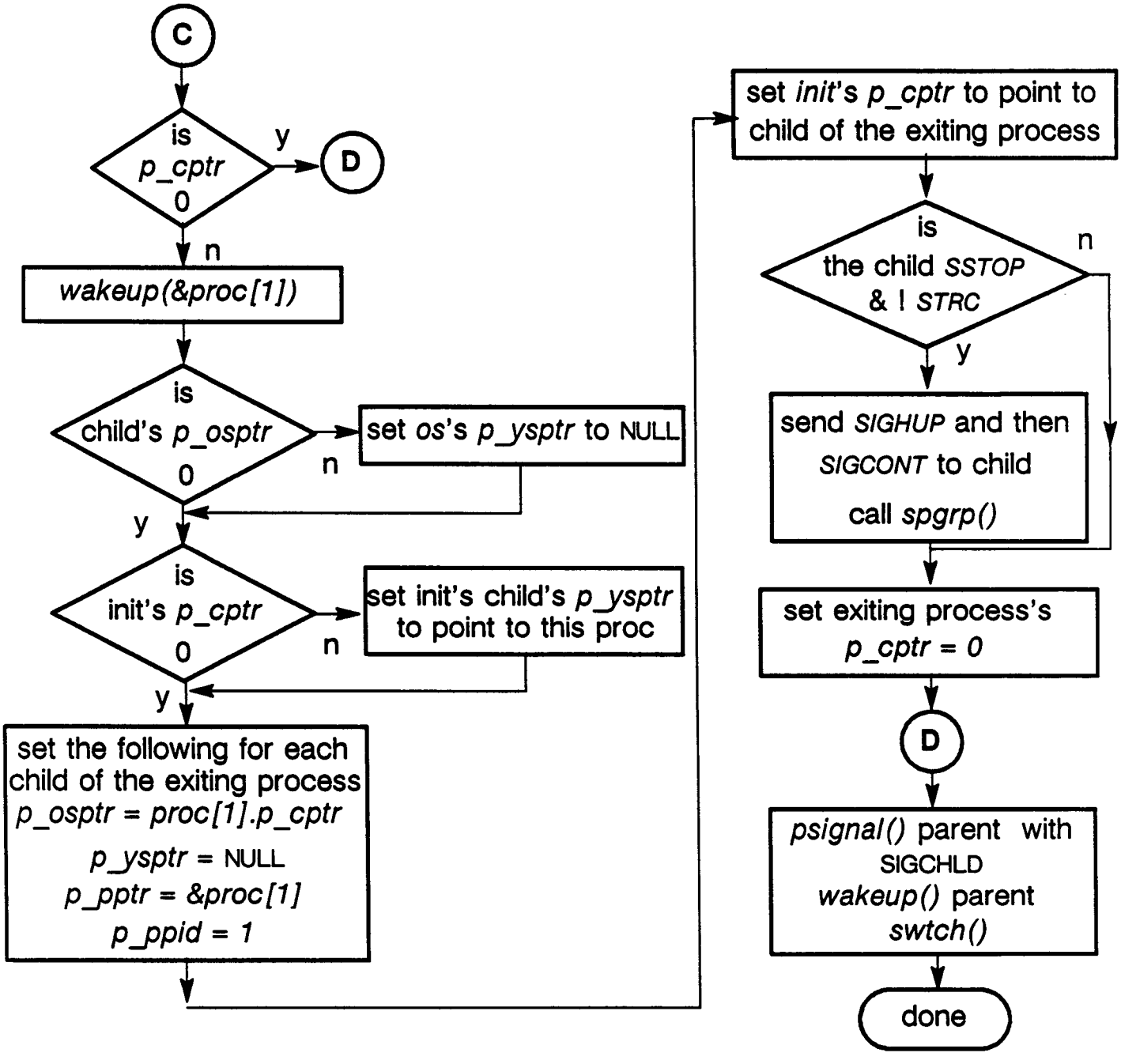
Search the zombproc list, for any *proc* that has *p_tpr* pointer to this *proc* struct:

set its *STRC* and *p_tpr* pointer to 0

signal the parent process *(SIGCHLD)*

wakeup the parent.

exit() (cont.)


C
is p_cptr 0
y
D
n
wakeup(&proc[1])
is child's p_osptr 0
n
set os's p_ysptr to NULL
y
is init's p_cptr 0
n
set init's child's p_ysptr to point to this proc
y
set the following for each child of the exiting process
p_osptr = proc[1].p_cptr
p_ysptr = NULL
p_pptr = &proc[1]
p_ppid = 1
set init's p_cptr to point to child of the exiting process
is the child SSTOP & ! STRC
n
y
send SIGHUP and then SIGCONT to child
call spgrp()
set exiting process's p_cptr = 0
D
psignal() parent with SIGCHLD
wakeup() parent
swtch()
done

exit() (cont.)

If there are any child processes of the exiting process:

Wakeup init (for zombies).

for each child :

If that child has an older sibling set the older sibling's *p_ysptr* to NULL.

Use the *p_osptr* pointer to get the next child proc to release.

If init has a child: (this is where init adopts abandoned children)

set that child's p_ysptr to point to this proc

set this process's *p_osptr* to init's *p_cptr*

set this process's *p_ysptr* to NULL, the *p_pptr* to point to init, and the *p_pid* to 1

set init's *p_cptr* to point to this proc.

If the child is stopped and not being traced:

send the child a *SIGHUP* signal

send a *SIGCONT* signal

call *spgrp()* to clear any pending stop signals.

Set exiting process's *p_cptr* to NULL

Signal parent with *SIGCHLD*.

Wakeup parent.

Call *swtch()* to let someone else run.

wait()

The final stage of a process, performed by the *wait()* routine will:

- search for a terminated process on the *zombproc* list
- collect the process status
- look for stopped traced children and collect their status
- look for other processes that we are tracing

wait()

The parent of an exiting process will typically call *wait4()* to search for a terminated child, remove it from the *zombproc* list, place the proc structure on the free list after leaving it in an initialized state.

The following fields are set to zero:

p_xstat

p_ru

p_stat

p_pid

p_ppid

p_pptr

p_tptr

p_ysptr

p_osptr

p_cptr

p_sig

p_sigcatch

p_sigignore

p_sigmask

p_pgrp

p_flag

p_wchan

p_cursig

Signals

SIGHUP 1 – hangup

SIGINT 2 – interrupt

SIGQUIT 3 – quit

SIGILL 4 – illegal instruction

SIGTRAP 5 – trace trap

SIGIOT 6 – IOT instruction will be called SIGABRT in future

SIGEMT 7 – EMT instruction emulator trap

SIGFPE 8 – floating point exception

SIGKILL 9 – kill (cannot be caught or ignored)

SIGBUS 10 – bus error

SIGSEGV 11 – segmentation violation

SIGSYS 12 – bad argument to system call

SIGPIPE 13 – write on a pipe with no one to read it

SIGALRM 14 – alarm clock

SIGTERM 15 – software termination signal from kill

Signals

Signals are the mechanism the system uses to inform a process of any unscheduled events. Signals may be hardware or software induced.

A signal may be handled by a specific signal handler or take some default action such as create a core dump and terminate the process or ignore the signal.

The signals SIGSTOP and SIGKILL may not be caught or ignored as they exist to terminate a runaway process.

Signals (cont.)

SIGURG 16 – urgent condition on IO channel

SIGSTOP 17 – stop signal not from tty (nonmaskable)

SIGTSTP 18 – stop signal from tty

SIGCONT 19 – continue a stopped process

SIGCHLD 20 – any change in child process state

SIGTTIN 21 – to readers pgrp upon background tty read

SIGTTOU 22 – like TTIN for output

SIGIO 23 – input/output is now possible signal

SIGXCPU 24 – exceeded CPU time limit

SIGXFSZ 25 – exceeded file size limit

SIGVTALRM 26 – virtual time alarm

SIGPROF 27 – profiling time alarm

SIGWINCH 28 – window changed

SIGLOST 29 – resource lost (eg, record–lock lost)

SIGUSR1 30 – user defined signal 1

SIGUSR2 31 – user defined signal 2

Signals (cont.)

Signals may only be posted to processes that have the same effective user id except for *SIGCONT* which may be posted to any descendant to the process.

SIGPOLL is the System V name for SIGIO.

Interrupt level processes or super user may post signals to any user process.

If a system call is interrupted by a signal it is automatically restarted whenever possible (unless the user requests the old behavior via the *SV_INTERRUPT* bit).

Signals are all the same priority. They are not stacked.

Signals (cont.)

In a user's *proc* structure there are three masks that determine what actions will be taken when a signal is delivered.

- *p_sigignore* – signals to be ignored
- *p_sigmask* – signals currently being blocked.
- *p_sigcatch* – signals for which a specified signal handler exists.

The following two fields track signals that are currently being delivered or are still pending delivery.

- *p_cursig* – signal currently being delivered.
- *p_sig* – bit mask of signals pending delivery.

Signals (cont.)

Signals being blocked are kept from being delivered by masking the delivered signal with *p_sigmask* in the *proc* structure.

Signals can be postponed until they are unmasked, by calling *sigblock(2)* or *sigsetmask(2)* to mask a specific set of signals.

In the process structure the *p_cursig* field identifies the signal currently being delivered. The *p_sig* field is the set of signals pending delivery.

When a process returns from a system call, a trap or is awoken, the system checks for pending signals *(p_sig),* to see if a signal is pending.

When signaled, the signal mask is updated by ORing the current signal with *p_sigmask.*

Process group identifiers are used to pick a collection of related processes for the purpose of signal delivery (ex. killp for a pipeline). The *p_pgrp* (process group identification number) is used to post signals to all processes that are members of a process group. This is used for job control such as terminal handling and the IPC mechanism.

Signal Routines

- *kill(2)* and *killpg(2)*
- *psignal()*
- *gsignal()*
- *sigblock(2)*
- *sigsetmask(2)*
- *sigvec(2)*
- *sigpause(2)*
- *sigstack(2)*
- *sigcleanup(2)*

Signal Routines

The *kill(2)* and *killpg(2)* routines, do some error checking and call *psignal(2)*.

The *psignal(2)* call is used to post a signal to a single process (see flowchart).

The *gsignal(2)* call is used to post a signal to an entire process group by searching the *allproc* list and calling *psignal(2)* for all processes with the appropriate *p_pgrp* number.

The *sigblock(2)* routine adds to the *p_sigmask* (the bits are ored).

The *sigsetmask(2)* routine replaces the current *p_sigmask*.

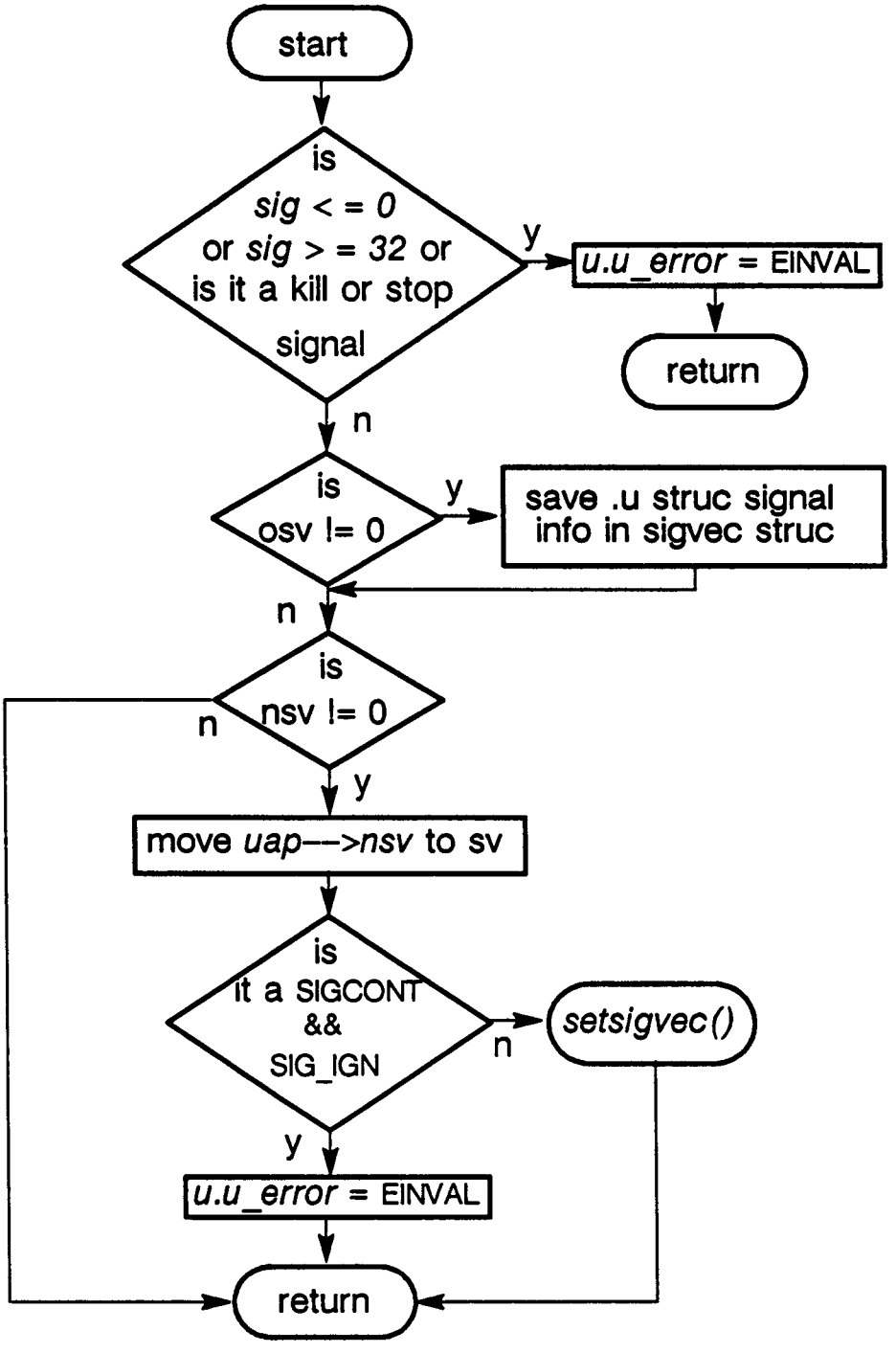
The *sigvec(2)* call points to an action to be taken: to respond to a signal such as take the default action, or 'catch' it with a specified handler or ignore it.

The *sigpause(2)* system call lets a user give up the CPU until it receives a signal.

The *sigstack(2)* system call allows specification of an alternative stack for signals such as stack overflow *(SIGSEGV)* handlers to execute on.

The *sigcleanup(2) (syscall(139))* call is used to pass a pointer to the kernel of a context block which describes the user state so it can be restored when returning from a signal handler.

sigvec()


start
is
sig < = 0
or sig > = 32 or
is it a kill or stop
signal
y
u.u_error = EINVAL
return
n
is
osv != 0
y
save .u struc signal
info in sigvec struc
n
is
nsv != 0
n
y
move uap—>nsv to sv
is
it a SIGCONT
&&
SIG_IGN
n
setsigvec()
y
u.u_error = EINVAL
return

sigvec()

The *sigvec(2)* routine assigns a handler to a particular signal.

First the signal range is checked. If it is out of range or its a kill or stop signal which we cannot catch return EINVAL.

If there was an old signal handler specified in the *sigvec()* call, save the information from the u structure about the handler so we can reinstall it if necessary.

```
struct  sigvec {
        void    (*sv_handler)();        - signal handler
        int     sv_mask;                - signal mask to apply
        int     sv_flags;               - see signal options below
}
```

Signal Options:

SV_ONSTACK 0x0001 – take signal on signal stack

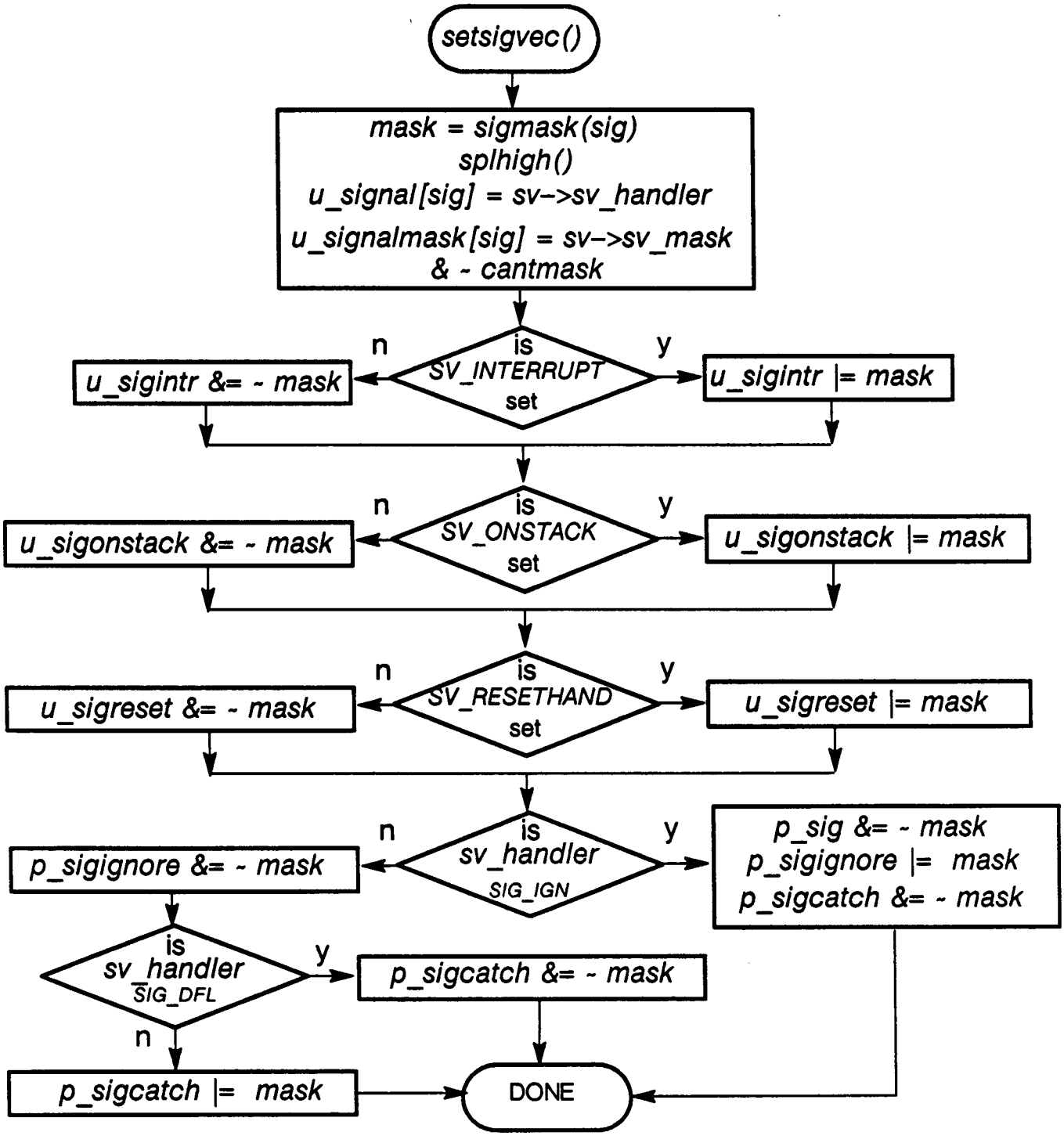
SV_INTERRUPT 0x0002 – do not restart system call on signal return

SV_RESETHAND 0x0004 – reset signal handler to SIG_DFL when signal taken

If a pointer to a new signal handler is specified move that to *sv* to make it accessible to the kernel.

If the signal handler is trying to ignore a continue signal, return EINVAL.

sigvec() (cont.)


setsigvec()
mask = sigmask(sig)
splhigh()
u_signal[sig] = sv->sv_handler
u_signalmask[sig] = sv->sv_mask
& ~ cantmask
is SV_INTERRUPT set
n
y
u_sigintr &= ~ mask
u_sigintr |= mask
is SV_ONSTACK set
n
y
u_sigonstack &= ~ mask
u_sigonstack |= mask
is SV_RESETHAND set
n
y
u_sigreset &= ~ mask
u_sigreset |= mask
is sv_handler SIG_IGN
n
y
p_sigignore &= ~ mask
p_sig &= ~ mask
p_sigignore |= mask
p_sigcatch &= ~ mask
is sv_handler SIG_DFL
y
p_sigcatch &= ~ mask
n
p_sigcatch |= mask
DONE

sigvec() (cont.)

The signal handler is permanently installed with the *sigvec(2)* system call, unless the SV_RESETHAND flag is set.

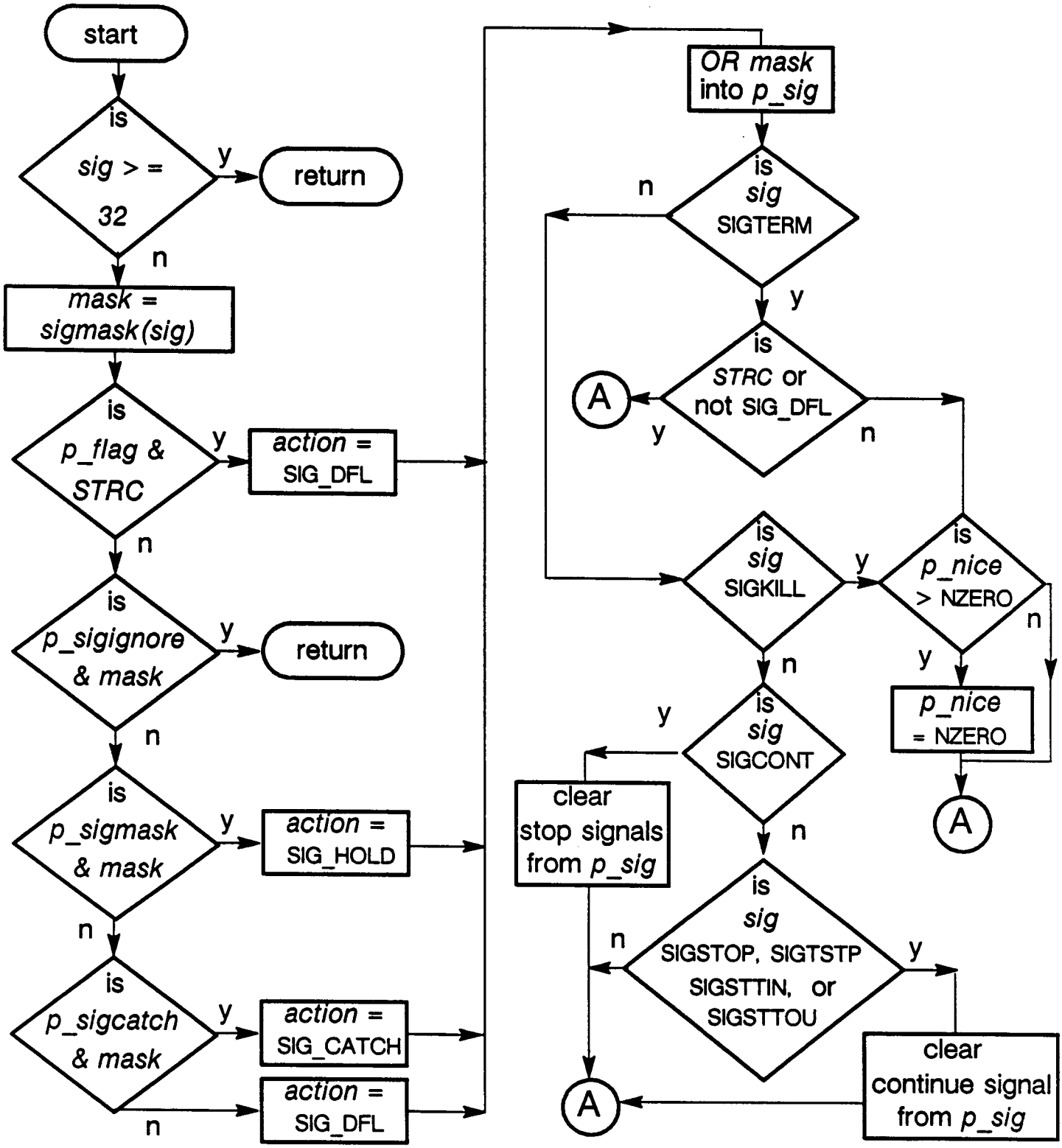
If the SV_RESETHAND flag is set in the *sv_flags,* the *sv_handler* for that signal will be set to the default in the flags(*SIG_DFL)* before invoking the handler. Furthermore the signal will not automatically be blocked.

Setting the *SV_INTERRUPT* bit in the *sv_flags* means that system calls will not be restarted on receipt of this signal.

The appropriate information is moved into the user structure from the sigvec structure. If the signal handler is SIG_IGN, remove that signal from the *p_sig* mask, if it is pending delivery to the process.

If the signal handler isn't ignoring the signal or taking the default action it must be catching the signal so set the *p_sigcatch* mask.

psignal(p,sig)



psignal(p,sig)

The psignal() routine is passed a signal number, *sig*, and a pointer to the proc structure, *p,* of the process the signal is being delivered to.

Check the signal number *sig* to make sure it's within range (1 - 31).

Set up a mask with the appropriate bit set in a 32 bit integer that corresponds to the signal number. For example if the signal number is 5 the mask would be 00000000000000000000000000010000.

If the process is being traced the action to be taken will be the default action. In other words we don't want the child to catch the signal we want the parent to able to intercept it first.

If the signal is being ignored we can just return.

If the signal is being masked the action will be held off till later.

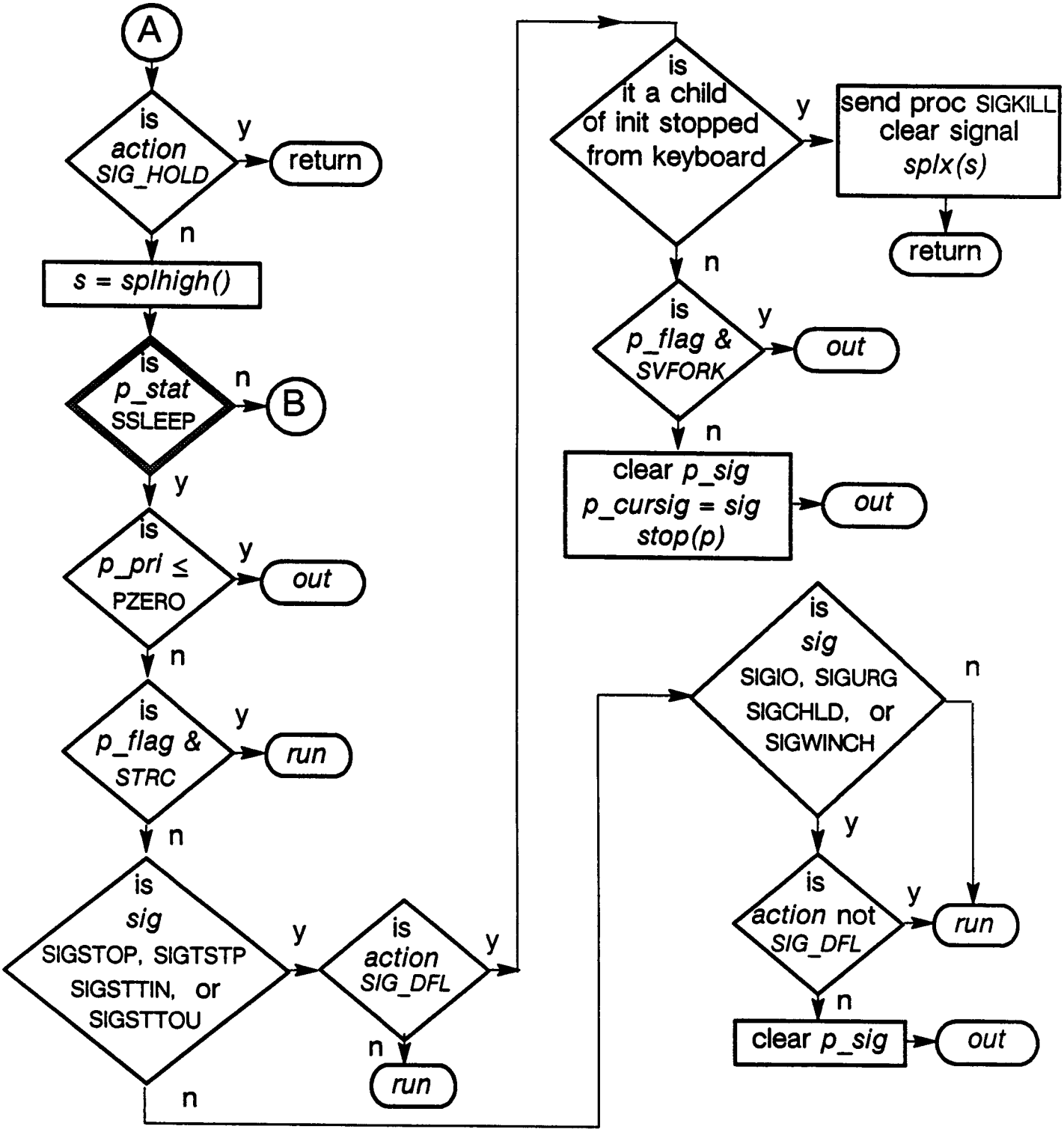
If the signal is being caught mark the action to be SIG_CATCH, otherwise mark it to SIG_DFL to indicate we must be taking the default action.

Set the bit corresponding to this signal in the pending signal mask (psig).

If it's a software termination signal from the terminal, SIGTERM, check if the signal is being traced or caught. If it is fall through to A, if not or if the signal is SIGKILL make sure that the p_nice value is set to the default so it doesn't take too long to kill the process.

If it's a continue signal clear any stop signals from p_sig, if it's a stop signal clear the continue signal from p_sig, in either case go to A.

psignal() (cont.)



psignal() (cont.)

(A)

Check if the action was to hold the signal because it was masked, if so, it has already been marked pending for the process, so return.

Raise the priority.

If the process is not sleeping go to B.

If the process is blocked (sleeping).

Check if it is sleeping at an interruptable priority, if not go to *out* to lower the priority, then return. If the sleeping process is being traced go to *run* to set the process running.

If the signal is a stop signal and the process is not taking the default action go to *run*.

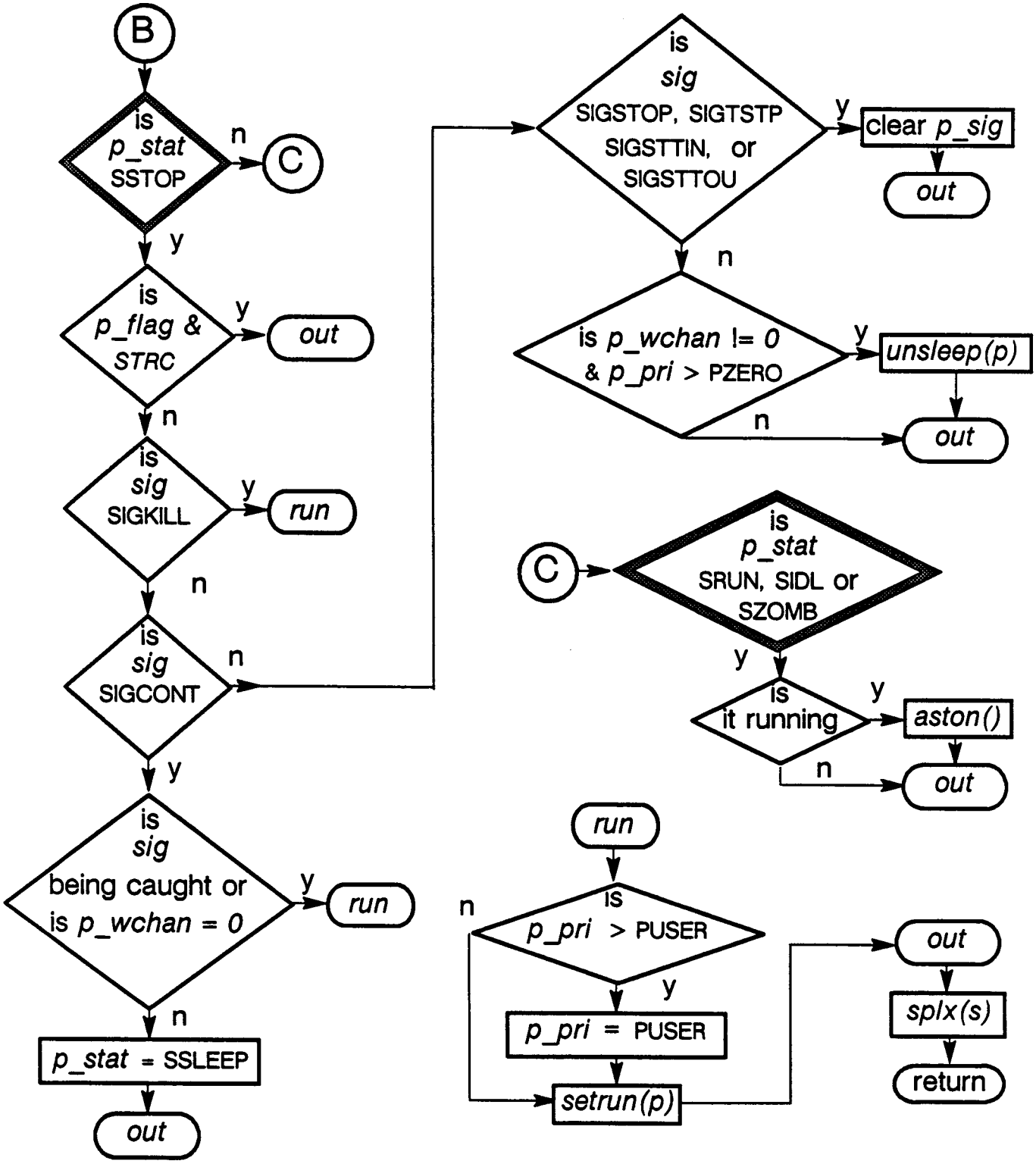
If it is a child of init stopped from the keyboard send the process a kill signal to terminate the process and return.

If the process was a result of a vfork go to *out*.

Otherwise clear the signal from *p_sig* and set *p_cursig* to indicate that *sig* is currently being delivered, call *stop(p)* to put the process in a stopped state, and go to *out*.

If it's not a stop signal, check if it's a signal that 's not delivered unless being caught. If it is being caught go to *run*. Otherwise just clear the pending signal and go to *out.*

psignal(p,sig)



psignal() (cont.)

Ⓑ

If the process is not in a stopped state go to C.

If the process is stopped:

Check if it's being traced. If it is, go to out (It's stopped by the parent).

If the process is being killed, go to *run* (so it can process the kill signal).

If *sig* is continue:

Check if the signal is being caught or the process is not blocked *(p_wchan=0)*, if this is the case, go to *run.*

If the process is not blocked set the status to *SSLEEP*, and go to *out.*

If *sig* is a stop signal clear the pending signal and go to *out.*

Otherwise check if the process is sleeping at interruptable priority and if it is, wake it up. Then go to *out* in either case.

Ⓒ

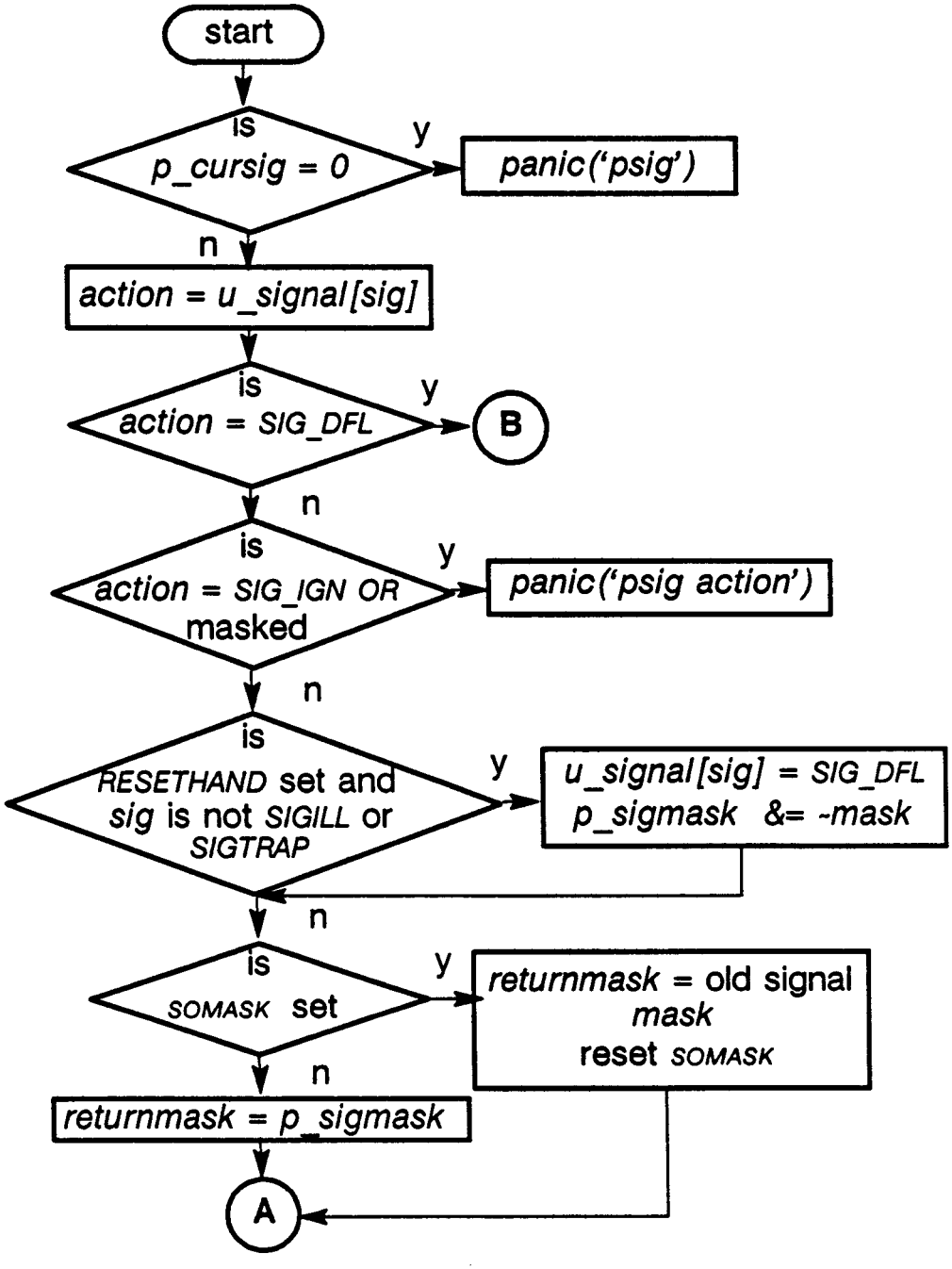
If the process is in a runnable, idle or zombie state: (this is the default to a switch statement that's why there is no branch for no)

Check if it's running. If it is, schedule an asynchronous trap so the process will notice the signal, then go to out. Otherwise just go to out.

run just makes sure that its priority is no greater than PUSER so the signal will be handled in a timely manner.

out returns the priority level to what it was and then returns.

psig()


start
is
p_cursig = 0
y
panic('psig')
n
action = u_signal[sig]
is
action = SIG_DFL
y
B
n
is
action = SIG_IGN OR
masked
y
panic('psig action')
n
is
RESETHAND set and
sig is not SIGILL or
SIGTRAP
y
u_signal[sig] = SIG_DFL
p_sigmask &= ~mask
n
is
SOMASK set
y
returnmask = old signal
mask
reset SOMASK
n
returnmask = p_sigmask
A

psig()

When a process wakes up or returns from a system call (kernel mode) it checks for a signal, with the routine *issig()*. The *issig()* routine clears the signal bit in the *p_sig* mask and stores the signal being delivered in *p_cursig*. The sequence to call psig() is: if (*issig()*) then *psig()*.

If the signal number *sig*, left in p_cursig, equals zero *panic('psig')*.

In the user structure look up the action to be taken by this process via the *u_signal[]* array.

If the action is not the default (*SIG_DFL*):

check if the action is SIG_IGN or is masked

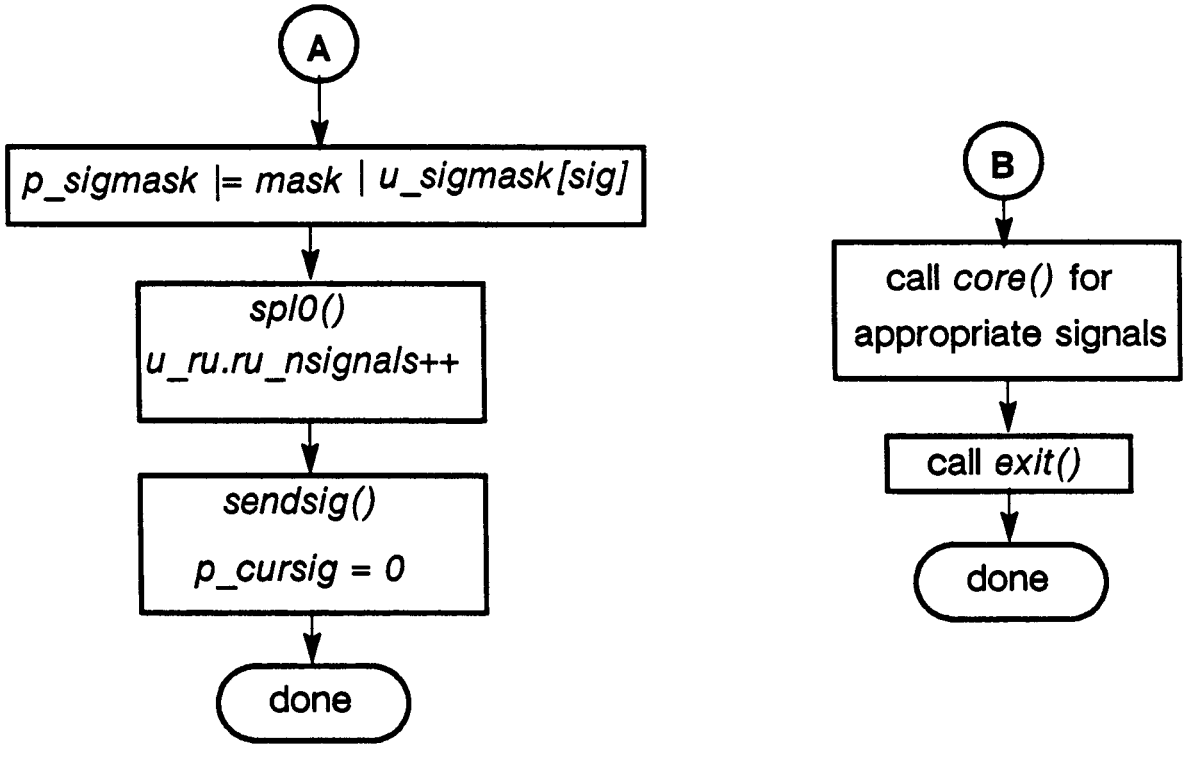
if it is, *panic('psig action')*, because this should have been caught in *psignal()* (we shouldn't be here).

If this signal should reset the signal handler when the signal is taken, and the signal is not *SIGILL* or *SIGTRAP* reset to *SIG_DFL* for the next time the signal is taken and take the signal out of *p_sigcatch* mask.

If we need to restore the old signal mask after taking a signal (*sigpause()*), then set returnmask to the old signal mask and reset the *SOMASK* in *p_flag*.

If this is not the case set the returnmask to the current value of *p_sigmask*.

psig()


A
p_sigmask |= mask | u_sigmask[sig]
spl0()
u_ru.ru_nsignals++
sendsig()
p_cursig = 0
done
B
call core() for
appropriate signals
call exit()
done

psig()

The *p_sigcatch* mask is set to mask out the current signal and any signal in the *u_sigmask[sig]* that was set by the *sigvec()* routine.

Lower the interrupt priority and increment the count of signals taken in the resource usage structure.

Call *sendsig()* to cause the process to execute *sigtramp()* upon resuming execution. The *sigtramp()* routine will call the real handler. The *sigtramp()* calls the *sigcleanup()* system call (#139) to return back to the original user context.

If taking the default action the following signals will generate a core dump by calling *core():*

SIGILL

SIGIOT

SIGBUS

SIGQUIT

SIGTRAP

SIGEMT

SIGFPE

SIGSEGV

SIGSYS

then call *exit()*

Module 4
Shared Libraries in SunOS

Objectives: Upon completion of this module, the student will be able to:

- Describe the implementation of shared libraries in the SunOS.
- Explain the differences between the new and the old C Compiler.
- Explain the changes made to the link editor to support shared libraries.
- Issue the command line to compile programs that will include specific shared libraries.
- Describe the various structures needed in the implementation of shared libraries.

Introduction

- Goal – Efficiency in Resource Utilization
  - Shared Text
  - Shared Libraries.

Introduction

One of the consistent goals of designers and implementers of operating systems is to make efficient use of computer system resources such as main memory and disk space. To achieve this goal, sharing of text has been implemented in the earliest versions of the UNIX operating system. More recently this sharing is being done through the level of the functions or libraries that are used by different programs. In this module we shall describe the SunOS's implementation of shared libraries.

Shared Library Implementation Goals

- Minimal Kernel Support
- Shared Libraries must not be a requirement
- Minimal Impact on Users
- Provide Functional Benefits.
- Performance

Shared Library Implementation Goals

- **Minimal Kernel Support** – Any modification of the kernel to support shared libraries will further complicate an already complex issue. It is ideal if the facility can be supported without any changes in the kernel.

- **Shared Libraries must not be a requirement** – Although the default system behavior will make use of shared libraries, their use however is not a requirement in the compilation of programs nor are they required in other system components.

- **Minimal Impact on Users** – The use of shared libraries must be transparent to application source code, program build procedures, and the use of standard utilities such as debuggers. It was also desirable to be able to use existing object files.

There should be no requirement to administer and coordinate the allocation of address space. Libraries should be able to evolve and be updated without requiring the rebuilding of the programs that used them as long as their interfaces are compatible, and mechanisms would have to be available to handle interface changes.

- **Provide Functional Benefits** – A great deal of flexibility was included in testing updates to libraries.

- **Performance** – Shared libraries represents a classic time vs. space trade–off opportunity. The work of incorporating library code into an address space is deferred to a later time in order to save both secondary and primary storage space. Thus, it is expected that programs using shared libraries pay a time penalty. However, the expectation was that if sharing of library code really occurred, then the I/O time required to bring in a program and get it executing would be greatly reduced. As long as the CPU time required to merge the program and its libraries did not exceed the I/O time saved, the apparent performance would be the same or potentially even better. This approach fails if sharing does not occur, or if the system is CPU saturated already.

SunOS Approach

- High Degree of Memory Sharing with Fine Level of Granularity
- New System Link Editor that Supports Dynamic Loading and Binding
- Changes to Compiler for Generating Position–Independent Code(PIC) that may be Directly Shared
- Built on Top of the Virtual Memory(VM) System
  - file mapping
  - sharing at page level
  - per page COW facility

SunOS Approach

- High Degree of Memory Sharing with Fine Level of Granularity(pages)
- New System Link Editor that Supports Dynamic Loading and Binding
- Changes to Compiler for Generating Position–Independent Code(PIC) that may be Directly Shared
- Built on Top of the Virtual Memory(VM) System
  - file mapping through the *mmap( )* system call
  - sharing at page level
  - per page COW facility allows run–time modification of shared object

Because of the above features which are internal to the SunOS VM system, the act of exec'ing a program is reduced to the establishment of copy–on–write mappings to the file containing the program. A "shared library" is added to the address space the same way, using the general file mapping mechanism.

New Link Editor

- Old *ld* Builds Complete Programs Statically
- New *ld* Builds "Incomplete" Programs with Linking of Certain Object Files Done at a Much Later Time
  - Performs same function at a different time
  - Minimizing dynamic link editing promotes sharing

New Link Editor

- The old link editor *ld* built programs statically with every executable file being a complete program that contained copies of the necessary library routines. These programs were created by link editing the programs in a single batch operation using *ld*. *ld* would refuse to build an incomplete executable file.

- The new *ld* builds "incomplete" programs, deferring the incorporation of certain object files at a much later time which is usually at execution. These deferred link editing operations use the VM system's facilities to map and thus share these objects directly. A "shared library" is simply the code and data constituting a library built as a shared object **(.so)** file. A **.so** is one of these "incomplete" a.out files that lack an entry point and could be any object such as a program or a library. Essentially, dynamic link editing is still the same operation as static link editing except occurring at a later time.

Position Independent Code

- C Compiler Generates Position Independent Code(PIC)
- PIC Needs Minimal Link Editing and thus, Promote Sharing

Position Independent Code

- Code that minimizes dynamic link editing promotes sharing and therefore is "best". To increase the probability of having the best code, the C Compiler was changed to generate position-independent-code **(PIC)**.
- **PIC** needs minimal link editing and thus promotes sharing.

Compiler Changes

- New Option (–pic) for Generating Position Independent Code(PIC)
- Uses Global Offset Table (GOT) for Accessing Global Objects
- Each Reference to a Global Data Object Dereferences a Pointer in the GOT
- Each Function Call is Generated as as Assembler Pseudo–operation Including a "Free Register"
- GOT's are Usually no Larger than 64k

Compiler Changes

- The C compiler has been changed to take a new option (**-pic**) for generating Position Independent Code(**PIC**).
- Each function prologue is extended to include the initialization of a register that is used as the base address of a linkage table to global objects, this is the Global Offset Table (**GOT**).
- Each reference to a global data object is generated as a dereference of a pointer in the **GOT**.
- Each function call is generated as an assembler pseudo–operation including a "free register".
- The linkage tables are usually no larger than 64k bytes, but in the event that a larger size is needed the compiler can be forced to generate more clumsy code sequences permitting linkage tables to a full 32 bits in size by expressing **-pic** as **-PIC**.

Assembler Changes

- Using the Compiler (–pic) Option Requires Support from the Assembler

- Assembler Generates some New Relocation Information for Certain Constructs

Assembler Changes

- The changes in the compiler to generate **PIC** code needs support from the assembler. The support required is that the assembler generate some new relocation information for certain constructs, and a change in the interpretation for some syntactic forms. This support is enabled by the assembler flag **-k**, and is generated automatically by the C compiler driver when invoking the assembler for a compilation that contained the **-pic** or **-PIC** options.

- When assembling a module with the **-k** flag enabled, the assembler:

Interprets a relocatable expression in an operand involving an "immediate" addressing mode as a PC-relative reference to any symbol involved and generates a PC-relative relocation record for the expression.

Interprets symbolic relocatable expressions in operands involving base-register relative addressing as a reference to the **GOT** entry for the symbol and generates a relocation record indicating such.

Generates a "procedure call" relocation type for all jbsr pseudo-operations it assembles.

Link Editor Changes

Ld can output:

- "simple objects" (.o files)
- "executables" (a.out files)

  program (has entry point)

  shared object (.so) (no entry point)

Ld can process following input:

- simple object files, .o's
- archives, .a's, libraries
- shared objects, .so's

  dynamically bound executables or shared libraries.

Link Editor Changes

The batch link editor, *ld* always sees to it that an a.out file is complete by resolving all references within the program. The new version however, stops working when it runs out of information and will produce an incomplete executable file based on the assumption that later events will provide more.

ld's output can be of two basic types:

– a "simple object" (.o file), produced by simply combining several .o's into a single, larger one using the **–r** flag;

– an "executable" (a.out), which is either a "program" (has an entry point) or a shared object (**.so**) (does not have an entry point).

What gets produced is dependent on what is passed to *ld* in the way of input files and command line options. *ld* will process the following kinds of input files:

– simple object files, .o's;

– archives, .a's, conglomerates of simple objects and also referred to as libraries; and

– shared objects, **.so**'s, also known as dynamically bound executables and sometimes called shared libraries.

Link Editor Changes (cont)

- (–l) flag for specifying object file to be used as a library

New flag

- (–B) flag that takes several keywords

  dynamic for dynamic binding (default)

  static for static binding

Link Editor Changes (cont)

The occurence of .so files in the command line will result in the storage of a structure describing that object in the resulting a.out file and will be utilized by the execution–time *ld.so* to effect dynamic loading and binding.

The **–l** flag is used to specify a short name for an object file to be used as a library. The full name of the object file is derived by adding the prefix lib and suffix of either .a or .so (for archive or shared library, respectively). The specific suffix applied depends on the binding "mode" *ld* is operating on at the time the **–l** flag is processed. The binding mode is specified by a new flag, **–B** that takes several keyword arguments:

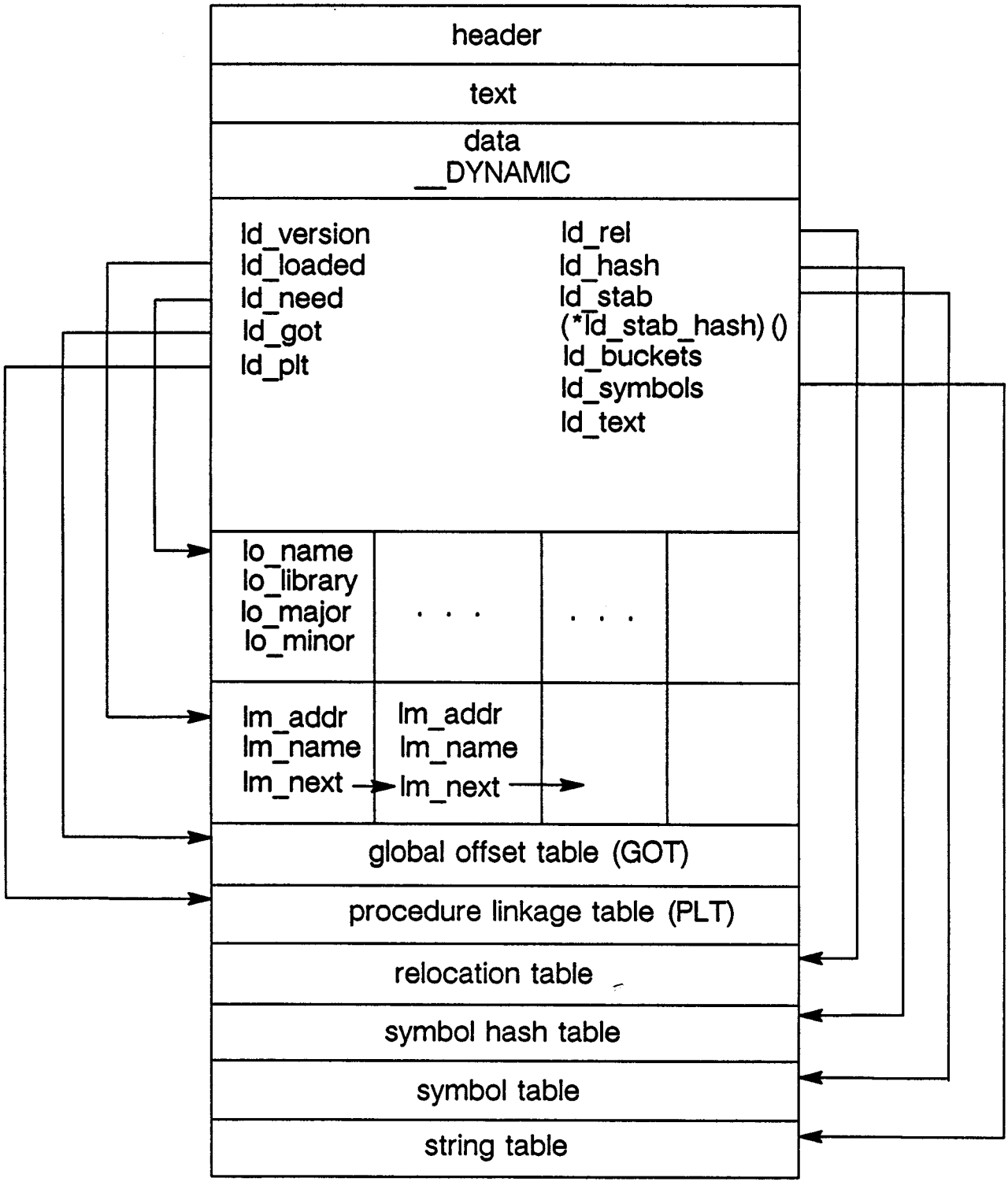
> *dynamic* – Allow dynamic binding, do not resolve symbolic references, and allow creation of execution–time symbol and relocation information. This is the default setting.
>
> *static* – Force static binding, implied by options that generate non–sharable executable formats.

–B*dynamic* and **–B***static* may be specified multiple times and may be used to toggle each other on and off. Like **–l**, their influence is dependent upon their location. When **–B***dynamic* is in effect, any **–l** searches may be satisfied by the first occurrence of either form of library(.so or .a), but if both are encountered the .so form is preferred.

If **–B***static* is in effect, however, *ld* will refuse to use any .so form of libraries it encounters and will continue searching for the .a form. Further, an explicit request to load a .so file is treated as an error.

From the all the input files and all the command line options, *ld* will try to to reduce all symbolic references to numerical offsets within the executable file it is building. To perform this "symbolic reduction", *ld* must know that all information relating to the program has been provided, in particular, no .so will be added at execution time; and/or this program has an entry point and symbolic reduction can be performed for all symbols having definitions existing in the material it has been provided.

The link_dynamic structure



The link_dynamic structure

If ld is able to perform all symbolic and relative reductions and there are no further relocations or definitions to perform, then *ld* has produced a completely linked executable–essentially its old behavior.

However, if any reductions remain, then it is necessary that link editing be done further at execution time for the executable to become useable. In the data spaces of such executables, ld creates an instance of a *link_dynamic* structure that has the label *__DYNAMIC*. The definition of the *link_dynamic* structure is as shown in the previous diagram. This structure is used by the execution–time *ld.so* to obtain .so's necessary to do further symbolic and relative reductions that remain to be done for the executable file.

a.out execution

- Program Prologue Module – crt0
- Execution–time Link Editor – ld.so

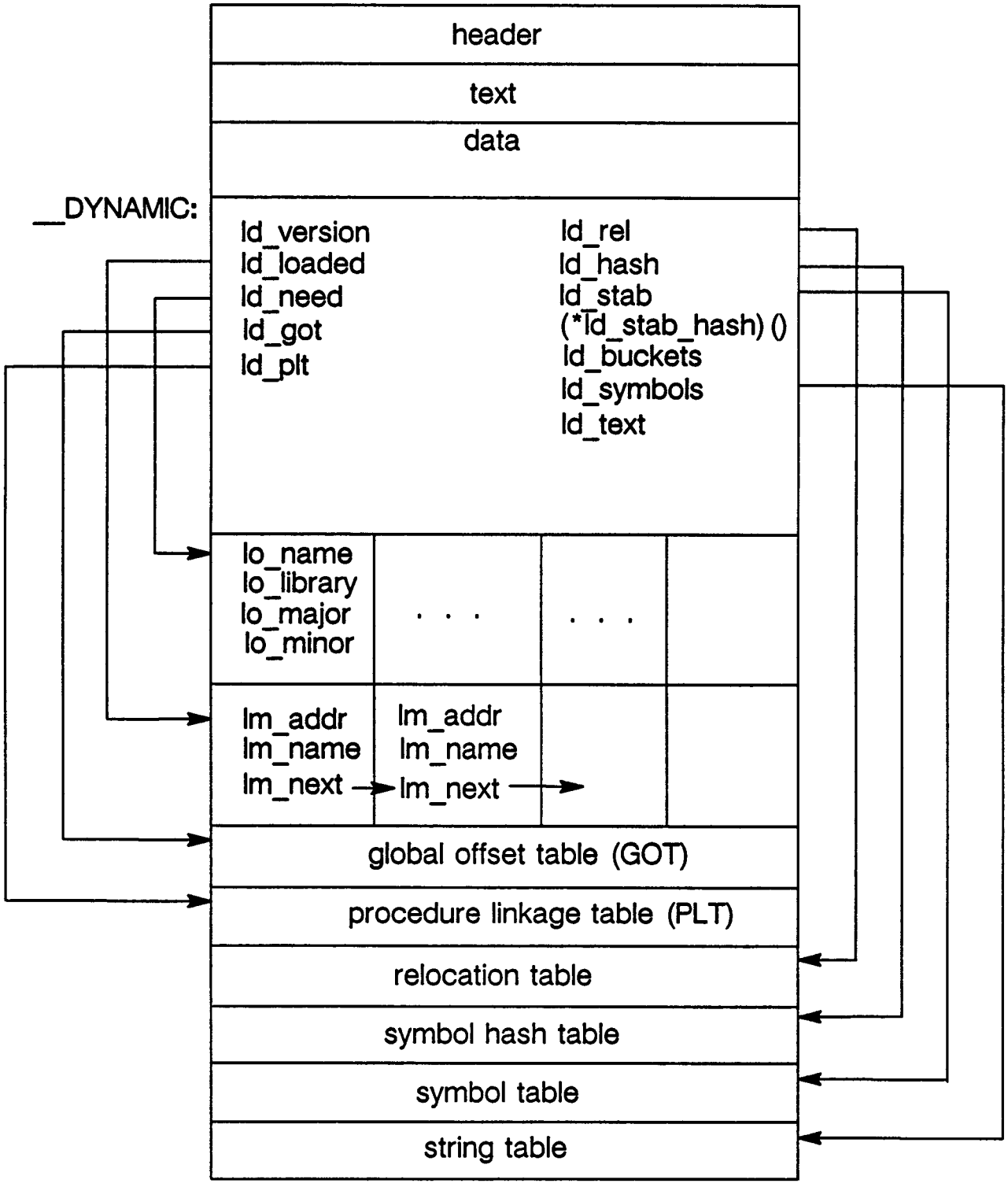
a.out execution

Every main program produced by the standard languages is linked with a program prologue module, *crt0* which actually contains the program's entry point and performs various initializations of the environment prior to calling the main function or logical starting point. *crt0* was modified to contain a reference to the symbol *__DYNAMIC* which is defined as the address of a *link_dynamic* structure that is created when the executable needs execution–time link editing operations. If the structure is not needed, any reference to *__DYNAMIC* is relocated to zero.

At program start–up time, *crt0* checks if *__DYNAMIC* is defined and if so, it opens the file */lib/ld.so* and requests the system to map it into the program's address space via the *mmap( )* system call. It then calls *ld.so*, and passes the address of the program's *__DYNAMIC* structure.

After *crt0* transfers control to *ld.so, ld.so* executes a short bootstrap routine that performs any relocation *ld.so* itself requires. The process of building *ld.so* results in very simple forms of relocation that can be easily handled by this bootstrap routine. *ld.so* then processes the information contained in the *__DYNAMIC* structure to perform the link editing that is required to start the execution of the program.

a.out execution (cont.)



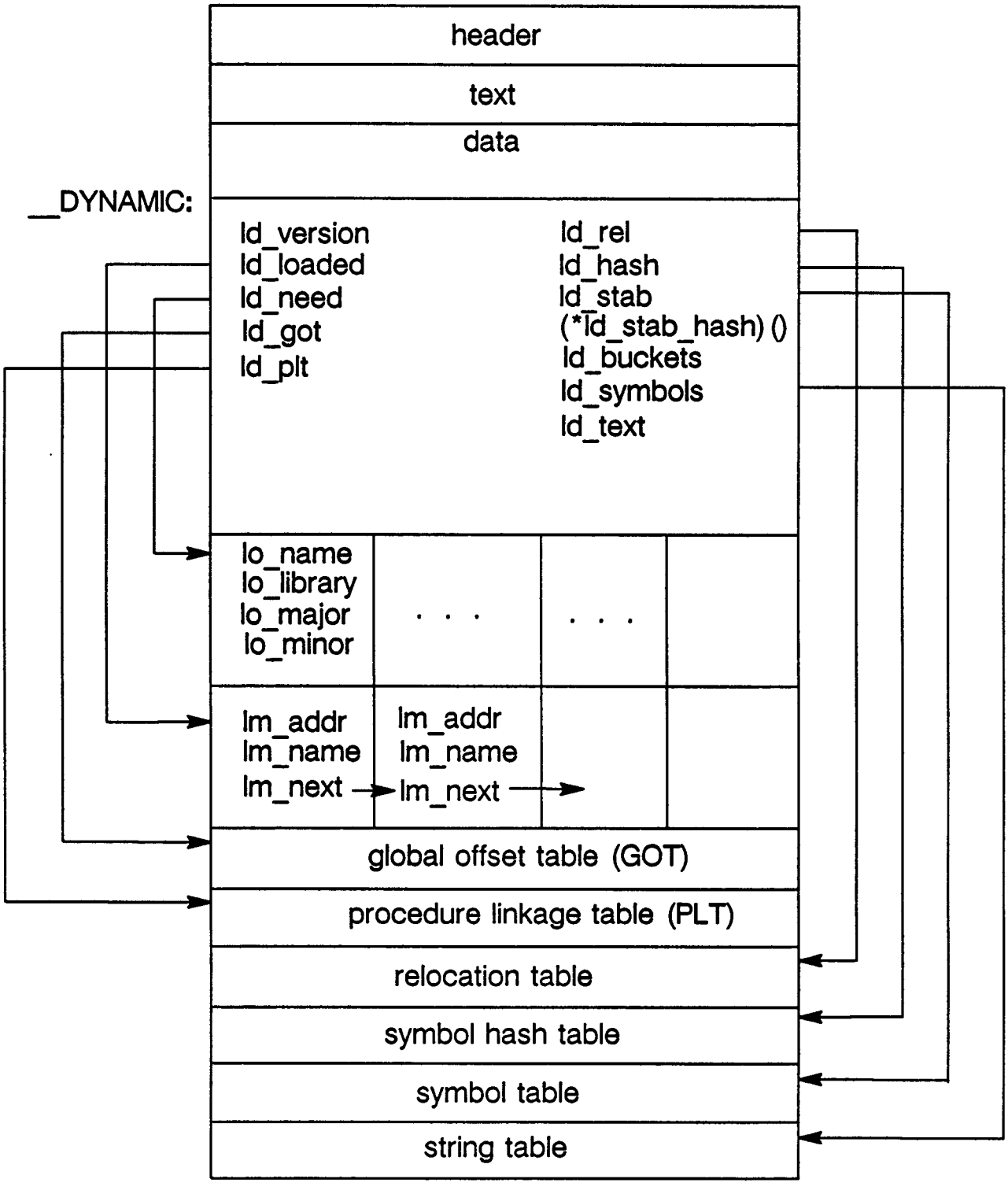
a.out execution (cont.)

ld.so's first action is to examine the *ld_need* entry of the program's *__DYNAMIC* structure which is an offset relative to the structure itself to an array of *link_object* structures which identifies the .so's that were specified in the command line used to build the program. These shared objects are the ones that must be added to the program's address space and link edited.

For each entry in the *ld_need* array, *ld.so* looks up the file identified and maps it into the process's address space, the location of which is decided by the system. A given .so may reside at different locations in the address spaces of different processes. At this point, failure to find a particular object is a fatal error and results in program termination. At the end of the initial program's *ld_need* array, *ld.so* examines the *__DYNAMIC* structure of the first .so file mapped in. It processes that .so's *ld_need* array, and proceeds likewise through all the mapped .so's. For each .so that is mapped, *ld.so* builds a *link_map* structure whose definition is shown in the previous diagram. This structure is placed in a single linked list in the order it was loaded. This list is rooted in the *ld_loaded* member of the initial program's *__DYNAMIC* structure. This ordering of the mapped or loaded .so's is used to establish the search order for undefined symbol look–ups.

After all the modules have been mapped, *ld.so* attempts to complete the link editing operations begun by *ld*. It will attempt to perform first symbolic and then relative reductions on all the references outside of procedure linkage tables left in the program. In particular this includes the allocation of uninitialized commons (since all information regarding their use is finally present). If all non–procedural references can not be reduced to absolute addresses, then it is because a definition for a given symbol is not available, in which case *ld.so* terminates the program with an "undefined symbol" diagnostic.

a.out execution (cont.)



a.out execution (cont.)

All non–reduced references in any **PLT**'s in the loaded executables are not processed during the program start–up. Rather, all such references are initialized to cause the initial calls to the procedures they reference to result in the transfer of control to *ld.so*. *ld.so* will then try to reduce the original reference to the appropriate absolute address and modify the referencing **PLT** entry to direct future calls directly to the targeted procedure. Deferring the binding of procedure entry points until their first reference saves performing perhaps thousands of unnecessary bindings to entry points programs may never call.

Interposition

- Capability to interpose targets for symbolic references.
- Consider building a program using libinterpose supplied as a .so file

```
% cc –o prog prog.o –linterpose
```

Interposition

The command on the opposite page invokes *ld,* and invisibly includes a reference to the *C library* after *libinterpose. libinterpose* defines entry points for read and write that in addition to performing the required system calls take statistics on their use.

In this case, both *libinterpose* and *libc* define entry points for read and write, the latter being the standard entry points for the "stub" routines performing the respective system calls. In the presence of multiple definitions of a symbol, the link editor resolves the issue of which to select by using the ordering established when prog was linked. Since *libinterpose* was specified first, its entry points are used for references to read and write. These entry points are used globally, so that read and write references made internal to the *C library* also use *libinterpose.*

Interposition offers the opportunity for programmers to provide "added value" to a "standard" system function by providing a new implementation of the function.

Construction of a Shared C Library

- Compile all C source modules using the –pic option
- Modification of assembly–language source files so that assembly source is also position–independent
- Link editing the collection of .o files to create libc.so.1.0

```
%ld –o libc.so.1.0 –assert pure_text *.o
```

- Run–time link editor is created

```
%ld –o ld.so –Bsymbolic –assert nosymbolic
```

... list of modules ...

Construction of a Shared C Library

The construction of the shared C library involves:

– The compilation of all of its C source modules using the **-pic** option;

– modifications of some assembly language source files so that the assembly source is also position–independent; and

– link editing the resulting collection of .o files to create *libs.so.1.0*

The ld operation is conceptually just:

% ld -o libc.so.1.0 -assert pure_text *.o

Since no entry point is supplied, and lacking any other clue to an absolute address, *ld* simply stops processing after it has combined all the object files, built the **GOT** and **PLT**'s, and performed any intra–library PC–relative relocations. The assertion request will cause *ld* to issue a diagnostic if the library requires further relocation to the code contained within it, a sign that a non–**PIC** object has found its way into the library.

The execution–time link–editor, *ld.so* is built with an *ld* command that has the form:

% ld -o ld.so -Bsymbolic -assert nosymbolic ...list of modules...

It is conceptually just like other .so files but, it also involves the use of a special binding control option **-Bsymbolic** and the assertion **nosymbolic**.

Normally when *ld* builds a program lacking an entry point or other absolute addressing information, it is unable to perform its symbolic reduction operations simply because it can not assume that symbols from other executable files will not be added later. However, *ld.so* must be self–contained, or else it would require itself to operate and would otherwise pollute the symbol space of the programs it link edits. The **-Bsymbolic** flag forces *ld* to perform symbolic reduction operations using the information it has now, leaving only relative reductions to be performed – something *ld.so* resolves as part of its bootstrapping operations, resulting in a self–contained program.

Module 5

Objectives

Objectives: Upon completion of this module, the student will be able to:

- Explain the advantages to the 4.3BSD file system.
- Describe the vnode interface layer to the file system.
- List the four fields in a directory entry.
- Describe the function of the superblock, and cylinder group structures.
- List at least 80% of the fields in the disk inode structure and explain what they are used for.
- Name the routines involved in determining the global placement policies.
- Explain the allocation routines using the flowcharts and describe how the fragments are located quickly.

Evaluation: quiz

Reference information: Course Guide

Introduction

- A disk drive is divided into one or more partitions, each partition contains one file system.
- Files are accessed in fixed length blocks on the disk.
- Information about a file is kept in an inode (index node).
- A directory references an inode for each named object on disk.
- Sun uses the 4.3 BSD implementation of the file system to speed disk access.

Introduction

From the user level a file is viewed as a stream of bytes. The underlying hardware manages the file data as blocks of bytes. The minimum size the hardware will write or read from disk is a sector, typically 512 bytes.

Sun uses a block I/O system that transfers blocks that are 16 contiguous sectors in length or 8,192 bytes (8 kb).

In order to avoid wasting disk space for small files a file may be allocated a fraction of the block. The fraction of the block is a multiple of a fragment which is 1024 bytes on Sun file systems.

The inode holds information about a file's permissions, owner, time stamps, and where the data blocks that comprise the file are located on disk.

Directories are treated as sequentially accessed files. A different set of routines are used to access directories, they basically parallel the routines used to manipulate files.

The Fast File System

The advantages to the the 4.3 BSD file system over the traditional file system are:

- the block size has been increased so more data is accessed with each disk I/O.
- portions of a disk block, fragments, can be used by different files to avoid wasting disk space.
- the seek distance between inodes and data blocks has been reduced by dividing the inodes in a file system into cylinder groups.
- the BSD file system is more robust because it saves redundant copies of the superblock in each cylinder group.

The Fast File System

The problem with the System V implementation of the file system is the large separation between inodes and their corresponding data blocks. Also a small block size caused the fragmentation of files to occur very rapidly and consequently many seeks within the file.

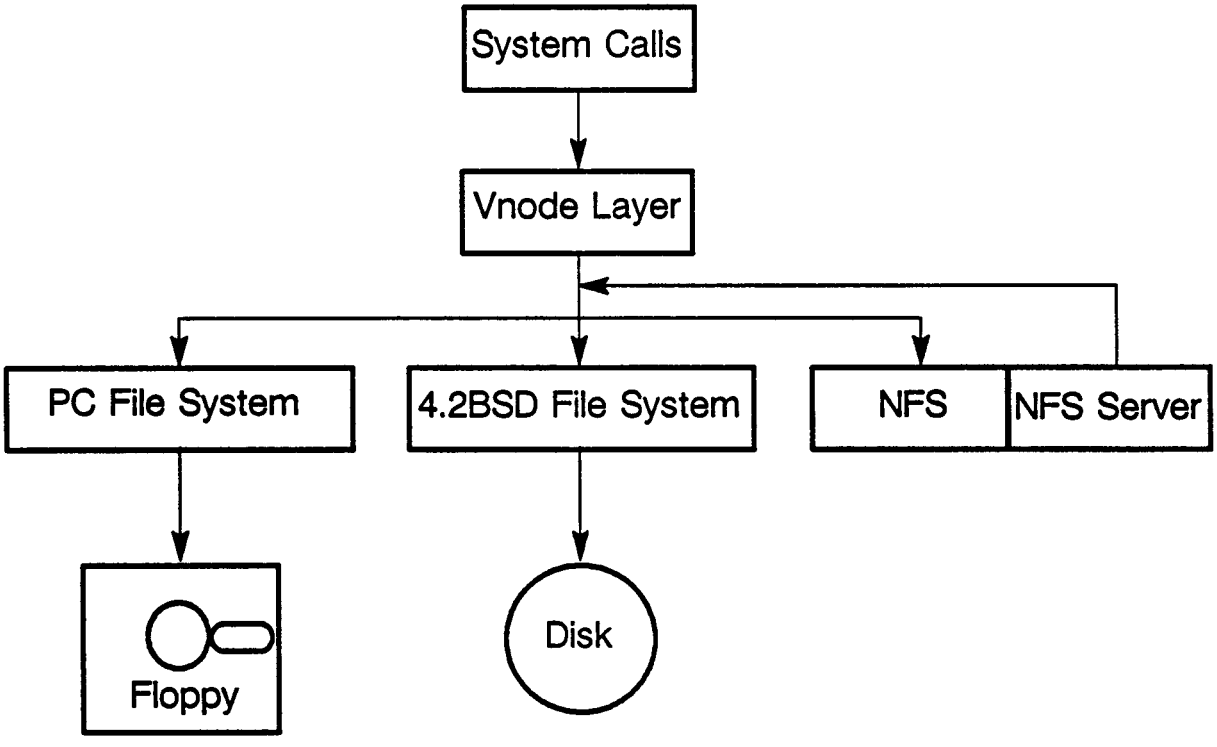
Performance was improved by using larger data blocks. This not only allowed the file to be accessed with fewer disk operations, it also allowed much larger files to be accessed before having to use single indirect or double indirect blocks.

To prevent a large number of small files from using up all the data blocks in a file system, multiple files are allowed to occupy the same disk block in multiples of fragments. Each fragment is 1024 bytes (1kb), so there are 8 fragments per data block.

To avoid the long seek between the inode and the data blocks, the file system is divided into cylinder groups. Each cylinder group contains a copy of the superblock, and an inode for each 2 kilobytes in the cylinder group. By spreading the inodes throughout the disk cylinders the inode to data block seeks are reduced.

By having multiple copies of the superblock throughout the fs there is less chance of losing the superblock information due to hardware failure. The offset to the superblock is at an offset calculated to be one track farther than the offset of the previous cylinder group's superblock. This means any single head or cylinder can be lost without losing all the copies of the superblock.

The Vnode Interface



The Vnode Interface

The SunOS operating system has the ability to access Unix file systems on local disks, remote file systems via NFS, or even other types of file systems, such as AT&T System V file systems, or MSDOS file systems. This is accomplished using vnodes.

If locking is required it has to be done at the file system specific layer, such as UFS, rather than the vnode layer. This means all file system operations must be atomic.

An object oriented approach was used to implement vnodes in the kernel. This means the data structures such as vnode.h, will contain a list of pointers to generic operations. These operations will be handled by a corresponding specific file system dependent routine, depending on the object being manipulated.

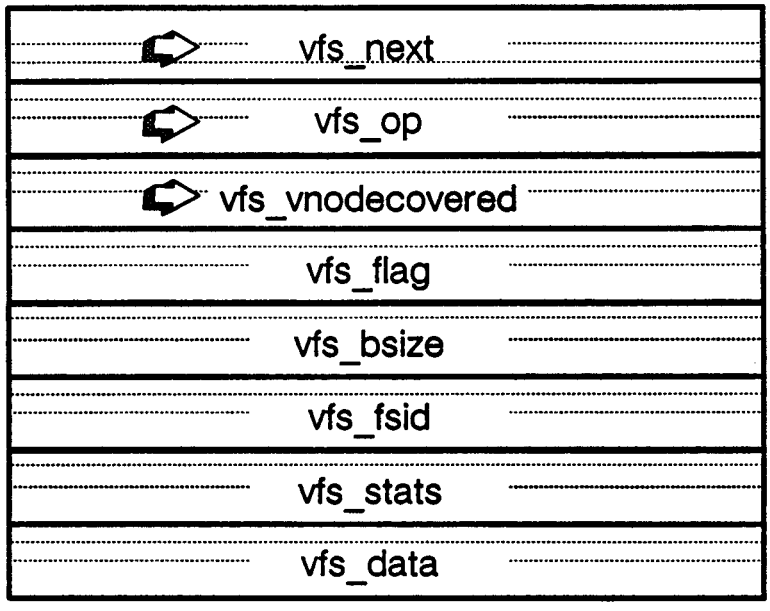
The *mount(2)* command now accepts a file system type and a pointer to file system specific data.

As shown in the diagram the vnode layer was placed just above the inode layer, and below the system call layer.

The operation is then carried out on the behalf of the process doing the system call, the process may be put to sleep during the course of the operation.

The different file systems are manipulated through the vfs (virtual file system) structure, which is linked to a list of mount table entries. The first file system is root. The vfs_data pointer points to the file system dependent data, for UFS it will point to a mount table entry.

The *vfs* structure



The *vfs* structure

vfs_next – file systems are linked starting with the root partition.

vfs_op – pointer to the *vfs_ops* structure (below).

vfs_vnodecovered – vnode of the file system this file system is mounted on.

vfs_flag – see Appendix B.

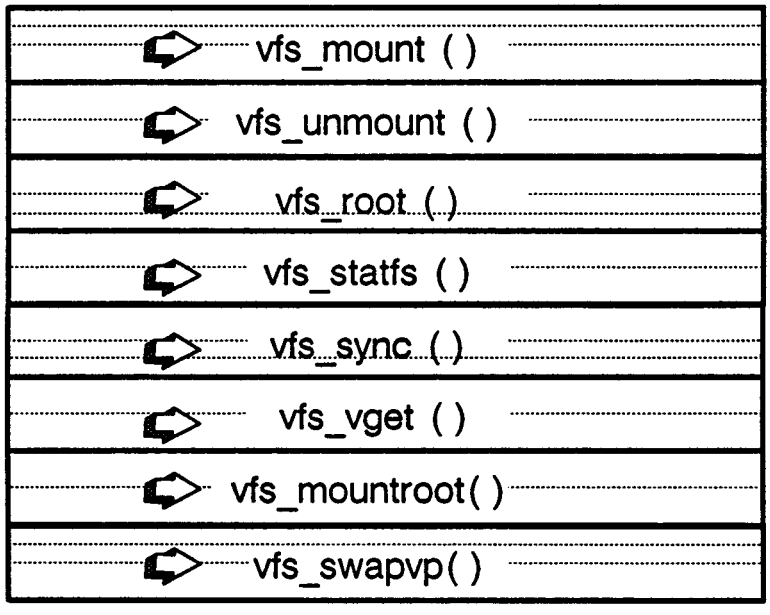
vfs_bsize – size of data block on this file system.

vfs_fsid – file system id.

vfs_stats – pointer to file system statistics.

vfs_data – pointer to specific information about the file system (for ufs it points to a struct mount, for nfs it points to a struct mntinfo).

vfsops


vfs_mount ( )
vfs_unmount ( )
vfs_root ( )
vfs_statfs ( )
vfs_sync ( )
vfs_vget ( )
vfs_mountroot( )
vfs_swapvp( )

The *vfsops* structure

vfs_mount( ) – mounts *vfs,* receives a pointer to the pathname *(pathp) and* a pointer to file system dependent data *(datap).*

vfs_unmount( ) – unmounts the indicated vfs.

vfs_root( ) – returns a pointer to the root vnode for this file system.

vfs_statfs( ) – returns a pointer to a *statfs* structure which contains information about the file system such as type, block size, free blocks, etc. (see Appendix B).

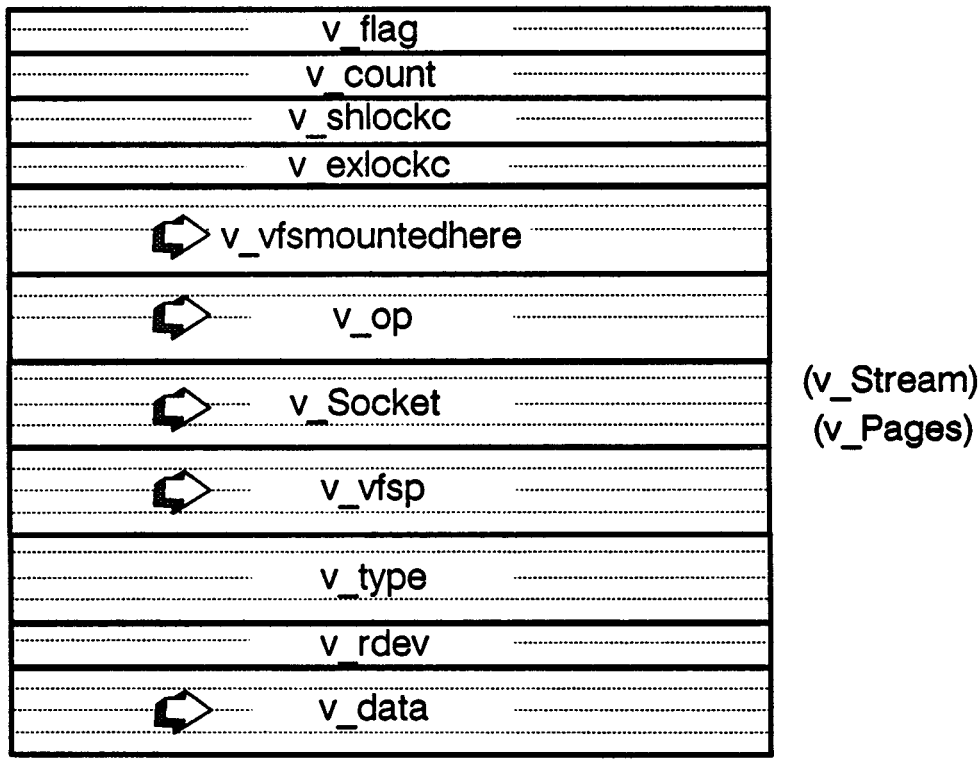
vfs_sync( ) – writes out all the cached *vfs* information.

vfs_vget( ) – get a pointer *(vpp)* to the *vn* for the *fidp.*

vfs_mountroot( ) – mounts the root file system.

vfs_swapvp( ) – returns a pointer to the vnode for the swap device.

vnode structure



vnode structure

v_flag – vnode flags where:

VROOT – 0x1 –file system root.

VEXLOCK – exclusive lock.

VSHLOCK – 0x04 – shared lock.

VLWAIT – 0x08 – proc is waiting on shared or exclusive lock.

VNOCACHE – 0x10 – don't keep cache pages on vnode.

VNOMAP – 0x20 – vnode cannot be mapped.

VISSWAP – 0x40 –vnode is part of virtual swap device.

v_count – reference count (manipulated by *VNHOLD* and *VNRELEASE)*

v_shlockc – the number of shared locks.

v_exlockc – the number of exclusive locks.

v_vfsmountedhere – points to the vfs structure for the file system mounted here.

v_op – vnode operations.

union

v_Socket – pointer to a socket structure

v_Stream – pointer to a stream structure

v_Pages – pointer to a page structure

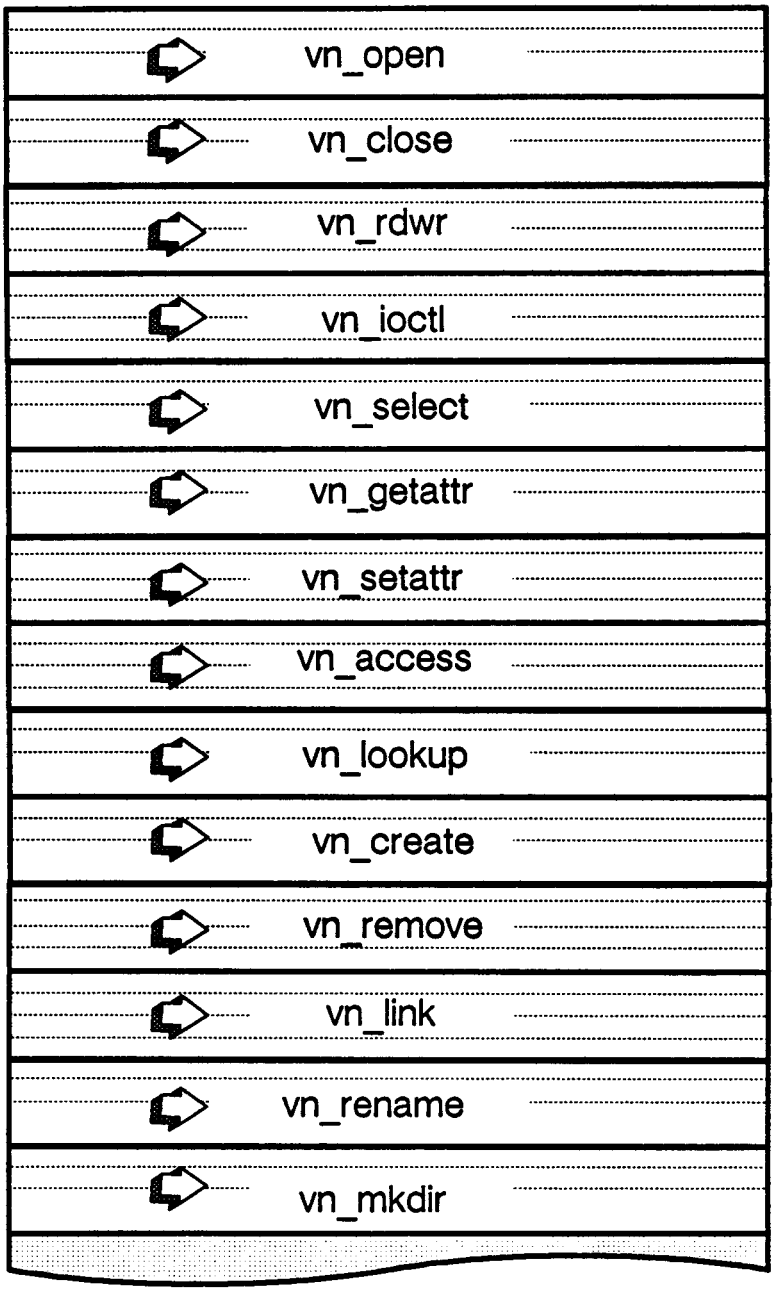
v_vfsp – pointer to the vfs structure for the file system we are in.

v_type – the type of vnode.

v_rdev – device number if it is of type VCHR or VBLK.

v_data – a pointer to the private data (for ufs this points to an inode).

vnodeops structure



vnodeops structure

vn_open– perform open on vnode.

vn_close – close vnode, called when closing the last reference to a vnode in the file table if the vnode is a device, closed on the last user close of a file descriptor otherwise.

vn_rdwr – reads or writes a number of bytes to the specified offset in the file.

vn_ioctl – perform ioctl on vnode

vn_select – perform a select on the vnode.

vn_getattr – get attributes for the vnode.

vn_setattr – set attributes

vn_access – check the access permissions for the vnode.

vn_lookup – lookup a component of a pathname in the specified vnode.

vn_create – create a new entry in the directory in the vnode.

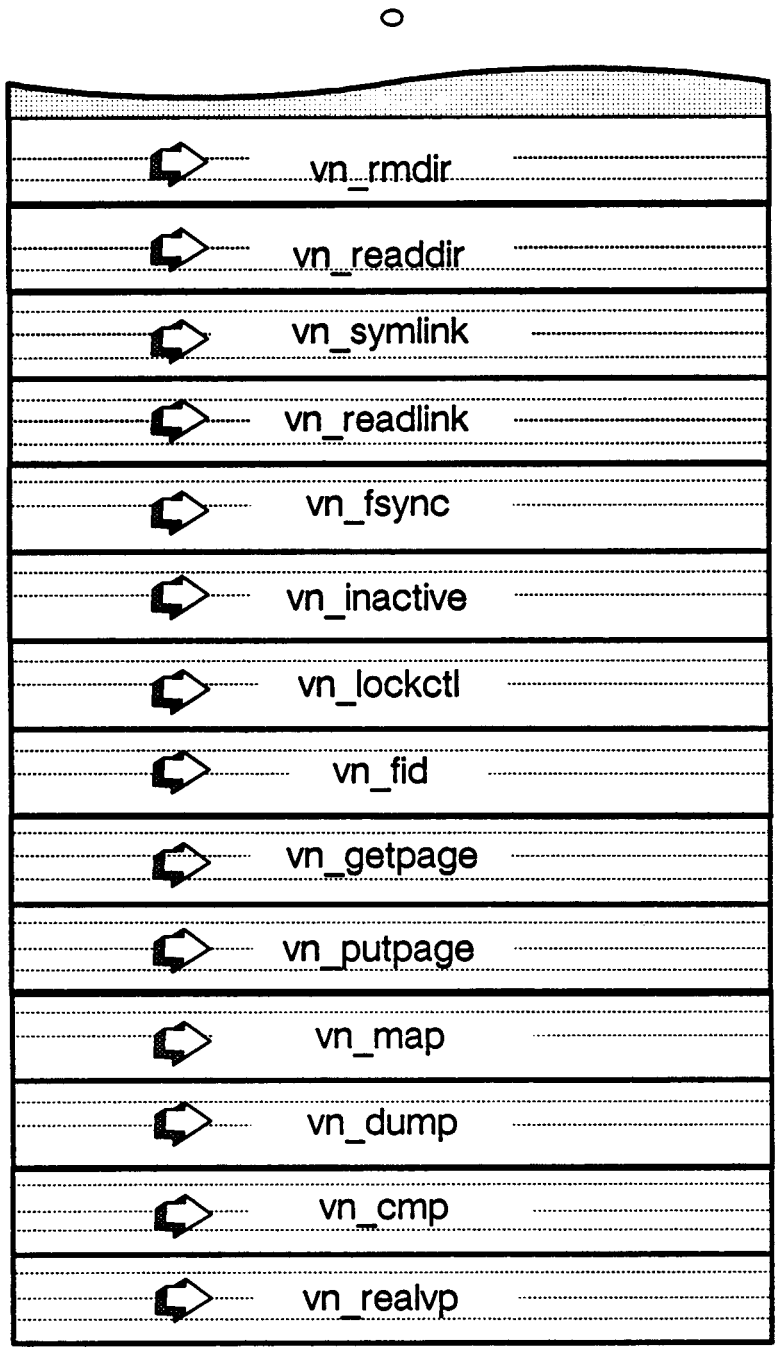
vn_remove – delete an entry in the directory in the vnode.

vn_link – link the vnode to a target name.

vn_rename – rename the file name in the directory vnode.

vn_mkdir – create a directory in the vnode.

vnodeops structure (cont.)



vnodeops structure (cont.)

vn_rmdir – remove a directory from the vnode.

vn_readdir – read directory entries from the vnode.

vn_symlink – create a symbolic link in the vnode.

vn_readlink – read symbolic link vnode.

vn_fsync – write out all cached information for the file vnode.

vn_inactive – the vnode is no longer referenced by the vnode layer and can be deallocated.

vn_lockctl – record locking request are passed to the local Lock–Manager daemon.

vn_fid – returns a *fid* structure for this vnode.

vn_getpage – gets a range of pages which may involve calling the device driver to read the page from disk.

vn_putpage – write out dirty pages in the specified range.

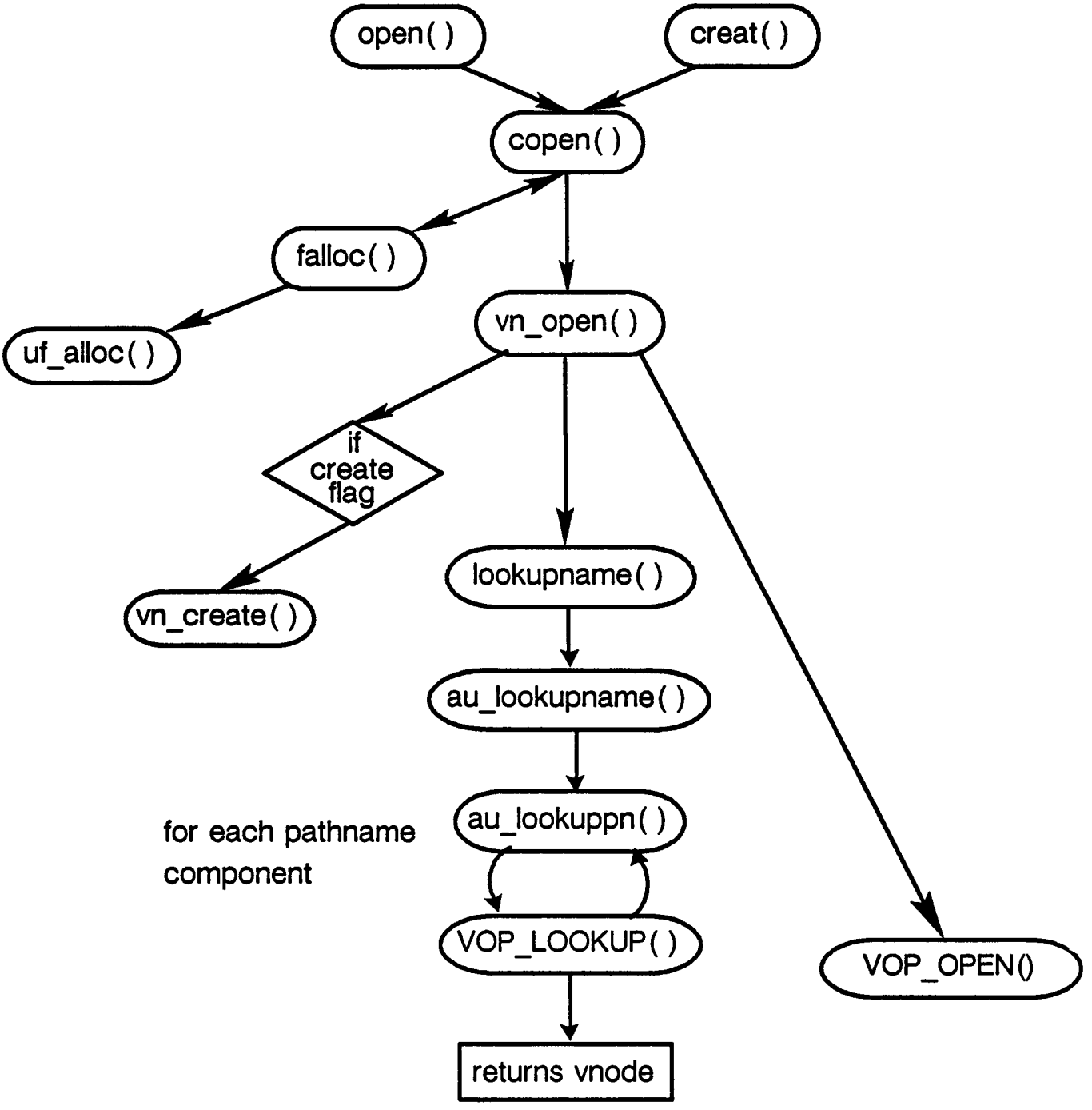
vn_map – create a mapping to the address space with specified protections and flags.

vn_dump – dumps a range of addresses within a page to a given file system block (badop of Unix filesystem).

vn_cmp – compare two vnodes for equality.

vn_realvp – used by specfs to hold a pointer to a 'shadow' node or the real vnode.

Opening a Vnode



Opening a Vnode

The *open()* or *creat()* routines call *copen()* (common open), this routine will in turn call *falloc()*.

The *falloc()* routine creates an entry in the system open file table and calls *uf_alloc()* to create an entry in the user's file descriptor table *(u_ofile)*. *copen()* then calls *vn_open()*.

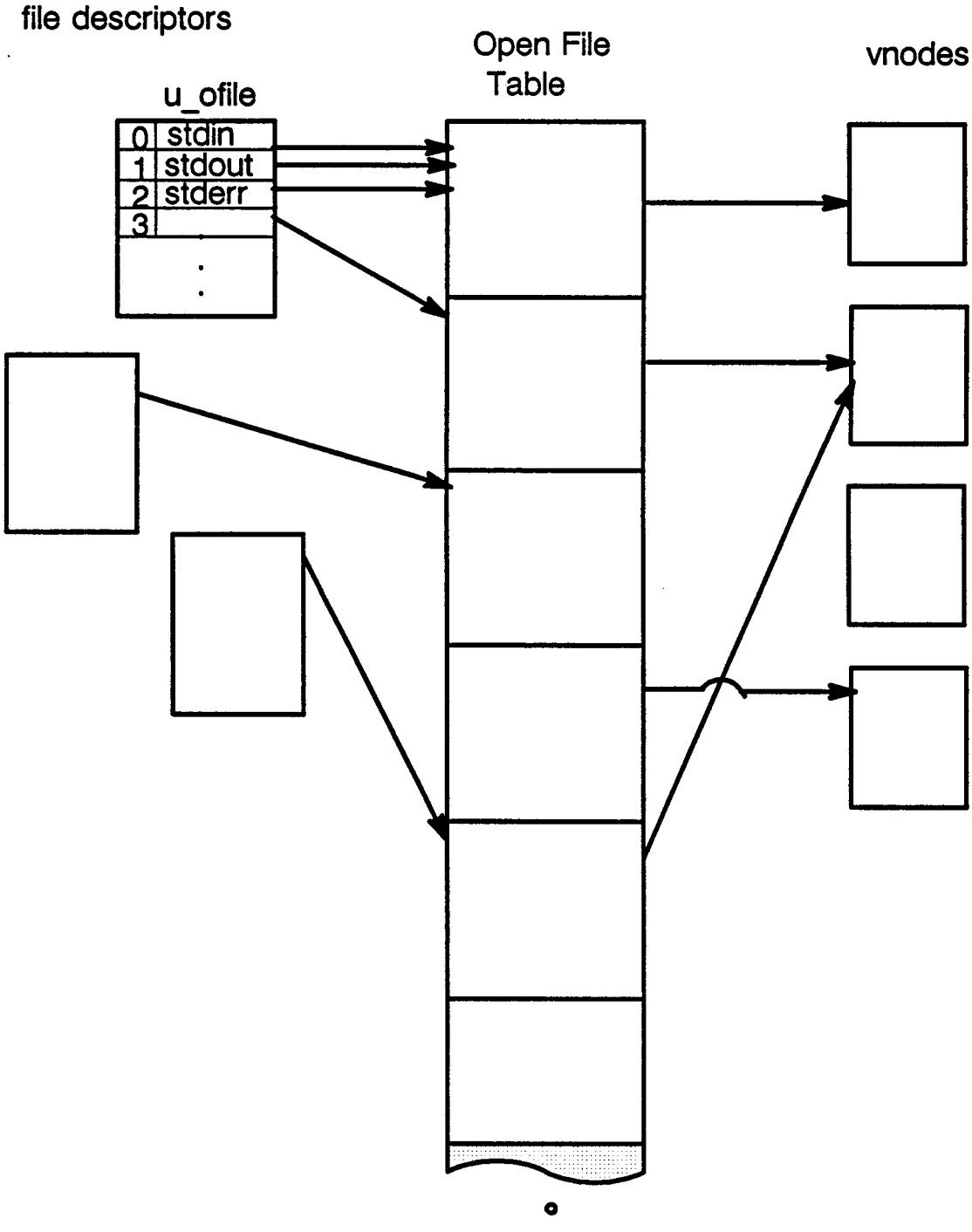
vn_open() creates a vnode structure if the create flag is set, otherwise it opens the vnode, opens the file and increments its reference count. It then calls *lookupname()* which just calls *au_lookupname()*. *au_lookupname()* allocates a buffer for the pathname through a call to *pn_get()*. Eventually the pathname buffer is released (when no longer needed by a call to *pn_free()*). *au_lookupname()* then calls *au_lookuppn()*.

au_lookuppn() is the vnode equivalent to the *namei()* routine. For each component of the pathname it calls the generic *VOP_LOOKUP()* which will call the file system specific lookup routine. For a vnode on a Unix file system this would cause the routine *ufs_lookup()* to be called, which would in turn call *dirlook()*, which is described later in this module.

au_lookuppn() calls the *VOP_LOOKUP()* routine until the last component of the pathname at which time it returns a vnode.

When we eventually return to *vn_open()* again with the vnode of the last component, *vn_open()* calls the generic *VOP_OPEN()* routine. This will call the file system specific routine to open the file represented by the vnode.

The System Open File Table



The System Open File Table

The *open()* call is passed a filename, the type of open, and if it is being called to create a file, it passes the file permissions.

The user's file descriptor entry will contain a pointer to the system open file table.

The open file table contains the current byte offset in the file, and a pointer to the vnode. The vnode has a pointer, *v_data,* that points to the file system specific data, for ufs this would be a pointer to the inode.

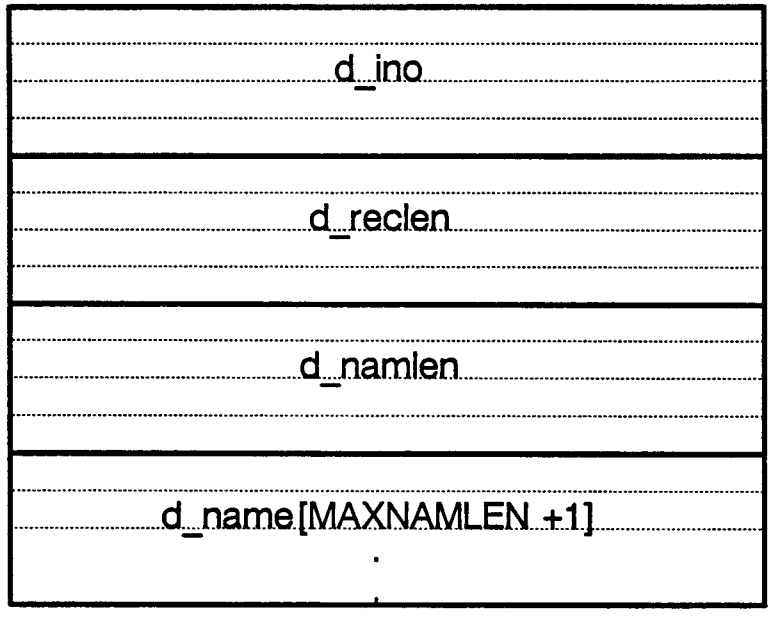
When open is applied to a symbolic link it returns an *fd* for the file pointed to by the pathname in the link file, rather than an *fd* for the link itself.

The entries in the open file table are defined in file.h which appears in Appendix B.

Directories

- Directories are divided into blocks 512 bytes long.
- Each directory entry consists of four fields defined in the *direct* structure.

direct structure



Directories

The directories are broken into blocks that can be written to disk in one operation. This is defined in *fsdir.h* as *DIRBLKSIZ* currently set at 512 bytes.

Directory entries are not allowed to span multiple directory blocks.

d_ino – inode number of this entry.

d_reclen – the length in bytes of this directory entry.

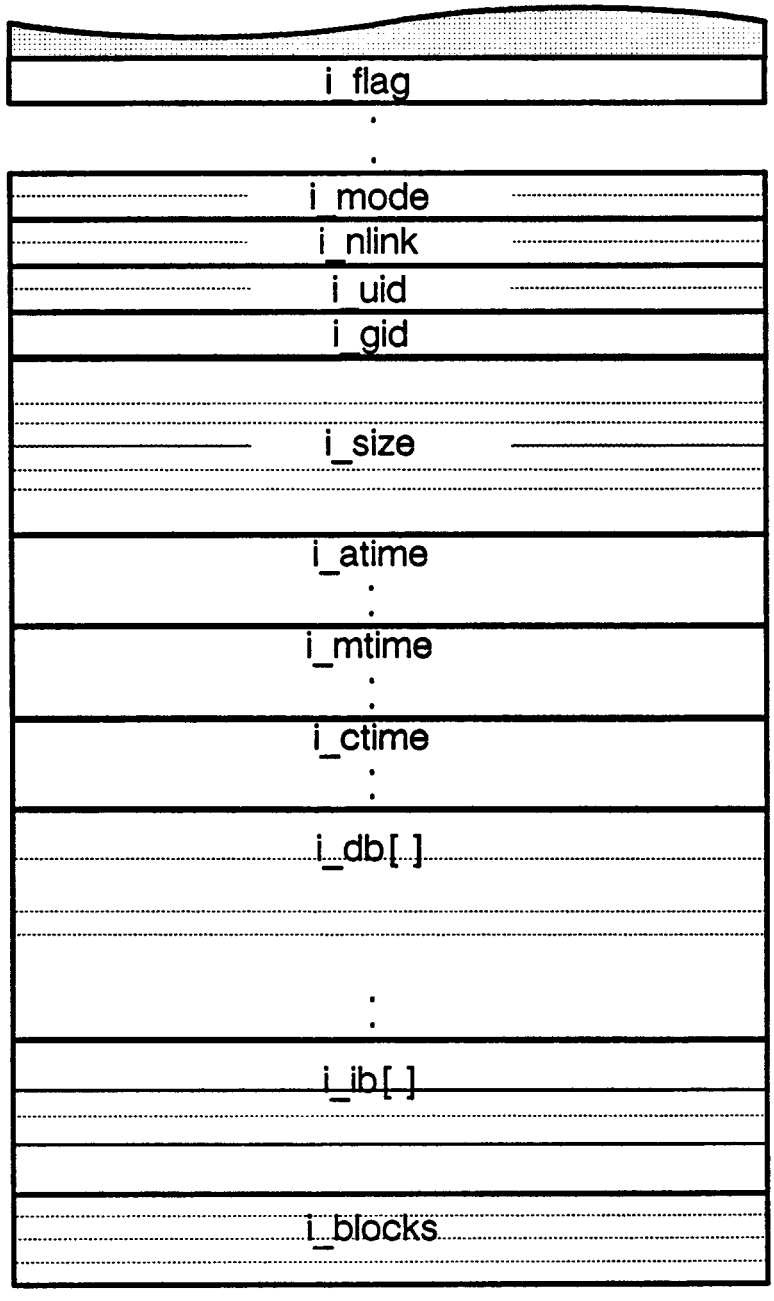
d_namlen – the length of the file name in bytes. The filename cannot be greater than 255 characters.

d_name[MAXNAMLEN + 1] – the rest of the entry consist of the characters that make up the filename. The name field is of variable length, up to 255 characters, it is null terminated, and padded to a 4 byte boundary.

Any holes in the directory caused by deletion or space at the end of a directory block is accounted for by having the previous entry include any unused bytes in its record length (*d_reclen*).

Deleted entries at the beginning of a 512 block are recognized by the inode number (*d_ino*) being 0.

inode structure



inode structure

This shows most of the fields in the inode structure, the entire structure is in inode.h, see Appendix B.

i_flag - where:

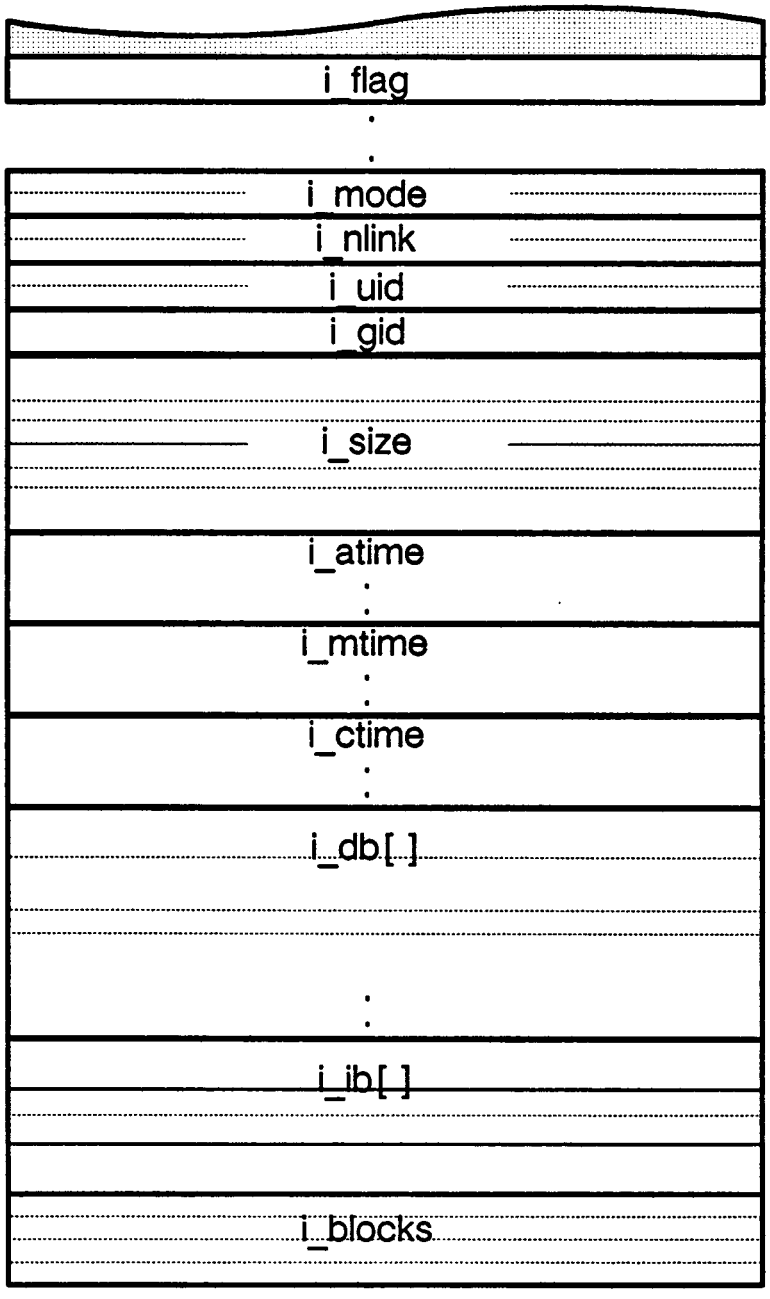
ILOCKED - inode is busy (locked)
IUPD - file has been modified
IACC - inode access time needs to be updated
IMOD - inode has been modified
IWANT - a process is waiting for the lock to be released
ISYNC - do all allocation synchronously
ICHG - inode has been changed
ILWAIT - a process is waiting on file lock
IREF - inode is being referenced
IMODTIME - modification time has already been set

i_mode - the mode and type of file, where:

IFMT - 0170000 - file type mask
IFIFO - 0010000 - named pipe
IFCHR - 0020000 - special character file
IFDIR - 0040000 - directory
IFBLK - 0060000 - special block file
IFREG - 0100000 - regular file
IFLNK - 0120000 - symbolic link
IFSOCK - 0140000 - socket

ISUID - 04000 - set uid on exec
ISGID - 02000 - set gid on exec
ISVTX - 01000 - save swapped text after use
IREAD - 0400 - read permission
IWRITE - 0200 - write permission
IEXEC - 0100 - execute permission

inode structure



inode structure

i_nlink – the number of links to a file.

i_uid – the user id of the owner.

i_gid – the owner's group id.

i_size – size of file in bytes.

i_atime – the time the file was last accessed.

i_mtime – the time the file was last modified.

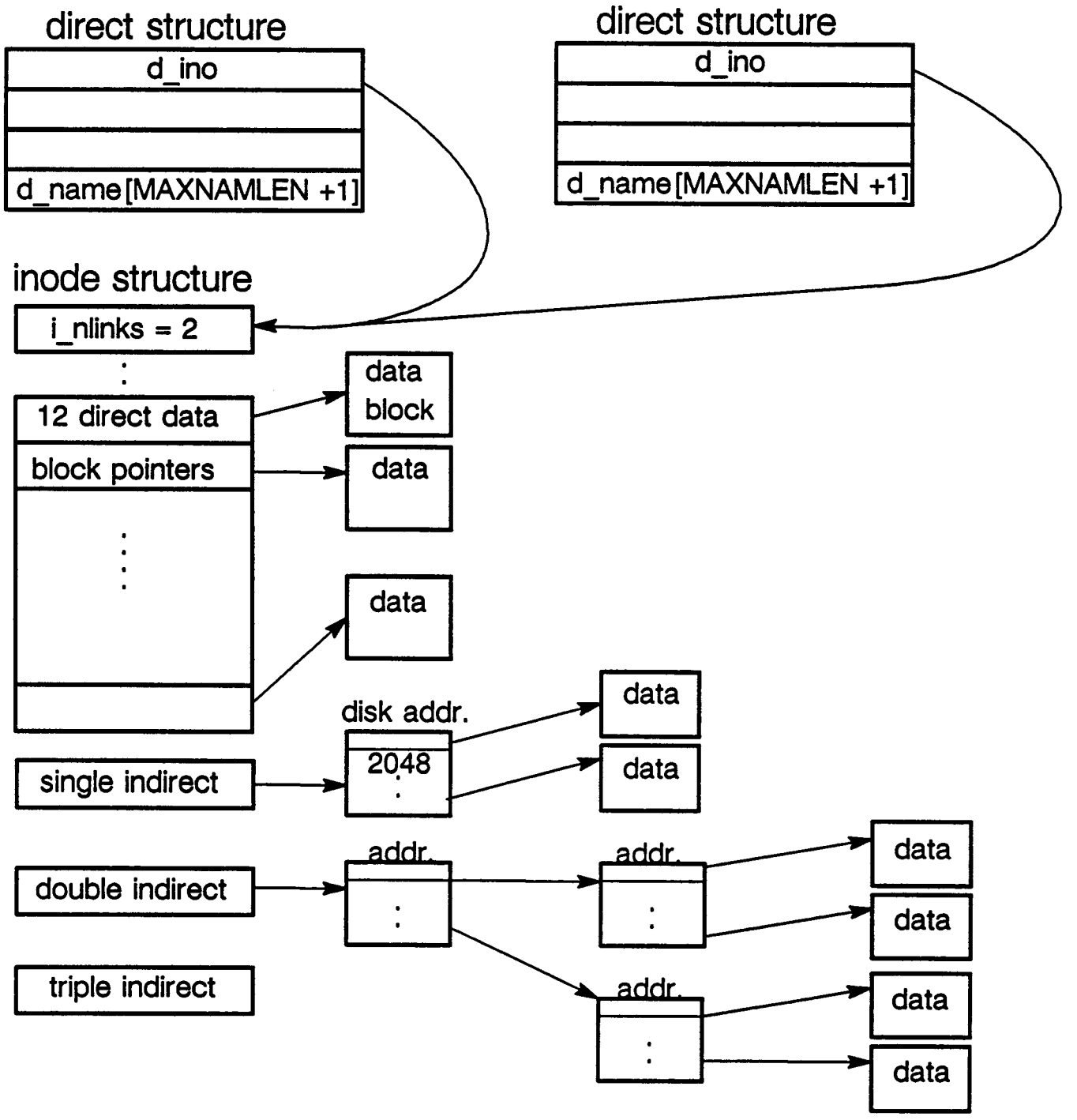
i_ctime – the time the inode was last modified.

i_db[NDADDR] – disk block addresses of direct blocks. NDADDR (number of direct addresses) is set at 12.

i_ib[NIADDR] – disk block addresses of indirect blocks. NIADDR (number of indirect addresses) is defined as 3.

i_blocks – number of blocks used by this inode.

inodes


direct structure
d_ino
d_name[MAXNAMLEN +1]
direct structure
d_ino
d_name[MAXNAMLEN +1]
inode structure
i_nlinks = 2
12 direct data
block pointers
data block
data
data
disk addr.
2048
single indirect
data
data
double indirect
addr.
addr.
data
data
triple indirect
addr.
data
data

inodes

The inode number, contained in the directory structure creates a 'hard link ' between the filename and the inode.

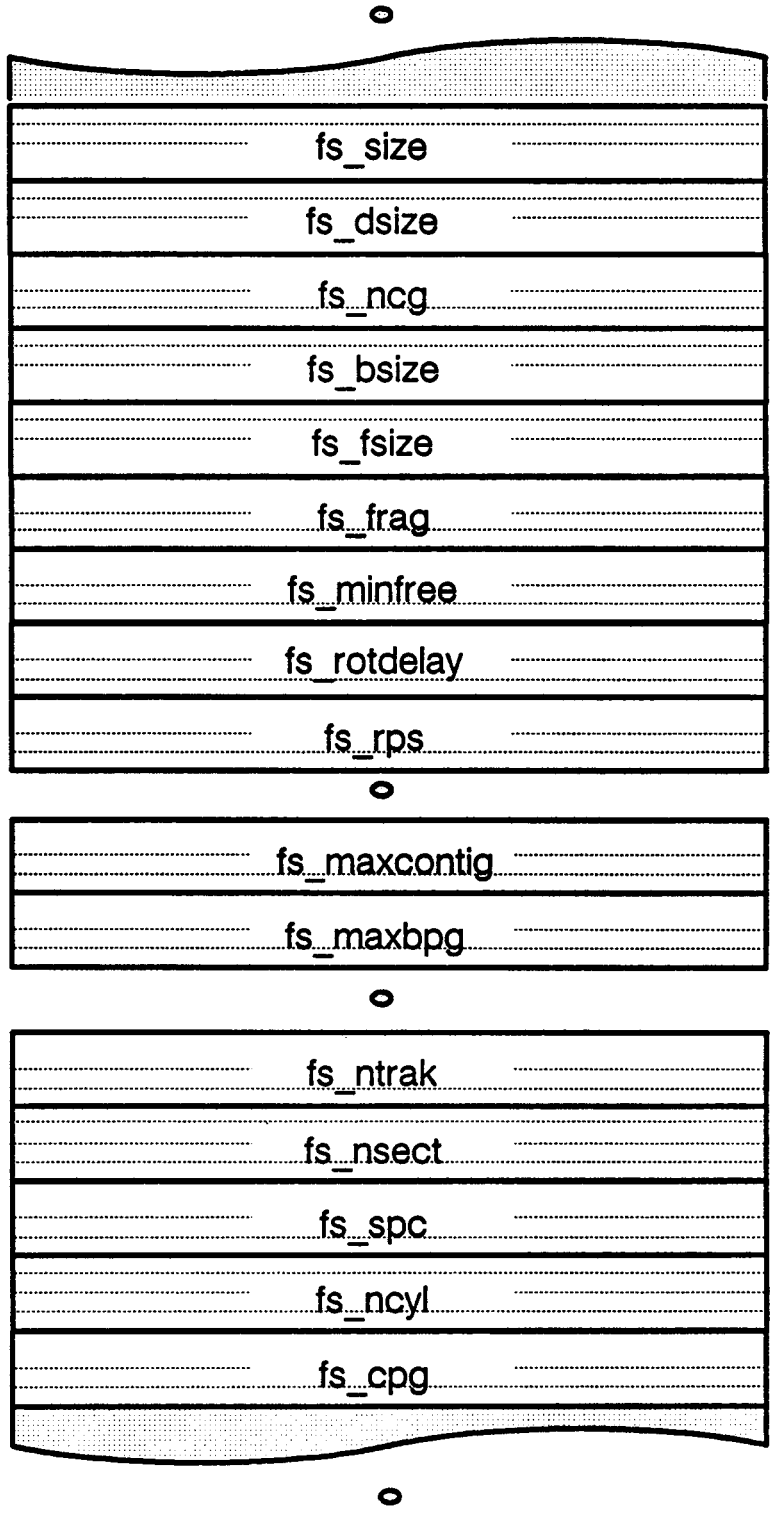
A file has a single inode but multiple directory entries may contain the same inode reference. The reference count in the inode structure indicates the number of links to this inode. When the last link goes away the inode can be deallocated.

Sun also supports a link to a file that is implemented by pointing to a file that contains a pathname. This is a soft or symbolic link and allows us to point to a file or directory on a different file system. This is necessary since hardlinks are not allowed to point to directories as this may cause an endless loop during pathname translation. The exception to this is root is allowed to create a hard link to a directory, but it cannot be removed by conventional means.

If the number of symbolic links encountered during a translation is greater than 20, it is assumed that a loop has been created and an error is returned.

The potential size of a file depends on the size of the blocks that hold the data.

The Superblock



The Superblock

The superblock contains the underlying physical and software defined parameters for a filesystem. There are only selected entries shown here the entire structure is in the include file fs.h in Appendix B.

fs_size – is the number of blocks in the file system.

fs_dsize – the number of data blocks in this fs.

fs_ncg – the number of cylinder groups in this fs.

fs_bsize – the size of a data block in bytes in this fs, typically 8,192.

fs_fsize – the size of a fragment in bytes in this fs, typically 1024.

fs_frag – the number of fragments in a data block.

fs_minfree – minimum percentage of blocks we want to keep free.

fs_rotdelay – number of milliseconds to field an interrupt, used to calculate optimal rotational block placement.

fs_rps – disk revolutions per second.

fs_maxcontig – number of contiguous blocks before rotational delay.

fs_maxbpg – maximum number of blocks a file can use in a cylinder group.

fs_ntrak – the number of tracks in a cylinder.

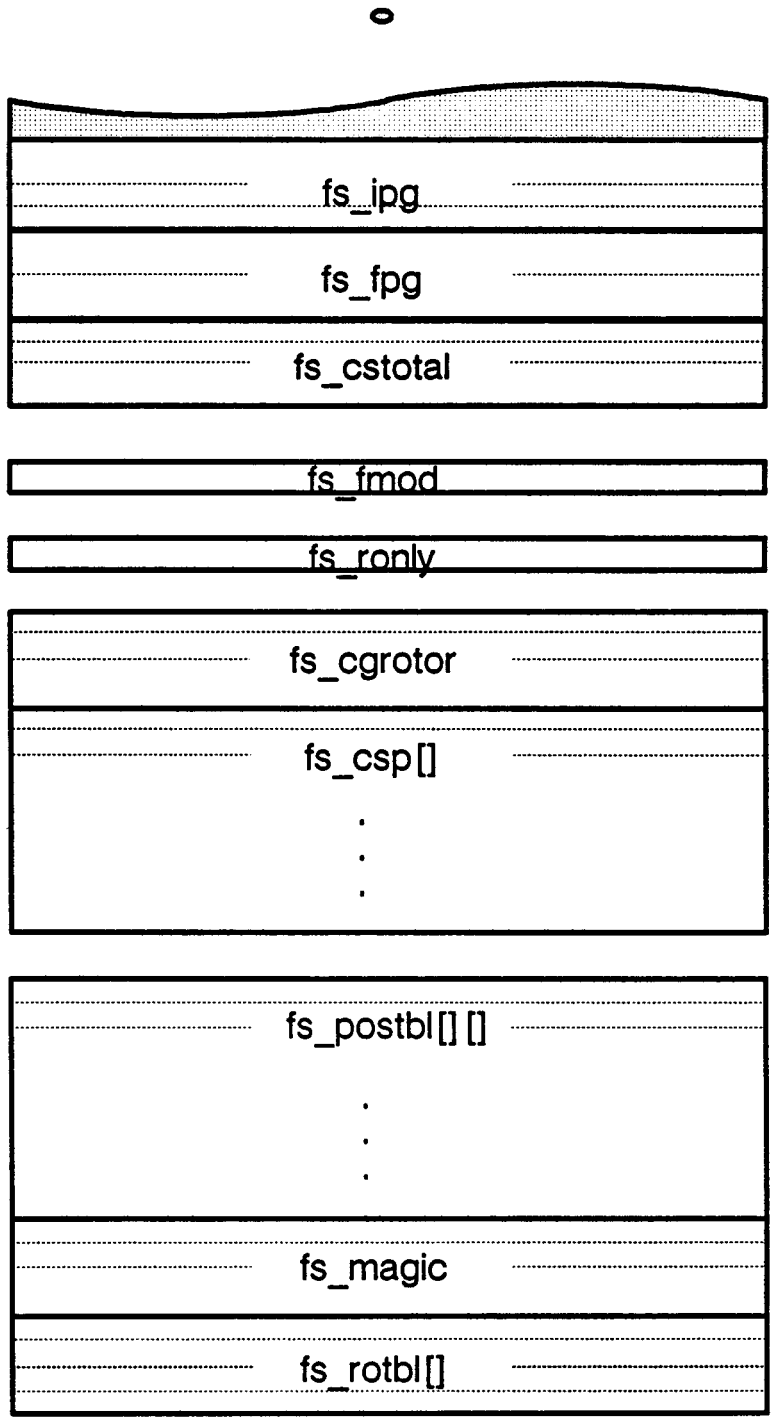
fs_nsect – the number of sectors in a track.

fs_spc – the number of sectors per cylinder.

fs_ncyl – the number of cylinders in the partition (fs).

fs_cpg – the number of cylinders per cylinder group.

The Superblock



The Superblock (cont.)

fs_ipg – the number of inodes per cylinder group.

fs_fpg – the number of fragments per cylinder group.

fs_cstotal – cylinder group summary information totaled for all cylinder groups within this file system.

fs_fmod – a flag to indicate that the superblock has been modified.

fs_ronly – a flag to indicate that this block is mounted read only.

fs_cgrotor – a hint to which cylinder block to start searching when allocating a block or inode.

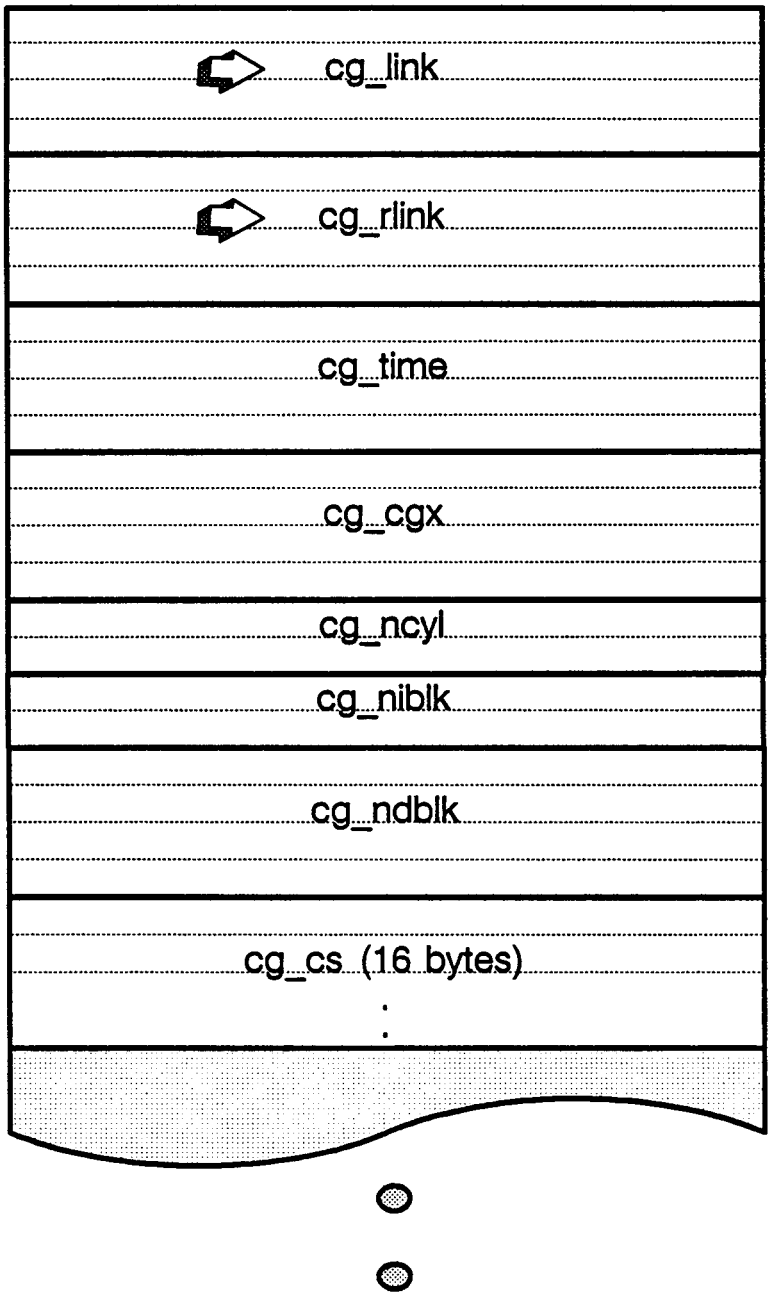
fs_csp[MAXCSBUFS] – pointer to an array of cylinder group summary information kept for each cylinder group in this file system. MAXCSBUFS is the number of buffers needed, there is one per cylinder group so the maximum is 32.

fs_postbl[MAXCPG][NRPOS] – This is the position table which is used for determining the availability of blocks at optimal rotational positions.

fs_magic – the magic number for fs.h (the superblock).

fs_rotbl[1] – a list of blocks available for each rotation.

cg structure



cg structure

The cylinder group parameters are defined when the file system is created with *mkfs*. The maximum size of the group is *MAXCPG* (defined as 32). The default size is 16 cylinders. The maximum number of inodes is *MAXIPG* (2048). The *cg*, *fs* (superblock), and *csum* structures are defined within *fs.h* (Appendix B).

cg_link – forward link for incore cylinder groups.

cg_rlink – reverse link for incore cylinder groups.

cg_time – time the *cg* was last written.

cg_cgx – index number of *cg* within the file system.

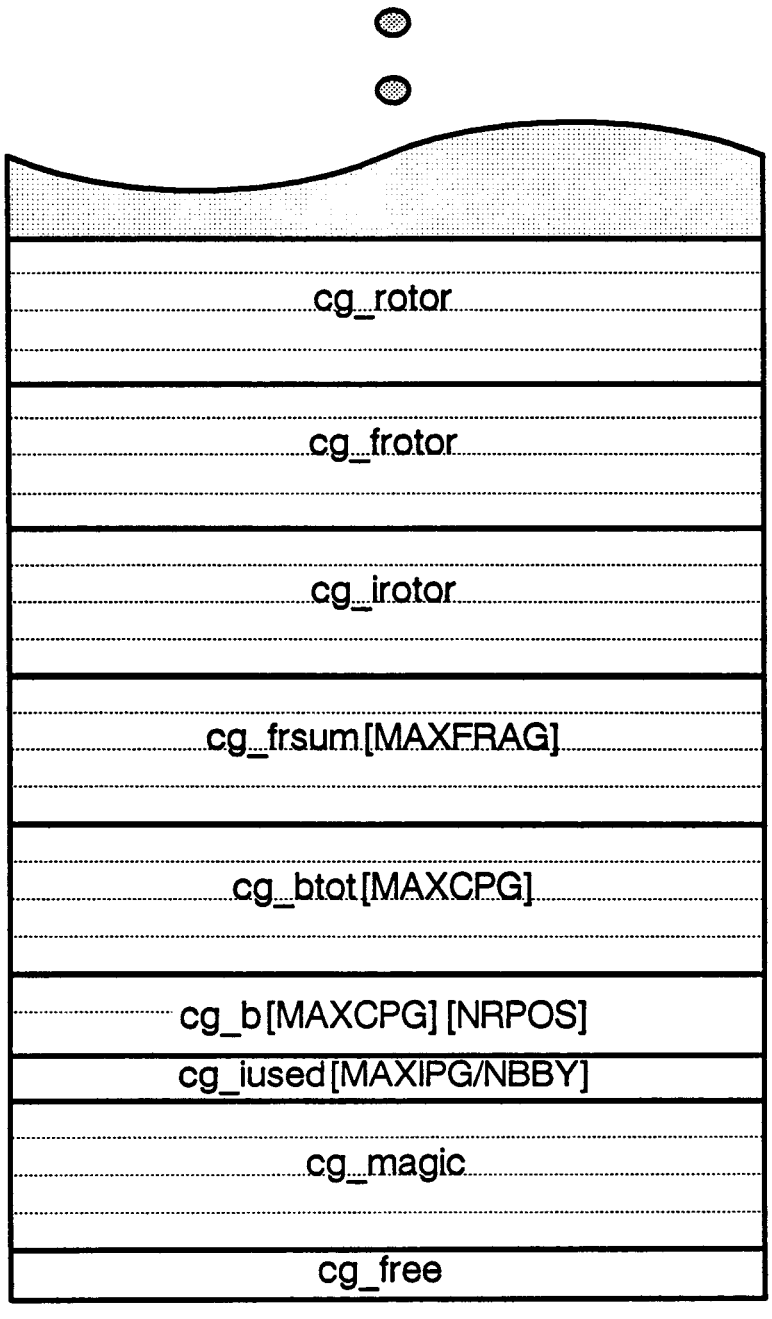
cg_ncyl – total number of cylinders in this cylinder group.

cg_niblk – total number of inodes in this cylinder group.

cg_ndblk – total number of data blocks in this cylinder group.

cg_cs – the cylinder group summary information, see the *csum* structure.

cg structure (cont.)



cg structure (cont.)

cg_rotor – position of last block used within the cg, used by allocation routines.

cg_frotor – position of last fragment used within the cg.

cg_irotor – position of the last inode allocated within the cg (see ialloc()).

cg_frsum[MAXFRAG] – an array that summarizes the count of available fragments of each possible size. MAXFRAG is defined in param.h as 8.

cg_btot[MAXCPG] – an array that summarizes the block totals for each cylinder, where MAXCPG is the maximum number of cylinders per group and defined as 32.

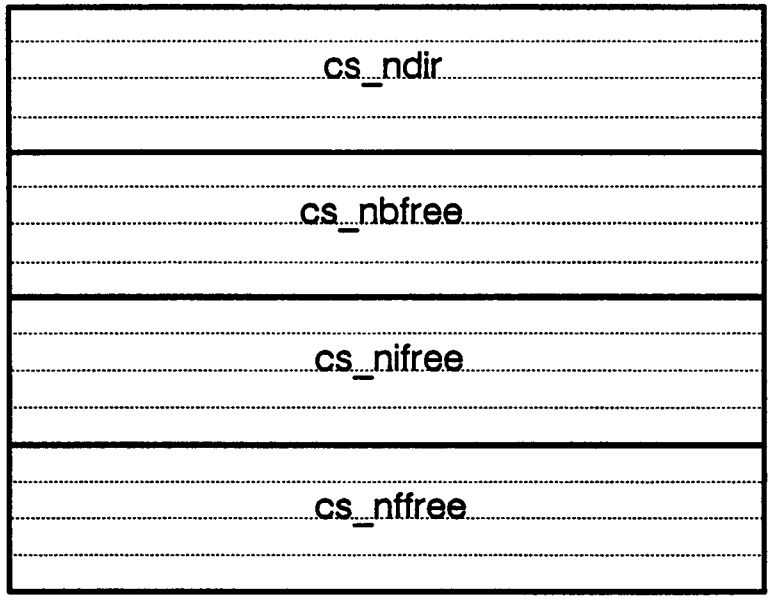
cg_b[MAXCPG][NRPOS] – an array that summarizes the number of blocks available in each rotational position in each cylinder. NRPOS is the number of rotational positions distinguished, defined as 8. Used by the global placement policy routine blkpref( ) to place a block in an optimal position.

cg_iused[MAXIPG/NBBY] – bit map of the inodes that are currently being used in the cylinder group. MAXIPG is the maximum number of inodes per group (2048) and NBBY is the number of bits per byte, defined in param.h as 8.

cg_magic – magic number used to identify the cg structure.

cg_free[1] – the start of the free block map. Each byte represents a block and each bit a fragment. Used by block allocation and deallocation routines.

csum structure



csum structure

There is a cylinder group summary block in each *cg*. There is also a *csum* structure in the superblock (*fs_cstotal*) that has the total of the summary information for all the cylinder groups in the *fs*. Also there is an array of pointers to the *csum* information for each *cg* within the superblock, *fs_csp[MAXCSBUFS].*

cs_ndir – The number of directories in the cylinder group. Used by *dirpref().*

cs_nbfree – The number of free blocks in the cylinder group.

cs_nifree – The number of free inodes in the cylinder group.

cs_nffree – The number of free fragments in the cylinder group.

The *dirlook()* Routine

dirlook() looks for a given name in a directory

It first checks the *dnlc.*

If the name is not there *dirlook()* checks *i_diroff.*

Set up the end of directory search address.

The *dirlook* Routine

The *dirlook()* routine is called with a pointer to the directory's inode structure (*dp*) and the name (*namep*) to locate. It will return a pointer to the inode associated with that name.

After checking the inode mode (*dp->i_mode*) to make sure it's a directory, it checks if the name is in the directory name lookup cache (*dnlc*).

If not in the cache it checks to see if we have already been searching this directory. This is useful if the directory is being accessed sequentially such as with an *ls* command.

The offset that we have already searched to is in the *i_diroff* field of the inode structure. It gets reset to zero if we change directories. If it is set we start to search the directory from the offset. If we get to the end of the directory and have not found the entry we were looking for the directory has to be searched again from the beginning to the offset.

The *dirlook* Routine (cont.)

The search loop reads in each entry sequentially and checks the entry to see:

- if it contains a nonzero inode number.
- it compares the d_namlen field of the directory entry to the name length of the namep.
- If it matches it checks the first character of d_name to the first character of namep.
- If they are the same it *bcmp()* the entire d_name to namep.

As soon as one of these conditions is false we add *d_reclen* to the *i_diroff* to move to the next entry.

If *namep* matches d_name then we call iget() to get a locked pointer to the inode. We then add it to the *dnlc,* and return.

The *dirlook* Routine (cont.)

The dirlook() routine first checks to see if the entry has been deleted by checking for a nonzero inode number. It then checks the number of characters in the name to see if it's the same length as the object we are looking for. It then does a quick check to see that the first character matches what we want. All this is done to avoid the long ascii comparison of the name fields if at all possible.

Global Placement Policies

There is a set of global placement policies used to determine an optimal block position.

The policies for data blocks are governed by *blkpref( )*, taking into consideration:

number of blocks used in the file

rotational layout

free space in the cylinder group

The inode placement policies are determined by *dirmakeinode( )* and *dirpref( )*.

Global Placement Policies

An important part of the 4.2 BSD file system is the placement policies which determine where a directory, inode or data block will be placed physically on disk.

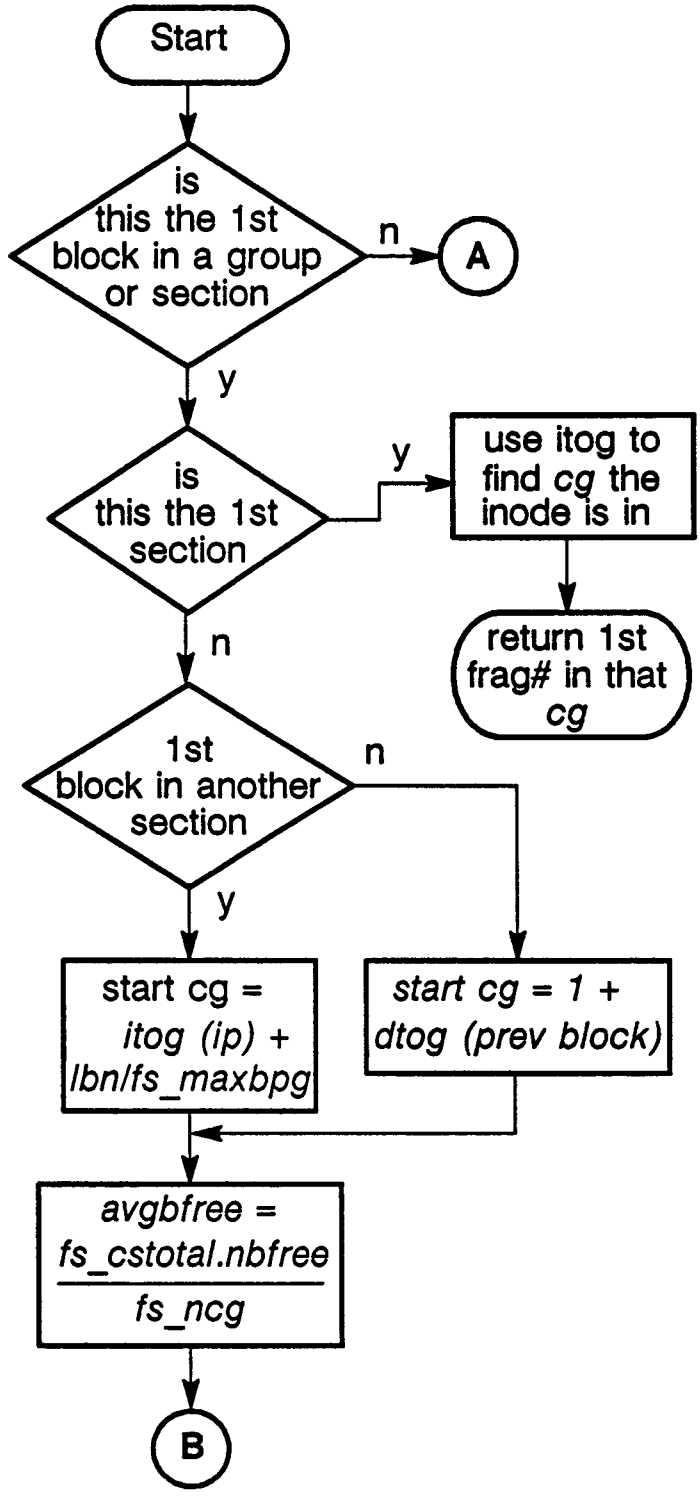
The position of an inode or data block is determined by a set of global and local placement policies.

The global placement policies for data blocks are handled in the *blkpref( )* routine that passes a preferred data block, *bpref,* to the local allocation routines that will determine if that block is actually available. The *blkpref( )* routine also determines when to place blocks in another cylinder group because the file has used its maximum number of data blocks in the cylinder group *(fs_maxbpg)*, or because the *cg* is too full. It also takes into consideration the optimal rotational position to place a block.

The global policies for inodes is much simpler and defined within *dirmakeinode( ).* A preference for a directory inode is given special consideration and specified in the routine *dirpref( ).*

The local placement policies are defined in a set of allocation routines broken out in flowchart form later in this module.

blkpref ( )


Start
is this the 1st block in a group or section
n
A
y
is this the 1st section
y
use itog to find cg the inode is in
return 1st frag# in that cg
n
1st block in another section
n
y
start cg = itog (ip) + lbn/fs_maxbpg
start cg = 1 + dtog (prev block)
avgbfree = fs_cstotal.nbfree / fs_ncg
B

blkpref ( )

The *blkpref( )* routine is used to select the position to place the next block of a file on disk. It is passed a pointer to the inode, *ip*, the logical block number in the file, *lbn*, and the block index into the section, *indx*.

The first section of a file is the portion that uses the direct block pointers in the inode. Each section after that is defined by the data blocks pointed to by an indirect block.

There is a limit to the number of data blocks that can be written to a cylinder group before the remainder of the file has to be written to another cylinder group *(fs_maxbpg)*. This is defined in mkfs to be the number of data blocks that can be referenced with one fragment.

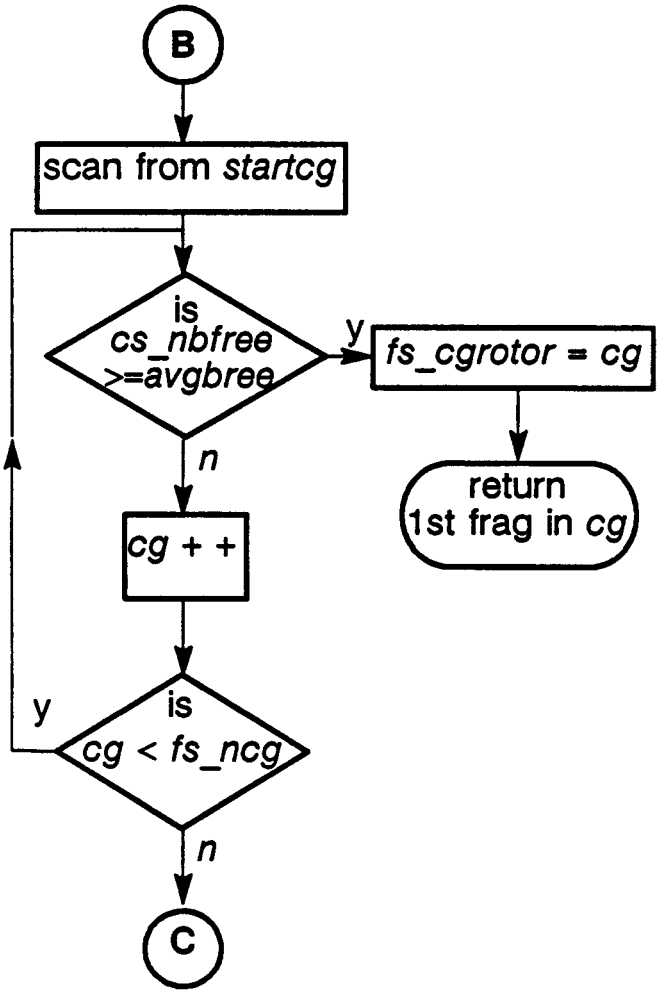
If this is the first block in the file find the *cg* of the inode using *itog()* and return the first block in that cylinder group.

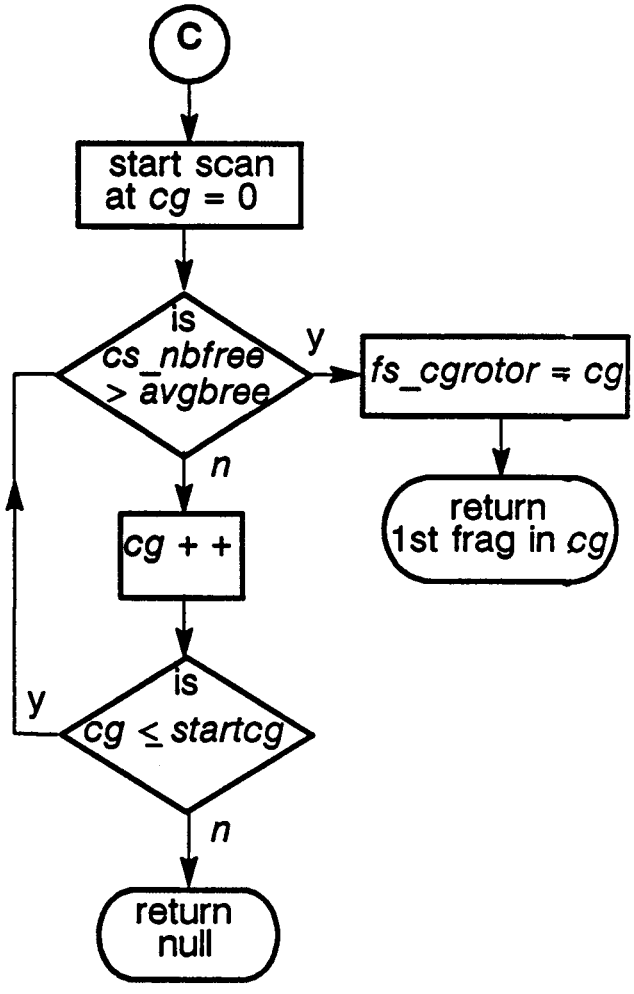
If it is the first block in a section other than the first section, start scanning the *cgs (startcg)*, for a free block at the *cg* past the inode *cg* by *lbn/fs_mbpg*.

If this is the first block in a *cg* but not the first in the section, use *dtog()* to figure out what *cg* was used for the last block written and add one to it for *startcg*.

Calculate the number of free blocks per cg (avgbfree).

blkpref( ) (cont.)





blkpref( ) (cont.)

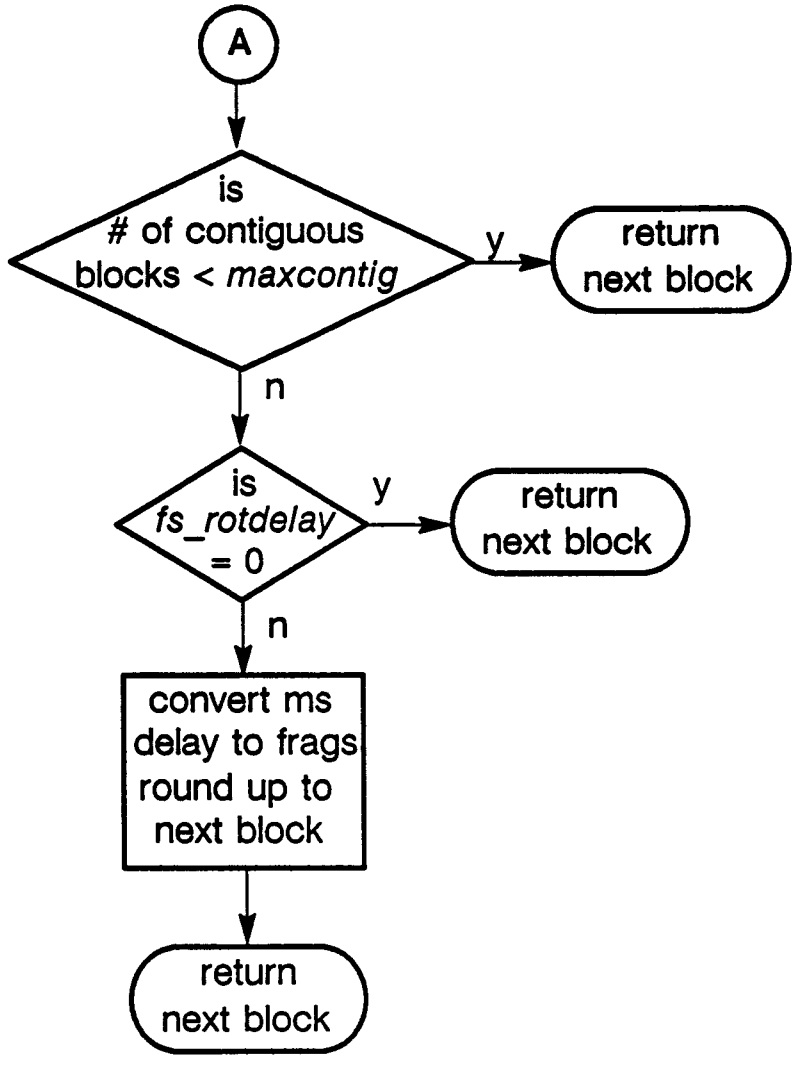
Starting from *startcg* search to the last *cg* in the *fs* to find the first *cg* with *cs.nb_free* greater than *avgbfree*.

Set *fs_cgrotor* =*cg*.

Return the first block in that *cg*.

If we don't find a *cg* from the *startcg* to the last *cg*, start the search from *cg* = 0 and search up to *startcg*. If we still haven't found a *cg* with a greater than average number of free data blocks return NULL.

blkpref( ) (cont.)



blkpref( ) (cont.)

If we are locating a block in the middle of a *cg*, check how many blocks have been laid out contiguously.

If it is less than *fs_maxcontig,* return the next contiguous block.

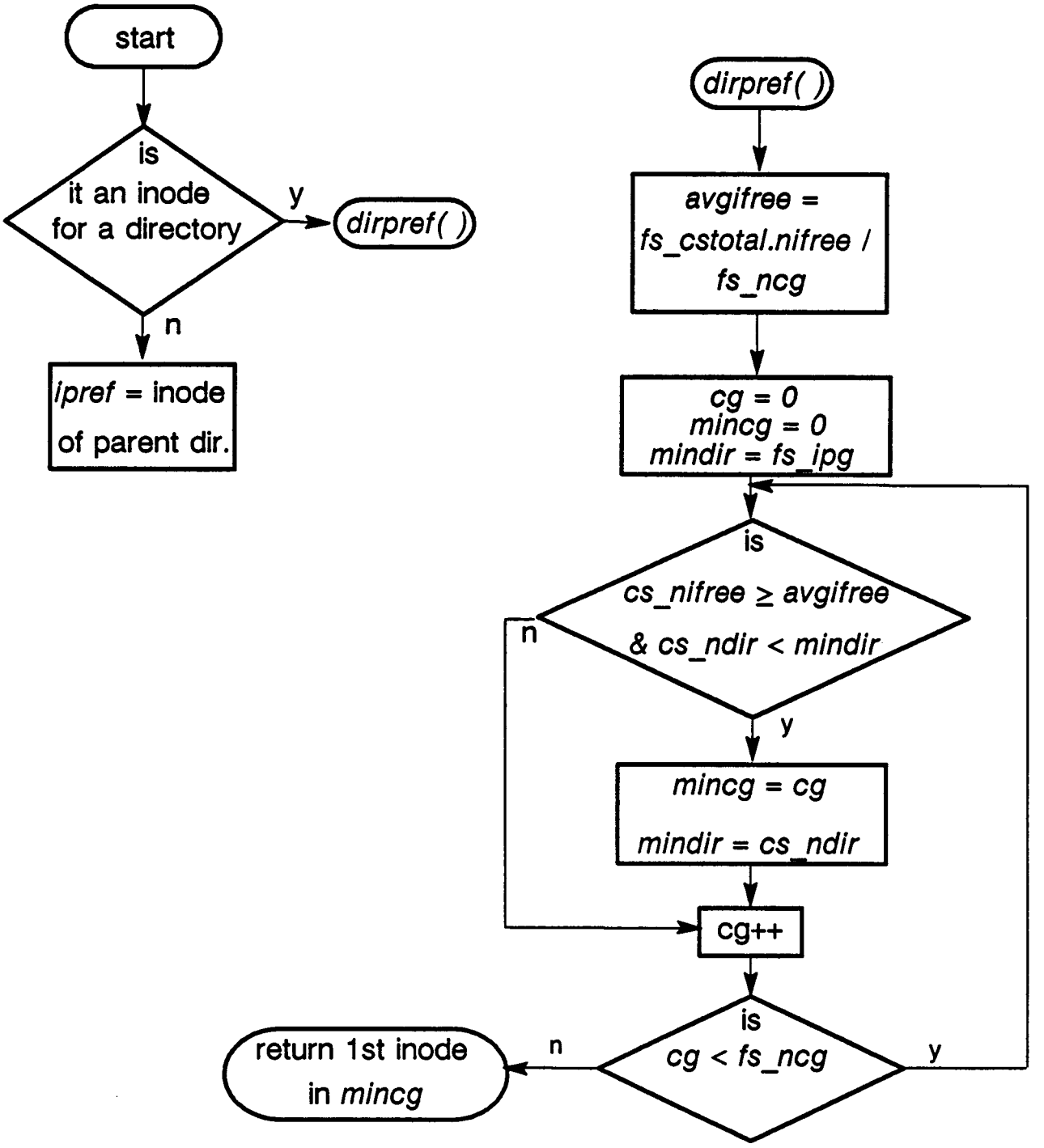
If the number of blocks is greater than fs_maxcontig then calculate the number of frags that would pass in under the heads during fs_rotdelay and round up to the next block number and return that block.

The calculation is as follows:

$$\text{the \# of frags} = \left\lfloor \frac{\text{fs_rotdelay} * \text{fs_rps} * \text{fs_nsect}}{\text{NSPF(fs)} * 1000} \right\rfloor$$

Where NSPF is the number of sectors per fragment in this file system.

Determining *ipref*

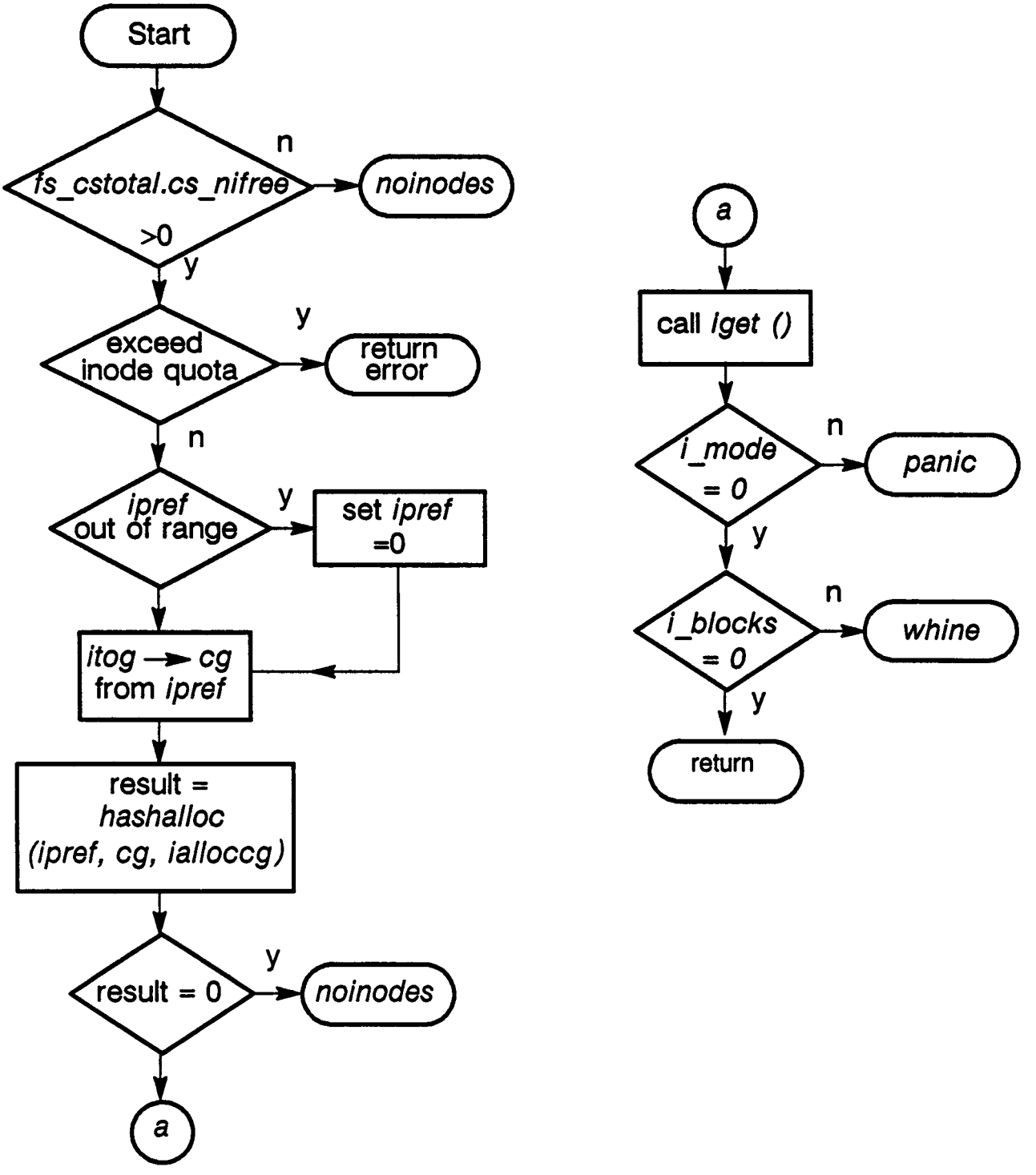


Determining *ipref*

The routine *dirmakeinode( )* checks if the inode is being created for a directory. If it is it calls *dirpref( )*, otherwise *ipref* is just set to the inode of the parent directory.

The routine *dirpref( )* calculates the average number of free inodes per cylinder group *(avgifree)*. It then looks through the cylinder groups for one that has greater than the average number of free inodes, and has the least amount of directories.

ialloc ( )


Start
fs_cstotal.cs_nifree >0
n
noinodes
y
exceed inode quota
y
return error
n
ipref out of range
y
set ipref =0
itog → cg from ipref
result = hashalloc (ipref, cg, ialloccg)
result = 0
y
noinodes
a
a
call iget ()
i_mode = 0
n
panic
y
i_blocks = 0
n
whine
y
return

ialloc

The *iget()* routine allocates a known inode an incore inode if it doesn't already have one.

The *ialloc()* routine assigns an inode to a newly created file. It can be called with a request for a given inode number *(ipref).*

First check in the super block the cylinder group summary information *(fs_cstotal)* to see if any inodes are available on the file system. If not report 'out of inodes' error.

Check to see if the user has exceeded any inode quota that may be set in the users' resource limits.

If the requested inode *(ipref)* is greater than the number of cylinder groups in this file system times the number inodes per cylinder group, set *ipref* to zero.

Calculate the preferred cylinder group from ipref (the preferred inode).

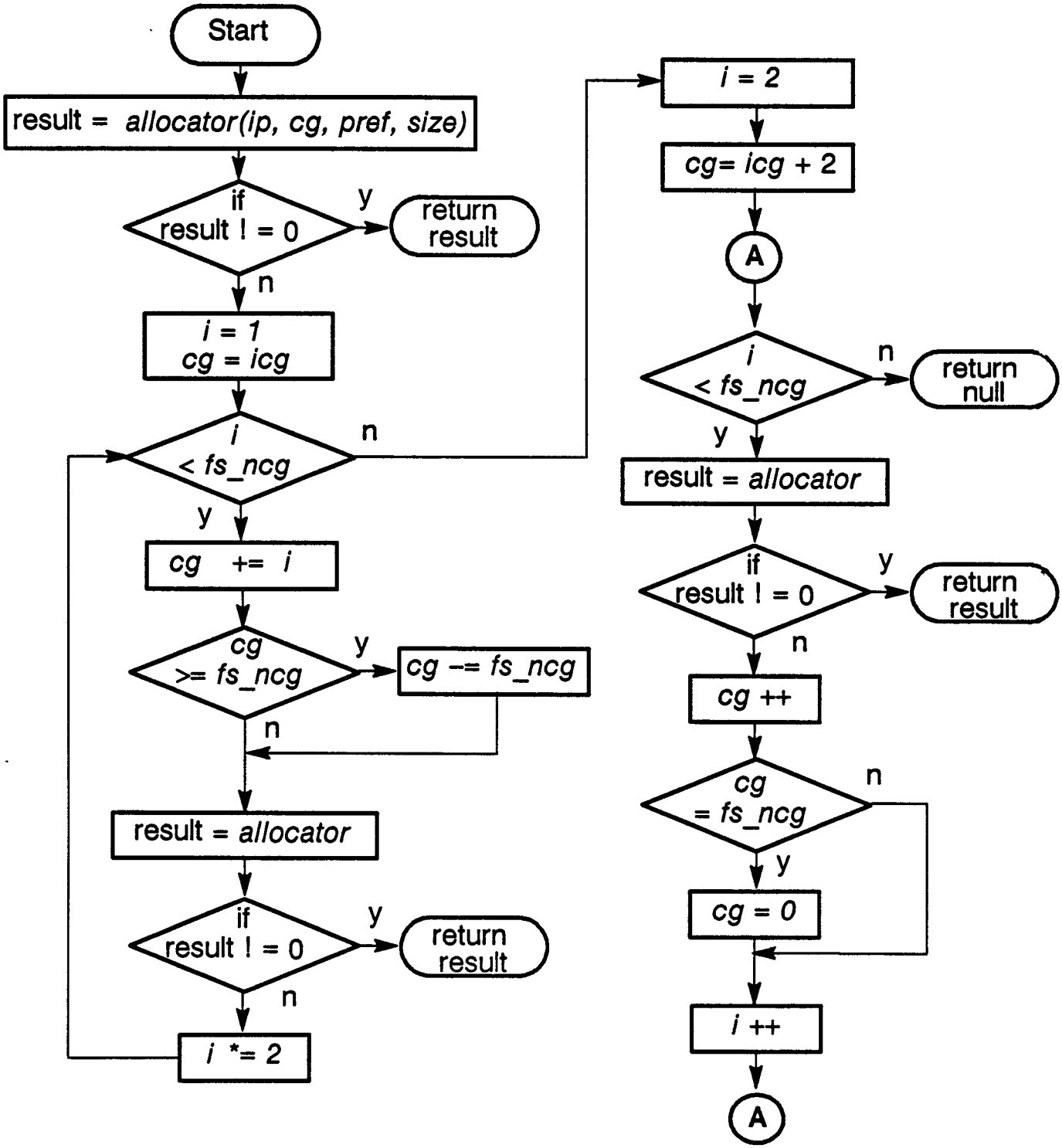
Call *hashalloc* to find a cylinder group with a free inode and pass it *ipref* to try and get that inode.

If no inode is returned, report 'out of inodes' error.

Call *iget()* to lock down an incore inode for the one just allocated.

If inode mode is not zero then it may still be allocated to another file, panic *(ialloc: dupalloc)*. The *i_mode* field should be set to zero in the inode when the reference to the inode goes to zero, and the incore inode is put on the free list in *iinactive().*

hashalloc


Start
result = allocator(ip, cg, pref, size)
if result ! = 0
y
return result
n
i = 1
cg = icg
i < fs_ncg
n
y
cg += i
cg >= fs_ncg
y
cg -= fs_ncg
n
result = allocator
if result ! = 0
y
return result
n
i *= 2
i = 2
cg= icg + 2
A
i < fs_ncg
n
return null
y
result = allocator
if result ! = 0
y
return result
n
cg ++
cg = fs_ncg
n
y
cg = 0
i ++
A

hashalloc

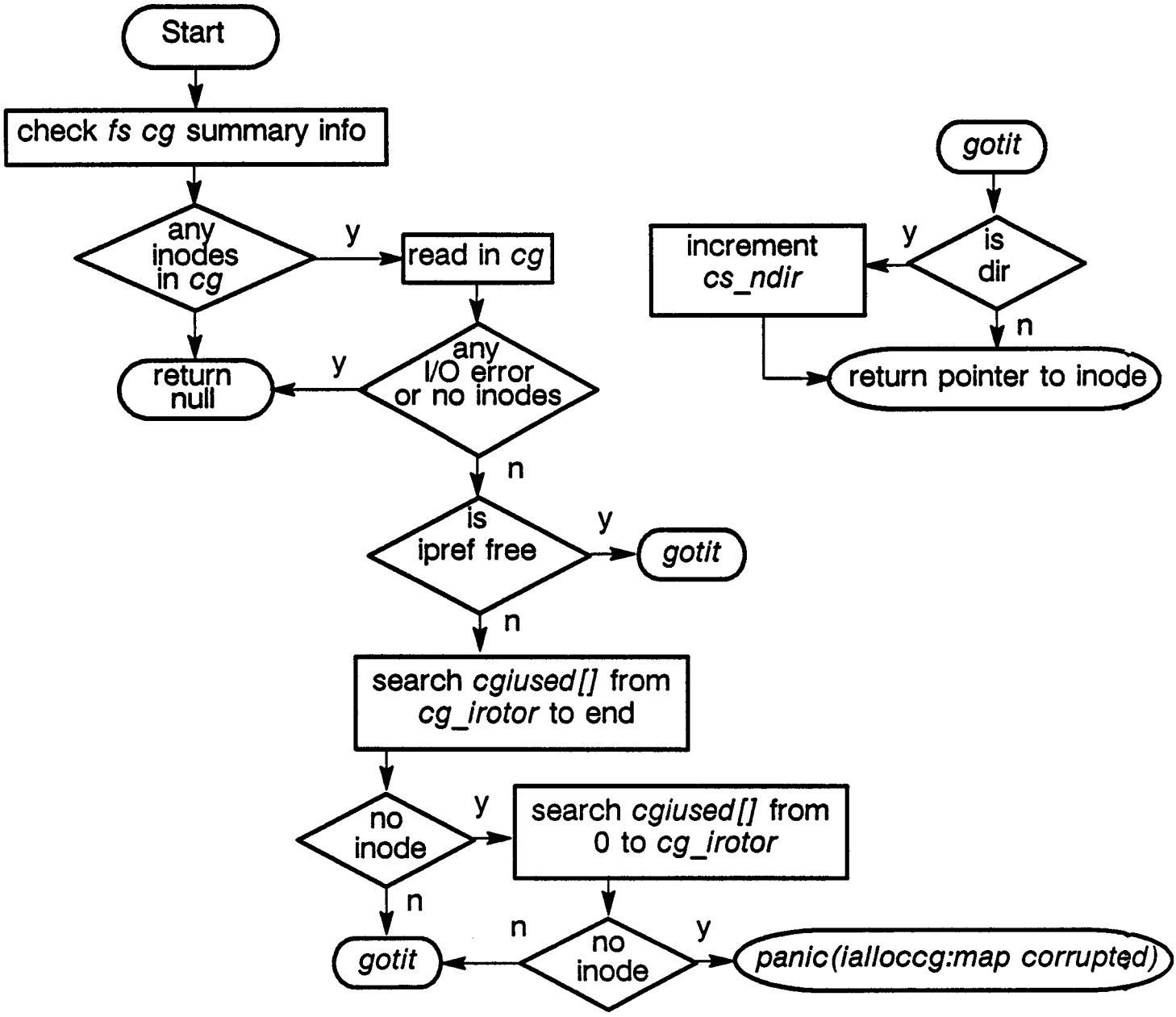
The *hashalloc()* routine gets called by different allocation routines to locate a free data block or inode.

It will first check the requested cylinder group for the requested inode or datablock. If looking for an inode, it will call *ialloccg()* to try and allocate the requested inode *(ipref)*. If looking for a data block *alloccg()* is called with a preferred block number *(bpref)*.

If the preferred data block or inode is not available a quadratic rehash is used to attempt to find a data block or inode as quickly as possible.

If the quadratic rehash fails to find a free data block or inode, then start a sequential search with the preferred cylinder group +2 through all the cylinder groups for a free inode or data block.

ialloccg()


Start
check fs cg summary info
any inodes in cg
y
read in cg
return null
y
any I/O error or no inodes
n
is ipref free
y
gotit
n
search cgiused[] from cg_irotor to end
no inode
y
search cgiused[] from 0 to cg_irotor
n
gotit
n
no inode
y
panic(ialloccg:map corrupted)
gotit
is dir
y
increment cs_ndir
n
return pointer to inode

ialloccg()

This routine was called by *hashalloc().*

The *cg* (cylinder group) structure keeps track of what inodes have been used in the group through a bit map *cg_iused[MAXIPG / NBBY]. MAXIPG* is the maximum number of inodes per group and is set at 2048. *NBBY* is the number of bits per byte, 8.

Check the super block's array of per cylinder group summary information for available inodes in this *cg.*

Read in the *cg* structure via *bread().*

If there is an I/O error, or the magic number indicates it's not a *cg*, or the cylinder group summary *(csum)* information indicates there are no free inodes, return NULL for the inode number.

Check the *cg_iused* bit map to see if the preferred inode *ipref* is available.

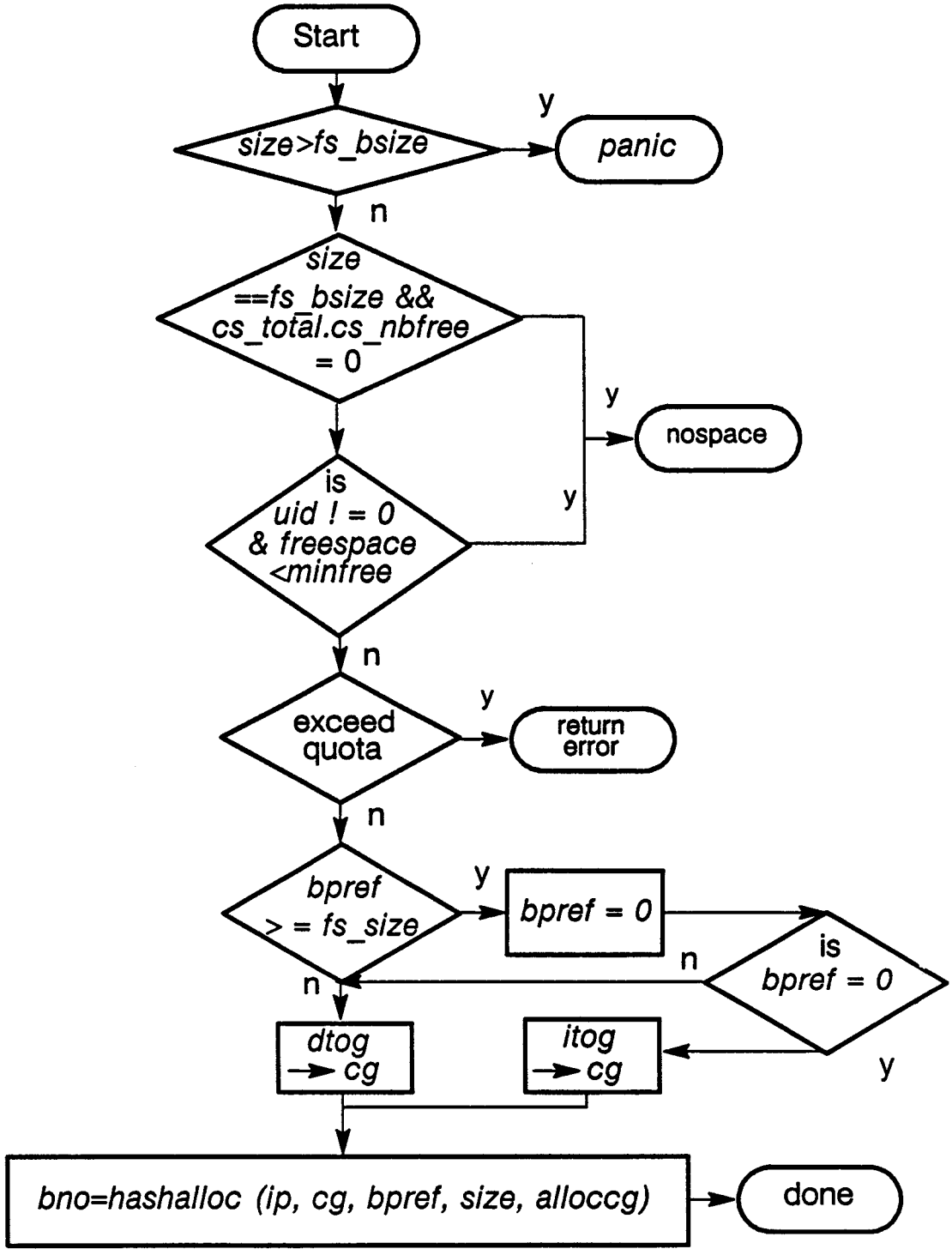
If it's not, start searching the *cg_iused* bit map at the position we found the last inode *(cg_irotor).* Search the map from the *cg_irotor* to the end, and then from the beginning to *cg_irotor*, if necessary to find a free inode.

If we don't find one, panic('ialloccg: map corrupted').

When we find an inode we decrement the number of inodes that are free in the cylinder group summary structures and mark the superblock as modified. If the inode was for a directory, increment the number of directories *(ndir)* in the cg summary structures.

Return the address of the inode to *hashalloc().*

alloc


Start
size>fs_bsize
y
panic
n
size ==fs_bsize && cs_total.cs_nbfree = 0
y
nospace
is uid ! = 0 & freespace <minfree
y
n
exceed quota
y
return error
n
bpref > = fs_size
y
bpref = 0
is bpref = 0
n
n
y
dtog → cg
itog → cg
bno=hashalloc (ip, cg, bpref, size, alloccg)
done

alloc

The *alloc()* routine is used to allocate a disk block or fragment off a disk block to a file system object. It is passed a pointer to the inode, *ip*, a size of the area to map, *size*, and a preferred block number *bpref*. It returns a pointer to an allocated data block, *bnp* (block number pointer).

The routine first checks the superblock to make sure the size is less than or equal to the the size of a file system block and a multiple of the fragment size. If it isn't then panic("alloc: badsize").

It then checks if there's available space in the file system to allocate *size*. To do this check *minfree* and *fs_cstotal.cs_nbfree*.

It then checks any possible user quota to see if the quota for number of disk blocks will be exceeded.

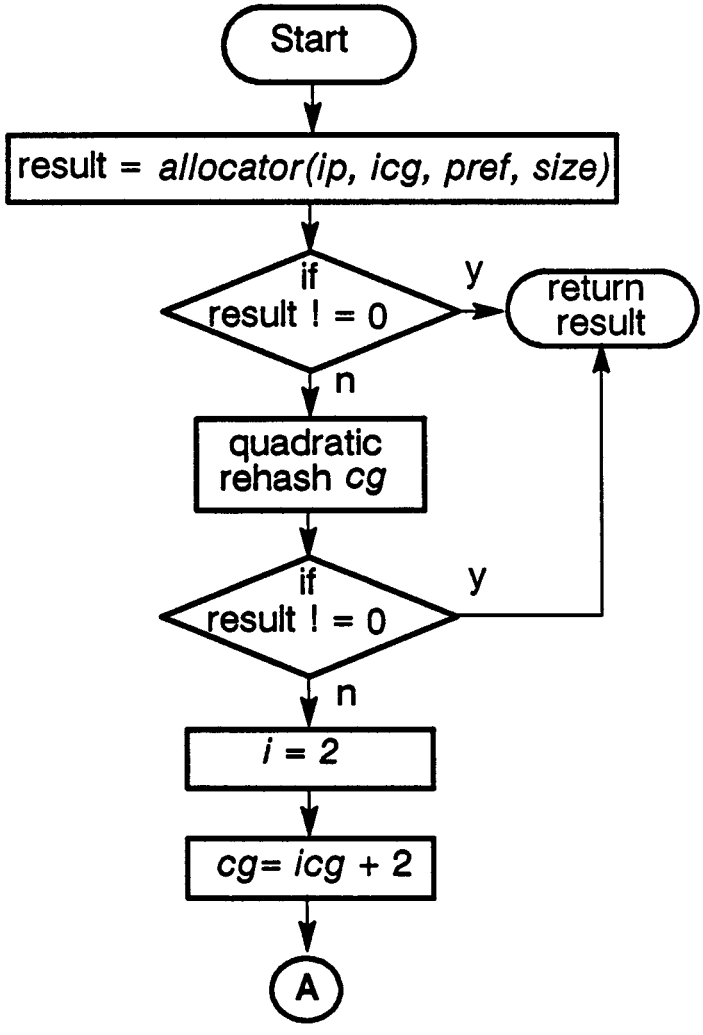
If the bpref is out of range set *bpref* to zero.

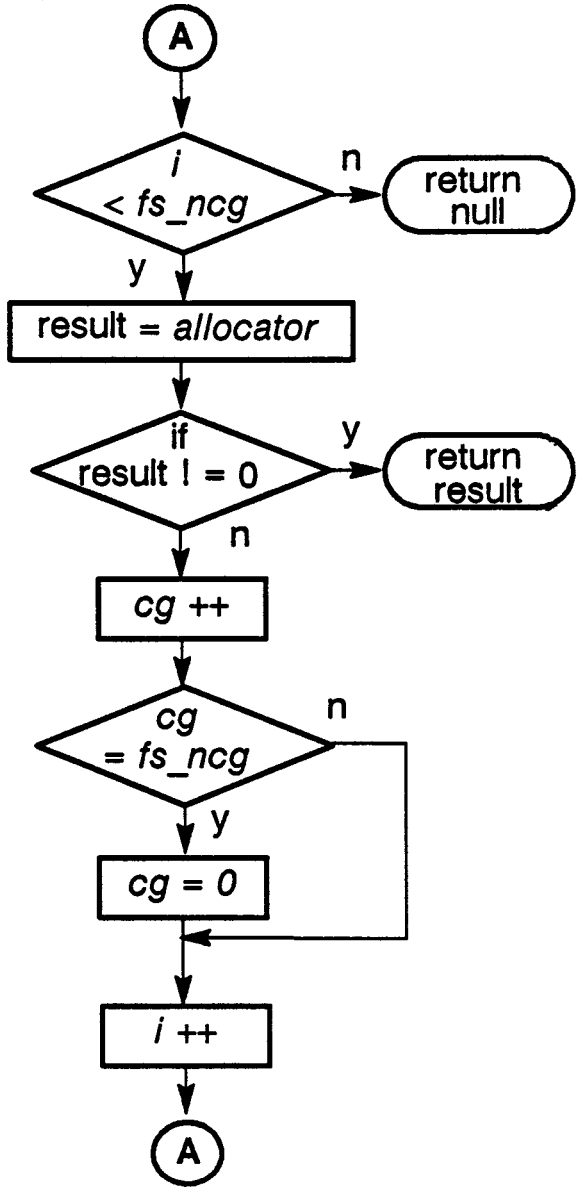
If no preferred block is specified *(bpref = 0)* then use *itog* to calculate a preferred cylinder group, from the inode.

Otherwise use *dtog* to calculate a preferred cylinder group, cg, from the data block address *(bpref)*.

Call *hashalloc()* with a pointer to the inode, *ip, cg, bpref, size* and a pointer to the routine *alloccg()*. If successful hashalloc() will return a block number.

hashalloc


Start
result = allocator(ip, icg, pref, size)
if result ! = 0
y
return result
n
quadratic rehash cg
if result ! = 0
y
n
i = 2
cg= icg + 2
A


A
i < fs_ncg
n
return null
y
result = allocator
if result ! = 0
y
return result
n
cg ++
cg = fs_ncg
n
y
cg = 0
i ++
A

hashalloc

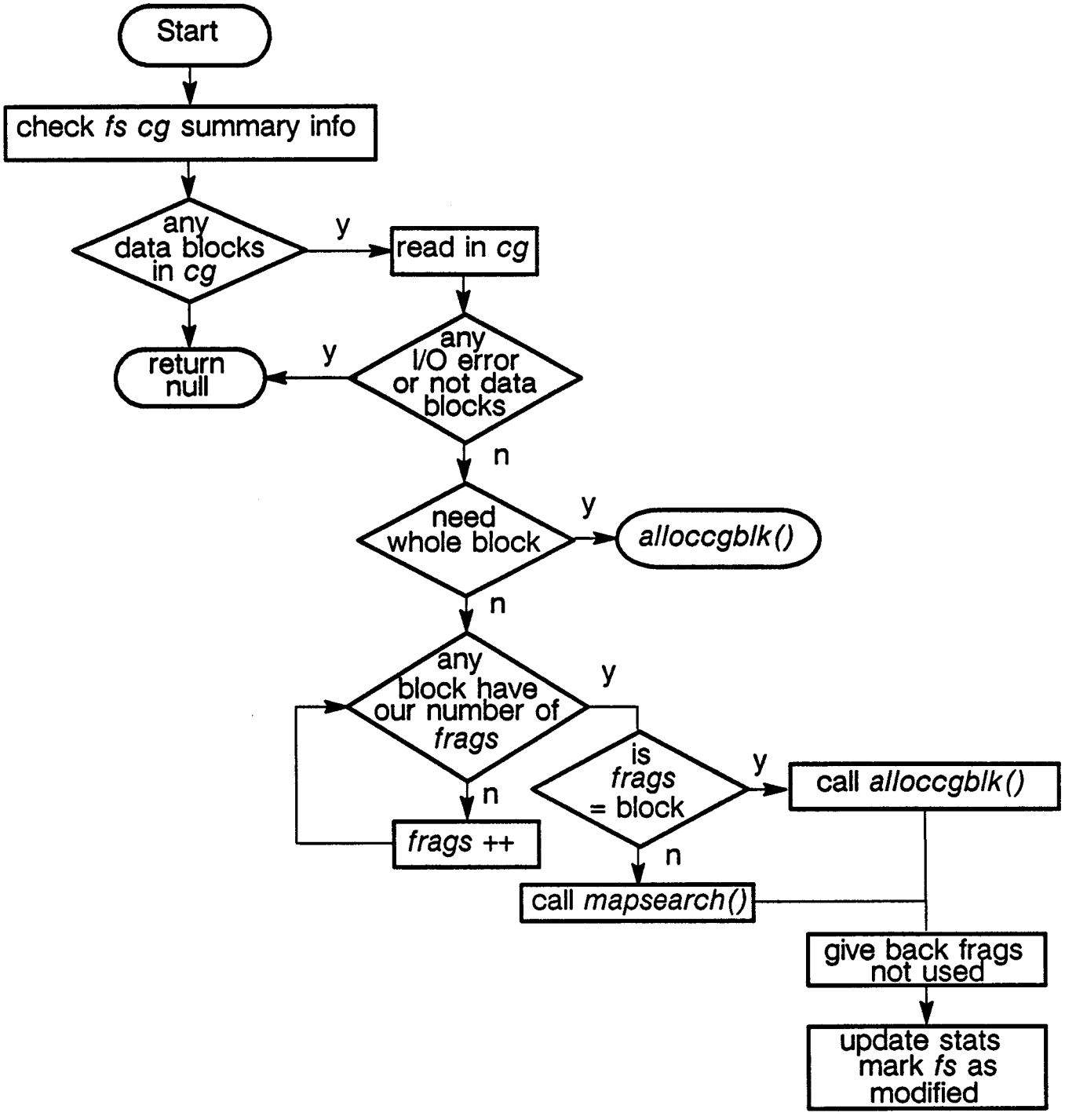
The *hashalloc()* routine gets called by different allocation routines to locate a free data block or inode.

It will first check the requested cylinder group for the requested inode or datablock. If looking for an inode it will call *ialloccg()*, to try and allocate the requested inode *(ipref)*. If looking for a data block *alloccg()* is called with a preferred block number *(bpref)*.

If the preferred data block or inode is not available a quadratic rehash is used to attempt to find a data block or inode as quickly as possible.

If the quadratic rehash fails to find a free data block or inode, then start a sequential search with the preferred cylinder group +2 through all the cylinder groups for a free inode or data block.

alloccg


Start
check fs cg summary info
any data blocks in cg
y
read in cg
any I/O error or not data blocks
y
return null
n
need whole block
y
alloccgblk()
n
any block have our number of frags
y
is frags = block
y
call alloccgblk()
n
frags ++
n
call mapsearch()
give back frags not used
update stats mark fs as modified

alloccg

The *cg* (cylinder group) structure keeps track of the positions of free data blocks in the group through a bit map *cg_b[MAXCPG][NRPOS]*. *MAXCPG* is the maximum number of cylinders per group and is set at 32. *NRPOS* is the number of rotational positions distinguished per track, 8.

Check the super block's array of per cylinder group summary information for available data blocks in this *cg*.

Read in the *cg* structure via *bread()*.

If there is a I/O error, or the magic number indicates it's not a *cg*, or the cylinder group summary *(csum)* information indicates there's no free data blocks, return NULL.

If an entire block is needed call *alloccgblk()* to allocate a block in this cylinder group, if possible.

If looking for a fragment the array of fragment summary information *(cg_frsum[MAXFRAG])* is used to keep a count of how many fragments of each size are available. *MAXFRAG* is equal to the maximum number of fragments in a block, 8. Start the search at the number of fragments we need *frags*. If cg_frsum[frags] is 0, check if any fragments of size frags + 1 are available. If cg_frsum[frags+1] is 0, keep checking up to a fragment size of *fs_frag*, at which point we are looking at whole blocks. At this time call *alloccgblk()* to get a whole block and then mark the unused fragments as free.

If there were any fragments that would accommodate the size we needed, call *mapsearch()* to locate the block number, *bno*, that the fragment exists in.

In either case the statistics *cg_cs.cs_nffree, fs_cstotal.cs_nffree*, and *cg_frsum[]* have to updated appropriately. Also the superblock is marked as modified.

alloccgblk

The *alloccgblk()* routine is called to allocate an entire block. It will attempt to locate a block in the following manner:

- First it will try to allocate the block requested by *bpref().*
- If that is not free it will attempt to allocate a rotationally optimal block in the same cylinder.
- If that fails it will allocate the next available block in the cylinder group, indicated by *cg_frotor.*

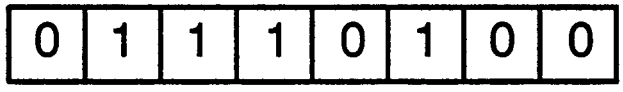
alloccgblk

A rotationally optimal positioned block is located by using the array *cg_b[MAXCPG][NRPOS]*. This array keeps track of the free blocks at each rotational position (*NRPOS*), for each cylinder in the group (up to *MAXCPG*).

To find a free block in the cylinder group the *mapsearch()* routine is called (next page).

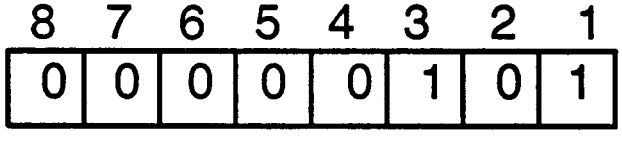
mapsearch()

bit pattern in free block map



fragment table

size
fragments
available



mapsearch()

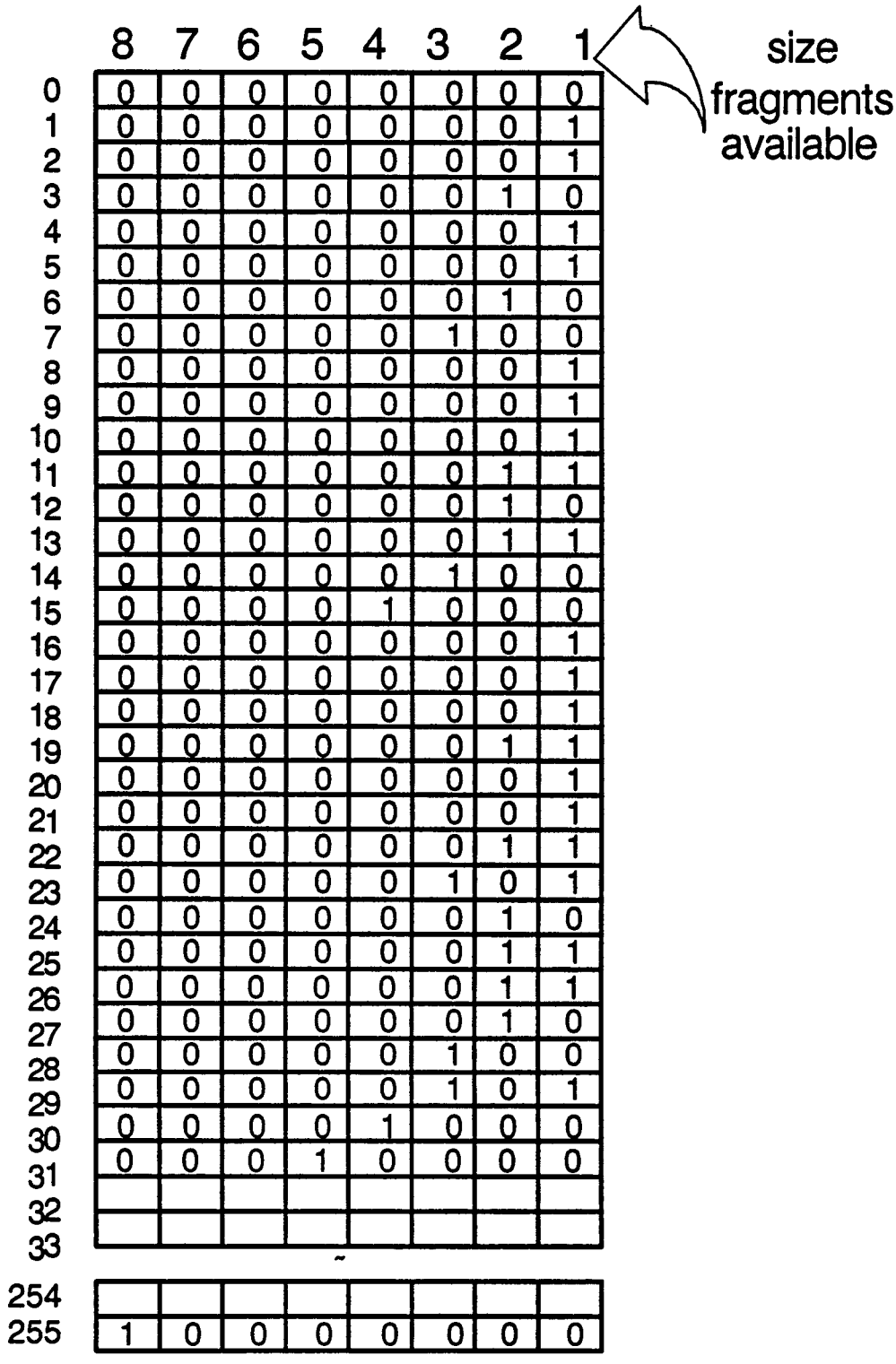
The *mapsearch()* routine works by looking at the entries in the free block map. Each entry is 8 bits long and each set bit corresponds to a fragment that is available.

For fast lookup the entry is also used as an index into an array that indicates for that bit pattern what size segments are available.

For example if the bit pattern in the free block map is 01110100, this indicates that one 1 fragment size and one 3 fragment size block is available. This entry (01110100 = 116) is used as an index into a fragment table that has a bit set for every size fragment available.

Now we can just 'logically and' the bits in the fragment table entry with a byte in which the bit corresponding to the size block we need is set.

fragment table



fragment table

The fragment table is represented on the opposite page. Notice if a bit is set it would indicate how many contiguous fragments are available within that block. Suppose the system is searching for a block on disk that is three fragments long. The system would create a mask of 100 to indicate it is looking for a block of three fragments.

The system would use the cg_free array entries as indexes into the fragment table. if the first several entrees in the array looked like this in binary:

00000100
00000000
00001100
00001101
00011011
00100000
00000111

The system would use the mask (100) to determine, with a 'logical and' operation, if the appropriate number of fragments were available. The first entry would index to entry four in the fragment table 'logically and' this with the mask and get a result of 0 indicating this data block would not work. The next entry in the cg_free map would index to the first entry in the fragment table which indicates that no fragments are available.

This would continue until the system gets to the last entry shown (00000111), which indexes to entry 7 in the fragment table. When 'anded' with the mask (100) it will indicate that there are three fragments available in this block and it will be used. Since the three fragments are used the entry in the *cg_free map* will be updated to contain all 0s. Future references to this entry in *cg_free* will index to entry 0 of the fragment map.

Module 6
Input/Output Management

Objectives: Upon completion of this module, the student will be able to:

- Describe the various interfaces to the device driver routines.
- Explain the differences between block and character devices.
- Explain the function of the major and minor device numbers.
- Explain the expansion of the different I/O system calls using the SunOS Virtual Memory System.

Reference information: Course Guide

Introduction

In this module we will cover:

- Device Files.
- The I/O path in the SunOS.
- Device driver routines.
- The character I/O interface.
- The block I/O interface.
- Input/Output Operations on regular files.

Introduction

The emphasis of this module is on the input/output operations that are done on a regular file using the *read( ), readv( ), write( ) and writev( )* system calls.

Types of Files

The SunOS Supports Many Types of Files:

- Regular Files
- Directory Files
- Symbolic Link Files
- Pipe Files
- Special or Device Files

Types of Files

The SunOS supports a number of different types of files including regular, directory files, pipe files, symbolic link files and device files. In addition, the SunOS can support different file system types which can introduce still more file types to the system.

Device Files

- Device Independence.
- Device Drivers.
- Configurable Kernel.

Device Files

Device Independence – One of the major features of the SunOS is providing application programs an independent view of the I/O hardware. This allows user processes to manipulate device or "special" files just like regular files. By making system calls, user processes can open, close, read from device files, or write to device files.

Device Drivers – Once the system call carries process execution into the SunOS kernel, however, it becomes clear why they are called "special" files. The kernel handles different types of files differently and translates the various system calls to device files to corresponding calls to their device drivers. It is this set of routines that makes it transparent to the user the differences between devices and regular files.

Configurable Kernel – The SunOS kernel is configurable which means that device drivers can be added to the system by rebuilding the kernel even if source code is not available. On Sun386i systems, the loadable driver capability makes it possible to add a driver without rebuilding or rebooting the system.

Types of Devices

- Co–processors.
- Disks and Tapes.
- Network Interfaces.
- SCSI Devices.
- Serial Communication Multiplexors.
- General DMA Devices.
- Programmed I/O Devices.
- Frame Buffers and Other Memory–mapped Devices.
- Pseudo Devices.

Types of Devices

Devices may be classified into the following types:

- Co-processors.
- Disks and Tapes.
- Network Interfaces.
- SCSI Devices.
- Serial Communication Multiplexors.
- General DMA Devices.
- Programmed I/O Devices.
- Frame Buffers and Other Memory-mapped Devices.
- Pseudo Devices.

SunOS Bus Structures

The SunOS Provides Device Driver Support For:

- Multibus Machines
- VMEbus Machines
- ATbus Machines
- SBus Machines

SunOS Bus Structures

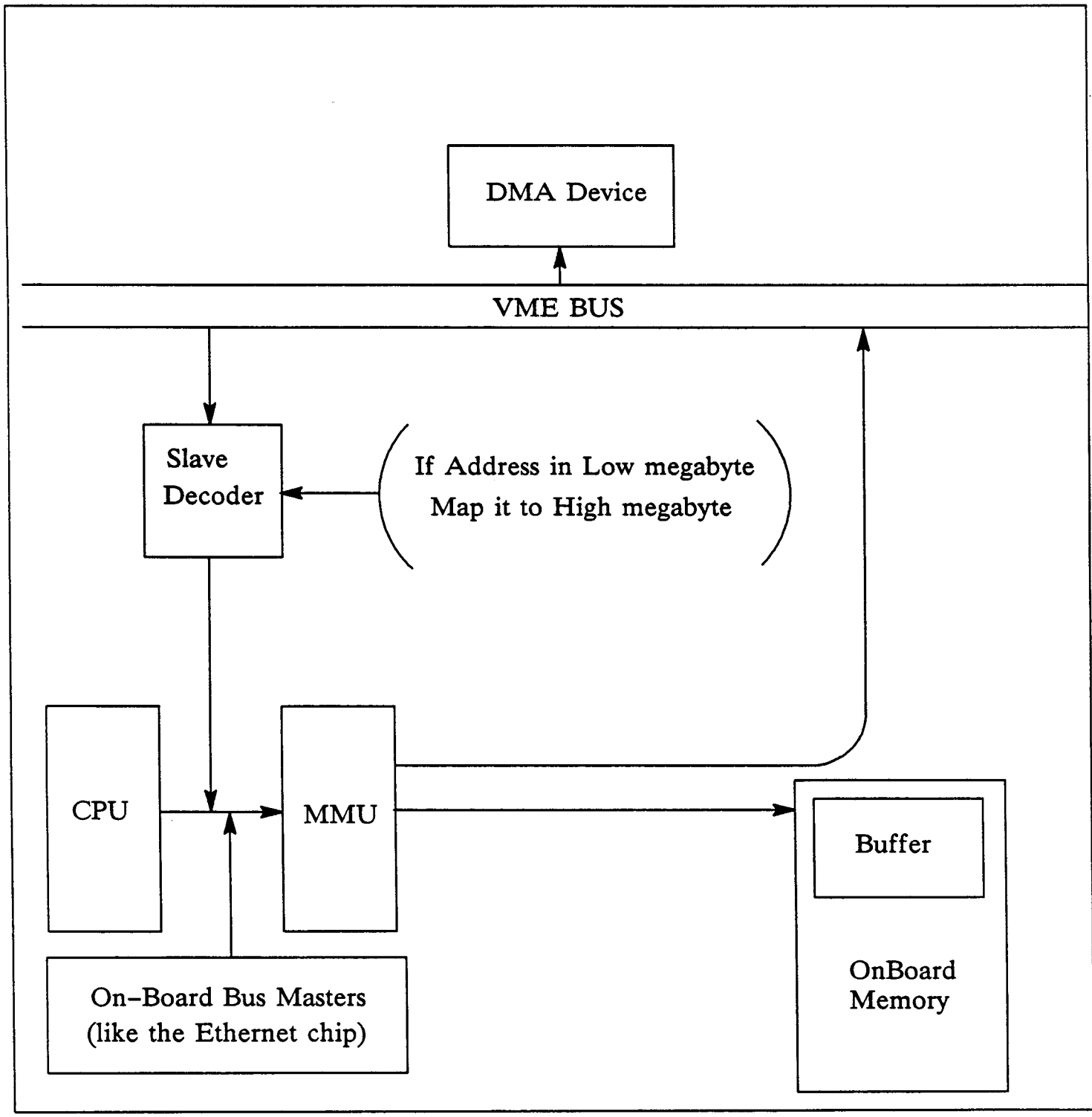
The SunOS supports a variety of hardware bus structures including Multibus, VMEbus, and the ATbus.

The Multibus was originally designed for processors that support two separate data spaces, one for storing in or fetching data from memory and another for transferring data to or from peripheral devices. The Sun Multibus system, however provides support for four address spaces corresponding to the four types of memory namely: On–Board Memory, On–Board I/O Space, Multibus Memory, and Multibus Space.

VMEbus machine architecture is generally more complex than Multibus machine architecture as it makes no distinction between I/O space and Memory space. It supports multiple address spaces for reasons of costs and flexibility.

The Sun386i uses the ATbus which supports the machine's capability of having any memory instruction to access any I/O port located in memory space for added programming flexibility.

DVMA


DMA Device
VME BUS
Slave Decoder
If Address in Low megabyte
Map it to High megabyte
CPU
MMU
Buffer
OnBoard Memory
On-Board Bus Masters
(like the Ethernet chip)

DVMA

Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun Memory Management Unit to allow devices on the Main Bus (either a VMEbus or a Multibus) to perform DMA directly to Sun processor memory. It also allows Main Bus master devices to do DMA directly to main Bus slaves without the extra step of going through the processor memory. DVMA works by ensuring that the addresses used by devices are processed by the MMU, just as if they were virtual addresses generated by the CPU. This allows the system to provide the same memory protection and mapping facilities to DMA devices as it does to the system CPU (and thus to programs).

Character vs. Block Device Drivers

Devices "appear" as *special* files that reside (by convention) in the */dev* directory. They are created with the *mknod(1)* command. A long listing of this directory might show:

Type	Permissions	links	owner	major #	minor #	date		name
c	rw--w--w-	1	henry	0,	0	Feb 21	09:45	console
b	rw-------	1	root	3,	0	Feb 21	09:45	xy0a
c	rw-r--r--	1	root	3,	1	Feb 21	09:45	kmem
c	rw-r--r--	1	root	3,	0	Feb 21	09:45	mem
c	rw-------	1	root	9,	0	Feb 21	09:45	rxy0a
c	rw-rw-rw-	1	root	13,	0	Feb 21	09:45	mouse
				.				
				.				
				.				

Block Devices are those that can contain a file system. Character Devices are all others.

The Major Device Number is used as an index into either a Block (*bdevsw*) or Character (*cdevsw*) Device switch table. These tables contain pointers to device driver functions. They are found in the *conf.c* file in the configuration directory.

Character vs. Block Device Drivers

The classification whether a device file falls under the block or character category is not dependent on the hardware itself, but rather on whether the device will be used to hold a file system or not. It is quite common that the same piece of hardware may have two interfaces: a block interface for holding a file system, and a character(or raw) interface for loading or unloading massive amounts of data using physical I/O. Anything that is not classified as a block device is a character device, which means that there usually will be a lot more character devices in the system than block devices.

Device Driver Routines

By convention, a device is given a code(or name) which prefixes all Device Driver function names. Device Driver **routines** may be categorized into three groups:

- **Auto Configuration Process** (boot time)
  xx*probe*() xx*attach*() xx*slave*() xxidentify()

- **Top–half** (System call request)
  xx*open*() xx*close*() xx*read*() xx*write*() xx*ioctl*()
  xx*mmap*() xx*select*() xxstrategy() xxdump()
  xxpsizel()

- **Bottom–half** (interrupt time)
  xx*poll*() xx*intr*()

Drivers commonly have one or more of these routines missing from their interface.

Device Driver Routines

Drivers commonly have one or more of these routines missing from their interface.

In this regard the dummy functions *nodev* and *nulldev* are used which return the error *NODEV* and *success (0)*, respectively.

Character Driver Top Half Interface

The top–half routines interface to the kernel via the character device switch table.

```
struct cdevsw {
    int (*d_open)();          /* driver's open function */
    int (*d_close)();         /* driver's close function */
    int (*d_read)();          /* driver's read function */
    int (*d_write)();         /* driver's write function */
    int (*d_ioctl)();         /* driver's control function */
    int (*d_reset)();         /* reset function – not used in class*/
    int (*d_select)();        /* driver's select function */
    int (*d_mmap)();          /* driver's mmap function */
    struct streamtab *d_str;  /* entry point for streams drivers */
};
```

The file *conf.c* initializes an array of cdevsw structures. Each *cdevsw* structure provides entry points to one device driver's top–half functions. The top–half functions are called via system calls made from user programs.

This definition is the SunOS 4.0 definition of struct *cdevsw*; the pre–4.0 definition is slightly different. Look at *conf.h* in the header file sub–directory of the configuration directory for the system you are building.

Character Driver Top Half Interface

xxopen( ) – This routine is called in response to the *open( )* system call and prepares the device for the I/O operation that is going to take place. The *open( )* routine performs a lot of checks like making sure that the device is connected, it is on–line, and it is error–free.

xxclose( ) – This routine is called on the last *close( )* to the device and performs the reverse of what *open( )* does. If *open( )* allocated certain structures that were used by the drivers, then *close( )* usually deallocates these and resets whatever flags were set in the opening.

xxread( ) – This routine is called when the *read( )* system call is issued for the device file. For raw devices, this routine may just call the *physio( )* routine which is passed the appropriate parameters which includes the block device driver *strategy( )* routine.

xxwrite( ) – Called in response to the *write( )* to the device file and similar to the *read( )* system call may just provide an entry point for raw devices to call the *physio( )* routine.

xxioctl( ) – This routine provides a lot of flexibility to the device driver as this routine is normally assigned to do operations which can not be done by the *read( )* and *write( )* routines. These operations vary from device to device and may include changing the speed, mode of operation, looking at the status of the device and almost anything that could be done on the device. This routine is not available in the block driver, but a separate character interface may always be written to provide the device this capability.

xxselect( ) – This routine checks whether data is available for reading or if space in the device is available for writing.

xxmmap( ) – A routine for mapping the contents of device into memory.

xxreset( ) – This will reset the device state after a bus reset is made by the bus–adapter support routines.

struct streamtab *d_str – Pointer to a streamtab structure in case the file is a streams device file. This pointer will get us into the streams routines.

Block Driver Top Half Interface

The top–half routines interface to the kernel via the block device switch table.

```
struct bdevsw {
    int (*d_open)();          /* driver's open function */
    int (*d_close)();         /* driver's close function */
    int (*d_strategy)();        /* driver's strategy function */
    int (*d_dump)();          /* driver's dump function */
    int (*d_psize)();         /* driver's psize function */
    int d_flags;          /* driver flags*/
};
```

The file *conf.c* initializes an array of bdevsw structures. Each *bdevsw* structure provides entry points to one device driver's top–half functions. The top–half functions are called via system calls made from user programs.

Block Driver Top Half Interface

xxopen( ) – Serves a similar function as the *open( )* routine for a character driver, although this routine may also be called when the device is being prepared for mounting a file system through the *mount( )* system call.

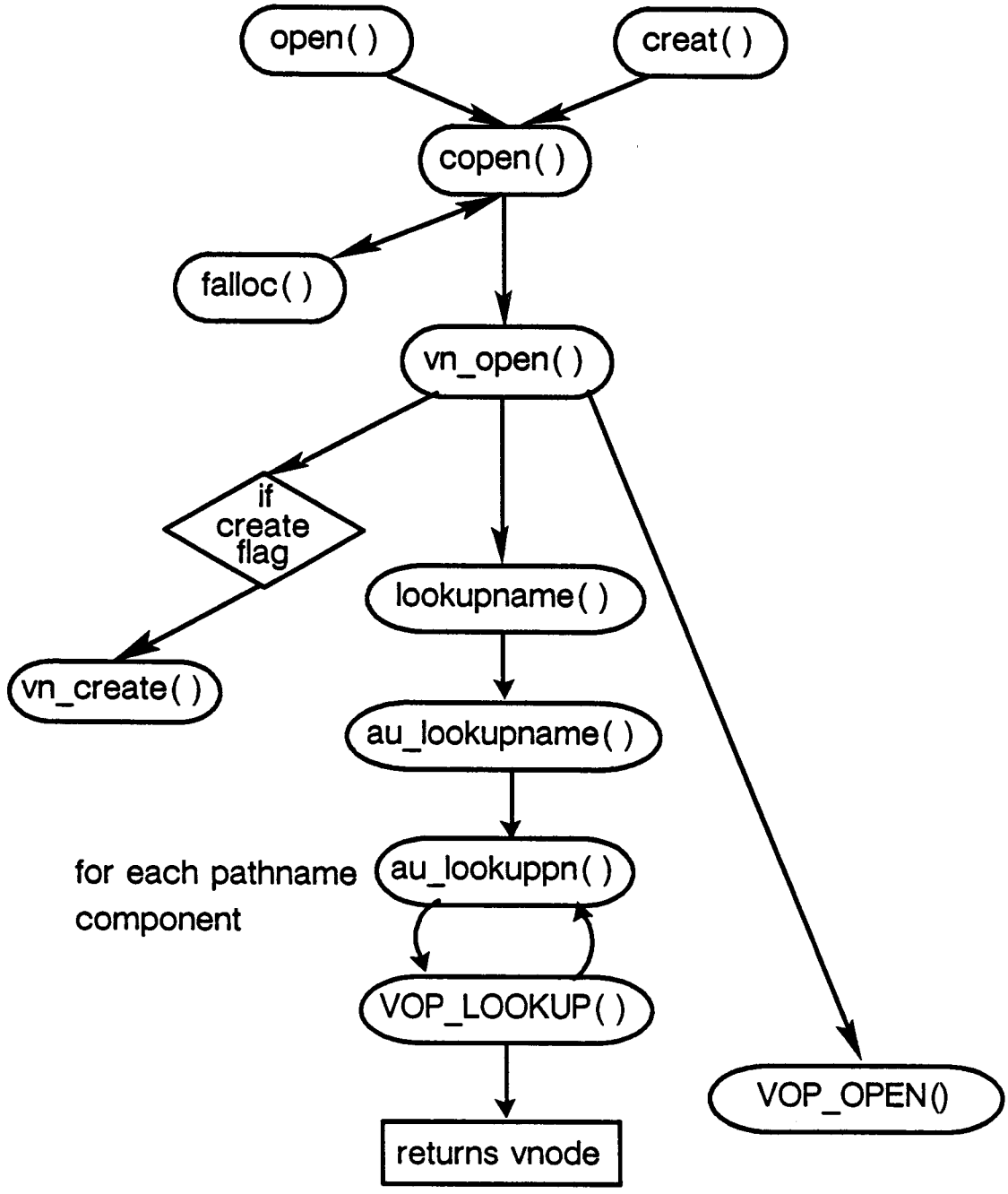
xxclose( ) – Will close the device just like in the character driver.

xxstrategy( ) – This routine performs the combined services usually done by both the *read( )* and *write( )* routines in the character driver. This routine, after initiating the reading and writing, returns immediately.

xxdump( ) – This routine is used to write or dump the contents of physical memory to the device.

xxpsize( ) – Will return the size of a disk partition.

Opening a Vnode (device)



Opening a Vnode

The *open()* or *create()* routines call *copen()* (comon open), this routine will in turn call *falloc()*.

The *falloc()* routine creates an entry in the system open file table and calls *uf_alloc()* to create an entry in the user's file desciptor table *(u_ofile)*. The *copen()* routine then calls *vn_open()*.

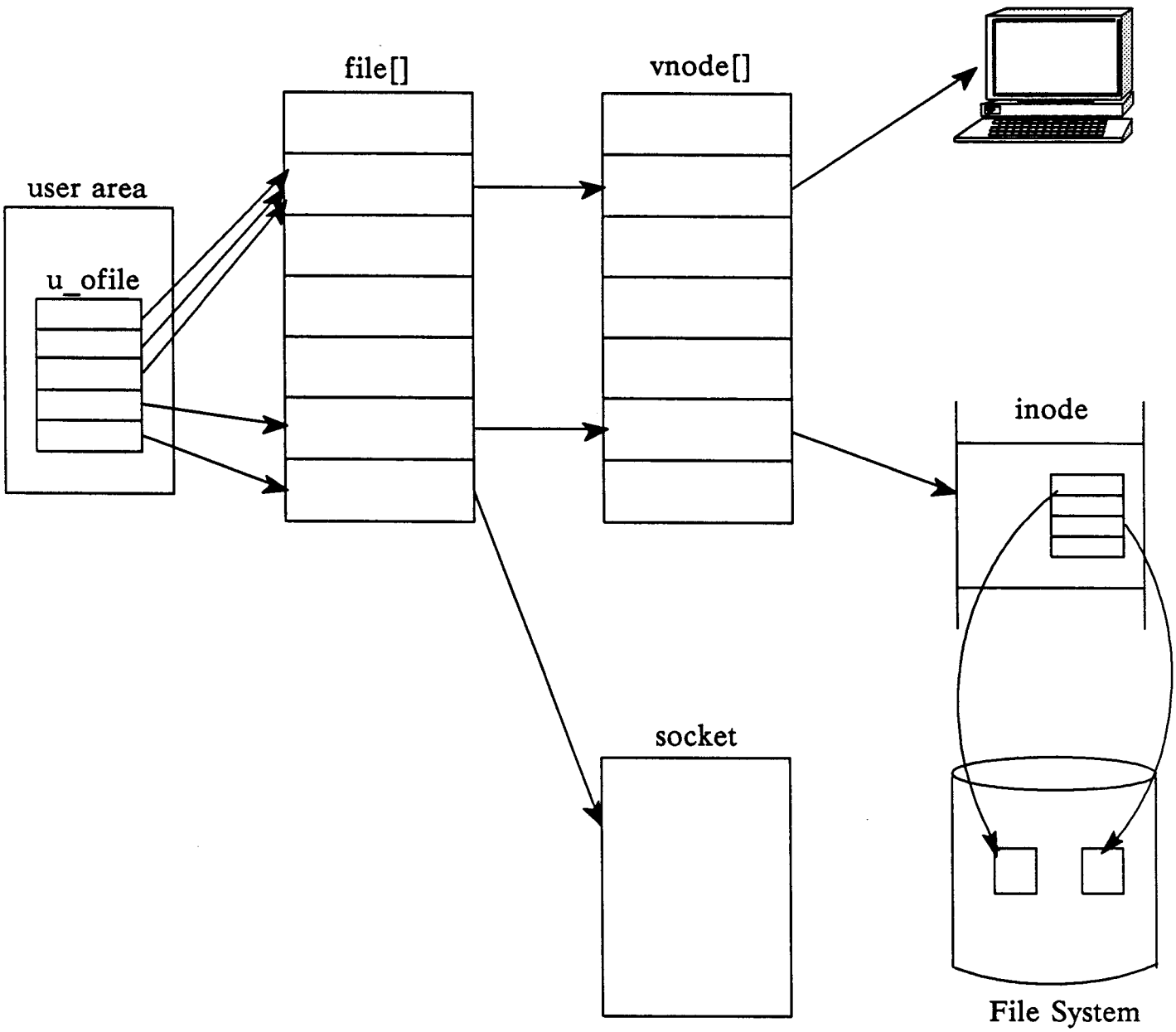
vn_open() creates a vnode structure if the create flag is set, otherwise it opens the vnode and increments its reference count. It then call *lookupname()* which just calls *au_lookupname()*. *au_lookupname()* allocates a buffer for the pathname through a call to *pn_get()*. Eventually the pathname buffer is released (when no longer needed by a call to *pn_free()*). *au_lookupname()* then calls *au_lookuppn()*.

au_lookuppn() is the vnode equivalent to the *namei()* routine. For each component of the pathname it calls the generic *VOP_LOOKUP()* which will call the file system specific lookup routine. For a vnode on a Unix file system this would cause the routine *ufs_lookup()* to be called, which would in turn call *dirlook()*, which was described in module 5.

au_lookuppn() calls the *VOP_LOOKUP()* routine until the last component of the pathname at which time it returns a vnode.

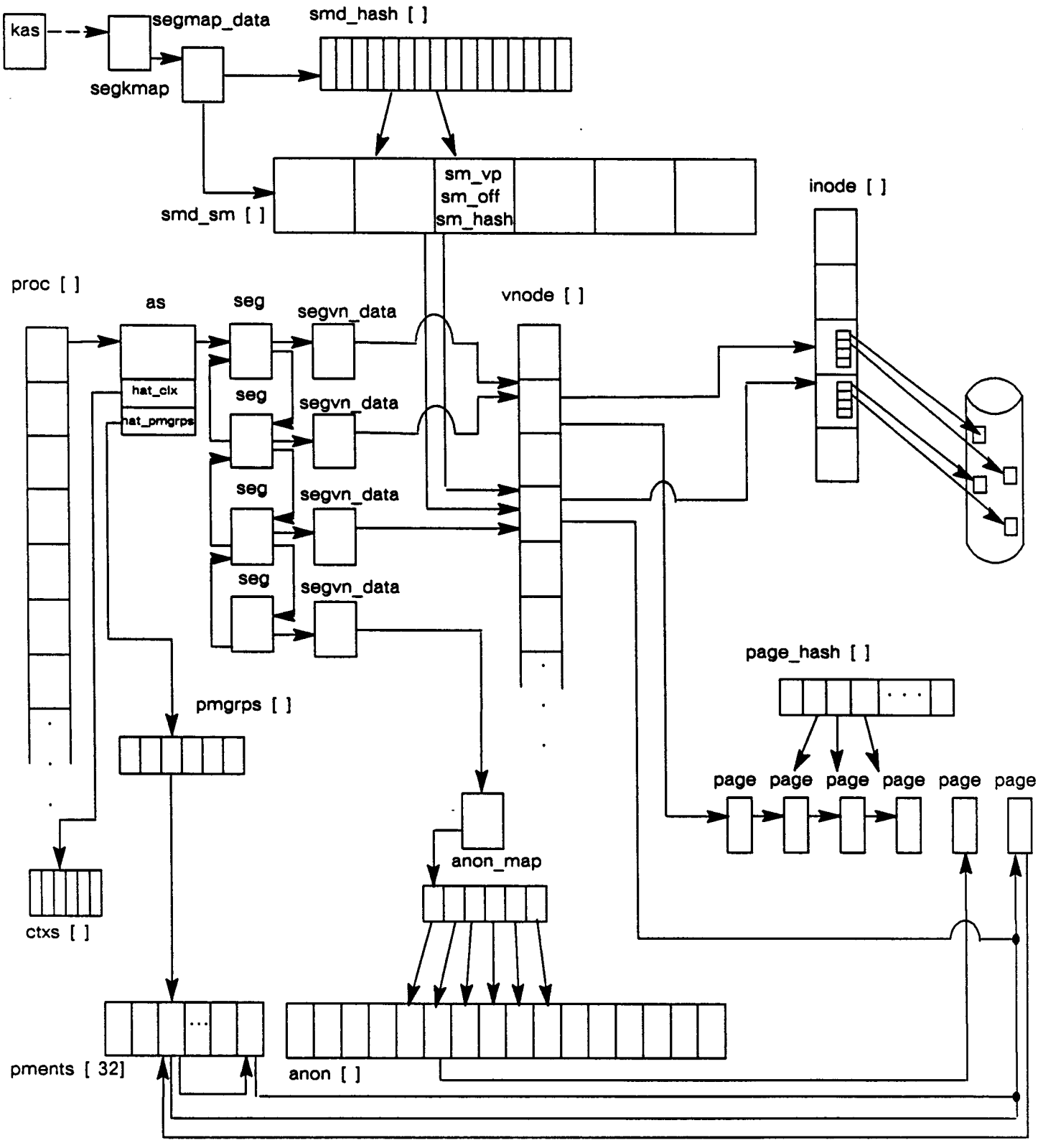
When we eventually return to *vn_open()* again with the vnode of the last component, *vn_open()* calls the generic *VOP_OPEN()* routine. This will call the file system specific routine to open the file represented by the vnode.

Pathname To A File



Pathname To A File

The diagram on the left hand page describes the pathname to a file starting with the descriptor (returned by a previous open) which is used as the index into the descriptor table in the user area. The entry in the descriptor table is a pointer to an entry in the file[] which in turn has the address of a private data structure. This file table entry may be pointing to a vnode structure that represents a special file if the vnode type is either VCHR or VBLK. It may also point to a vnode that represents either a regular file (VREG), a directory file (VDIR), a symbolic link file (VLNK), or a pipe file (VFIFO). The file[] entry may also point to a socket which is used for interprocess communication.

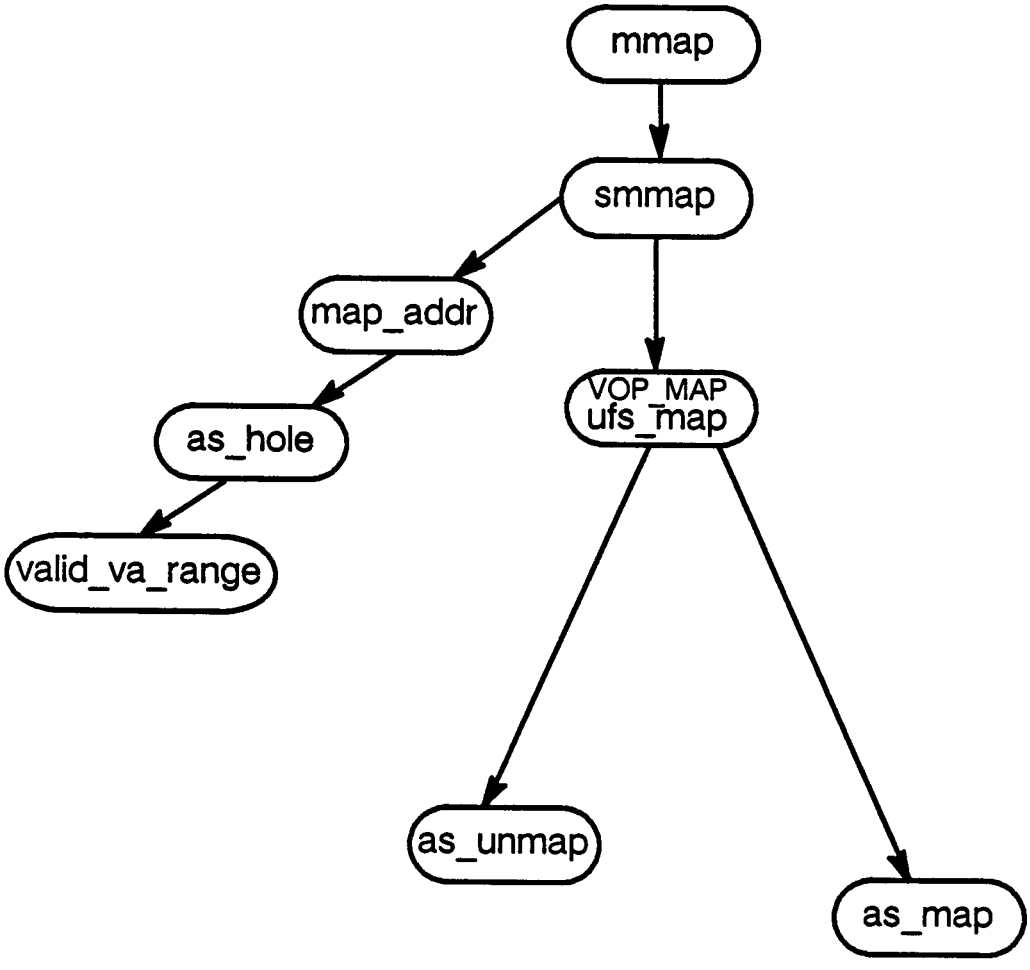

kas
segmap_data
segkmap
smd_hash [ ]
smd_sm [ ]
sm_vp
sm_off
sm_hash
inode [ ]
proc [ ]
as
seg
segvn_data
vnode [ ]
hat_ctx
hat_pmgrps
seg
segvn_data
seg
segvn_data
seg
segvn_data
page_hash [ ]
pmgrps [ ]
page
page
page
page
page
page
anon_map
ctxs [ ]
pments [32]
anon [ ]

Virtual Memory System Map

The Virtual Memory System Map on the previous page will be used to trace through the *read()* and *write()* system calls to show how the SunOS uses the VM System to map the file which will be read or written by the process. This map will also be used in conjunction with several flow diagrams that follow to illustrate what structures are manipulated by different routines that are shown in the flow diagrams.

The flow diagrams represent the expansion of system calls to the underlying routines that are called starting from the process address space layer *(as)*, to the segments that make up the as, to the object defined by its vnode and page structures, down to the hardware address translation layer which is responsible for loading and unloading translations into and from the memory management unit of the system.

mmap( ) system call



mmap( ) system call

The *mmap( )* system call is probably the most basic system call in the SunOS as mapping is the focal point of the Virtual Memory System(VM). The *mmap( )* system call calls the internal version of the routine *smmap( )* which first checks that the correct flags are passed and that either MAP_SHARED or MAP_PRIVATE is specified. If this is a "new" *mmap( )* call, it will call *map_addr( )*, a machine dependent routine to determine at what address the object (file) will be mapped to.

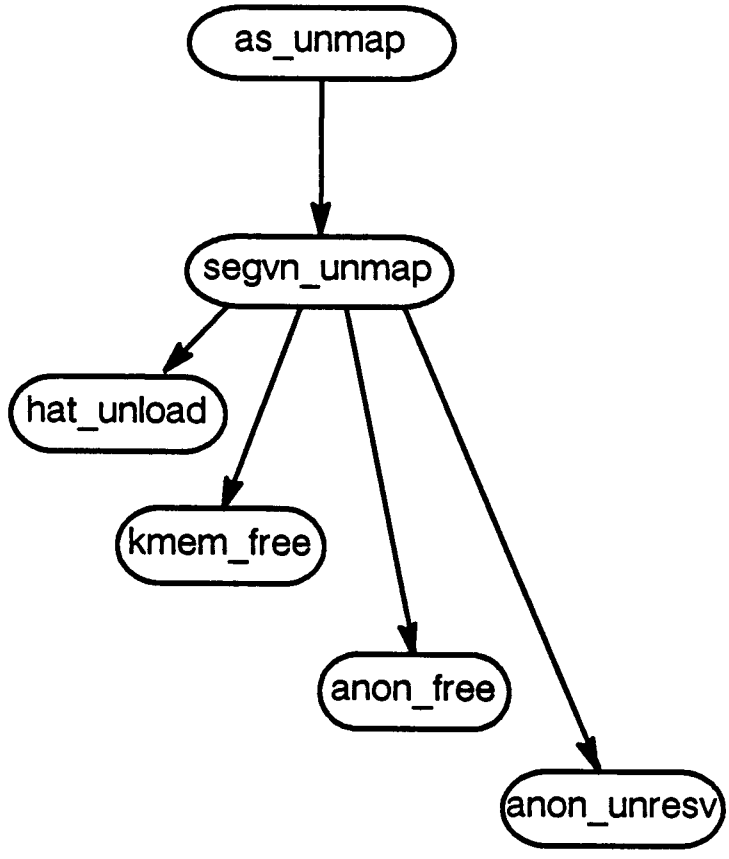
map_addr( ) makes sure that the addresses are shm aligned and will make adjustments to make it so. It calls *as_hole( )* to search for a large enough hole starting from below the stack limit which in turn calls *valid_va_range( )* to check the validity of the address range passed.

If the user requests an address to be used, *smmap( )* will do page alignment checks and boundary checks before calling *getvnodefp( )* to get the file pointer corresponding to the descriptor. The file[] entry holds the vnode pointer in its private data member. If the vnode is not cacheable, the map request is refused as this violates the don't cache attribute. *smmap( )* will also check that the proper protections are set before calling the vnode map routine which for UFS is the *ufs_map( )* routine.

The *ufs_map( )* does not do much except check that the offset and the length are not negative and that the vnode is for a regular file. It also initializes a segvn_crargs structure that will be passed to the as*_map( )* routine. It calls *as_unmap( )* to remove existing mappings on the address range before calling *as_map( )* to map in the file.

as_unmap( ) will search the address space of the process for the segments that fall within the address and size that are given and will call the segment driver unmap routine for each segment that is found. The *segvn_unmap( )* routine calls *hat_unload( )* to unload any hardware translations in the address range taken out.

as_unmap( )


as_unmap
segvn_unmap
hat_unload
kmem_free
anon_free
anon_unresv

as_unmap( )

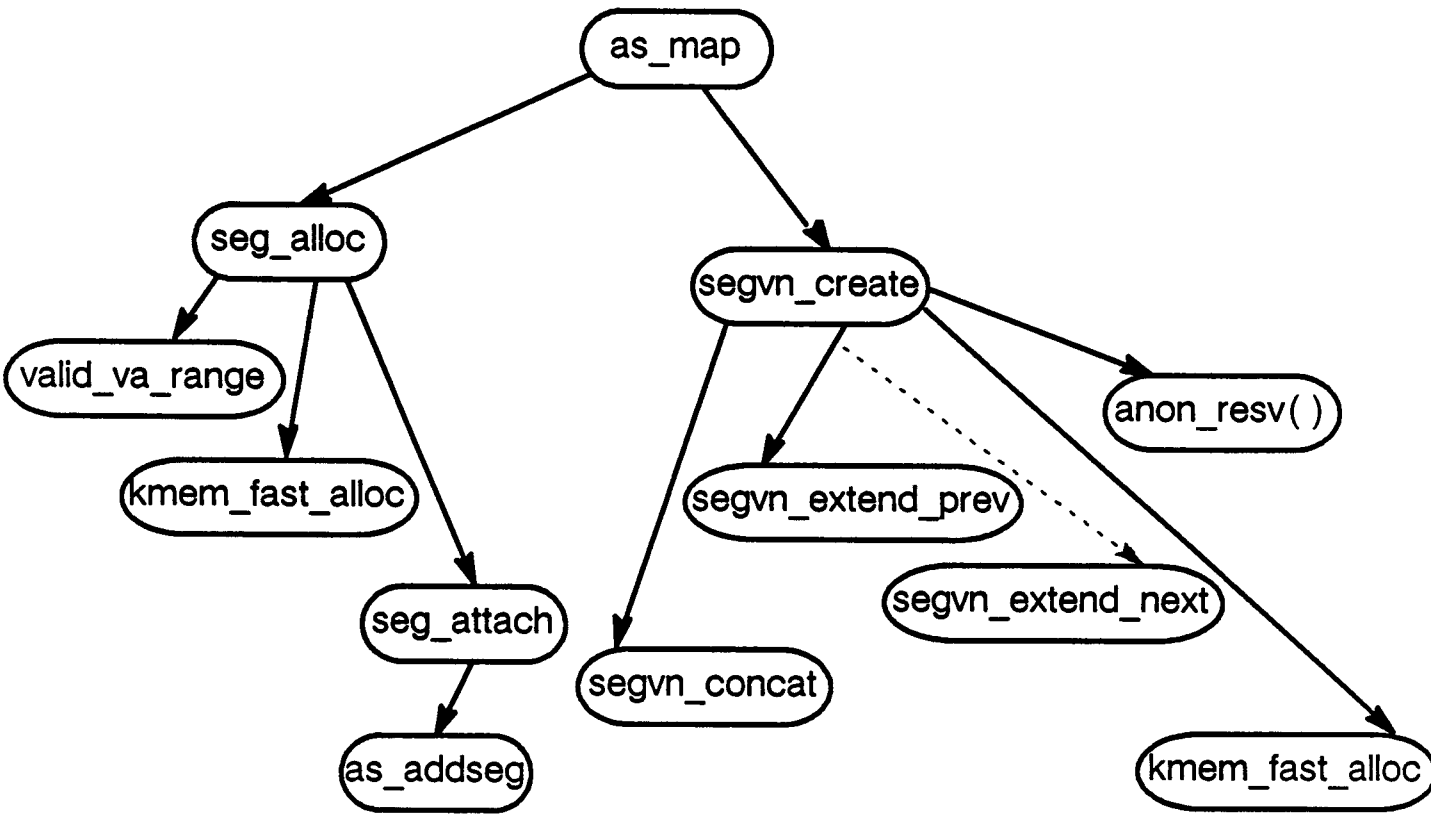
as_unmap( ) loops through all the segments in the address range and for each valid segment within the address range, calls the segment driver unmap routine which is segment dependent. For the vnode segment, this will be the *segvn_unmap( )* which will unmap mappings within this segment.

segvn_unmap( ) calls *hat_unload( )* to unload all hardware translations for the address range specified in the parameters. If the entire segment is being unmapped, we can let go of the whole segment by calling *seg_free( )*. If it is not, *segvn_unmap( )* will check if the address range being unmapped is at the beginning of the segment and it calls *kmem_free( )* to free vpage structures if there are any allocated.

If the segvn_data structure has a non-null pointer to an anon_map structure and the reference count on the structure is still one or if the segment is mapped MAP_PRIVATE, then we have to free the now unused parts of the anon_map array by calling *anon_free( )*. If anonymous pages were reserved earlier, then we give up this reservation by calling *anon_unresv( )*. *segvn_unmap( )* will then update the segment structure and return success to *as_unmap( )*.

If the address range to be unmapped is at the end of the segment, then the above procedure will be done on that particular portion of the segment and it means the segment will be shorter than it was before. The slightly more complicated case, is when the address range being unmapped is in the middle of the segment in which case the segment will actually be split into two with a new segment being allocated for the high end.

as_map( )


as_map
seg_alloc
segvn_create
valid_va_range
kmem_fast_alloc
seg_attach
as_addseg
segvn_concat
segvn_extend_prev
segvn_extend_next
anon_resv( )
kmem_fast_alloc

as_map( )

The *as_map( )* routine calls *seg_alloc( )* which in turn calls the *valid_va_range( )* routine to determine if the address plus the size may contain a mappable range of addresses of a minimum size. *seg_alloc( )* then calls *kmem_fast_alloc( )* to allocate the segment structure for the address range. It then calls *seg_attach( )* which turns around and calls the *as_addseg( )* routine. This routine adds the segment into its proper place in the segments list in the process address space *as*. After these routines return successfully, *as_map( )* calls *segvn_create( )*.

segvn_create( ) checks that the type MAP_PRIVATE and MAP_SHARED are not specified at the same time, and that if a pointer to an anon structure is given, it is illegal to also specify a vnode pointer or MAP_PRIVATE. If MAP_PRIVATE is specified with write protections, this is a case that might need private or anonymous pages, so call *anon_resv( )* to reserve storage in the swap area.

If there is more than one segment in the address space, and the segment is adjacent virtually, *segvn_create( )* will try to concatenate it to these segments. Concatenation is not done if an explicit anon_map structure is supplied or if this is a segment for the kernel's address space. *segvn_create( )* calls *segvn_extend_prev( )* to see if the new segment can be concatenated to a previous one. If this succeeds, it calls *segvn_concat( )* to concatenate the resulting segment to the next one. If *segvn_extend_prev( )* fails, then it will just try to concatenate the new segment to the next one by calling *segvn_extend_next( )*.

If it fails in all of these, then it means that the new segment should remain on its own and *segvn_create( )* calls *kmem_fast_alloc( )* to allocate a segvn_data structure for this new segment and returns success to *as_map()*.

The Read and Write Routines

The *read()/write()* routines operate by:

- Mapping the file to/from the kernel address space
- Copying the data from disk to the physical pages pointed to by the mapping
- Any process now using the object mapped will reference the mapping in the kernel address space
- When the last reference to the object goes away the object can be unmapped.

Since *read(2)* and *write(2)* use mapping you get a coherent access to the file no matter how you access it.

The Read and Write Routines

The *read(2)* and *write(2)* calls are still compatible with the older versions from the user perspective.

The seg_map driver performs the *read(2)* and *write(2)* calls using an internal version of *mmap* with a *type* of shared mapping.

For example when you do a read or write:

1. A portion of the kernel's address space is mapped to the file you're accessing.
2. Another portion of the kernel's address space is mapped to your allocated user buffer space.
3. The memory to memory copy is performed using the *copyin* and *copyout* routines.
4. The mappings may then removed as needed.

Since it uses mapping, a file that is accessed and modified using the *write(2)* call and then accessed by *mmap(2)* will see the modified portion of the file.

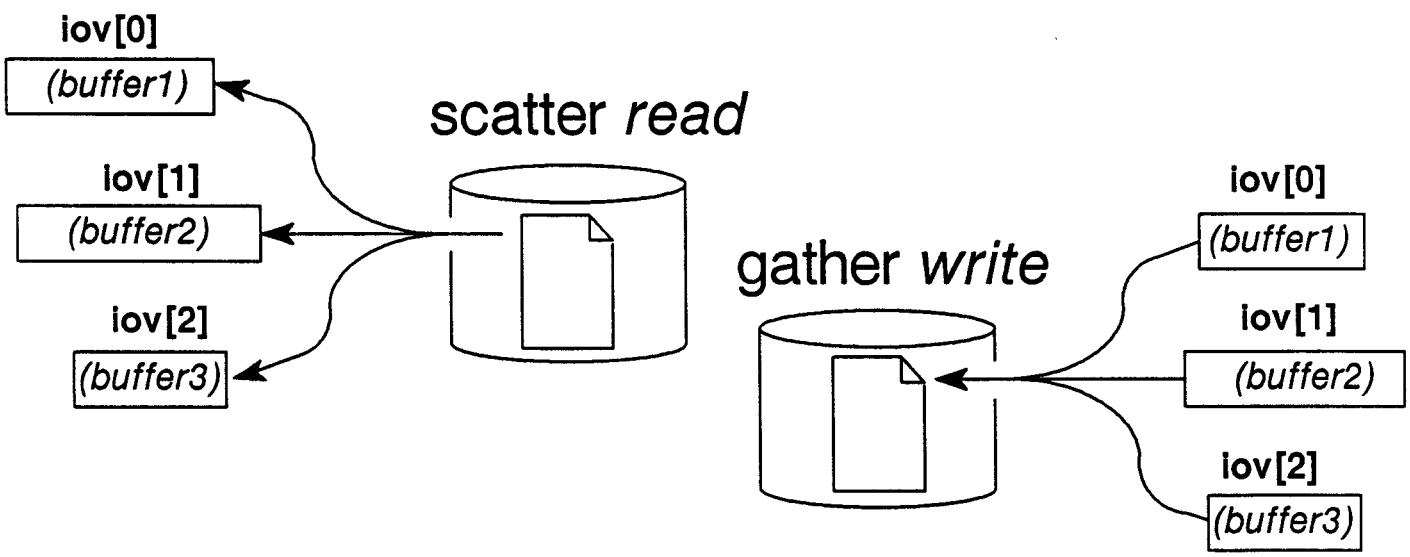
Scatter Read and Gather Write

- Struct iovec for use with scatter–gather:

```
#include <sys/types.h>
#include <sys/uio.h>
struct iovec {
        caddr_t         iov_base;
        int             iov_len;
};
```

- scatter read input, gather write output – *readv(2), writev(2).*

```
cc = readv(fd, iov, iovcnt)
cc = writev(fd, iov, iovcnt)
int cc, fd;
struct iovec *iov;
int iovcnt;
```



Scatter Read and Gather Write

Scatter–gather I/O is ideal for reading and/or writing into non–contiguous, dynamic memory, or when the information is produced and/or utilized by different parts of a program. Scatter–gather allows for an arbitrary number of buffers in a single I/O operation by using an "I/O vector", and each buffer can be of an arbitrary size.

readv() will always fill one buffer completely before proceeding to the next area. As with *read()*, when the returned value is zero, end–of–file has been reached.

The buffers for reading to and writing from are defined using the structure *iovec*. The I/O vector is an array of *struct iovec*, and the vector count is set in *iovcnt*:

iov[0], iov[1], ..., iov[iovcnt–1]

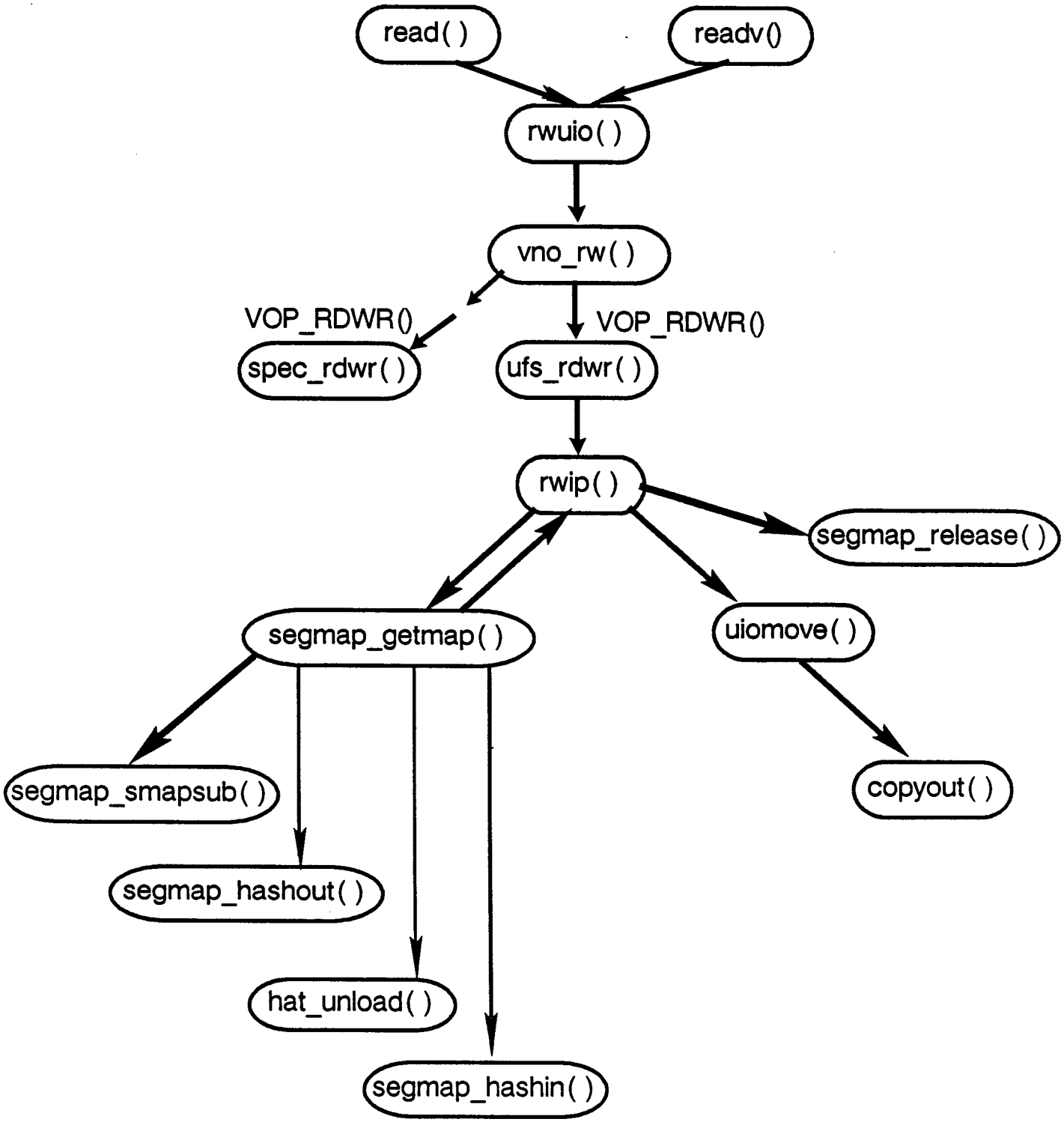
Within each *struct iovec* in the vector/array are two fields:

iov_base ..... pointer to a storage area/buffer (array).
iov_len ..... length in bytes of the in–memory array pointed to by *iov_base*.

If scatter–gather is supported by the disk controller hardware, it will be significantly faster than multiple *reads* and/or *writes*.

Scatter–gather file I/O is a BSD extension to UNIX.

read( ), readv( )


read( )
readv()
rwuio( )
vno_rw( )
VOP_RDWR()
VOP_RDWR()
spec_rdwr( )
ufs_rdwr( )
rwip( )
segmap_release( )
segmap_getmap( )
uiomove( )
segmap_smapsub( )
copyout( )
segmap_hashout( )
hat_unload( )
segmap_hashin( )

read( ) system call

The *read( )* and *readv( )* are similar in that they both call the same routine *rwuio( )*. The main difference is in the parameters passed to them, *read( )* gets a file descriptor, the address of a buffer and a count of the bytes to be transferred and this information is used to initialize the iovec and uio structures that will be used later. On the other hand *readv( )* gets a file descriptor and an array of iovec structures that allows it to do scatter/gather I/O from several buffers in the user data space.

rwuio( ) gets the file pointer associated with the file descriptor and uses the contents of the file[] entry to continue the initialization of the uio structure started by *read( )* and *readv( )* earlier. It then calls a routine pointed to by the fileops structure which in this case is *vno_rw( )* which turns around and calls the *ufs_rdwr( )* routine through the appropriate function pointer which is a member of the vnodeops structure. This is the object oriented way of referencing the function which allows the implementation of several file system types whose file system dependent routines can be accessed in a uniform way.

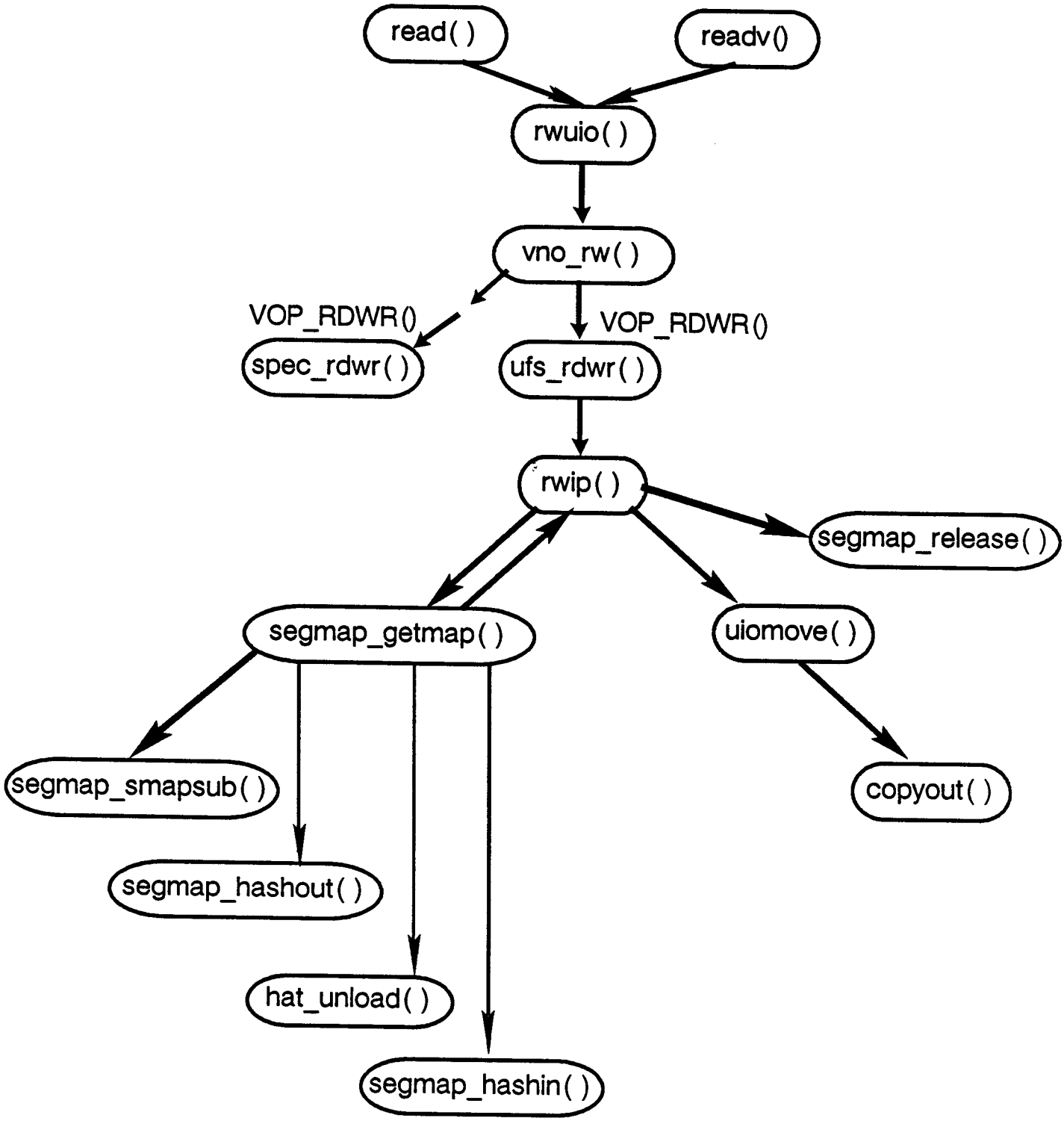
ufs_rdwr( ) checks if the operation will be writing with append to a regular file, and if so it sets the offset to the file size. It then calls *rwip( )* which does the real work for *read( )* and *write( )* in the UNIX File System (ufs).

rwip( ) checks for a valid rw flag, makes sure that the file type is only regular, directory or symbolic link file, that the offset is valid, and then locks the inode corresponding to the file. It collects statistical information about the system call and then calls *segmap_getmap( )* which is one of the segmap driver routines. This is the routine that will determine in the fastest time possible if the block of data that is requested is in memory already.

segmap_getmap( ) will hash on the vnode and offset combination to find a hash slot in the smd_hash table and then traverse it to find the particular vnode and offset pair. If we get a non-null pointer to an smap structure, this is an indication that a mapping is found and we determine if another process is using it, in which case we increment the smap reference count.

Sun Education
Not to be copied

read( ), readv( )



read( ) system call (cont.)

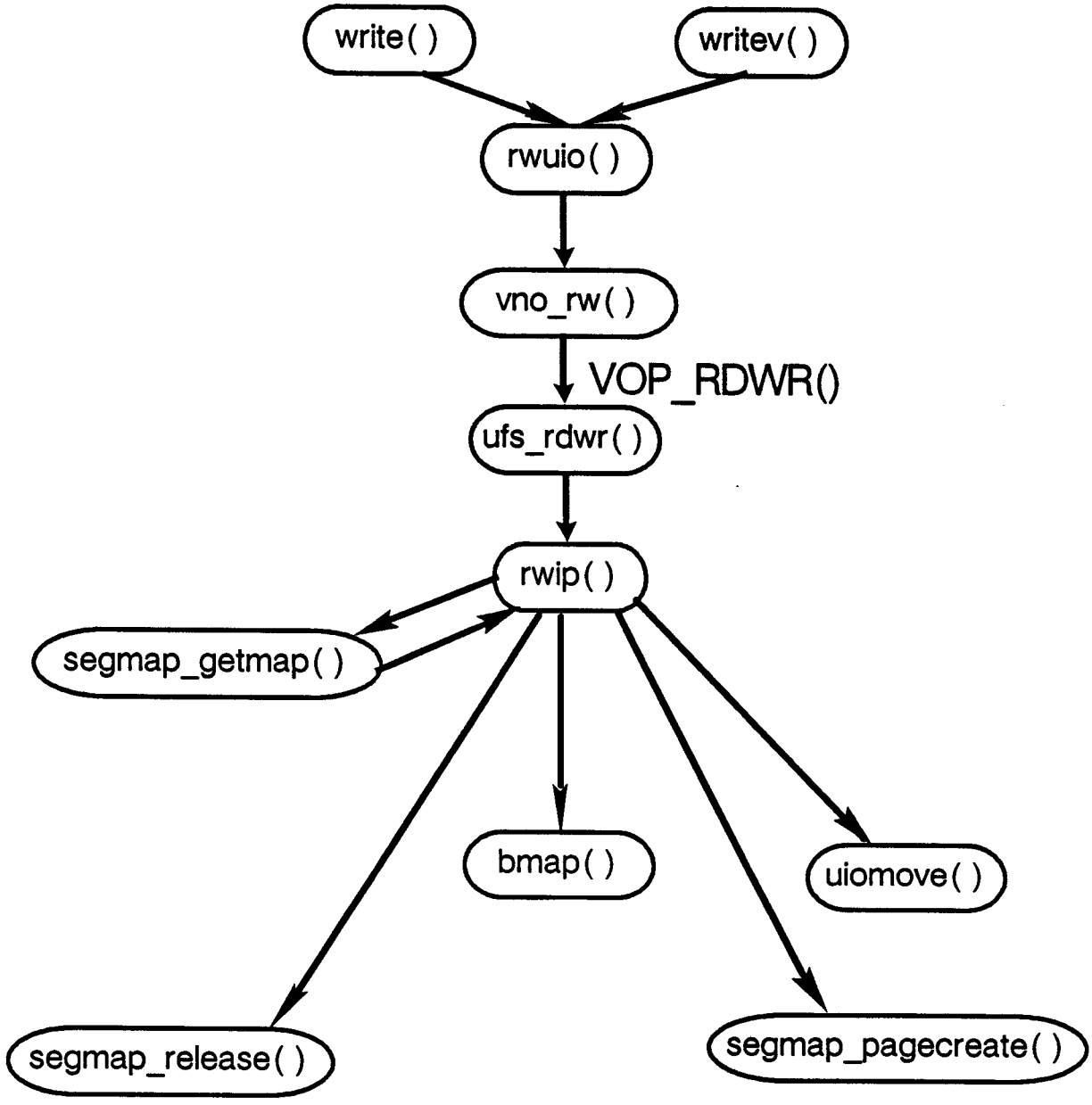
If no process is referencing the mapping, then we reclaim the mapping by incrementing the segmapcnt reclaim counter and call *segmap_smapsub()* to take out the mapping from the free list. It will be allowed to stay in its hash slot.

If the mapping is not found, then a mapping is taken off the free list if it is not empty, otherwise the smd_want flag is turned on and *sleep( )* is called with the address of the free list. If one is available, it is taken out of the free list by calling *segmap_smapsub( )*.

If this mapping still contains the old association with the vnode, it must be destroyed by calling *segmap_hashout( )* to take it out of its present hash slot and then to call *hat_unload( )* to unload the hardware translations for the mapping. *segmap_getmap( )* then calls *segmap_hashin( )* to hash the mapping into a slot in smd_hash[] based on the vnode offset combination that will be mapped. *segmap_getmap( )* then returns to *rwip( )* with the address of the mapping.

rwip( ) then calls *uiomove( )* to copy the data from the address into user data space. If the address returned is that of a valid page then the *read( )* is completed right at this point. If the page does not have a valid hardware translation, a pagefault will occur when *copyout()* which is called by *uiomove()* tries to access the address in question. The handling of a pagefault will be discussed in a following diagram.

write( ), writev( )



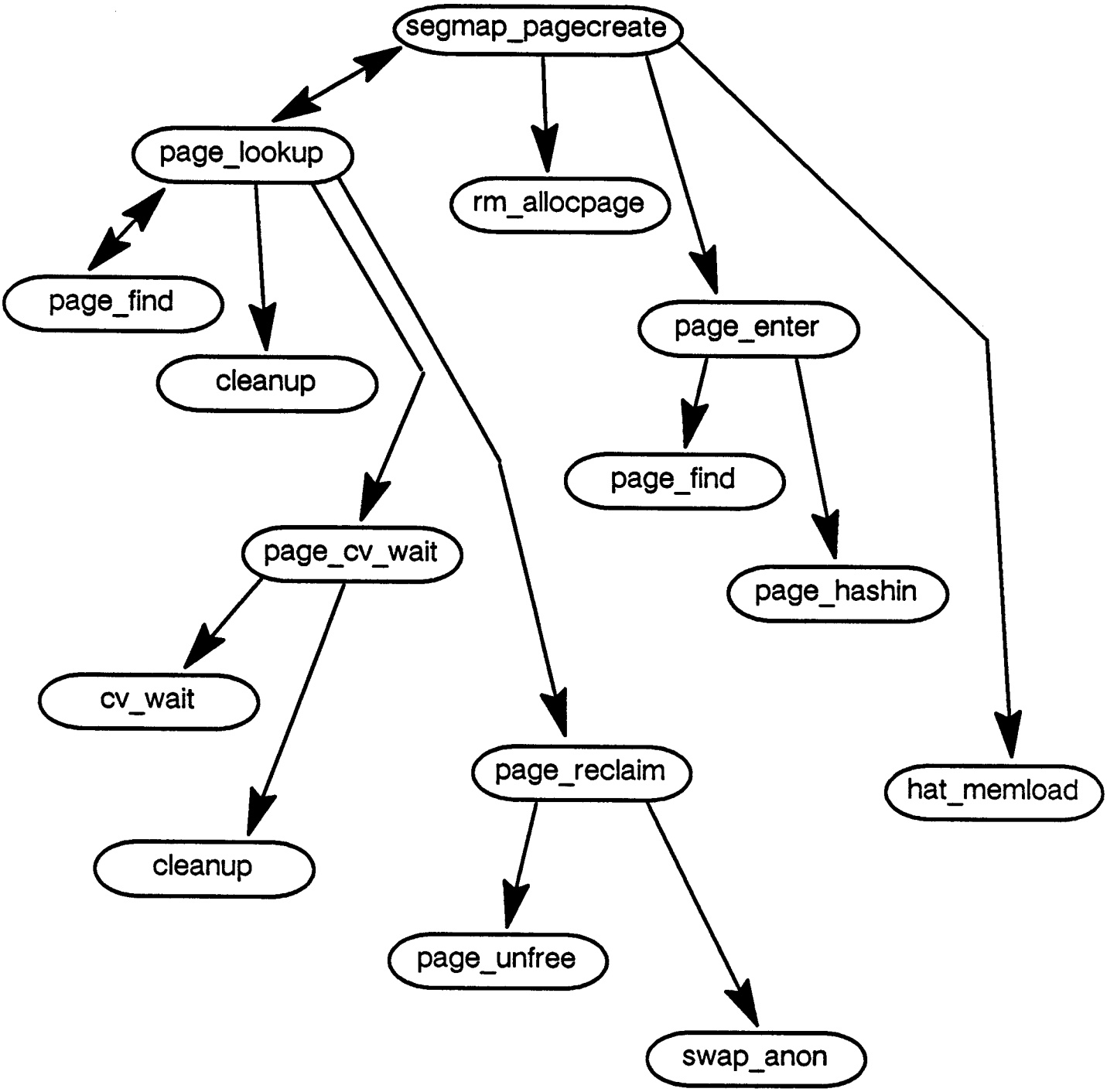
write( ) system call

The *write( )* and *writev( )* system calls follow the same initial path as that of the *read( )* and *readv( )* routines. *write( )* and *writev( )* both calls *rwuio( )* which perform the same initializations on the uio structure and then calls the fileops pointer to the read/write routine. This routine pointed to is *vno_rw( )* which makes sure that the file system is writable and then calls the object-oriented vnodeops routine for read/write which for the UNIX file system is the *ufs_rdwr( )* routine. This routine gets the inode pointer corresponding to the vnode that is passed and then sets the offset to the file size if the operation is writing with append to a regular file after locking the inode. It then calls *rwip( )* which performs additional checks not done when just reading from the file, like checking if we are going to exceed the maximum file size in this particular write operation. If the maximum file size will be exceeded, a SIGXFSZ signal will be sent to the process, the error set to EFBIG, the inode unlocked and then the error returned to the calling routine.

If the maximum file size will not be exceeded, then it checks if the sync flag is on for the opening of the file, in which case the inode sync flag is also turned on so that the inode is written out synchronously. *rwip( )* then calls *segmap_getmap( )* to return the address of the page in main memory representing the block of the file that we want to write to. If we are extending the length of the file, the *bmap( )* routine will be called to make sure that if new blocks need to be allocated that it is done at this point before the file size is increased.

bmap( ) is also called if we are not extending the file but we are doing a whole mappings worth in which case we must be sure that the needed blocks are allocated first. If we are extending the file and we are writing from the beginning of the mapping, we do not need to read in the pages. We'll just call *segmap_pagecreate( )* to create the page and we also call it, when we are not extending the file but we are writing an entire page. After the base address is returned to *rwip( )*, *rwip( )* then calls *uiomove( )* to move data from the process address space to the page representing the block of the file. *rwip( )* then calls *segmap_release( )*.

segmap_pagecreate( )



segmap_pagecreate( )

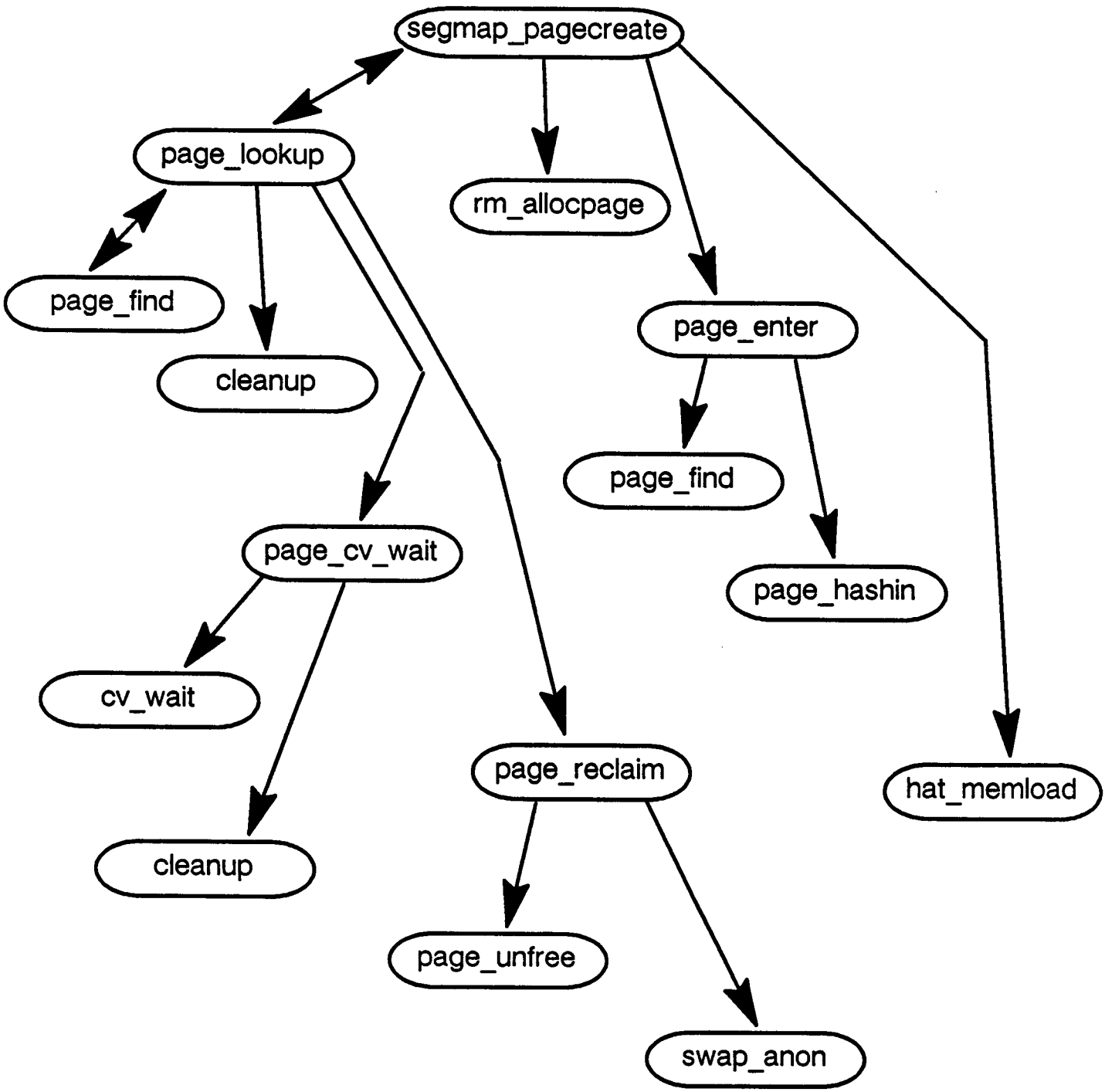
This routine is called whenever we are writing to a page and we do not have to read in the page. There are two cases, one is when we are extending the size of the file and we are writing at the beginning of the page, *segmap_pagecreate( )* is called. The other case is when we are not extending the file size but we are writing a whole page worth, in which case we do not need the page but we can just create the page we are going to write to and load up the translation.

segmap_pagecreate( ) calls *GET_SMAP( )* to return the smap structure corresponding to the address. It gets the vnode pointer from the smap and calculates the offset from the address 'anded' with MAXBOFFSET plus the offset stored in the smap. It then loops through the address range a pagesize at a time, and calls *page_lookup( )* which calls *page_find( )*.

page_find( ) uses the page_hash[] to determine if the page is already in memory by hashing on the vp, offset combination. If data for the page is in transit and data is not valid and the I/O queue is not empty, it then calls *cleanup( )* to take the buffer off the queue and to unload mappings to the page if it is marked remapped. If the page is found but is locked and in transit then *page_cv_wait( )* is called to wait until the page becomes available. If the page is now on the free list, then it is reclaimed by calling *page_reclaim( )*. *page_lookup( )* verifies the contents of the page before returning the page to *segmap_pagecreate( )*.

If *page_lookup( )* fails, then *segmap_pagecreate( )* calls *rm_allocpage( )* which calls *page_get( )* to allocate enough pages for the number of bytes passed. After getting the pointer to the page structure, *segmap_pagecreate( )* calls *page_enter( )* to enter the page in the hash chains and vnode page list. It then calls *page_unlock( )* to unlock the page and wake up all processes wanting the page.

segmap_pagecreate( )

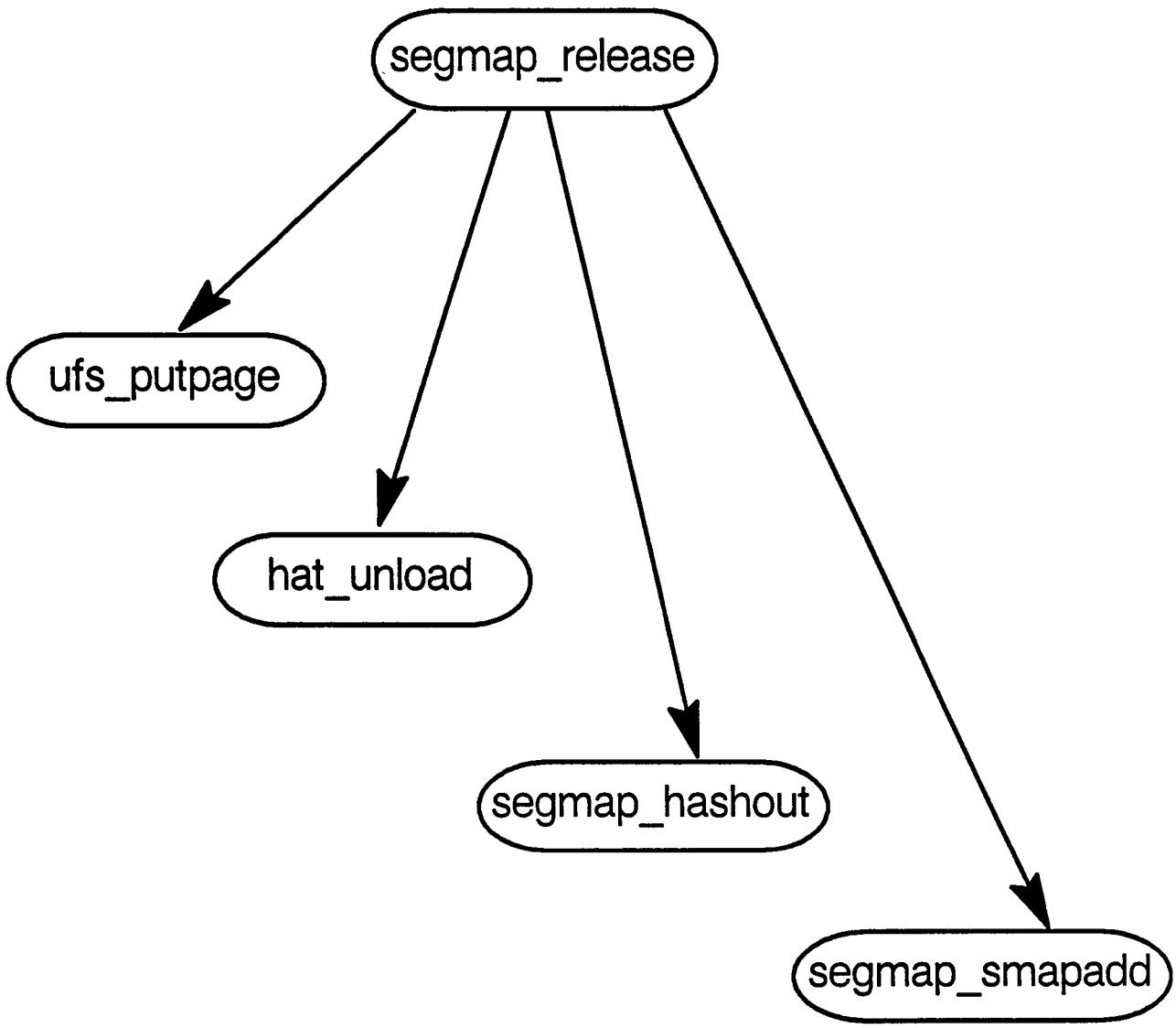


segmap_pagecreate( ) (cont.)

If we are softlocking the page, then we call hat*_memload( )* with the lock flag and we do not call PAGE_RELE( ) which will decrement the keepcnt and calls *cv_broadcast( )* if keepcnt goes down to zero and the page is wanted.

It the page is found, and we are softlocking, then we call *PAGE_HOLD( )* to increment the keepcnt and then call *hat_memload( )* with the lock flag, otherwise only *hat_memload( )* is called.

segmap_release( )



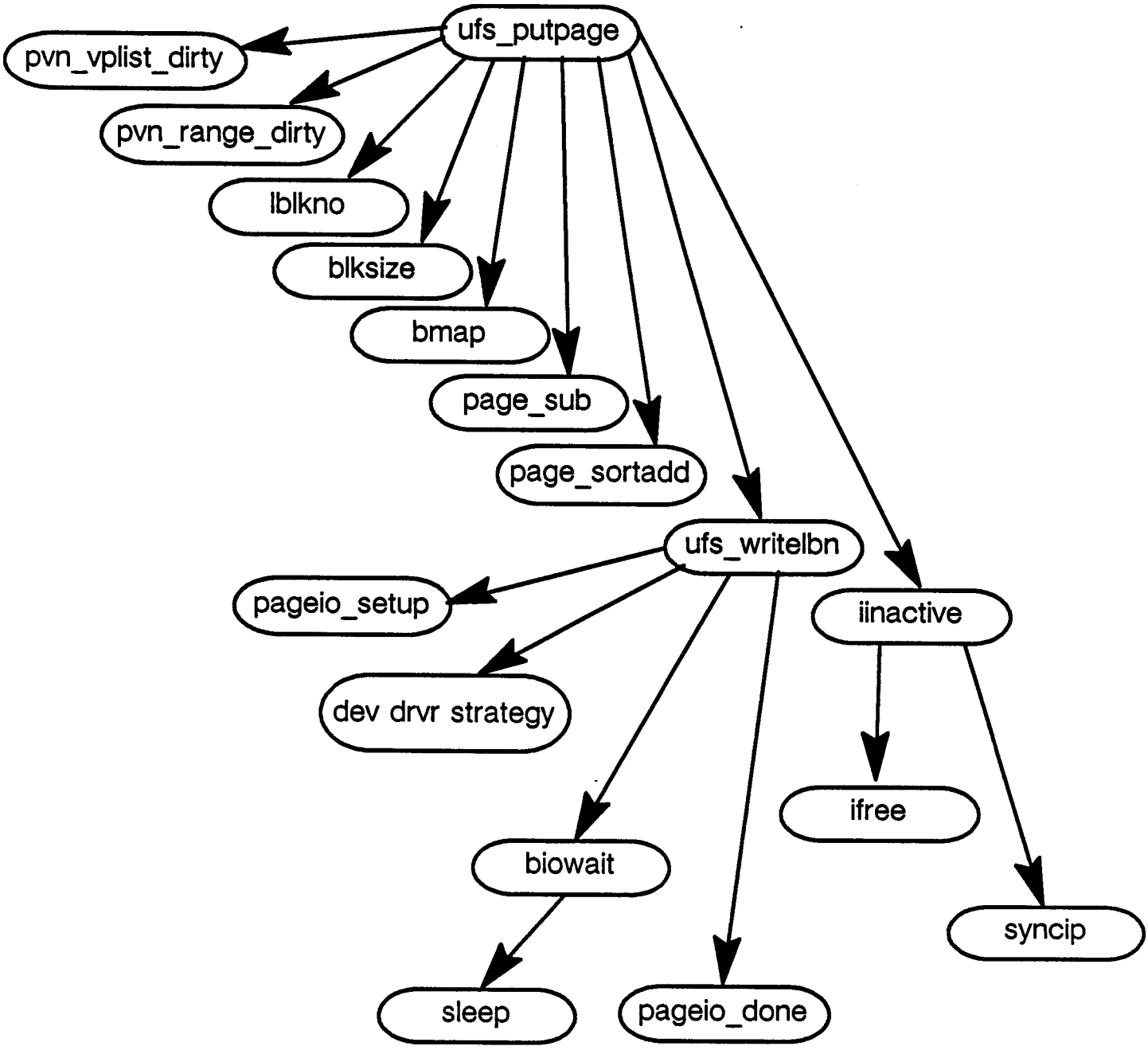
segmap_release( )

segmap_release( ) is called in the *rwip( )* routine when we have to release the page because an error has occurred during the reading or writing. But more importantly, this routine is called when we have written to the page and we want to be sure that the object to which the page maps to also gets updated.

segmap_release( ) first checks if the address passed to it is within the segment, if not *panic( )* is called. It gets the smap structure corresponding to the address and checks the flags on the smap. If the there are no flags set or if the only flag set is SM_DONTNEED, then we shall not bother writing out the page at all.

Otherwise, we call the vnodeops putpage routine which for the UFS is the *ufs_putpage( )*. When this routine returns successfully to *segmap_release(),* it means that the page has been written out. After returning to *segmap_release( )*, it then decrements the reference count on the smap structure and if it gets down to zero and the SM_INVAL flag is on, it calls *hat_unload()* to unload the translations to the page. It then calls *segmap_hashout( )* to remove the page from its hash slot and then calls *segmap_smapadd( )* to add it to the free list.

ufs_putpage( )


ufs_putpage
pvn_vplist_dirty
pvn_range_dirty
lblkno
blksize
bmap
page_sub
page_sortadd
ufs_writelbn
pageio_setup
dev drvr strategy
iinactive
ifree
biowait
syncip
sleep
pageio_done

ufs_putpage( )

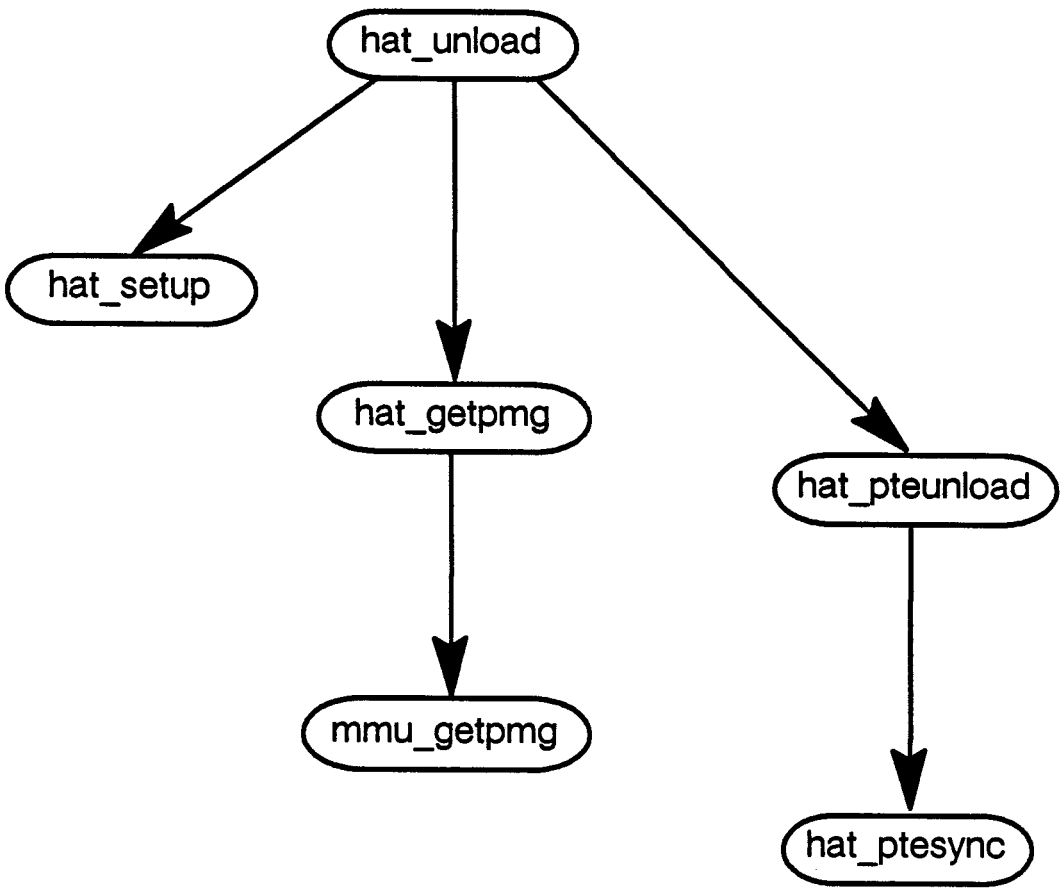
This routine is called by *segmap_release( )* when we have written to the page and the object it maps to needs to be updated. This routine gets called via the vnodeops structure pointer and could be different for different file system types.

ufs_putpage( ) calls *pvn_vplist_dirty( )* if it is passed a len == 0, which means that the entire vp list of pages must be searched for dirty pages. If the len and offset are not zero, then it calls *pvn_range_dirty( )* which will search for dirty pages within the range of addresses specified. In either case, the two routines will return a list of dirty pages that need to be written back. *ufs_putpage( )* checks if the file system to write back to is writable and if not, returns an error. Otherwise, It loops through all the dirty pages and calls *lblkno( )* and *blksize( )* to return the logical block number and block size respectively. It then calls *bmap( )* with a flag of S_OTHER because blocks should have been allocated for these dirty pages already.

ufs_putpage( ) then calls *page_sub( )* to remove the first page from the dirty list and initializes the io_list. It then loops through the rest of the dirty pages and calls *page_sub( )* to take the pages out of the dirty list and then calls *page_sortadd( )* to add the pages into the io_list. *ufs_putpage( )* then calls *ufs_writelbn( )* which calls *pageio_setup( )* to allocate and initialize the buffer structure for the pageio and then calls the device driver strategy routine to get the I/O operation underway. u*fs_writelbn( )* also calls *biowait( )* which calls *sleep( )*. When the process is awakened after I/O completion, *pageio_done( )* is called to unload the mappings and remove the block from the buffer hash list.

After *ufs_writelbn( )* returns successfully to *ufs_putpage( )*, *ufs_putpage( )* checks if the vnode reference count goes down to zero after it is decremented and if so the *iinactive( )* routine is called which will truncate the file if the number of links to the inode is already zero and calls *ifree( )* which will put back the inode into the free map. If there are still links to the file, then *syncip( )* is called which will asynchronously write the pages of the vnode and then update the inode information on disk.

hat_unload( )


hat_unload
hat_setup
hat_getpmg
hat_pteunload
mmu_getpmg
hat_ptesync

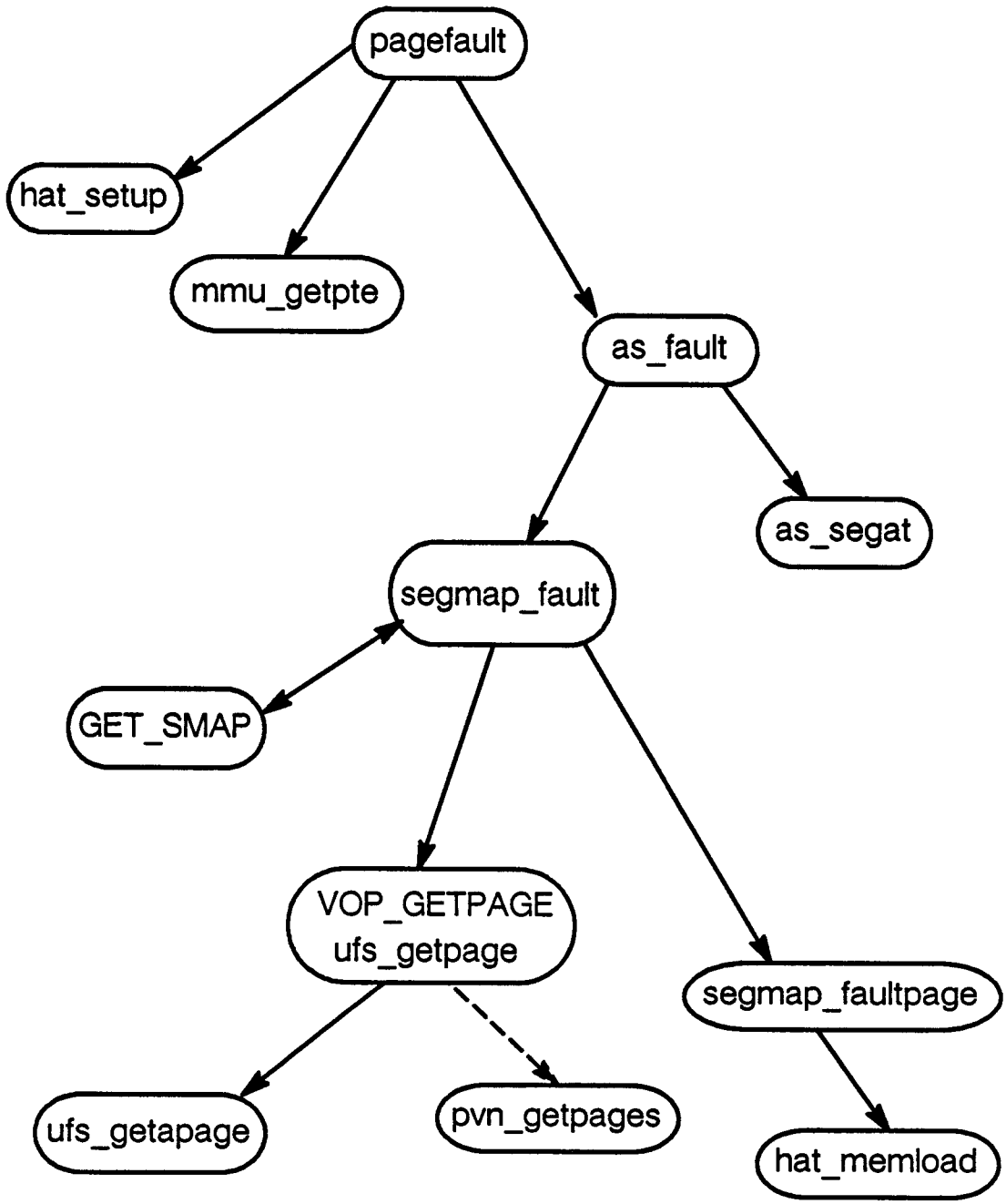
hat_unload( )

hat_unload( ) is a machine independent hat routine that is used to unload mappings within the address range specified by addr and len. This routine first checks that there are allocated pmgrps for the address space given and if none, it will return immediately. Otherwise, it calls *hat_setup( )* which is called by UNIX during pagefaults to insure that a context is allocated for this address space and that the ctx time is updated.

hat_unload( ) will then loop through the address range looking for invalid pmgrps and for every one that is found its pmg_keepcnt is decremented. It calls *hat_getpmg( )* for the specific address which then calls *mmu_get-pmg( )* which returns the address of the pmgrp using the 16 bit segment map entry corresponding to the given address. If indeed this pmgrp is invalid, then the address will be bumped up to point to the next pmgrp in the address range to avoid checking every pme within the invalid pmgrp. When a valid pmgrp is found, then we proceed to throw away its mappings.

The *hat_pteunload( )* is called to unload the mapping for the pme which calls *hat_ptesync( )* to unload the translation and then remove the pme from the list of pme's mapping the page. *hat_ptesync( )* also synchronizes the hardware and software structures for a pte by updating the hardware nocache bit, the software reference and modify bits, and invalidating the pte's.

pagefault( )


pagefault
hat_setup
mmu_getpte
as_fault
as_segat
segmap_fault
GET_SMAP
VOP_GETPAGE
ufs_getpage
segmap_faultpage
ufs_getapage
pvn_getpages
hat_memload

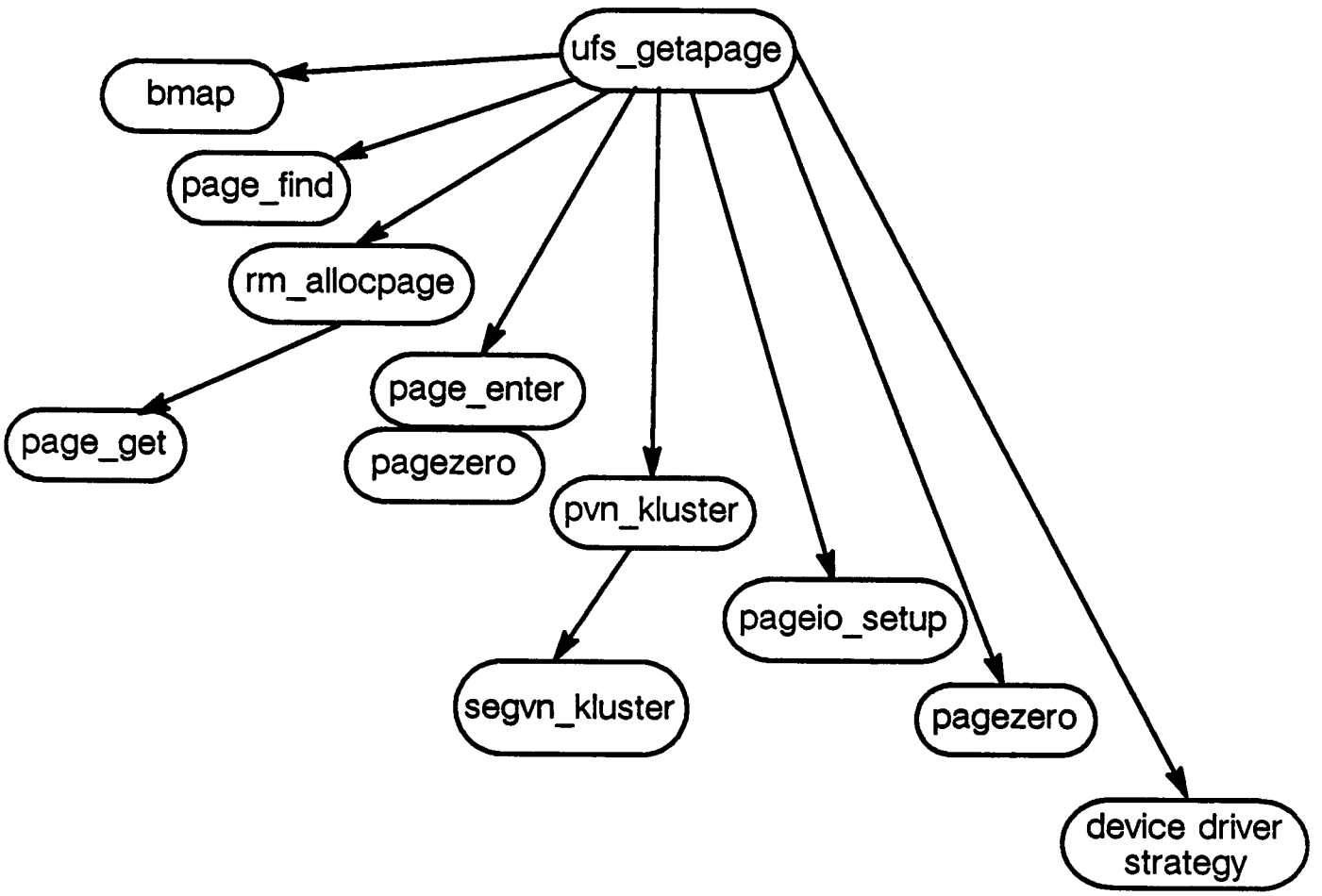
pagefault( )

When we are reading from a page and the translation or the mapping to that page is invalid the *copyout( )* routine will detect it and will cause a page fault that will result in the machine specific *pagefault( )* routine being called. The *pagefault( )* routine checks that the faulting address is valid so that it can return a real fault if it is not. It calls *hat_setup( )* that will check if there is a context and if none gets one and update the context time. *pagefault( )* then calls *mmu_getpte( )* which gets the *pte* from the mmu corresponding to the address passed. If this pte is valid and the protections are OK, the fault is satisfied right at this point and *pagefault( )* will return success.

If the pte is invalid, then *pagefault( )* will call the *as_fault( )* routine. This routine first calls *as_segat( )* to determine the segment to which the faulting address belongs and if it is not found, then returns null which says that there is no mapping at the fault address. If *as_segat( )* is successful in returning the segment to *as_fault( )*, the segment driver fault routine will now be called via the object-oriented method using the seg_ops structure pointer to the fault routine. In the case of a read fault on a file that is mapped into kernel address space, then the *segmap_fault( )* routine will be called and if the fault is on user address space, then *segvn_fault( )* will be called.

segmap_fault( ) is called if we are reading from a file mapped into kernel address space. This routine calls *GET_SMAP( )* to get the smap structure corresponding to the segment and address and then calls the vnodeops getpage routine *ufs_getpage( )*. This routine calls either *ufs_getapage( )* or *pvn_getpages( )* depending on whether we are trying to get up to a pagesize or more. *pvn_getpages( )* will end up calling *ufs_getapage( )* for each page that it wants. In either case, *ufs_getapage( )* will try to find the page using the page_hash[], and if it is not in memory will allocate a page to hold the data block and will call the device driver strategy routine to bring the block from the file into main memory. It then returns to *ufs_getpage( )* and *ufs_getpage( )* returns to *segmap_fault( )* which in turn calls the *segmap_faultpage( )* routine. This in turn calls *hat_memload( )*.

ufs_getapage( )



ufs_getapage( )

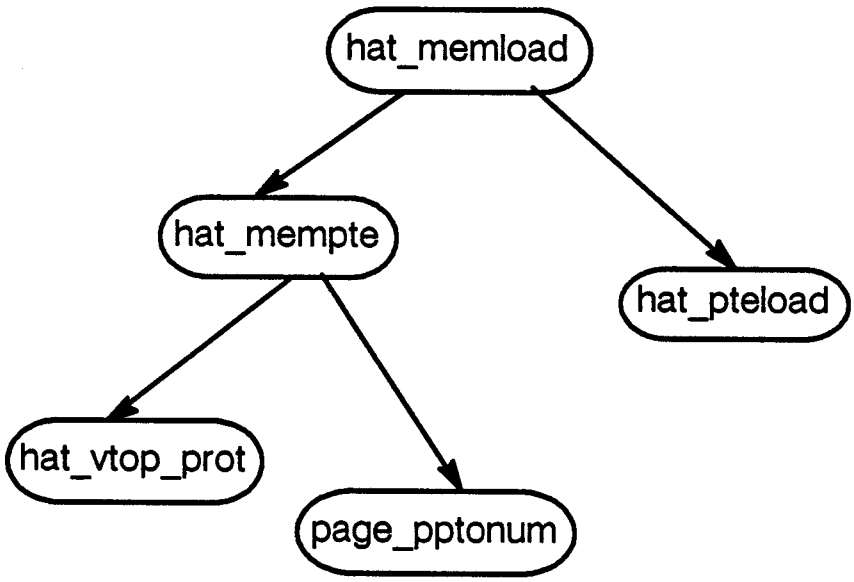
This is a routine that is called by the *ufs_getpage( )* or the *pvn_getpages( )* routines to get a particular page. When this routine is called the inode is already locked. The file system block size can be either 4k or 8k and if it is set to 4k, then *ufs_getapage( )* will do two reads to get the page and will not do the read ahead.

ufs_getapage( ) calls *bmap( )* to translate the logical block number to physical and also to get the read ahead block. It then calls *page_find( )* that will search for the page by hashing on the <vp, offset> combination to find a slot in the page_hash[]. If the page is not found, then it must be allocated by calling r*m_allocpage( )*. This routine calls *page_get( )* to allocate enough pages for the number of bytes that is passed as a parameter. In either cases, a page structure is returned to *ufs_getapage( )* which calls *page_enter( )* to enter the page into the hash chains and vnode page list. If we are reading from an unallocated page (hole) then *pagezero( )* is also called to zero out the contents of the page.

If we are really reading from the file (not hole) for blocks that are not in memory, we call *pvn_kluster( )* to see if we can bring the blocks in clusters. This routine calls *segvn_kluster( )*, which will approve the request if we are still within the segment and mapping from the same <vp, offset>. After verifying whether we can do clustering or not, we check if the requested data fits into the pages allocated.

We call *pageio_setup( )* to allocate and initialize a buffer structure to be used for the pageio. If we are just reading a portion of the page from disk, then we call *pagezero( )* to zero out the portion of the page that will not be read from disk. After I/O is set up, *ufs_getapage( )* will call the device driver strategy routine to get I/O started on the page. If a second read is indicated either for read ahead or for the second block of the page, then the *pageio_setup( )* and the device driver strategy routine will be called a second time.

hat_memload( )

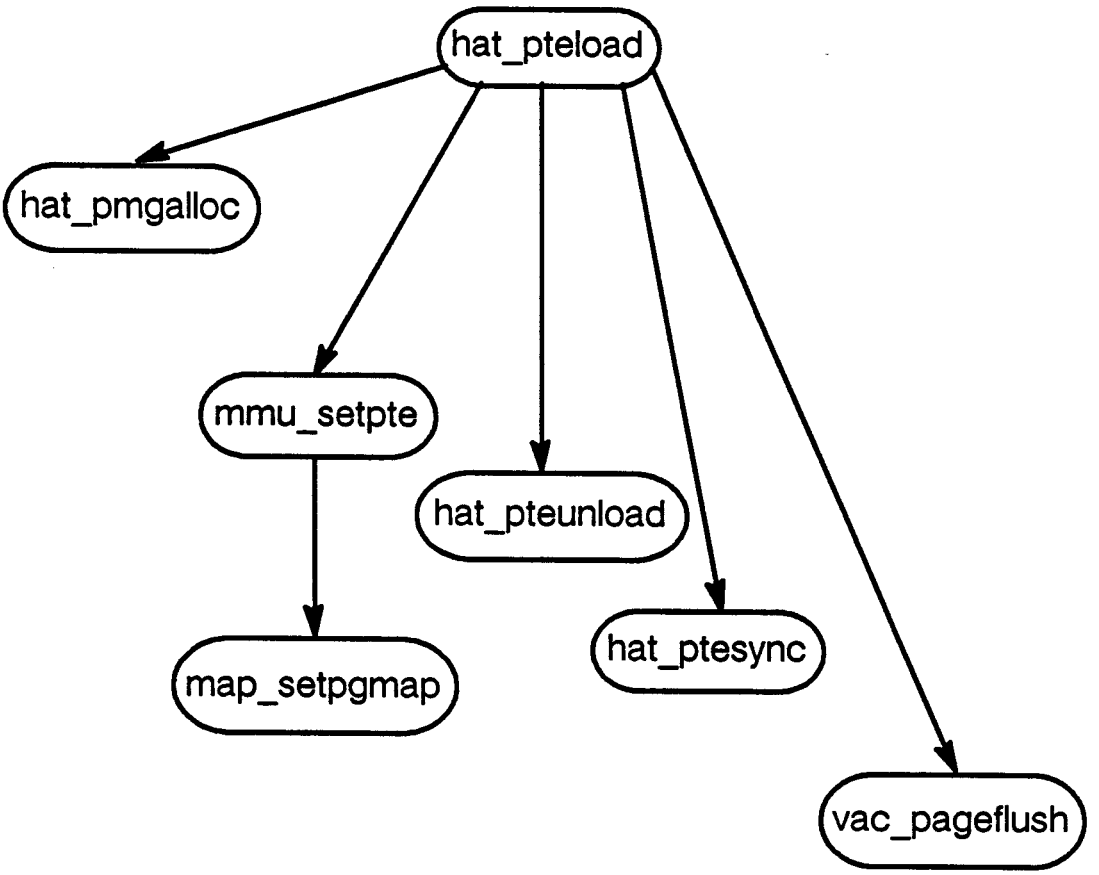

hat_memload
hat_mempte
hat_pteload
hat_vtop_prot
page_pptonum

hat_memload( )

hat_memload( ) is one of the machine independent hat routines that is used to map a particular address to a particular page with a set of protections. This routine calls *hat_mempte( )* which calls *hat_vtop_prot( )* which translates the protection specified in the vnode to protections that will be set in the pte structure.

hat_mempte( ) also sets the valid bit on, sets the modified and referenced bits to what they are in the page structure, sets type to OBMEM and sets the page frame number to what is returned by the *page_pptonum( )* routine. This is a simple routine that returns the page frame number corresponding to a given page. *hat_memload( )* then calls *hat_pteload( )*.

hat_pteload( )


hat_pteload
hat_pmgalloc
mmu_setpte
hat_pteunload
hat_ptesync
map_setpgmap
vac_pageflush

hat_pteload( )

hat_pteload( ) is the common routine called by the machine-independent hat routines *hat_memload( )* and *hat_devload( )* which will set the address in the segment to use the pte passed and then to map this pte to the page structure passed. *hat_pteload( )* initially calls *hat_pmgalloc( )*.

hat_pmgalloc( ) will allocate a pmgrp to map the specified address. It returns with the keepcnt incremented for the particular pmgrp used. It first looks for something in the free list and if there is none will steal one that is currently used. It uses a simple round-robin algorithm to find a used one to steal, skipping over the first few that are permanently allocated to the kernel and starting at pmgrpmin.

hat_pteload( ) makes sure that setting the pte values and then adding it to the list of mappings is atomic to it calls *splvm( )* to raise the interrupt priority level before doing this. If there is no page structure or if the virtual address cache is off (vac) or if the no-cache bit is on, it then sets the pte no-cache bit, sets the pme valid bit on and then calls *mmu_setpte( )* which calls *map_setpgmap( )* to set the pme corresponding to the base address to the software pte that was passed. In other words, the pte is loaded into the mmu or a translation is loaded.

Upon return to *hat_pteload( )*, it will check if the pme needs to be added to the list of pme's mapping to the same page. It sets the pointer in the pme to the address of the page structure and the pointer in the page structure to point to the pme. The memory claim for the address space is incremented. If the *vac* is active, then the routine will check for conflicts. A conflict exists if the new and existing mappings do not match in their "shm_alignment" fields and one of them is writeable. If conflicts exist, the extant mappings are flushed unless one of them is locked, in which case the mappings are flushed and deconverted into non-cacheable mappings by calling *hat_pteunload( )*. If conflicts are found, mappings to the current page are converted to non-cacheable by calling *hat_ptesync( )* after the appropriate bits are turned on in the pte and page structures. The routine *vac_pageflush( )* will then be called (in case of vac active) to ensure no protection mismatches between the cache and the mmu.

Module 7
Interprocess Communication – Sockets

Objectives: Upon completion of this module, the student will be able to:

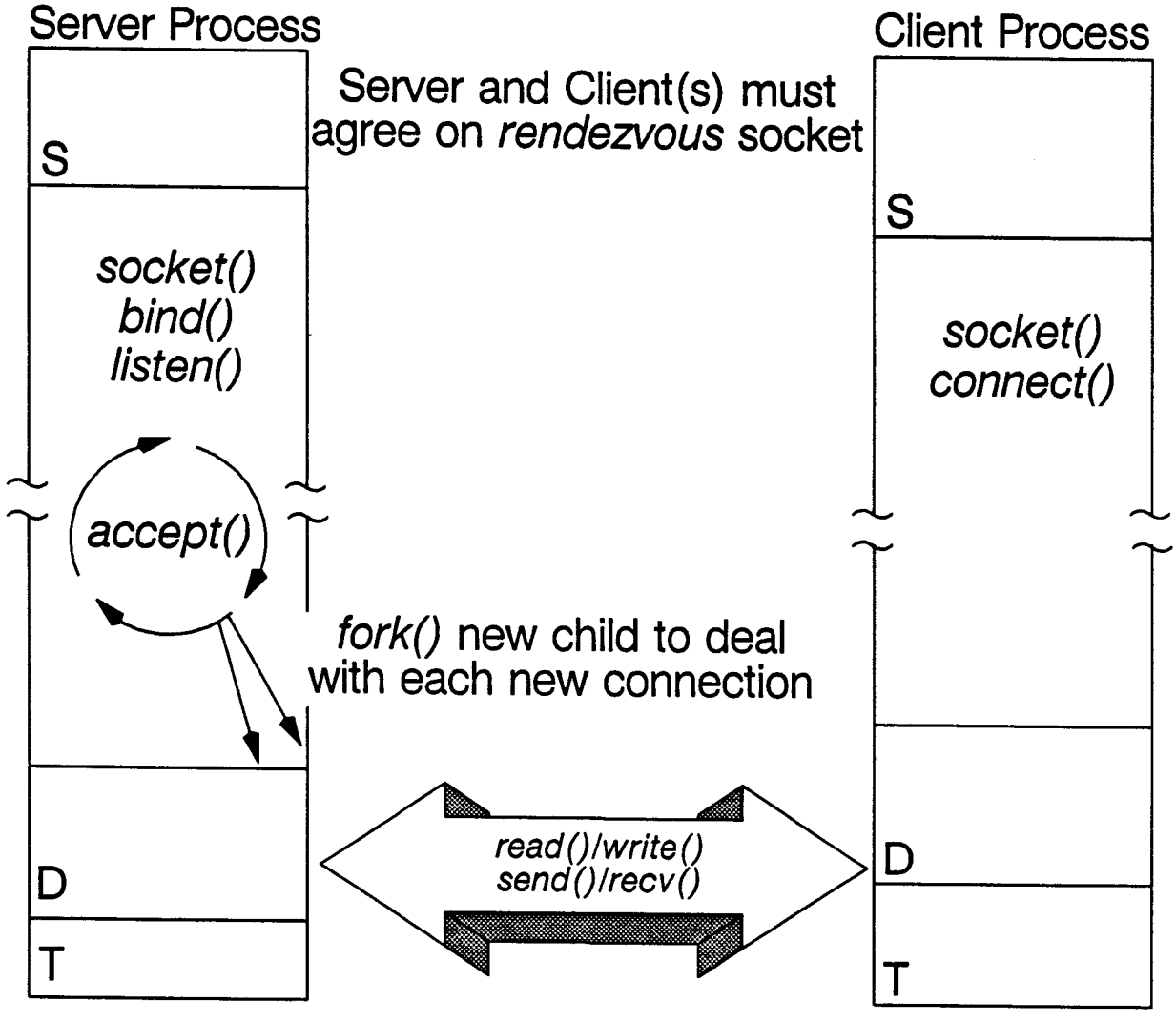
- Describe what happens in the system when the *socket( )*, *bind( )*, *connect( )*, *accept( )*, *send( )*, *recv( )* and other related system calls are executed.

- Describe the different structures that are used by the above mentioned system calls. This includes the socket, mbuf, protosw, unpcb, and domain structures.

- Become familiar with the different structures that are used by the above mentioned system calls. This includes the socket, mbuf, protosw, unpcb, and domain structures.

Server/Client Model

- Diagram illustrating Server/Client socket model:



Server/Client Model

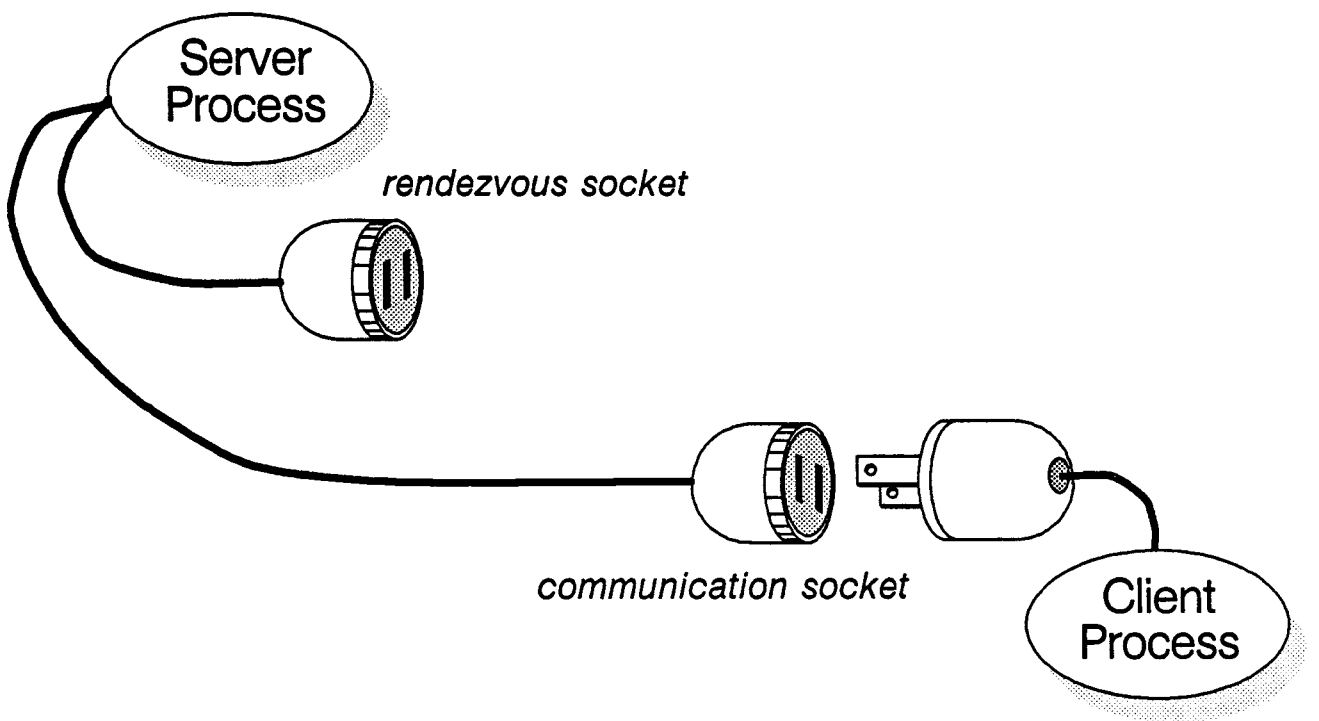
Sockets are generally used in a client–server model. The server and client(s) processes must agree on the *rendezvous socket's* address: a file in the UNIX domain (*AF_UNIX*), and an Internet address and a Port number in the Internet domain (*AF_INET*). With this model, the server executes *socket(), bind(), listen(), accept(),* and then loops around the *accept()*, *fork*ing a child process to deal with each connection. The client process just executes the *socket()* and *connect()* system calls. Once the client process is connected, I/O may take place using either the *read*() and *write()* (*SOCK_STREAM* only) system calls, or the newer, more specialized, *send()* and *recv()* system calls.

Sockets (BSD)

- Interprocess Communication; *sockets – socket(2)*

```
#include <sys/types.h>
#include <sys/socket.h>

sd = socket(domain, type, protocol)
int sd, domain, type, protocol;
```



Sockets (BSD)

socket() creates an endpoint for interprocess communication and returns a valid file (*socket*) descriptor *sd* upon success. This *socket* descriptor is then used in subsequent calls to *read(), write(), send(), recv()* etc.

The argument *domain* is the "Address Family" to be used (<*sys/socket.h*>):

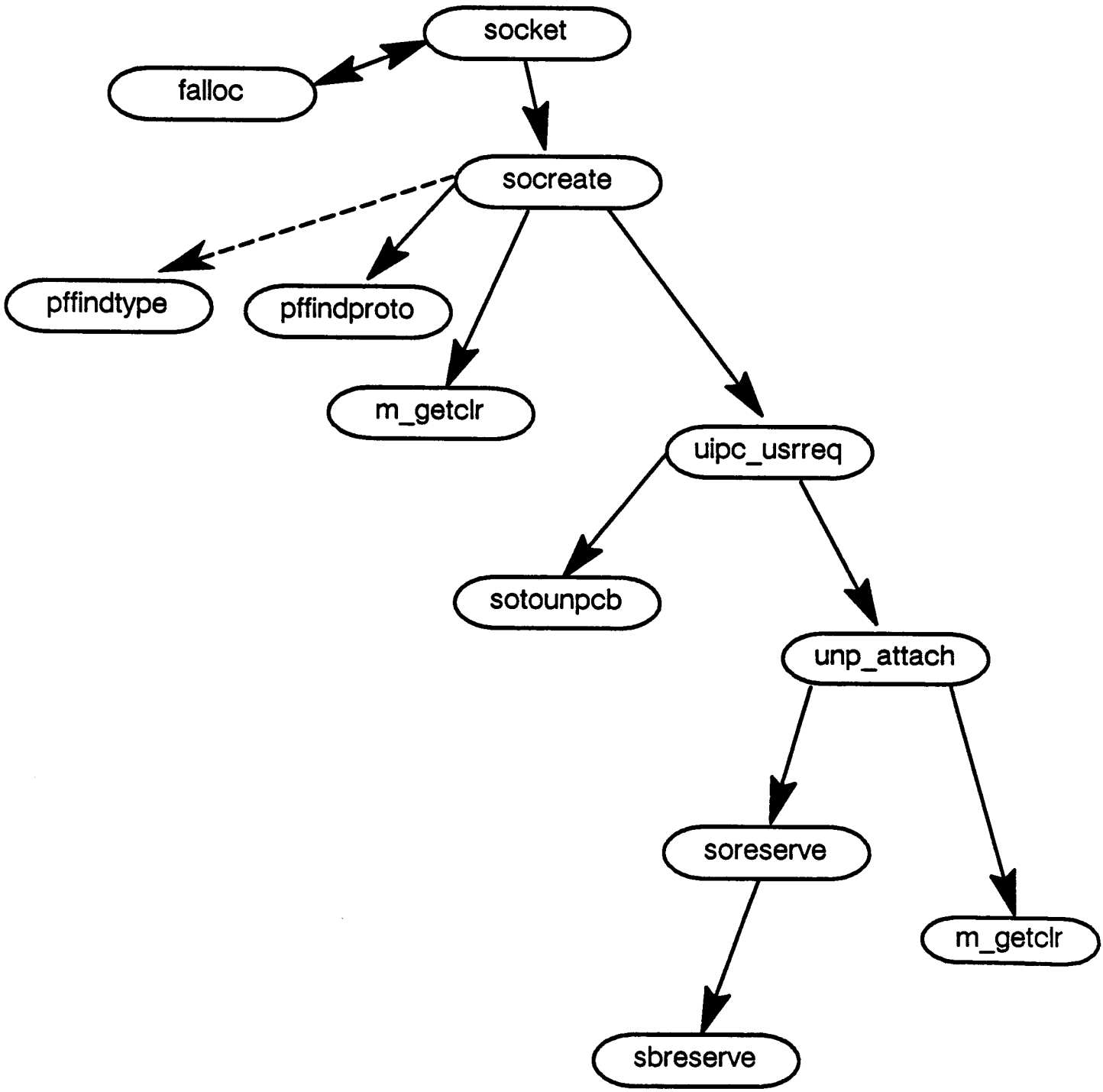
AF_UNSPEC	unspecified family, let system choose.
AF_UNIX	UNIX system internal protocols.
AF_INET	ARPA Internet protocols.

The *type* parameter specifies the type of socket (usually *SOCK_STREAM*):

SOCK_STREAM	sequenced, reliable, 2-way, byte stream; *SOCK_STREAM* socket protocols insure that data is not lost or duplicated.
SOCK_DGRAM	unreliable, connectionless, fixed-length, usually small messages (less overhead).

The *protocol* parameter is usually zero, because there is typically only one possibility within the given *domain–type* pair.

socket( )

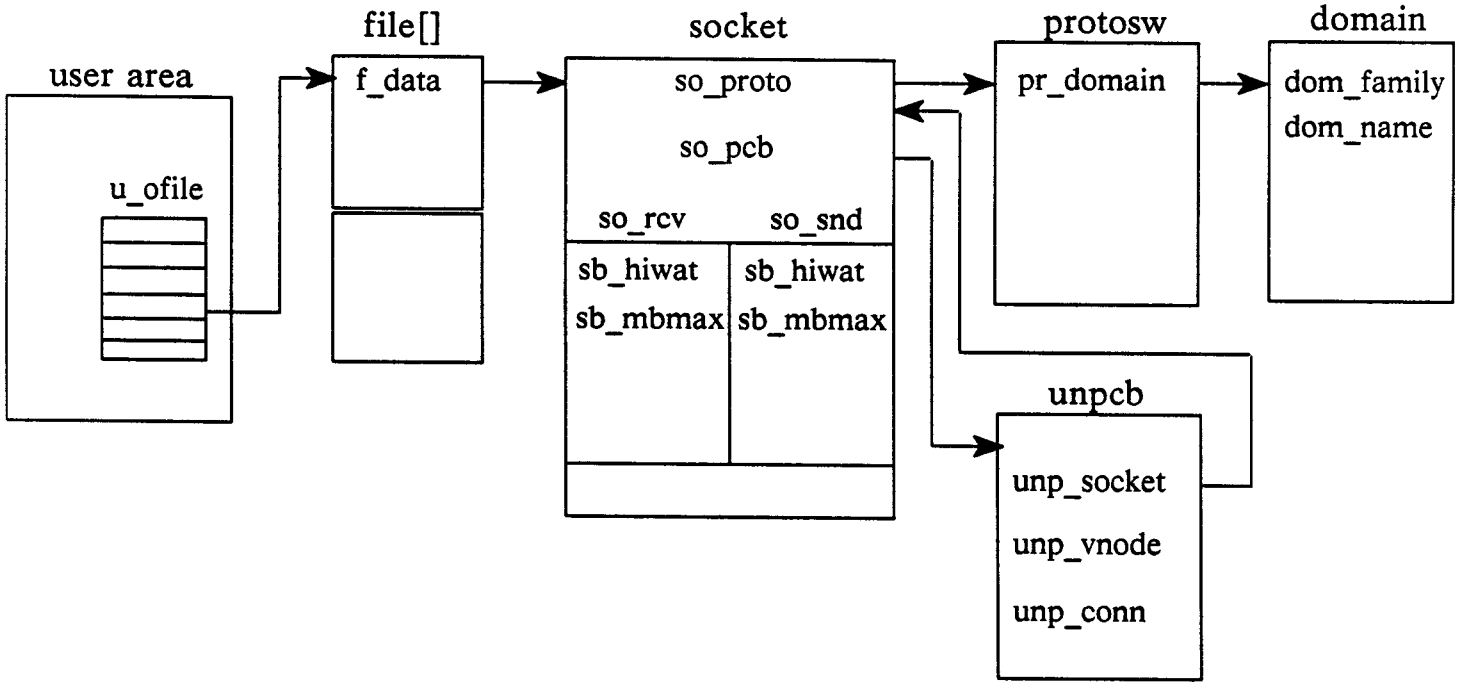

socket
falloc
socreate
pffindtype
pffindproto
m_getclr
uipc_usrreq
sotounpcb
unp_attach
soreserve
m_getclr
sbreserve

The socket( ) system call

The *socket()* routine calls *falloc()* to allocate a file[] entry and *falloc()* calls *ufalloc()* to allocate an entry in the descriptor table in the user area of the process which is set to point to the file[] entry. The flag in the file[] entry is set to FREAD and FWRITE and the type to DTYPE_SOCKET. *socket()* then calls the *socreate()* routine which calls *pffindproto()* or *pffindtype()* depending on whether we have specified a particular protocol within the domain. In either case, what is returned is a pointer to the protosw structure representing that protocol. If the type specified in the protosw structure is not the same as the type passed to socket(e.g. SOCK_STREAM), then return with an error. If there are no errors, the *m_getclr()* routine is called to allocate an mbuf cleared with zeroes for the socket structure. The socket is then initialized, with the type set to the type parameter, the state to SS_PRIV (broadcast privilege in case of root) and the pointer to the protosw structure. It then calls the protocol dependent *pr_usrreq()* routine with the PRU_ATTACH request.

uipc_usrreq() calls *sotounpcb()* to get the pointer to the protocol control block which should be null at this time. It then switches on the request and for PRU_ATTACH will call the *unp_attach()* routine which then calls *soreserve()* which calls *sbreserve()* which sets the maximum limits for mbufs in the sockbuf structures for the send and receive side of the socket. *unp_attach()* then calls *m_getclr()* to allocate an mbuf for the protocol control block for this socket. *unp_attach()* then sets the pointers so that the socket points to the protocol control block and the protocol control block points to the socket.

After socket()



After socket()

After the *socket( )* system call, a socket structure is allocated which is pointed to by the file[] entry that was also allocated. The socket structure is initialized so that the appropriate member is pointing to the protosw structure representing a particular protocol and this protosw structure in turn points to a domain structure representing the family.

Socket Structure

```
struct socket {
        short     so_type;
        short     so_options;
        short     so_linger;
        short     so_state;
        caddr_t   so_pcb;
        struct    protosw *so_proto;

        struct    socket *so_head;
        struct    socket *so_q0;
        struct    socket *so_q;
        short     so_q0len;
        short     so_qlen;
        short     so_qlimit;
        short     so_timeo;
        u_short   so_error;
        short     so_pgrp;
        u_long    so_oobmark;
```

Socket Structure

```
struct socket {
        short    so_type;        /* generic type, see socket.h */
        short    so_options;     /* from socket call, see socket.h */
        short    so_linger;      /* time to linger while closing */
        short    so_pcb;         /* internal state flags SS_*, below */
        caddr_t  so_pcb;         /*  protocol control block */
        struct    protosw *so_proto;     /*protocol handle */

        struct socket *so_head;  /* back pointer to accept socket */
        struct socket *so_q0;    /* queue of partial connections */
        struct socket *so_q;     /* queue of incoming connections */
        short  so_q0len;         /* partials on so_q0 */
        short  so_qlen;          /* number of conections on so_q */
        short  so_qlimit;        /* max number queued connections */
        short  so_timeo;         /* connection timeout */
        u_short so_error;        /* error affecting connection */
        short  so_pgrp;          /* pgrp for signals */
        u_long so_oobmark;       /* chars to oob mark */
/*
 * Variables for socket buffering.
*/
        struct  sockbuf {
                u_long  sb_cc;  /* actual chars in buffer */
                u_long  sb_hiwat;        /* max actual char count */
                u_long   sb_mbcnt;       /* chars of mbufs used */
                u_long  sb_mbmax;        /* max chars of mbufs to use */
                u_long  sb_lowat;        /* low water mark (not used yet ) */
                struct  mbuf *sb_mb;     /* the mbuf chain */
                struct  proc *sb_sel;    /* process selecting read/write */
                short  sb_timeo;         /* timeout (not used yet) */
                short sb_flags;          /* flags, see below */
```

Socket Structure

```
/*
 * Variables for socket buffering.
*/
        struct    sockbuf {
                  u_long  sb_cc;
                  u_long  sb_hiwat;
                  u_long  sb_mbcnt;
                  u_long  sb_mbmax;
                  u_long  sb_lowat;
                  struct    mbuf *sb_mb;
                  struct    proc   *sb_sel;
                  short     sb_timeo;
                  short     sb_flags;
        } so_rcv, so_snd;
#define SB_MAX          (64*1024)
#define SB_LOCK         0x01
#define SB_WANT         0x02
#define SB_WAIT         0x04
#define SB_SEL          0x08
#define SB_COLL         0x10

        struct    wypalt *so_wupalt;
};
```

Socket Structure

```
        } so_rcv, so_snd;
#define SB_MAX          (64*1024)       /* max chars in sockbuf */
#define SB_LOCK         0x01    /* lock on data queue (so_rcv only) */
#define SB_WANT         0x02    /* someone is waiting to lock */
#define SB_WAiT         0x04    /* someone is waiting for data space */
#define SB_SEL          0x08    /* buffer is selected */
#define SB_COLL         0x10    /* collision selecting */
/*
 * Hooks for alternative wakeup strategies.
 * These are used by kernel subsystems wishing to access the socket
 * abstraction.  If so_wupfunc is nonull, it is called in place of wakeup any
 * time that wakeup would otherwise be called with an argument whose
 * value is an address lying within a socket structure.
 */
        struct wupalt *so_wupalt;
};
```

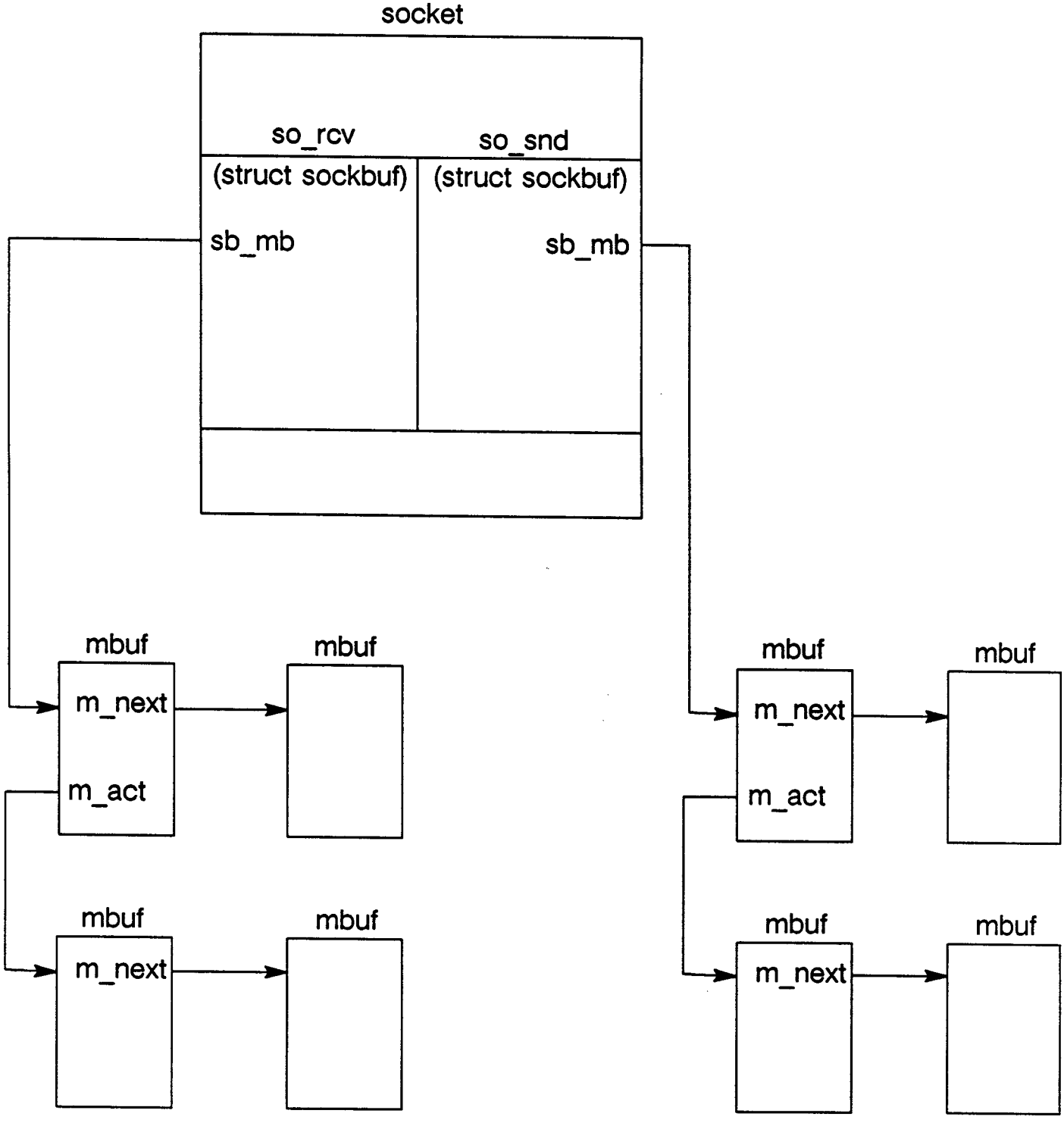
Mbuf Structure

```
struct mbuf {
        struct    mbuf *m_next;
        u_long    m_off;
        short     m_len;
        short     m_type;
        union {
                u_char   mun_dat[MLEN];
                struct {
                        short   mun_cltype;
                        int     (*mun_clfun) ();
                        int     mun_clarg;
                        int     (*mun_clswp) ();
                } mun_cl;
        } m_un;
        struct    mbuf *m_act;
};
```

Mbuf Structure

```
struct mbuf {
        struct   mbuf *m_next;          /* next buffer in chain */
        u_long  m_off;                  /* offset of data */
        short   m_len;                  /* amount of data in this mbuf  */
        short   m_type;                 /* mbuf type (0 == free) */
        union {
                u_char  mun_dat[MLEN];          /* data storage */
                struct {
                        short  mun_cltype;      /* "cluster" type */
                        int    (*mun_clfun) ();
                        int    mun_clarg;
                        int    (*mun_clswp) ();
                } mun_cl;
        }m_un;
        struct   mbuf *m_act;           /* link in higher-level mbuf list */
};
```

The socket, sockbuf and mbuf structures



The socket, sockbuf, and mbuf structures

The opposite diagram shows the relationship among the socket, sockbuf and mbuf structures. There are two sockbuf structures embedded in the socket, one for the receive side and one for the send side. The sockbuf structures contain the pointers to the mbuf structures representing the messages that are sent and received at the socket.

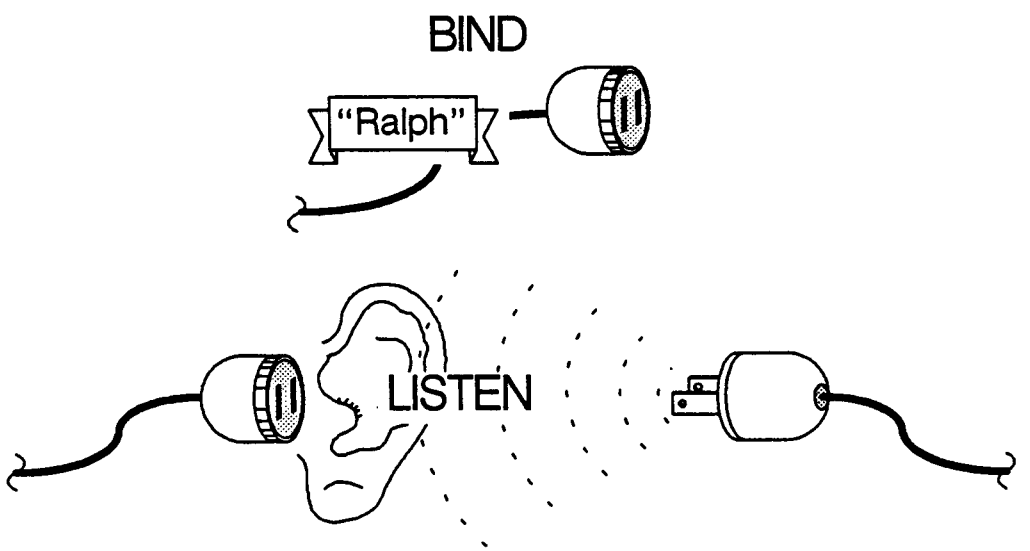
Socket Functions

- Ancillary socket functions:

bind(*sd, name, namelen*)
listen(*sd, backlog*)
int sd, namelen;
int backlog;
*struct sockaddr *name;*

- Structure definition for *sockaddr*:

```
struct sockaddr {
        u_short  sa_family;          /* address family */
        char     sa_data[14];        /* path */
};
```



Socket Functions

bind() assigns a name (or address) to an unnamed *socket*. When a *socket* is created with the *socket(2)* call, it exists in a "name space" but has no name assigned. The argument *sd* is a valid *socket* descriptor. For UNIX domain sockets, the name string pointed to by *name* is assigned to the *socket*. The parameter *namelen* specifies the absolute length of the name string, plus the size of a short integer (2 bytes):

(sizeof(name.sa_data) + sizeof(u_short))

If a name longer than 14 bytes is needed, then the file <*sys/un.h*> should be included, and the structure *struct sockaddr_un* used (which has room for strings up to 108 bytes), instead of the structure *struct sockaddr:*

```
struct sockaddr_un {
        short   sun_family;             /* AF_UNIX */
        char    sun_path[108];          /* path name */
};
```

For the INTERNET domain, the structure *sockaddr_in* is used <*netinet/in.h*>:

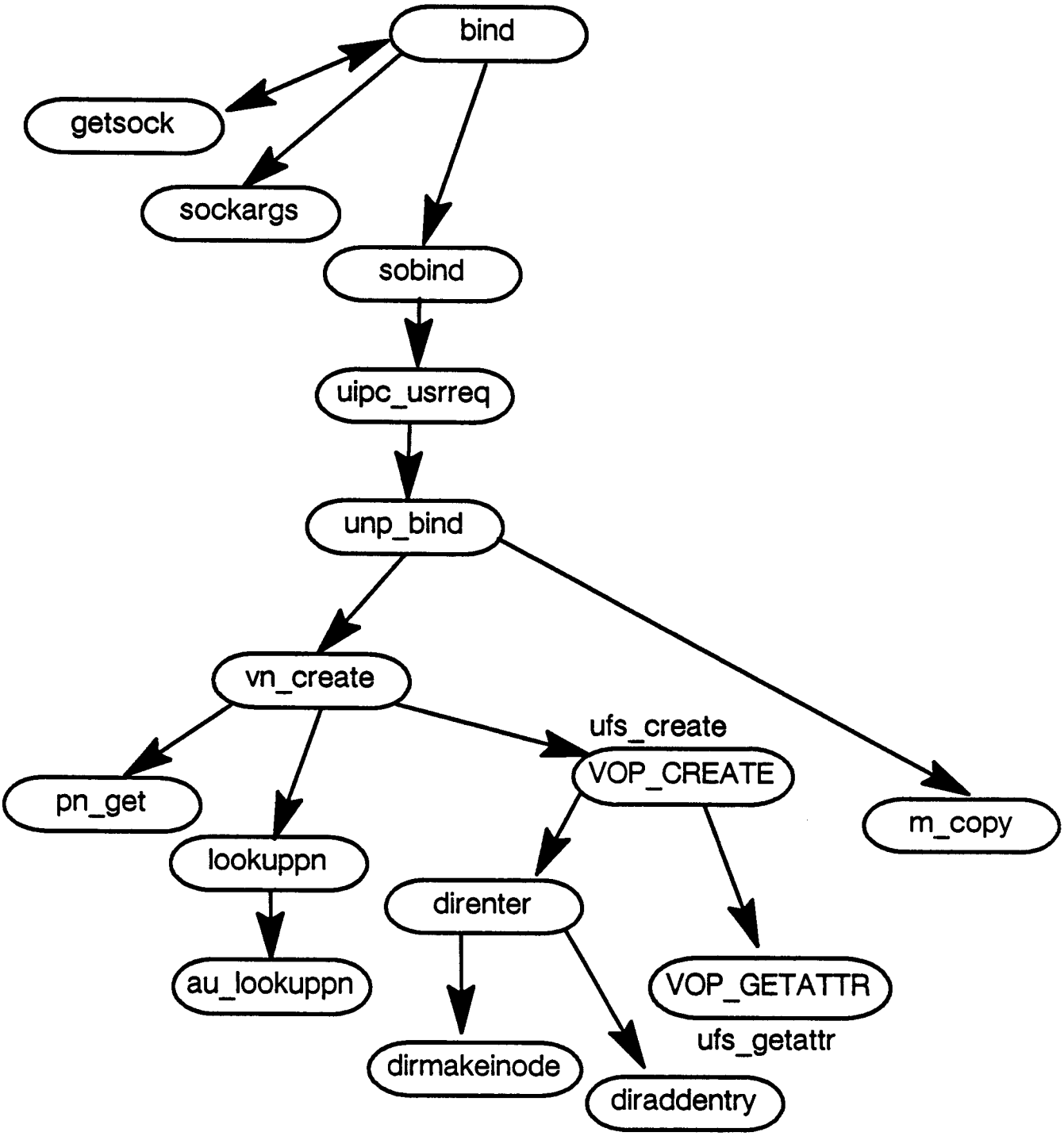
```
struct sockaddr_in {
        short   sin_family;             /* AF_INET */
        char    sin_port;               /* port number */
        struct  in_addr  sin_addr;      /* address structure */
        char    sin_zero[8];
};
```

The return value from a successful call to *bind()* is 0; otherwise –1.

NOTE: In the UNIX domain, binding a name creates a named *socket* in the file system, which must be physically deleted (when no longer required) by *unlink(2)* or *rm(1)*.

listen() is used to listen for connections on a *socket*. The argument *sd* is a valid *socket* descriptor returned from *socket(2)*, and the argument *backlog* defines the maximum length of the queue of pending connections (current maximum is silently enforced at 5). The *listen()* call applies only to *sockets* of type *SOCK_STREAM.* The return value from a successful call to *listen()* is 0; otherwise –1.

bind( )


bind
getsock
sockargs
sobind
uipc_usrreq
unp_bind
vn_create
ufs_create
VOP_CREATE
pn_get
lookuppn
direnter
m_copy
au_lookuppn
VOP_GETATTR
ufs_getattr
dirmakeinode
diraddentry

bind( )

bind( ) calls *getsock( )* to return the file pointer from the socket descriptor and then calls *sockargs( )* to copy the name into an mbuf structure of type MT_SONAME. It then calls *sobind( )*.

sobind( ) simply calls *splnet( )* to raise the interrupt priority level and then calls *uipc_usrreq( )* with the PRU_BIND request which results in the *unp_bind( )* routine being called.

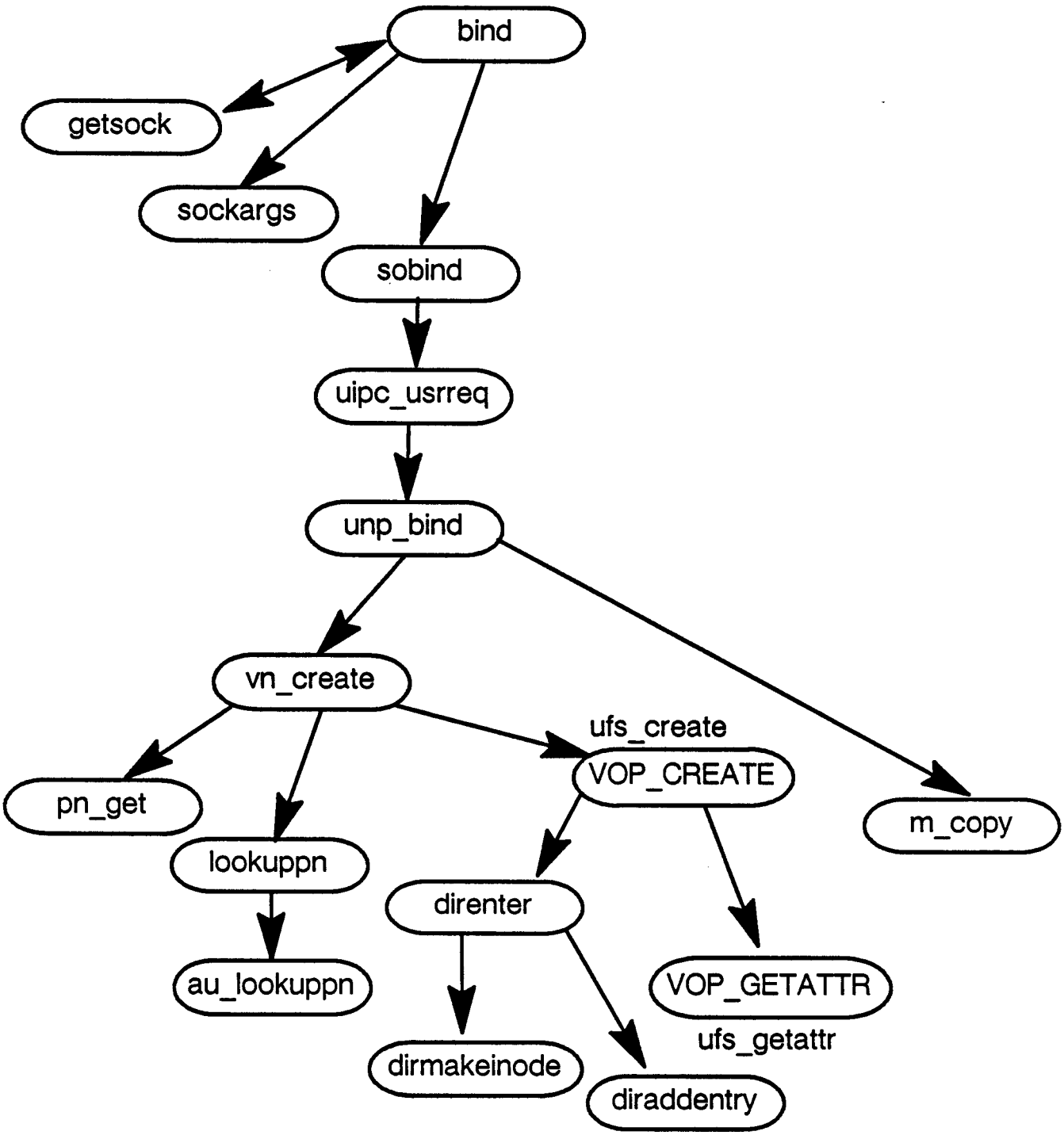
unp_bind( ) extracts the socket address from the mbuf to which the name or the address was copied to earlier by *sockargs( )*. If the protocol control block pointer to the vnode is not null or if the data length in the mbuf is too long then *unp_bind( )* returns with an error.

If everything is still fine, it will set the contents of the mbuf at offset plus the length to zero to indicate the end of the address. Then it will set the attributes in the attribute structure to null initially and then set the type to socket and the mode with all permissions.

It then calls *vn_create( )* with the exclusive flag to create the vnode for the socket so that it can exist in the file system (essentially a named socket). *vn_create( )* calls *pn_get( )* to pull the pathname and to allocate a buffer that will be used for pathname resolution. *vn_create( )* will then call *lookuppn( )* which calls *au_lookuppn( )* to return the vnode of the parent directory where the node will be created.

Then the file system dependent vnode create routine *ufs_create( )*, will be called to create the entry. *ufs_create( )* calls direnter( ) to create the new directory entry. direnter( ) verifies if we are creating and if we have write permissions in the directory, and if so we call *dirmakeinode( )* to create the inode and then *diraddentry( )* to add the entry to the directory.

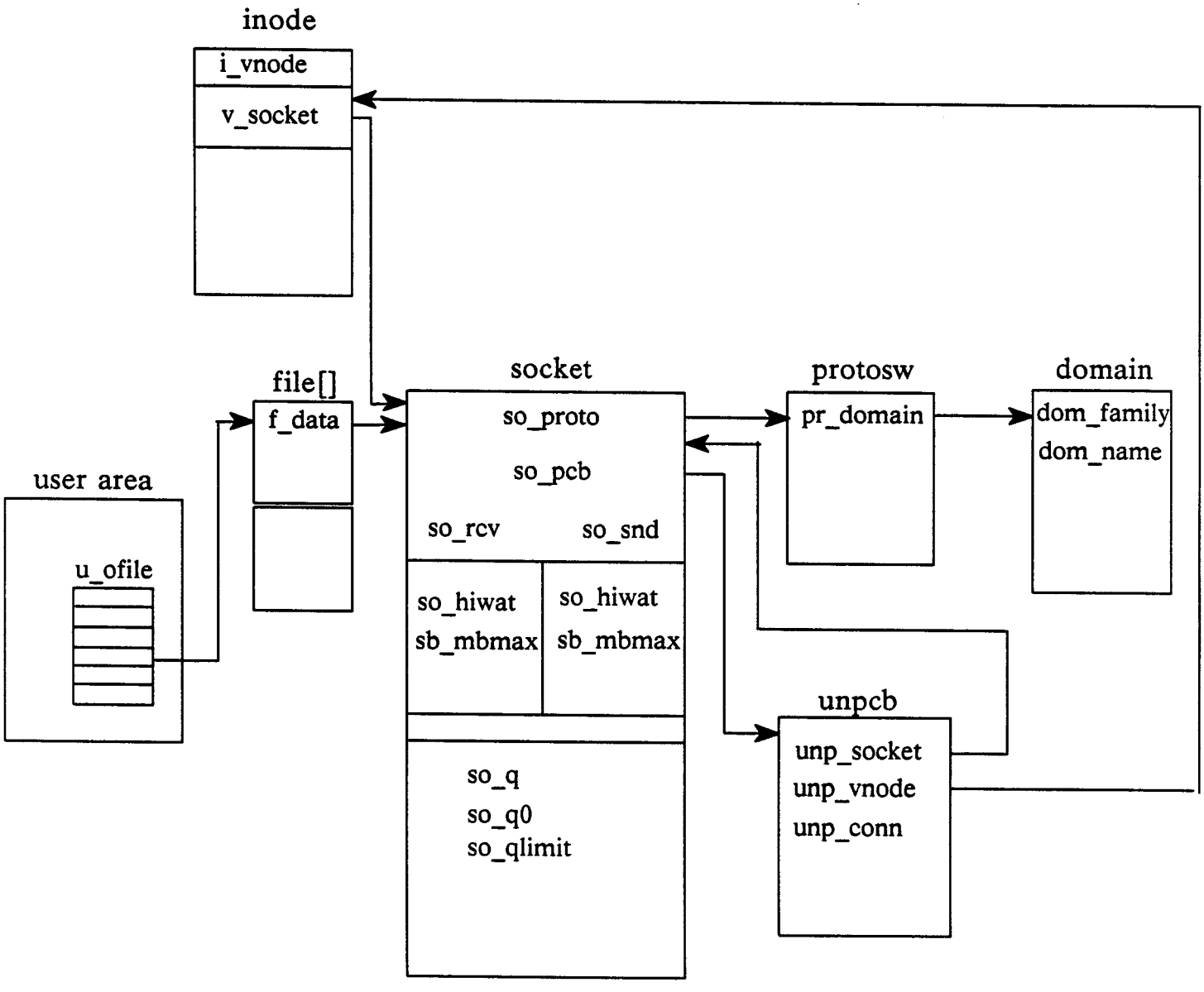
bind( )


bind
getsock
sockargs
sobind
uipc_usrreq
unp_bind
vn_create
ufs_create
VOP_CREATE
pn_get
lookuppn
m_copy
direnter
au_lookuppn
VOP_GETATTR
ufs_getattr
dirmakeinode
diraddentry

bind( ) (cont)

When *dirmakeinode( )* creates the inode, it gets the type and the mode field from the attributes structure, so that this inode's embedded vnode's type is set to VSOCK. Then *direnter( )* returns successfully to *ufs_create( )* which then calls *ufs_getattr( )* which sets the attribute structure for the socket with values coming from the inode. The attribute structure's va_nodeid is set to the inode number.

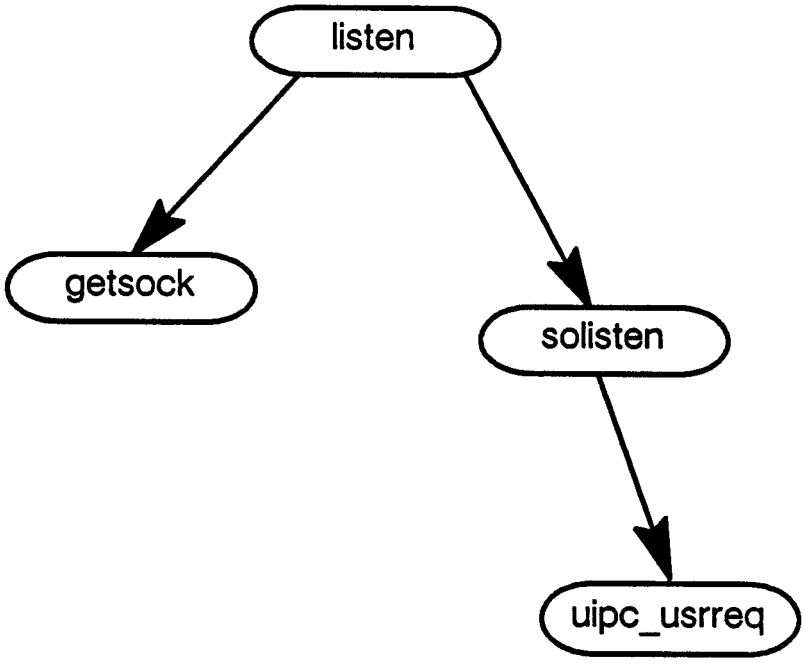
After socket(), bind()



After socket( ), bind( )

The diagram on the opposite page shows what was allocated after *bind( )* was called. We show an inode created for the socket entry and this inode contains the embedded vnode for the socket.

listen( )



The listen( ) system call

The *listen()* routine first calls *getsock()* which returns the file pointer corresponding to the descriptor. *Listen()* then calls *solisten()* with the pointer to the socket structure(obtained from the file[] entry) and the maximum number of pending connections as parameters.

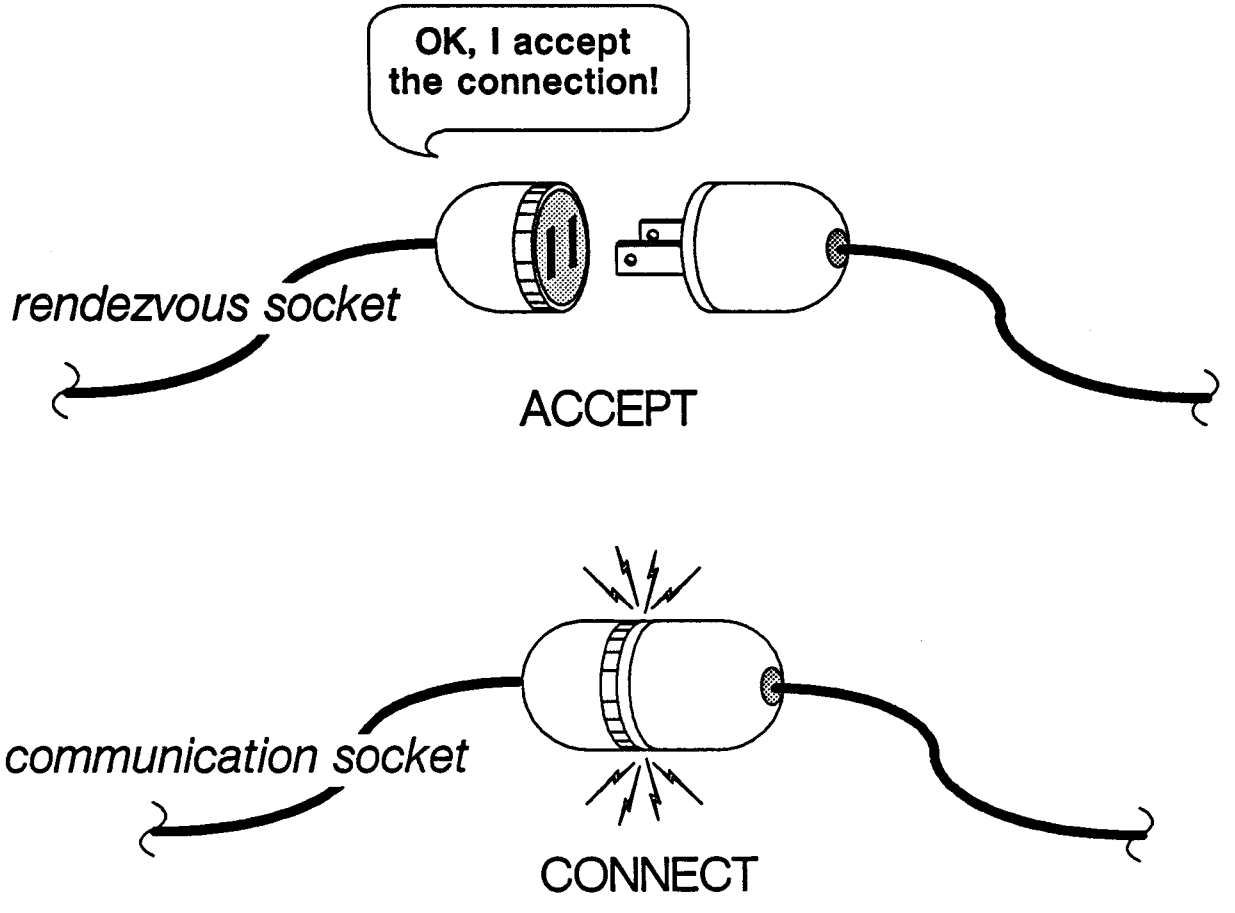
solisten() calls the protocol independent routine *pr_usrreq()* with the PRU_LISTEN request passed as one of the parameters. For AF_UNIX this routine points to the *uipc_usrreq()* and this routine simply switches on the request and for PRU_LISTEN, it simply checks that the vnode pointer in the protocol control block is not zero and it should not be if we have done a *bind()* system call previously.

After this routine returns to *solisten()*, *solisten()* initializes the socket members with the socket queue pointers set to point to itself, the options set to SO_ACCEPTCONN, and the maximum number of queued connections to the lower of SOMAXCONN or the parameter passed.

Socket Functions

- More ancillary socket functions:

ns = **accept**(*sd, addr, addrlen*)
connect(*sd, name, namelen*)
*int sd, ns, namelen *addrlen;*
*struct sockaddr *name, *addr;*



Socket Functions

accept() is the call that accepts a connection on a *socket*. The argument *sd* is a *socket* descriptor that was created using *socket(2)*, bound to an "address" using *bind(2)*, and is listening with *listen(2)*. The argument *addr* is a result parameter which is filled with the address of the connecting entity (the identity of the "client process"). *addrlen* is a value–result parameter which will initially contain the amount of space pointed to by *addr*, and on return will contain the actual length in bytes of the address returned. The accepted *socket* (*ns*) is used to read and write data to and from the *socket* which connected to *sd*, and is NOT used for accepting any more connections. The original *socket sd* remains open for accepting more connections. The *socket* passed to *accept()* is termed the "*rendezvous socket*", and the *socket* returned is called the "*communication socket*". In practice, client processes try to connect to the *rendezvous socket*, and, when connected, the server process communicates with the client process through the *communication socket* (this behavior is transparent to the client). The return value on a successful call is a descriptor for the accepted *socket*, otherwise a –1.

NOTE: If no connections are pending, *accept()* blocks until one comes in.

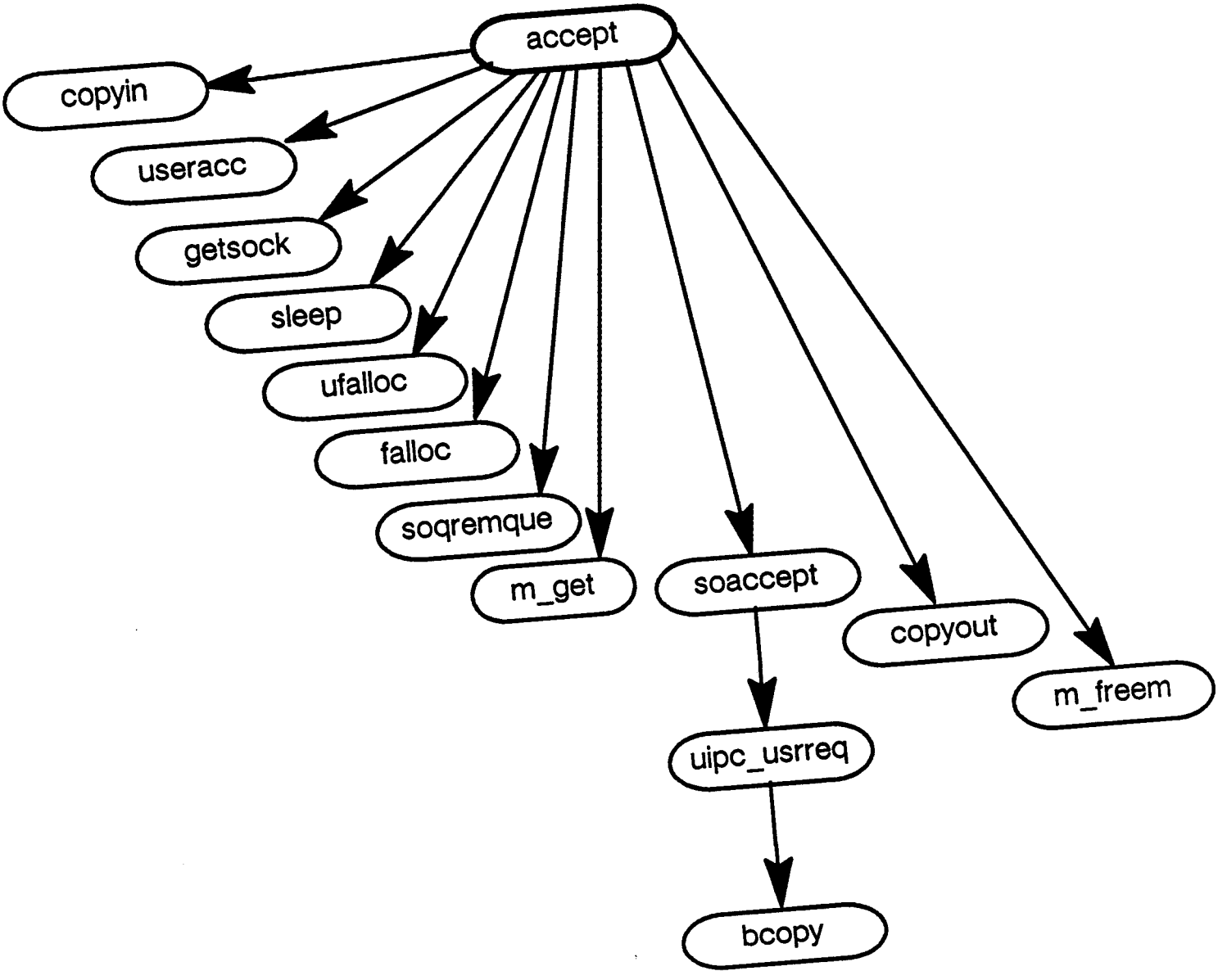
connect() initiates a connection on a *socket* (server's *rendezvous socket*). The parameter *sd* is a valid *socket* descriptor. If the *socket* is of type *SOCK_DGRAM*, then the *name* parameter specifies the peer with which the *socket* is to be associated; i.e. the address is that to which datagrams are to be sent, and the (only) address from which datagrams can be received. If the *socket* is of type *SOCK_STREAM*, then the call attempts to make a connection to another socket, and *name* is the name of the other socket to connect to. *namelen* is the same as that for *bind(2)*. This call will block until a connection is made.

Datagram *sockets* can be disconnected by using *connect()* and specifying a NULL or invalid address (*name*).

The return value from a successful call to *connect()* is a zero; otherwise –1.

Sun Education
Not to be copied

accept( )



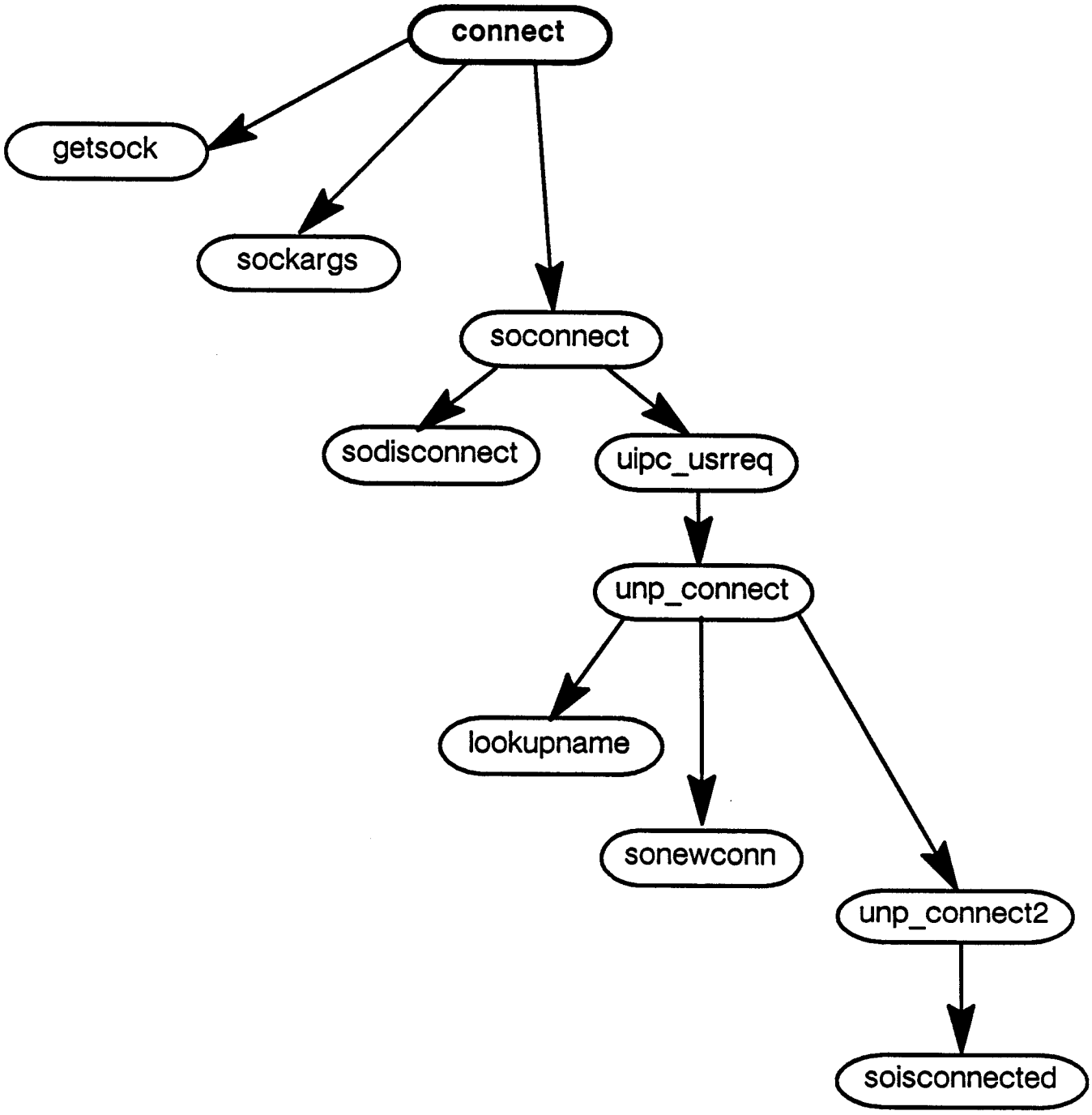
accept( ) system call

The *accept()* routine performs some error checking like making sure that a valid name has been passed as a parameter, and then calls *copyin()* to copy the name length parameter into another variable that will be manipulated later on. It then calls *useracc()* to make sure that the name passed has write permission. It calls *getsock()* to return the file pointer and from the private structure pointed by the file structure (socket) determines if the options are set for accepting a connection, and if the state indicates non–blocking operations, in which case if the number of connections is equal to zero, will return with an error immediately. Otherwise, it will simply go into a while loop waiting for connections to be established and calling *sleep( )* if there are none.

If it is awakened from its sleep because a connection has been initiated, it checks for errors and if there are none calls *ufalloc()* to allocate a descriptor and *falloc()* to allocate a file[] entry. Then *accept()* calls *soqremque()* with a second parameter of 1 which means that it should get the socket from the queue of incoming connections and not from the partial connections. *soqremque()* then takes out the socket from the queue of incoming connections, decrements the count, and zeroes out the pointers in the selected socket and returns to *accept()*.

accept() then calls *m_get()* to allocate an mbuf for the socket name and then calls soaccept() with the socket and the mbuf as parameters. *soaccept()* makes sure that the SS_NOFDREF flag is not off yet as it will turn this flag off in the socket state. It then calls *pr_usrreq()* with the PRU_ACCEPT request which will simply call *bcopy()* to pass back to the client the name of the connected socket if it is still bound and connected. It then returns to *accept()* which then calls *copyout()* to copy the name and the size into the original parameters passed to *accept()* and then calls *m_freem()* to free the mbuf.

connect( )


connect
getsock
sockargs
soconnect
sodisconnect
uipc_usrreq
unp_connect
lookupname
sonewconn
unp_connect2
soisconnected

The connect( ) system call

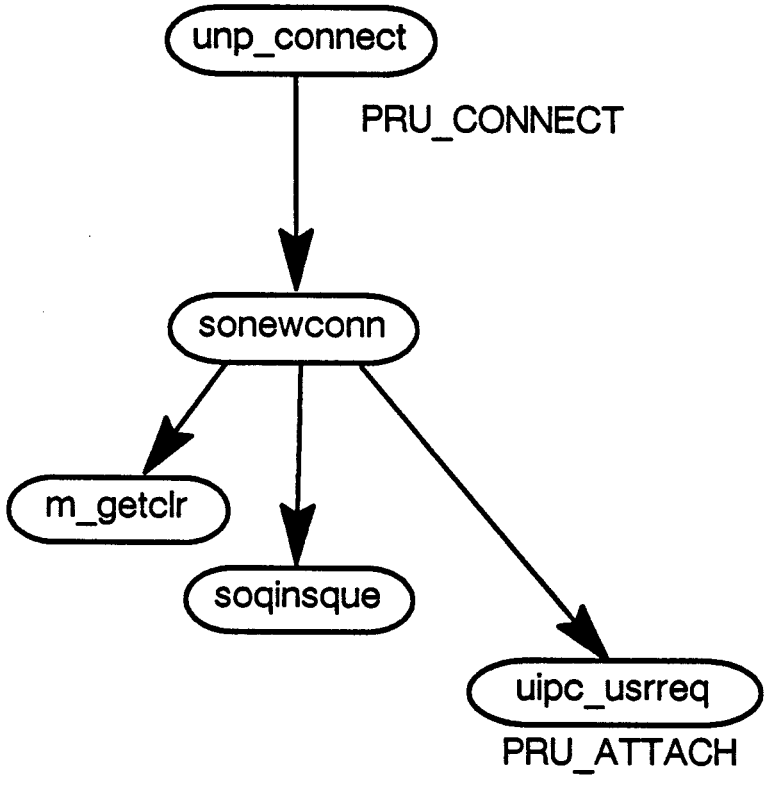
The *connect()* system call receives the descriptor, the name and the namelen as parameters. It calls *getsock()* to return the file pointer corresponding to the descriptor and extracts the pointer to the socket from the file table entry. It then starts checking if the socket state is set to non-blocking(SS_NBIO) and if the state flag is also set to being connected(SS_ISCONNECTING) and if so, it sets the error flag and returns.

connect() then calls *sockargs()* to copy the name passed into an mbuf structure whose type is MT_SONAME. It then calls *soconnect()* with two parameters, the pointer to the socket and the mbuf structure which has the name.

soconnect() calls *splnet()* to protect what it is doing from interrupts as it will modify the socket queues. For a connection-based protocol such as the one for AF_UNIX, connection can only be done once and if the socket state shows that it is connected or is connecting already, then the *sodisconnect()* routine is called to allow the user to disconnect, otherwise the protocol dependent routine is called with the PRU_CONNECT request. For the AF_UNIX domain, this is *uipc_usrreq()* which calls the *unp_connect()* routine.

unp_connect() checks first if the name passed in the mbuf is within the legal limit and then it calls the *lookupname()* that searches for the name that is passed to *connect()* and gets the corresponding vnode pointer. If the vnode type shows it is not type socket, then it will set the error flag, release the vnode and return. *unp_connect()* then checks if the vnode points to a valid socket structure, if not it means the connection was refused and again the error flag is set, the vnode is released and the routine returns. If there is a valid socket pointer, it then checks if the types for the two sockets, the one being held by the client that issued the connect system call and the one returned from the server are the same. It then calls the *sonewconn()* routine with the returned socket as the only parameter.

sonewconn( )


unp_connect
PRU_CONNECT
sonewconn
m_getclr
soqinsque
uipc_usrreq
PRU_ATTACH

sonewconn()

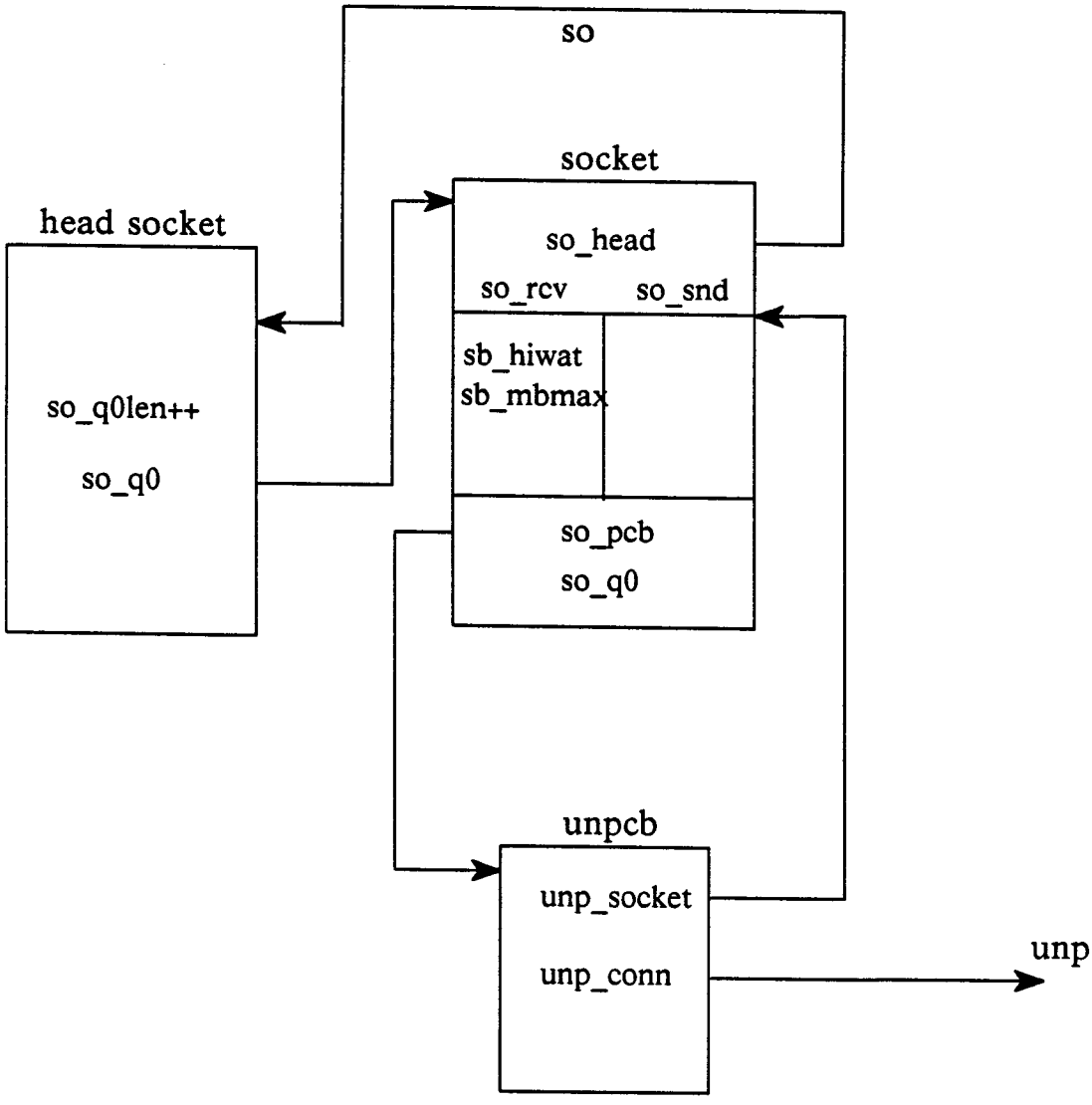
sonewconn() is called when a new connection on a socket is attempted. This routine calls *m_getclr()* to allocate an mbuf of type MT_SOCKET which will be used for the connection on the server side. It is noteworthy to point out that the original socket created in the server does not get used for communication purposes but for control and administrative purpose. Therefore this need for more sockets to be allocated when new connections are requested by clients. The mbuf or socket structure is initialized with information from the original socket with some exceptions, like the SO_ACCEPTCON option is turned off and SS_NOFDREF state flag is turned on. *sonewconn()* then calls *soqinsque()* with the original socket, the new socket and a flag of zero as parameters.

soqinsque() will add the socket to the head of the so_q0 queue of the original socket if this is a new, or what is termed a partial, connection. Otherwise, it will be added to the so_q which is the queue of incoming connections. The respective counts are also updated.

sonewconn() then calls the protocol dependent routine with the PRU_ATTACH request to attach the socket, allocate a process control block and to allot mbufs to this new socket. It then returns to *unp_connect()* which then calls *unp_connect2()* with the client socket and the newly allocated server socket as parameters.

unp_connect2() checks the socket types and sets the pointer in the client protocol control block to the address of the protocol control block of the new socket. The pointer in the protocol control block of the new socket is then set to the address of the protocol control block of the client socket. It then calls *soisconnected()* twice, once with the new socket as the parameter and another time with the client socket.

After connect( )



After Connect( )

After the *connect( )* system call is issued by the client process, a connection is established between the socket of the client and the socket of the server process.

Socket I/O

- *Socket* data transmission functions – *send(2), sento(2), sendmsg(2)*:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(sd, msg, len, flags)
int sendmsg(sd, msg_s, flags)
int sd, len, flags, tolen;
char *msg;
struct sockaddr *to;
struct msghdr *msg_s;
```

- Structure definition for *struct msghdr*:

```
struct msghdr {
    caddr_t  msg_name;        /* optional address */
    int      msg_namelen;     /* size of address */
    struct iovec *msg_iov;    /* scatter/gather array */
    int      msg_iovlen;      /* # of msg_iov indices */
    caddr_t  msg_accrights;   /* access, sent/recv */
    int      msg_accrightslen;
};
```

Socket I/O

send() is a system call used to transmit a message to another *socket. send()* may only be used when the *socket* to send to is in a *connected* state. The argument *sd* is a valid *socket* descriptor, the argument *msg* is the string containing the message to be sent, the argument *len* is the length of the string pointed to by *msg,* and the argument *flags* is built using one or more of the following masks (defined in the include file *<sys/socket.h>*):

MSG_OOB . . . . . send "out-of-band" data on *sockets* that support it (*SOCK_STREAM*).

MSG_DONTROUTE . . . this option will be turned on for the duration of the operation, typically used only by diagnostic or routing programs.

sendto() is also used to transmit a message to another socket specified by the argument *sd*. The arguments *msg, len,* and *flags* are the same for *send(2)*. Unlike *send()*, this function can send a message to a socket that has NOT been connected by specifying the *socket* address in the *struct sockaddr* argument *to,* with the length of the message in *tolen*.

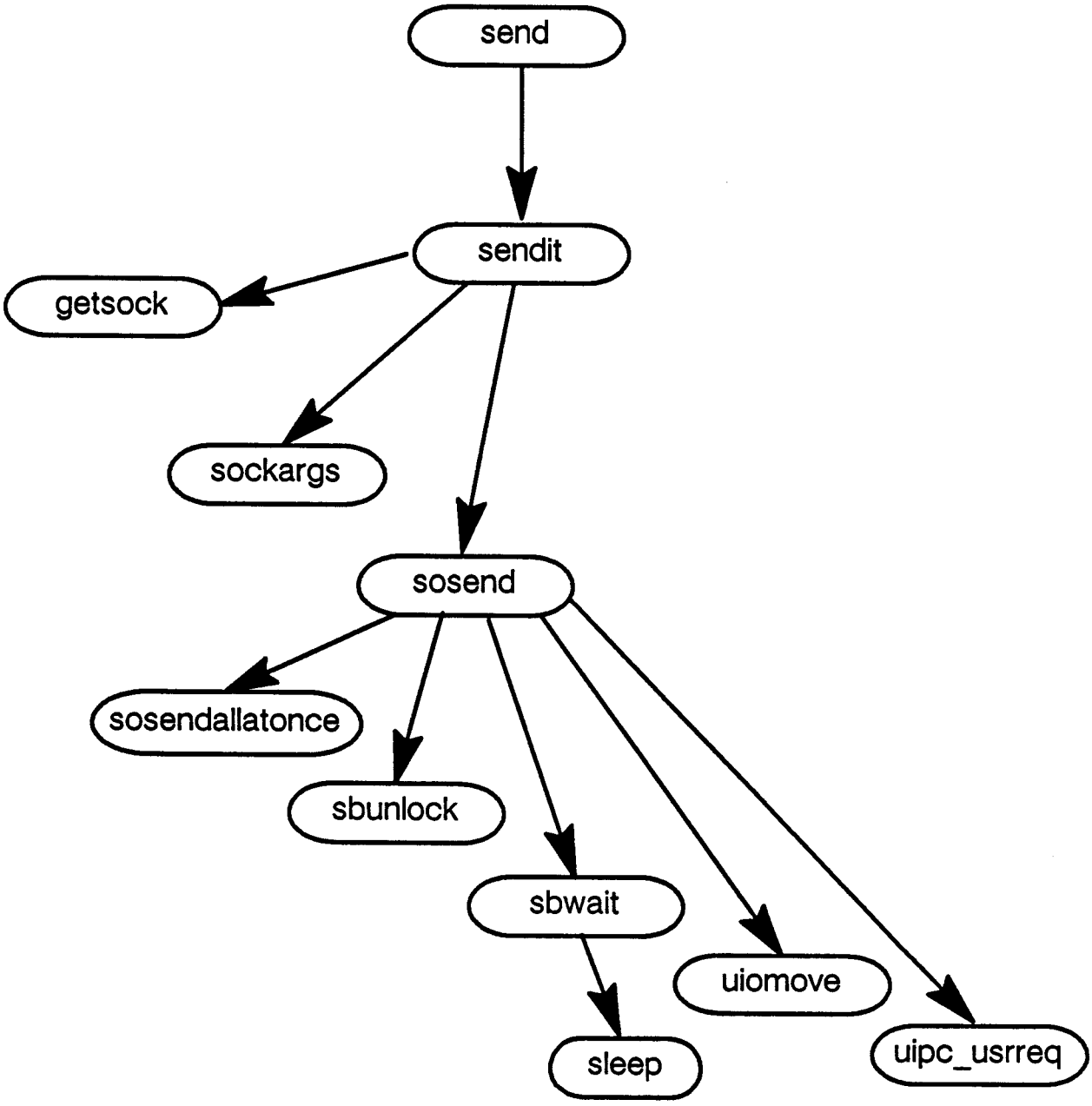
sendmsg() can also send a message to a *socket* that hasn't been connected with *connect(2)*. This function facilitates the use of *scatter/gather* I/O on the *socket* descriptor (see module 5: "I/O on File Descriptors"). The bulk of the arguments to *sendmsg()* are stored in the *struct msghdr msg_s* (found in *<sys/socket.h>*):

msg_name . . . . . optional address (name) of the *socket.*
msg_namelen . . . . . size in bytes of the (optional) address.
msg_iov . . . . . the *scatter/gather* structure array.
msg_iovlen . . . . . the number of elements in the I/O vector.
msg_accrights . . . . . access rights of message(s).
msg_accrightslen . . . . . the length of the *msg_accrights* field.

The return value on a successful call is the number of bytes sent; else –1.

NOTE: *read(2)* and *write(2)* can also be used to perform I/O on a *SOCK_STREAM socket.*

send( )



PRU_SEND

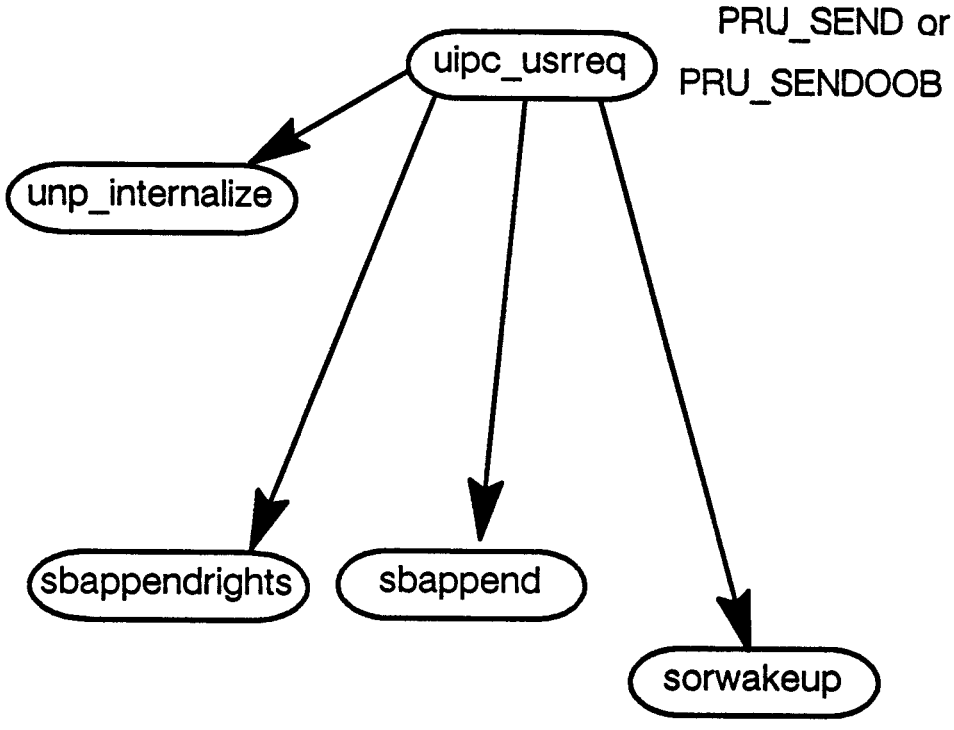
send( ) system call

The *send()* system call is one of a family of system calls for sending data through the socket. Send() initializes the msghdr and iovec structures and then calls *sendit()* which is a common routine that is called also by *sendto()* and *sendmsg()*. Sendit() calls getsock() to return a file pointer from the descriptor and then initializes the uio structure and perform additional validity checks on the parameters passed. It then calls *sockargs()* to copy the message header name or the access rights into an mbuf structure and then calls *sosend()*.

sosend() calls the routine *sosendallatonce()* to determine if the message needs to be sent all in one piece and if so, it checks if the amount of data to be transmitted is beyond the high water mark, in which case it will return an error indicating that the message is too big. If the data need not be sent in one piece then *sosend()* will break this up and send it in smaller messages. If there is not enough space in the socket buffers, and non–blocking operations is indicated, then it will release the socket and return an error.

Otherwise it will call *sbunlock()* to unlock the socket and then *sbwait()* to wait and call *sleep()* until data drains from the socket buffers and generates more space. If there is enough space, it will try to send as much as possible until the space in the socket runs out. *sosend()* calls *uiomove()* to copy the data from user space into mbufs that will be linked into the mbuf chains that will be transmitted. Then *sosend()* calls *pr_usrreq()* with either PRU_SEND or PRU_SENDOOB depending on whether the flags indicate out of bounds data or not.

uipc_usrreq( )


uipc_usrreq
PRU_SEND or
PRU_SENDOOB
unp_internalize
sbappendrights
sbappend
sorwakeup

uipc_usrreq( ), send( ) cont'd

uipc_usrreq() is called with either the PRU_SEND or PRU_SENDOOB requests. PRU_SENDOOB is currently not supported by this protocol and will result in an error being returned. For PRU_SEND, if access rights are being passed, then the *unp_internalize()* routine will be called to convert the key held by the sending process into an internal form that can be passed as data in a message. *uipc_usrreq()* then calls either *sbappendrights()* or *sbappend()*.

sbappend() will append an mbuf chain representing the data being transmitted to the last record in the socket buffer of the receiving socket. *sbappendrights()* on the other hand will append both data and access rights which are represented by different mbuf chains into the receive buffer of the receiving socket.

Lastly, *uipc_usrreq()* will then call *sorwakeup()* to wake up any reader processes that are awaiting data on the receiving socket.

Socket I/O

- *Socket* data reception functions – *recv(2), recvfrom(2), recvmsg(2)*:

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(sd, buf, len, flags)
int recvfrom(sd, buf, len, flags, from, fromlen)
int recvmsg(sd, msg, flags)
int sd, len, flags, fromlen;
char *buf;
struct sockaddr *from;
struct msghdr *msg;
```

Socket I/O

recv() is used to receive messages from another socket which has been connected with the *connect(2)* system call. If no messages are available at the *socket*, the receive call waits for a message to arrive (unless the *socket* is non-blocking; see *ioctl(2)*). The argument *sd* is a valid *socket* descriptor, the argument *buf* is the space allocated by the calling process to contain the received message, *len* is the size of *buf*, and the *flags* argument may be *or*-ed with one or both of the following:

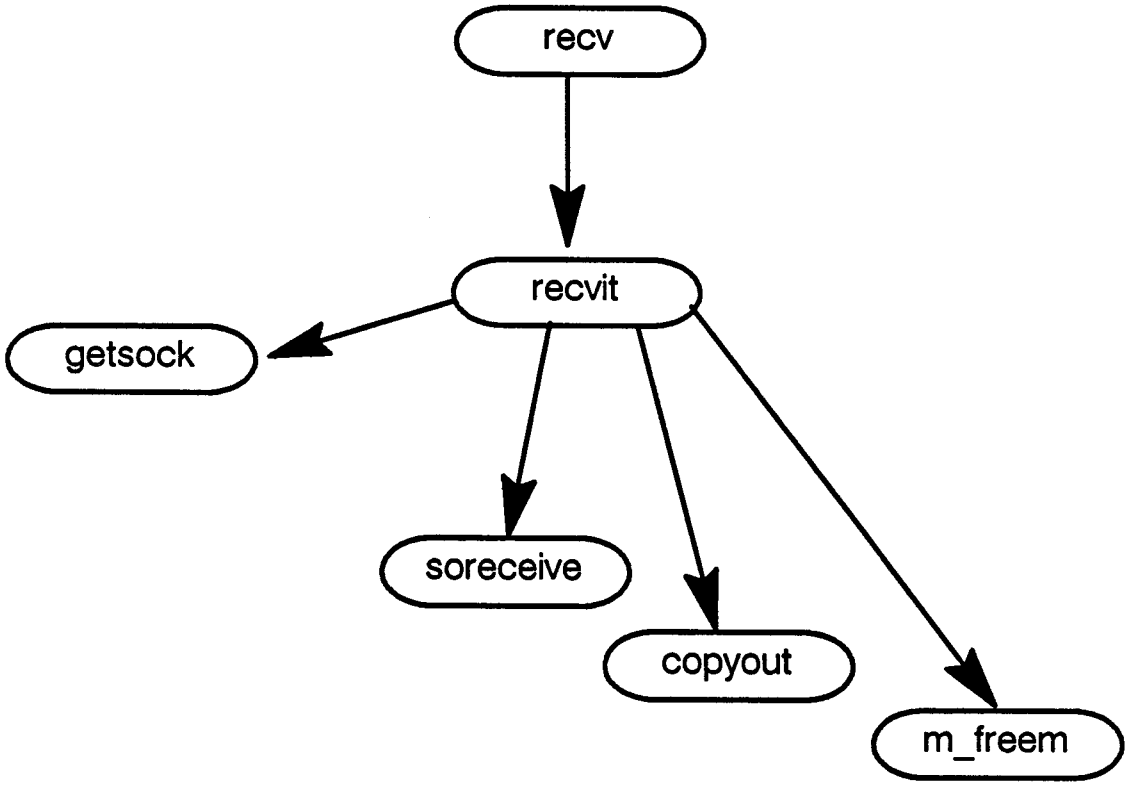
MSG_OOB	read any "out-of-band" data present on the *socket,* rather than "in-band".
MSG_PEEK	peek at the data present on the *socket*; the data is returned, but a subsequent receive operation will have access to the same data.

recvfrom() is the counterpart to the *sendto()* function, and as such it need not receive messages from connected *sockets.* If *from* is not a NULL pointer, the source address of the message is filled in. *fromlen* is a value/result parameter initialized to the size of the buffer associated with *from*, and filled on return with the actual size of the address stored in *from*.

recvmsg() is the counterpart to the *sendmsg()* function, which also need not receive message from a connected *socket*. The *struct msghdr* fields *msg_name* and *msg_namelen* specify the destination address if the *socket* is unconnected. As with *sendmsg()*, this function can perform *scatter/gather* I/O with the *socket.*

Upon a successful call these receive functions will return the number of bytes received; otherwise a -1.

recv( )



recv( ) system call

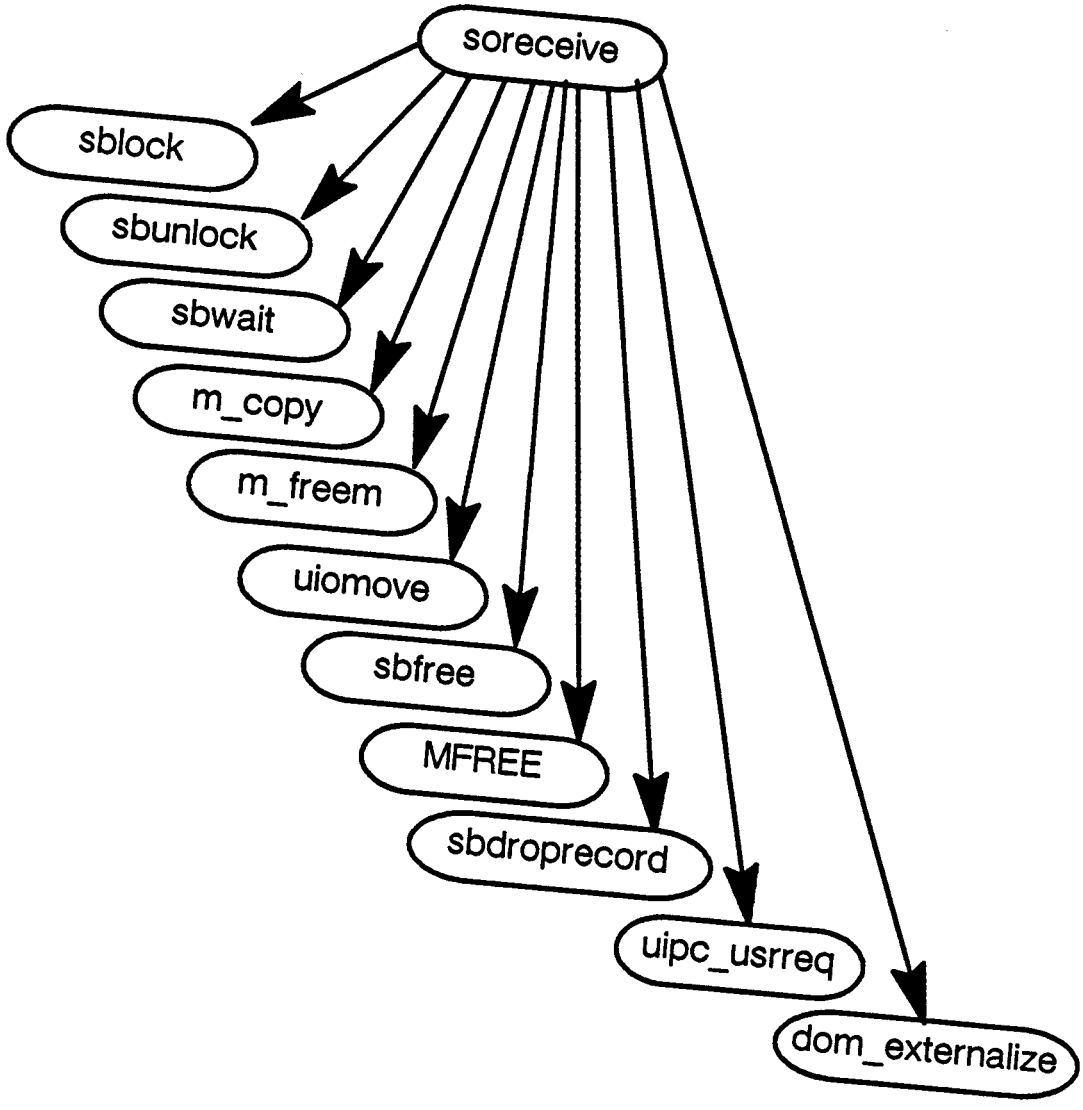
recv() is a system call which simply initializes the message header and iovec structures before calling the common routine *recvit()* which is also called by *recvfrom()* and *recvmsg()*. *recvit()* performs similar checks done by *sendit()* and calls *getsock()*. It then calls *soreceive()* which works on out of band data first.

Unfortunately, the AF_UNIX domain does not support a protocol that handles out of band data. *soreceive()* does a lot of validity checks for non–blocking operations, for existing connections, and for capability to receive more data. It also calls *sbunlock()* and then *sbwait()* if there is no data available and if the flags indicate that non_blocking operations is possible.

soreceive(), after receiving the messages will return to *recvit()* and *recvit()* will call *copyout()* to copy the address and access rights respectively from the system variables into process buffers and will call *m_freem()* to free the variables afterwards.

Sun Education
Not to be copied

recv( ) system call(cont)



recv( ) system call (cont)

If the flags in the protosw structure indicate that addresses must be passed with messages, then *soreceive()* looks for a MT_SONAME type and if it does not find any, will call *panic()*. If flag indicate MSG_PEEK, it will make sure that the mbufs holding the information are not disposed of. *soreceive()* will call *sbunlock()* to unlock the receive socket buffer and then sbwait if the non-blocking flag is on and the buffer is empty. If messages are available, it then will call *m_copy()* to copy the address and access rights messages into system variables and will call *m_freem()* if the MSG_PEEK flag is not on.

It then calls *uiomove()* to move the data from the mbuf chain into user address space and if the flag is not MSG_PEEK, it gets rid of the mbufs by calling *sbfree()* which will update the counters in the sockbuf structure to reflect the freeing of the mbufs. It then calls *MFREE( )* which will actually free the mbufs.

After receiving the required data, *soreceive()* checks if there are remaining mbufs in the chain and if so, checks for the atomic flag and if it is on, it calls *sbdroprecord()* to get rid of the entire chain from the sockbuf.

If the flags indicate PR_WANTRCVD, then *pr_usrreq()* is called with the PRU_RCVD request and for SOCK_STREAM type of socket will result in the backpressure being adjusted on the sender and wakeup any processes waiting to send messages on the socket.

if access rights have been passed, *soreceive()* calls the *dom_externalize()* routine to convert the access rights from a format used in transmitting it to one that could be used by the receiving process. The socket receive buffer will then be unlocked after the messages are received.

Module 8
System Startup

Objectives: Upon completion of this module, the student will be able to:

Describe the steps involved in system startup from power up until the first login can take place.

Evaluation: none

Reference information: Course Guide

Introduction

When a workstation is turned on the program's binary image must be loaded into main memory.

To initialize the state of the cpu:

- initialize the run time stack
- intitialize the virtual memory mapping

Initialize machine dependent operations by:

- setting up vm page tables
- configuring I/O devices

Initialize machine independent operations by:

- mounting the root file system
- initializing system data structures

Introduction

The vm has to be done early to minimize specialized assembler programs that have to reference physical addresses.

Bootstrap

The bootstrap program is stored on the disk (or on the server for a diskless client), and is responsible for loading in the actual executable. It is essentially a *very* stripped version of the kernel, containing only the minimum code needed to load in the real executable. It must:

_start:

- Allocate Memory – avoid colliding with *kadb*
- Set up buffer table for buffer cache
- Set up the system call trap handler
- Map virtual addressing (skipping around on–board frame buffer memory for Sun–3s)
- Get boot arguments to determine where to boot from
- If we cannot mount the root filesystem query user for boot device

Bootstrap

The system sets up a buffer cache to use with unamed objects such as indirect blocks, cylinder group, and superblock structures.

Bootstrap

Boot: *filename*

- If filename is " * "

  list root directory

- Open *filename*

  if it fails query user for bootpath

readfile:

- Read in header – check magic number

Size: *textsize*

- Make sure that when we load in the text section it will not overlay the boot program (verify *LOAD + textsize < start)*

- *addr = (char *)(LOAD + textsize)*

Bootstrap

Bootstrap

- Read in text section to virtual address *LOAD* (0x4000)
- Round (and zero) text section up to next segment address (set *addr* to this new value)
- Make sure that when we load in the data section it will not overlay the boot program (verify *addr* + *datasize* < *start)*

+ *datasize*

- Read in data section to virtual address *addr*
- Close file
- *addr* += *datasize*

Bootstrap

Bootstrap

- Allocate and zero BSS. Make sure that the BSS segment does not overlay boot program. This is the only case that is not an error, because at this point we won't be needing the boot code for much longer. If the BSS does overlap, however, the remainder will *not* be initialized to zero.

```
while (i < bsssize && addr < start) {
        *addr++ = 0;
        i++;
}
```

+*bsssize* bytes

- Determine starting address of executable. This will either be

  loadaddr = LOAD + sizeof (struct exec)

  for ZMAGIC executables, or just

  loadaddr = LOAD

- Execute program:

  *(*loadaddr)(args_from_boot);*

Bootstrap

Vmunix

_start:

- Block interrupts and enter kernel mode.

- Initialize MMU with kernel mappings.

- Enable virtual address translation by executing a non–pc relative jump:

  jmp cont
  cont:

  By setting up the mappings carefully, we can simply jump to the next instruction to enable translation.

- Map in EEPROM and special registers (clock, memory error register, interrupt control register, ecc registers.)

- Set up initial U–area mappings (which will become the U–area for process 0), using physical page 0.

- Create kernel stack for process 0

Vmunix

Vmunix

- Create user stack for process 1
- Build fake stack frame for process 1
- Call *main*. In a tricky manner, main will return as process 1 (*init*), which is why we build its stack and stack frame.

main:

- Initialize run and sleep queues
- Call *startup()*

startup:

- Determine boot options (such as single–user)
- Initialize interrupt/trap vector tables

Vmunix

Vmunix

- Enable interrupts.
- Determine cpu type
- Size DVMA regions
- If vac is present, enable it.
- Enable memory error register
- Size physical memory
- Initialize VM system
  hat_init
  kvm_init

SunOS Release 4.0 (JODY) #1: Mon May 8 19:00:12 PDT 1989
Copyright (c) 1988 by Sun Microsystems, Inc.

Vmunix

Vmunix

mem = ***memsize*K (0x*bytes*)**

- Determine amount of free memory available

avail mem = ***freemembytes***

- Allocate space for system data structures:
  - inode table
  - open file table
  - segudata
  - callout queue
  - process table

- Initialize system data structures

- Determine Ethernet address

Ethernet address = ***ethernet address***

Vmunix

Vmunix

- Configure devices (with *configure()*). This will call all the individual *probe* routines for each device, and print messages if they are found.

device_name unit# **at** *address_space address info*
bwtwo0 at obmem 0x100000 pri 4

- Initialize U–area for process 0

- Initialize process structure for process 0.

- Start *roundrobin()* and *schedcpu()*. Set up global clock for paging.

- Mount the root filesystem, and set up the swap and dump areas.

root on *root_device* **fstype** *fstype*
swap on *swap_device* **fstype spec size** *size***K**
dump on *dump_device* **fstype spec**

- Fork to create *init* (process 1)

Vmunix

Vmunix

- Fork *pagedaemon* (process 2)
- Call *sched()* (become process 0, the *swapper*)
- Init now runs, and executes several startup scripts
  - */etc/rc.boot*
  - */etc/rc*
  - */etc/rc.single or*
  - */etc/rc.local*

checking filesystems
fsck output
Fri Sep 1 01:08:31 PDT 1989
checking quotas: done
starting rpc and net services: portmap ypbind routed
starting system logger
starting additional services: biod
starting local daemons: sendmail statd lockd
link-editor directory cache
preserving editor files
clearing /tmp
standard daemons: update cron uucp
starting network daemons: inetd printer
Fri Sep 1 01:09:17 PDT 1989

Vmunix

Vmunix

- Init reads */etc/ttytab* to enable logins (by running *getty* processes on terminals)

yavin login:

- Init now *wait*s for its children to die, checking to see if they were login processes, so it can respawn *getty*s.

Vmunix

Appendix A

Labs and Questions

Overview Questions

1. What distinguishes the SunOS from most other Unix operating systems?

2. What is the buffer cache used for after 4.0?

3. How does a system call enter the kernel mode?

4. What is the *sysent[]* array used for?

5. How does a system call pass any error information to the user mode process?

Overview Questions (cont.)

6. What are two reasons a system call could abort?

7. Give some examples of why a user mode process would have to enter the kernel mode?

8. Why can't a user process block waiting for some system resource?

9. What is the difference between being pre-empted and interrrupted?

10. Why would a kernel routine have to be concerned about the number of routines it called?

Overview Questions (cont.)

11. When would the top half of the kernel be non-interruptable?

12. Why isn't it possible to do real time processing on most Unix operating systems?

13. Why can't the bottom half of the kernel call *sleep()*?

14. Why is the clock handler divided into a *hardclock()* routine and a *softclock()* routine?

15. What's a tick?

Overview Questions (cont.)

16. What happens if a process profile timer goes to 0.

17. What is the timeout queue used for?

18. What does the number of ticks in the queue structure represent?

Memory Questions

1. When is copy–on–write performed?

2. Why does memory have to share objects at a page level?

3. How large is a page on a Sun–3? Sun–4? Sun–2? Sun386i?

4. What are three definitions of a segment?

5. What defines a process's context?

6. Why do we use virtual memory?

Memory Questions (cont.)

7. What is the problem of using virtual memory if we're trying to do real time processing?

8. What is the first thing determined by the MMU during virtual to physical address translation?

9. With 4,096 page table entries on a Sun–3, how much memory can we address (or keep track of)?

10. Why can we just copy the last 13 bits of a virtual address to the last 13 bits of the generated physical address?

11. How many contexts can we have on a Sun–4?

12. With 16,384 page table entries how much physical memory can we have on a Sun–4?

Memory Questions (cont.)

13. What does the **nc (x)** bit mean in the PTE?

14. Why do we keep track of the modified status of a page in the PTE (the **m** bit)?

15. What makes cache memory so fast?

16. Why don't we just use cache memory?

17. If typical disk access time is 20–50 milliseconds, typical main memory access time is 200 nanoseconds, and cache access is 30 nanoseconds, how many trips to main memory could we make in the time it takes to do one disk access? how many trips to cache? how many instructions could we have executed on a 4 MIP processor?

Memory Questions (cont.)

18. What does locality of reference have to do with the cache hit rate?

19. What is the difference between a write-through and a write-back cache? Which is usually the fastest?

20. When must we synchonize cache and main memory?

Lab 1 Memory

1. Invoke adb -kw /vmunix /dev/mem or boot kadb

   Dump 10 4 byte entries of hex data at the symbol Usrptmap by typing:

   Usrptmap, 10/X or

   Sysmap, 10/X

   Where Usrptmap is the address, in this case a symbol from the vmunix namelist,

   10 is the repeat count,

   / means take the data from memory,

   and X means dump a 4 byte hex number.

   What you will see are some PTE entries.

   Select the second and third entry to answer the following questions.

   Is this a valid translation?

   Is it writable?

   Can you write to it as a user?

   What physical page numbers are they using?

   Is it user or kernel address space?

   Has it been referenced?

   Does it matter that the reference bit is set?

   Can it be placed in cache?

Lab 1 Memory

Invoke adb on the system by typing **adb -kw /vmunix /dev/mem**

Locate the address of the first page structure on the system by typing **pages,1/X.**

Dump the first several page structures by typing
(address from previous step), 40/X.

Answer the following questions by examining the first three page structures.

Are any of the three pages wanted?

Are any on the free list?

Is the reference bit set for any of the pages?

Were any of the pages brought in as the result of a read ahead operation?

What are the addresses of the vnodes that these pages came from?

Are there any other pages in the vnodes?

How many address translations exist for these pages?

Are there any outstanding I/O requests needed for these pages?

VM questions

1. What structure or object represents the highest level of abstraction in the VM system?

2. What structures make up the address space of the process?

3. What are the four types of segment drivers?

4. What two *as* routines are called by the *mmap(2) system call*?

5. How is a page coming from a file identified in the VM system?

VM questions

6. What structures get allocated when the system maps objects that do not have backing from the file system?

7. When a mapping exists for a page where does the *p_mapping* member of the page structure point?

8. For a particular segment mapping to anonymous pages, how do we determine how much space is allocated for the segment?

9. What structures get allocated during an *mmap(2)* system call?

10. What is the struct *seg_ops*?

VM questions

11. What is the purpose of the *as_hole* routine?

12. What will happen if a process tries to access (read or write to) an address to which an object is mapped but has no valid translation loaded?

13. What will the *anon_resv()* routine do?

14. Where is the *hat* structure used by a process located?

15. What is the use of *a_seglast*?

Lab1 (cont.)

1. su to root.
2. Run adbwrap

   Does the number of physical pages *(physmem)* make sense for your machine?

3. Type **u$<u** (use control s to stop the output after one screen).
4. Get the pointer to the *proc* structure (it is labeled *procp*).
5. Dump the proc structure by typing the address of the structure from step 4 followed by **$<proc**
6. From the proc structure note the address of the *as* structure.
7. Dump the *as* structure by typing (address of *as* from step 6) **$<as.**

   What is this process's current working set size in pages?

   What is the address of the first segment structure in this process's address space?

   What is the address of the seg structure of the segment that contained the last page referenced by this process?

   What type of segment is this? (hint: dump the address of the s_ops pointer ex. addr,10/X)

8. Use the data pointer and the appropriate script in /usr/lib/adb (ex. segvn or segmap) to dump out the private segment information.
9. Is this segment backed by swap or the file system?
10. Use the next pointer in the seg structure to look at the next segment in your address space. If either of these segments have a vnode pointer and you could get the inode number can you think of a way to get the pathname?

Process Scheduling Questions

1. What is a proc structure used for?

2. On what three lists could the proc structure be placed?

3. Should a process ever be on two of these lists at the same time? What if a proc is not on any of the lists?

4. What are the two sets of queues that procs are found on?

5. In what case would a process not be on one of the sets of queues?

Process Scheduling Questions

6. Why must the proc structure be resident in memory?

7. Where is an unused proc structure found?

8. Which field links procs on the *allproc* list?

9. What is the difference between p_usrpri and p_pri in the proc structure?

10. What is p_time in the proc structure used for?

11. What are maxrss and p_rssize used for?

12. What system call uses the p_realtimer?

Process Scheduling Questions

13. What is the p_idhash used for and what is an example of a system call that uses it?

14. What process is marked as idle (p_flag == SIDL)?

15. What is the difference between a process in the SSLEEP state and the SSTOP state?

16. Can a process be in more than one state at a given time such as stopped and blocked?

17. Give an example of a system call that would set the SKEEP flag?

18. What processes could have a flag code of 3?

Process Scheduling Questions

19. How could you set the process flags to give a process better execution time?

20. What is a red zone for?

21. How many sleep queues are there?

22. How many run queues are there?

23. Why are there 128 different priority levels?

24. What routines will move a process from a sleep queue to one of the run queues?

Process Scheduling Questions

25. Can you call sleep from the user level?

26. Can an interrupt routine call sleep?

27. When a process exhausts its time slice does it do a voluntary context switch?

28. What happens if the process has to wait for a page in?

29. How is it determined what sleep queue a process will wait on?

30. How is it determined which run queue a process goes on?

31. What is the priority range for user processes and kernel processes?

Process Scheduling Questions

32. What is special about priority PZERO (25)?

33. What routine runs every time slice and puts the next highest priority process in the run queues into execution?

34. Who calls this routine?

35. Who updates your CPU usage if you've been sleeping for more than a second?

36. What is the difference between unsleep and wakeup?

37. If the system average load is 2 and your process went to sleep with a CPU usage (p–cpu) of 200, what will your cpu usage be if you wake up five seconds later?

Process Scheduling Questions

37. Are most context switches voluntary or involuntary?

38. Why do we have to raise the priority at the beginning of the sleep routine?

39. Why does sleep call swtch() and then check for a signal?

40. After a process has been placed on a sleep queue according to its wait channel, we check if it received a signal. Then we check if the wait channel address is 0. Why?

41. During wakeup, why do we have to check if p_wchan is the channel we are waking?

42. In wakeup, if the p_stat does not equal SSLEEP, we set the p_slptime to 0. Why?

Process Scheduling Questions

43. What is the *wantin* flag used for?

44. What is *runrun* used for?

45. If the base user priority is PUSER (50), why do we clamp p_usrpri between 1 and 127?

46. After setting runrun in setpri() why do we call aston()?

47. For the process who slept for 5 seconds in the earlier question what would its priority be after *updatepri()* calls *setpri()*?

48. How often does *schedcpu()* run and what is its job?

Process Scheduling Questions

49. Say that a process that had slept for five seconds now executes for one second, and *hardclock()* runs 100 times per second. At the end of the second *schedcpu()* runs and our load factor has dropped to one. After *schedcpu()* runs, what is our *p_cpu*?

50. At the end of *schedcpu()* we call *setpri()* to recalculate our priority. What will *p_usrpri* be?

Process Creation Questions

1. What are some reasons the *fork* call may fail?

2. Why was the *vfork* call initially removed from 4.0?

3. Why did they have to put it back in?

4. If you're not the superuser (**root**), and there is only one free proc, will a fork succeed?

5. Does *vfork* call *as_dup*?

6. How do we avoid searching the *allproc* and *zombie* list whenever we need a new PID?

Process Creation Questions

7. What is the first state a process is in?

8. When creating a new process, how are the following fields initialized?
   p_sigmask
   p_wchan
   p_ysptr
   p_osptr
   p_idhash

9. In the *fork* call why is the parent's *p_flag*'s SKEEP bit set?

10. If the call was a *vfork*, why does the parent have to sleep until the child process *exec*s or *exit*s?

11. When the parent wakes up after the child *exec*s or *exit*s, why do we have to copy the text, data, and stack sizes back from the child?

Process Creation Questions

12. What is *forkutl* used for?

13. Why does the fork operation use less memory resources than before 4.0? Why is it faster?

14. What does the first bit of an executable file indicate?

15. What structure contains the effective group id and effective user id associated with an executable file?

16. What are the minimum rights needed to execute the file?

17. What if the process is being traced?

18. Why don't OMAGIC files use a text segment?

Process Creation Questions

19. If the executable file does not have a legitimate magic number, what must the first 2 characters of the file be? What if those characters aren't present.

20. Why do we have to copy the arguments to the executable into kernel address space?

21. How is the possible race condition in the *as_hole* routine prevented?

22. Is there any way to check the swap space requirements for an executable file with the a_dynamic bit set?

23. If a file is marked as fill-on-demand, what routine will bring in the text and data pages from disk to main memory?

24. What conditions would cause all the text pages to be brought in at *exec* time (in *getxfile*)?

Process Creation Questions

25. How does VOP_MAP treat data pages differently than text pages?

26. If the flag SVFORK is set in the process calling *relvm*, who gets awakened?

27. If an exiting process is a process group leader, what signal is sent by what routine?

28. Why is the priority of the exiting process raised via *splclock*?

29. How does an exiting process find processes that it is tracing?

30. What happens to the children of exiting processes?

31. What routine moves a proc structure to the free list?

Signal Questions

1. What are the three ways a process may react to a signal?

2. What signal is sent by the *kill(2)* command with no argument by default?

3. What is an example of why a process may be sent a SIGSEGV?

4. What is SIGTTIN used for?

5. What is the difference between *p_sigignore* and *p_sigmask*?

6. Is *p_cursig* a signal mask?

7. What does it mean if a process is "catching a signal?"

Signal Questions

8. What system call allows a process to specify a signal handler to be called in response to a given signal?

9. What is *sigpause()* used for?

10. Why would we want to possibly handle signals on a separate stack?

Lab 2 Process Management part a

1. Login into your student account *int#*, '**su** –' to root. Use **ps aux** to find the *pid* of your login process.
2. Use the **pstat –p** command to get the address of your login proc structure. It will be in the first column under Locs.
3. Invoke adb by typing the command line **adb –kw /vmunix /dev/mem**. You could also use kadb for this part of the lab.
4. Type in **(address of the proc structure from step 2), 40/X** to display your proc structure.
5. What is the address of your as structure?
6. What is your user and current priority?
7. Which flags are set for your process?
8. What state is your process in?

Lab 2 Process Management part b

1. Login into your student account *int#*, '**su -**' to root. Use **ps aux** to find the *pid* of your login process.

2. Use the **pstat -p** command to get the address of your login proc structure. It will be in the first column under Locs.

3. Invoke adb by typing the command line **adb -kw /vmunix /dev/mem**. You could also use kadb for this part of the lab.

4. Using the address you found with the *pstat* command dump your proc structure by typing **address_of_proc $< proc.** This will use the script in /usr/lib/adb (for adb) to display yoour process structure in a labled readable format. Note that some of the fields will be in different radix.

5. Find the pointer to your ucred structure in your proc structure and dump the ucred structure by typing **address_of_ucred $< ucred.** Within your ucred structure find the offset to your uid. Carefully calculate the address of the uid field, and change it to 0 (that of super user) by typing **address_of_uid / w 0.** Dump your ucred structure again to insure the uid field was the one that was changed. If you changed another portion of your ucred structure change it back to what it was before you proceed.

6. Verify the success of your operation by exiting adb (control D or **$q**) and invoking it from your login id as in step 1.

Lab 2 Process Management
part c

1. **cd** into the Lab2 directory, and make the t1, t2, t3 and t4 programs with **make all.** Execute the programs, start with t4, then t3, then t2, and t1. Do not look at the source.

2. Use **ps aux** to find the *pids* of processes t1, t2, t3, and t4. Use **pstat -p** to get the addresses of proc structures for each process. The addresses will be in the first column under Locs.

3. Now invoke adb by typing the command line **adb -kw /vmunix /dev/mem** (you might want to prefix this with **nice -20**). You must use *adb* for this section of the lab rather than *kadb.*

4. Dump the run queues by typing **qs,40/X**
   where **qs** is the label in the vmunix symbol table for the runqueues,
   ,40 means repeat the command 40 times and,
   /X means display a 4 byte hex number from the core file (in this case the core is /dev/mem, physical memory).

5. In the queues you should be able to locate the addresses of the proc structures. By looking at the queues and the proc structures answer the following questions.

   Why isn't your proc structure (the process running adb) in the run queues?

   What are the priorities of the processes t1, t2, t3 and t4?

   Does this make sense as far as what queue you found them on?

   Is queue 0 at the top or bottom of your screen?

Lab 2 Process Management part c

Why do the processes all start out near the same queue and then begin to spread out in the queues? (hint: look at the flow chart for *setpri())*

Why do two of the processes stay close together on the queues?

6. Now find your proc structure (hint use **u $< u**). Set the SFAVORD bit in your *p_flags* in your proc structure, do the same for process t1.

   Check the queues again, now what happens? Why?

   Are processes t1, t2, and t3 I/O bound or compute bound?

   Where are all the rest of the processes on the system?

Lab 2 Process Management
part d process creation

1. **cd** into the Lab2 directory, and make the f program. Execute the program.
2. Use **ps** and **pstat -p** to get the addresses of proc structures for each process. The addresses will be in the first column under Locs.
3. Now invoke adb by typing the command line **adb -kw /vmunix /dev/mem** . You must use *adb* for this section of the lab rather than *kadb.*
4. From the proc structure determine which process was created using *vfork().* There are at least two ways of doing this.

   Where is the parent process? (hint: see flow chart for *fork()*)

Lab 2 Process Management part e Signals

1. Login as root, halt the system using **/etc/halt.** Now boot the system with kadb by typing **b kadb**.
2. Login as int#
3. Type **L1a** instead of halting the system should now be in *kadb* with the prompt **kadb>.**
4. Using kadb, find out and list what signals your process is masking, catching and ignoring.
5. Use the command **psignal + 4:b** to set a breakpoint in psignal.
6. Use **:c** to continue SunOS.
7. Wait until you enter kadb due to the breakpoint. What might have caused you to encounter this breakpoint.
8. Now set a breakpoint in *psig()* by typing **psig + 4:b.** Look at the signal masks and p_cursig. Does it make sense.

File Questions

1. What are two advantages of a large disk block?

2. How large a block does Sun use?

3. What is the disadvantage of a large block and how do we compensate for that?

4. Why is the superblock stored at varying offsets in the cylinder group?

5. What is the minimum number of inodes per cylinder group?

6. How large is a fragment?

7. Why do we use cylinder groups?

File Questions

8. What is a VFS?

9. What is a vnode?

10. Explain the reason for the vnode interface?

11. What is object oriented programming and why does it work well with vnode layer operations?

12. Why do we have file descriptors?

13. What information is held in the open file table?

File Questions

14. Why isn't the open file table information just put in the file descriptors?

15. Why are directories accessed in 512 byte chunks?

16. What are the four fields in a directory entry?

17. How long can a file name be?

18. How does the system recognize a deleted directory entry?

19. How are holes in a directory handled?

File Questions

20. What is the incore inode and how does it differ from a disk inode?

21. How much data can be referenced on a Sun with the direct blocks?

    single indirect blocks?

    double indirect blocks?

    triple indirect blocks?

22. Can the size field support a double indirect block, a triple indirect block (see inode.h in App B)?

23. What does the IWANT flat in the incore inode indicate?

24. What would the file type mask IFMT in the incore inode's inode field be used for?

25. What do execute permissions on a directory allow you to do?

File Questions

26. Is it possible to update the file without having to update the inode?

27. Would you ever have to update the inode and not the file?

28. Why can't a hard link point to a directory?

29. How is a loop caused by a symbolic link detected?

30. What happens if the number of blocks in a file system falls below minfree?

File Questions

31. What determines the maximum number of blocks we can write to a cg before doing a long seek?

32. What is *fs_rotdelay* used for?

33. Would fs_rotdelay need to be modified for an intelligent disk controller?

34. What is the maximum number of cylinders per cylinder group?

35. What is the default number of cylinders per cylinder group?

36. What routine would use the rotational block position table in the superblock (fs_postbl[])?

File Questions

37. What structure contains a bit map of the available blocks and fragments?

38. Why do we keep track of the number of directories in a cylinder group?

39. How many data blocks are there in a section of a file, other than the first section?

40. How many data blocks in a file can be placed in a cylinder group (using 1 KB fragments)?

41. What is the routine that is used to determine the preferred block placement at the global placement policy level?

File Questions

42. How do we determine where the first block of a new section should be placed, where *lbn* is the logical block number and *fs_mbpg* is the maximum number of blocks per cylinder group?

43. What is the strategy used to place a new directory in a file system?

Lab 3 File System

1. In this lab we will examine the relationship between the user's file descriptor table , the system open file table the vnode, the page structures, and the inode structures for a given file. We will also take a look at the super block structure. This lab will be done using adb and an existing core dump rather than on a live system. The syntax will be:
   adb -k ~int0/Lab3/vmunix.sun3 ~int0/Lab3/vmcore.sun3
   where vmunix is the name of the executable file and vmcore is the name of the core file. These are located in the **Lab3** directory off of user **int0**'s home directory. If you are working on a Sun-4 use the '.sun4' files.
2. Look at the u structure for the current process at the time of the core dump.
3. Get the u_ofile pointer to the system open file entry.
4. Dump the file structure of the entry using the script **file** (in /usr/lib/adb).
5. In the open file table entry there is a pointer to the vnode structure, dump the vnode structure.
6. Use the vnode information to find the inode.

   How many bytes are in the file?

   How many blocks are used by the file? Why?

   What is the address of the pment[] entry associated with the file?

   What flags are set in the inode?

   What is the reference count in the vnode? In the inode?

   What will be the offset into the data block for the next write operation?

Lab 3 File System (cont.)

Are there any exclusive locks or shared locks on the file?

What is the size of the data block on this system (hint: look at the struct vfs)?

What are the major and minor device numbers of the device this is located on?

What are the permissions on the file?

When was the file last changed?

What is the address of the superblock?

What is the inode block number? What is this relative to?

What is the number of cylinder groups in the file system?

What is the fragment size in this file system?

What is the number of fragments per data block?

What is the minimum number of free blocks?

What is the rotational delay in milliseconds.

What would this work out to in blocks? (optional –equation in book)

How many revolutions per second is the disk spinning?

What is the max contiguous number of blocks before a rotational delay?

What is the max number of blocks per cylinder group for one file?

I/O Management Questions

1. What are the two categories of device files and how are devices classified into these categories?

2. How does the operating system interface with the top-half(base-level) routines of the device driver?

3. In the cdevsw[] structure, there is a member that is not a function pointer, what is it?

4. What base-level routine in the character driver gives it a lot of flexibility?

5. What is the purpose of the block device driver strategy routine?

6. What are the two ways discussed in this module by which a process can access the contents of a regular file?

7. When read( ) or write( ) are used to access the file after it is opened, what will read( ) or write( ) do to the file?

8. What common routine is called by the read( ), readv( ), write( ) or writev( ) system calls?

9. What segment driver routine determines if the block that the process wants to read in is already mapped in or not?

I/O Management Questions

10. What will the segmap_hashout( ) routine do? The segmap_hashin( )?

11. What routine is called to make sure that physical blocks are allocated when we are writing to a file?

12. In what two instances is the segmap_pagecreate( ) routine called when we are writing to a file?

13. What routine determines if a page is already in main memory or not?

14. What routine is called when a page needs to be written out to the file system?

Appendix B

Kernel Header Files

file.h

```
/*      @(#)file.h 2.12 88/02/08 SMI; from UCB 7.1 6/4/86       */
/*
 * Copyright (c) 1982, 1986 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef _FILE_
#define _FILE_

#ifdef KERNEL
/*
 * Descriptor table entry.
 * One for each kernel object.
 */
struct  file {
        int             f_flag;                 /* see below */
        short           f_type;                 /* descriptor type */
        short           f_count;    /* reference count */
        short           f_msgcount;/* references from message queue */
        struct          fileops {
                        int             (*fo_rw)();
                        int             (*fo_ioctl)();
                        int             (*fo_select)();
                        int             (*fo_close)();
        } *f_ops;
        caddr_t         f_data;                 /* ptr to file specific struct (vnode/socket) */
        off_t           f_offset;
        struct          ucred *f_cred;          /* credentials of user who opened file */
};

struct  file *file, *fileNFILE;
int     nfile;
struct  file *getf();
struct  file *falloc();
#endif KERNEL

#include <sys/fcntlcom.h>
```

file.h (cont.)

```
/*
 * flags - see also fcntlcom.h
 */
#define     FOPEN           (-1)
#define     FREAD           00001           /* descriptor read/receive'able */
#define     FWRITE          00002           /* descriptor write/send'able */
#define     FMARK           00020           /* mark during gc() */
#define     FDEFER          00040           /* defer for next gc pass */
#define     FSHLOCK         00200           /* shared lock present */
#define     FEXLOCK         00400           /* exclusive lock present */

/* bits to save after open */
#define     FMASK           (FREAD|FWRITE|FAPPEND|FSYNC|FASYNC|FNBIO)
#define     FCNTLCANT       (FREAD|FWRITE|FMARK|FDEFER|FSHLOCK|FEXLOCK)

/*
 * User definitions.
 */

/*
 * Flock call.
 */
#define     LOCK_SH         1       /* shared lock */
#define     LOCK_EX         2       /* exclusive lock */
#define     LOCK_NB         4       /* don't block when locking */
#define     LOCK_UN         8       /* unlock */

/*
 * Access call.
 */
#define     F_OK            0       /* does file exist */
#define     X_OK            1       /* is it executable by caller */
#define     W_OK            2       /* writable by caller */
#define     R_OK            4       /* readable by caller */
```

file.h (cont.)

```
/*
 * Lseek call.
 */
#define     L_SET               0           /* absolute offset */
#define     L_INCR              1           /* relative to current offset */
#define     L_XTND              2           /* relative to end of file */

#ifdef KERNEL
#define     GETF(fp, fd) { \
            if ((unsigned)(fd) >= NOFILE || ((fp) = u.u_ofile[fd]) == NULL) { \
                        u.u_error = EBADF; \
                        return; \
            } \
}

#define     DTYPE_VNODE         1           /* file */
#define     DTYPE_SOCKET        2           /* communications endpoint */
#endif KERNEL

#endif !_FILE_
```

fs.h

```
/*      @(#)fs.h 2.13 88/02/08 SMI      */

/*
 * Copyright (c) 1982, 1986 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 *
 *      @(#)fs.h    7.1 (Berkeley) 6/4/86
 */

/*
 * Each disk drive contains some number of file systems.
 * A file system consists of a number of cylinder groups.
 * Each cylinder group has inodes and data.
 *
 * A file system is described by its super-block, which in turn
 * describes the cylinder groups. The super-block is critical
 * data and is replicated in each cylinder group to protect against
 * catastrophic loss. This is done at mkfs time and the critical
 * super-block data does not change, so the copies need not be
 * referenced further unless disaster strikes.
 *
 * For file system fs, the offsets of the various blocks of interest
 * are given in the super block as:
 *      [fs->fs_sblkno]         Super-block
 *      [fs->fs_cblkno]         Cylinder group block
 *      [fs->fs_iblkno]         Inode blocks
 *      [fs->fs_dblkno]         Data blocks
 * The beginning of cylinder group cg in fs, is given by
 * the "cgbase(fs, cg)" macro.
 *
 * The first boot and super blocks are given in absolute disk addresses.
 */
#define BBSIZE                  8192
#define SBSIZE                  8192
#define     BBLOCK              ((daddr_t)(0))
#define     SBLOCK              ((daddr_t)(BBLOCK + BBSIZE / DEV_BSIZE))
```

fs.h (cont.)

```
/*
 * Addresses stored in inodes are capable of addressing fragments
 * of 'blocks'. File system blocks of at most size MAXBSIZE can
 * be optionally broken into 2, 4, or 8 pieces, each of which is
 * addressible; these pieces may be DEV_BSIZE, or some multiple of
 * a DEV_BSIZE unit.
 *
 * Large files consist of exclusively large data blocks. To avoid
 * undue wasted disk space, the last data block of a small file may be
 * allocated as only as many fragments of a large block as are
 * necessary. The file system format retains only a single pointer
 * to such a fragment, which is a piece of a single large block that
 * has been divided. The size of such a fragment is determinable from
 * information in the inode, using the "blksize(fs, ip, lbn)" macro.
 *
 * The file system records space availability at the fragment level;
 * to determine block availability, aligned fragments are examined.
 *
 * The root inode is the root of the file system.
 * Inode 0 can't be used for normal purposes and
 * historically bad blocks were linked to inode 1,
 * thus the root inode is 2. (inode 1 is no longer used for
 * this purpose, however numerous dump tapes make this
 * assumption, so we are stuck with it)
 * The lost+found directory is given the next available
 * inode when it is created by "mkfs".
 */
#define     ROOTINO             ((ino_t)2)  /* i number of all roots */
#define LOSTFOUNDINO            (ROOTINO + 1)

/*
 * Cylinder group related limits.
 *
 * For each cylinder we keep track of the availability of blocks at different
 * rotational positions, so that we can lay out the data to be picked
 * up with minimum rotational latency.  NRPOS is the number of rotational
 * positions which we distinguish.  With NRPOS 8 the resolution of our
 * summary information is 2ms for a typical 3600 rpm drive.
 */
#define     NRPOS               8           /* number distinct rotational positions */
```

fs.h (cont.)

```
/*
 * MAXIPG bounds the number of inodes per cylinder group, and
 * is needed only to keep the structure simpler by having the
 * only a single variable size element (the free bit map).
 *
 * N.B.: MAXIPG must be a multiple of INOPB(fs).
 */
#define     MAXIPG              2048        /* max number inodes/cyl group */

/*
 * MINBSIZE is the smallest allowable block size.
 * In order to insure that it is possible to create files of size
 * 2^32 with only two levels of indirection, MINBSIZE is set to 4096.
 * MINBSIZE must be big enough to hold a cylinder group block,
 * thus changes to (struct cg) must keep its size within MINBSIZE.
 * MAXCPG is limited only to dimension an array in (struct cg);
 * it can be made larger as long as that structures size remains
 * within the bounds dictated by MINBSIZE.
 * Note that super blocks are always of size SBSIZE,
 * and that both SBSIZE and MAXBSIZE must be >= MINBSIZE.
 */
#define MINBSIZE    4096
#define     MAXCPG              32          /* maximum fs_cpg */

/*
 * The path name on which the file system is mounted is maintained
 * in fs_fsmnt. MAXMNTLEN defines the amount of space allocated in
 * the super block for this name.
 * The limit on the amount of summary information per file system
 * is defined by MAXCSBUFS. It is currently parameterized for a
 * maximum of two million cylinders.
 */
#define MAXMNTLEN 512
#define MAXCSBUFS 32
```

fs.h (cont.)

```
/*
 * Per cylinder group information; summarized in blocks allocated
 * from first cylinder group data blocks.  These blocks have to be
 * read in from fs_csaddr (size fs_cssize) in addition to the
 * super block.
 *
 * N.B. sizeof (struct csum) must be a power of two in order for
 * the "fs_cs" macro to work (see below).
 */
struct csum {
        long    cs_ndir;        /* number of directories */
        long    cs_nbfree;      /* number of free blocks */
        long    cs_nifree;      /* number of free inodes */
        long    cs_nffree;      /* number of free frags */
};
```

fs.h (cont.)

```
/*
 * Super block for a file system.
 */
#define     FS_MAGIC            0x011954
struct      fs {
            struct      fs *fs_link;            /* linked list of file systems */
            struct      fs *fs_rlink;           /*     used for incore super blocks */
            daddr_t     fs_sblkno;              /* addr of super-block in filesys */
            daddr_t     fs_cblkno;              /* offset of cyl-block in filesys */
            daddr_t     fs_iblkno;              /* offset of inode-blocks in filesys */
            daddr_t     fs_dblkno;              /* offset of first data after cg */
            long        fs_cgoffset;            /* cylinder group offset in cylinder */
            long        fs_cgmask;              /* used to calc mod fs_ntrak */
            time_t      fs_time;                /* last time written */
            long        fs_size;                /* number of blocks in fs */
            long        fs_dsize;               /* number of data blocks in fs */
            long        fs_ncg;                         /* number of cylinder groups */
            long        fs_bsize;               /* size of basic blocks in fs */
            long        fs_fsize;               /* size of frag blocks in fs */
            long        fs_frag;                /* number of frags in a block in fs */
/* these are configuration parameters */
            long        fs_minfree;             /* minimum percentage of free blocks */
            long        fs_rotdelay;            /* num of ms for optimal next block */
            long        fs_rps;                         /* disk revolutions per second */
/* these fields can be computed from the others */
            long        fs_bmask;               /* "blkoff" calc of blk offsets */
            long        fs_fmask;               /* "fragoff" calc of frag offsets */
            long        fs_bshift;              /* "lblkno" calc of logical blkno */
            long        fs_fshift;              /* "numfrags" calc number of frags */
/* these are configuration parameters */
            long        fs_maxcontig;                   /* max number of contiguous blks */
            long        fs_maxbpg;              /* max number of blks per cyl group */
/* these fields can be computed from the others */
            long        fs_fragshift;           /* block to frag shift */
            long        fs_fsbtodb;             /* fsbtodb and dbtofsb shift constant */
            long        fs_sbsize;              /* actual size of super block */
            long        fs_csmask;              /* csum block offset */
            long        fs_csshift;             /* csum block number */
            long        fs_nindir;              /* value of NINDIR */
            long        fs_inopb;               /* value of INOPB */
            long        fs_nspf;                /* value of NSPF */
            long        fs_optim;               /* optimization preference, see below */
            long        fs_sparecon[3];                 /* reserved for future constants */
/* a unique id for this filesystem (currently unused and unmaintained) */
```

fs.h (cont.)

```
        long    fs_id[2];               /* file system id */
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;              /* blk addr of cyl grp summary area */
        long    fs_cssize;              /* size of cyl grp summary area */
        long    fs_cgsize;              /* cylinder group size */
/* these fields should be derived from the hardware */
        long    fs_ntrak;               /* tracks per cylinder */
        long    fs_nsect;               /* sectors per track */
        long    fs_spc;                 /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long    fs_ncyl;                /* cylinders in file system */
/* these fields can be computed from the others */
        long    fs_cpg;                         /* cylinders per group */
        long    fs_ipg;                         /* inodes per group */
        long    fs_fpg;                         /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct  csum fs_cstotal;        /* cylinder summary information */
/* these fields are cleared at mount time */
        char    fs_fmod;                        /* super block modified flag */
        char    fs_clean;                       /* file system is clean flag */
        char    fs_ronly;               /* mounted read-only flag */
        char    fs_flags;               /* currently unused flag */
        char    fs_fsmnt[MAXMNTLEN];            /* name mounted on */
/* these fields retain the current block allocation info */
        long    fs_cgrotor;             /* last cg searched */
        struct  csum *fs_csp[MAXCSBUFS];/* list of fs_cs info buffers */
        long    fs_cpc;                         /* cyl per cycle in postbl */
        short   fs_postbl[MAXCPG][NRPOS];/* head of blocks for each rotation */
        long    fs_magic;               /* magic number */
        u_char  fs_rotbl[1];            /* list of blocks for each rotation */
/* actually longer */
};/*
 * Preference for optimization.
 */
#define FS_OPTTIME 0            /* minimize allocation time */
#define FS_OPTSPACE             1       /* minimize disk fragmentation */
```

fs.h (cont.)

```
/*
 * Convert cylinder group to base address of its global summary info.
 *
 * N.B. This macro assumes that sizeof (struct csum) is a power of two.
 */
#define fs_cs(fs, indx) \
        fs_csp[(indx) >> (fs)->fs_csshift][(indx) & ~(fs)->fs_csmask]

/*
 * MAXBPC bounds the size of the rotational layout tables and
 * is limited by the fact that the super block is of size SBSIZE.
 * The size of these tables is INVERSELY proportional to the block
 * size of the file system. It is aggravated by sector sizes that
 * are not powers of two, as this increases the number of cylinders
 * included before the rotational pattern repeats (fs_cpc).
 * Its size is derived from the number of bytes remaining in (struct fs)
 */
#define     MAXBPC  (SBSIZE - sizeof (struct fs))

/*
 * Cylinder group block for a file system.
 */
#define     CG_MAGIC            0x090255
struct      cg {
        struct      cg *cg_link;                    /* linked list of cyl groups */
        struct      cg *cg_rlink;                   /*     used for incore cyl groups */
        time_t      cg_time;            /* time last written */
        long        cg_cgx;                         /* we are the cgx'th cylinder group */
        short       cg_ncyl;            /* number of cyl's this cg */
        short       cg_niblk;           /* number of inode blocks this cg */
        long        cg_ndblk;           /* number of data blocks this cg */
        struct      csum cg_cs;         /* cylinder summary information */
        long        cg_rotor;           /* position of last used block */
        long        cg_frotor;          /* position of last used frag */
        long        cg_irotor;          /* position of last used inode */
        long        cg_frsum[MAXFRAG];  /* counts of available frags */
        long        cg_btot[MAXCPG];    /* block totals per cylinder */
        short       cg_b[MAXCPG][NRPOS];        /* positions of free blocks */
        char        cg_iused[MAXIPG/NBBY];      /* used inode map */
        long        cg_magic;           /* magic number */
        u_char      cg_free[1];         /* free block map */
/* actually longer */
};
```

fs.h (cont.)

```
/*
 * MAXBPG bounds the number of blocks of data per cylinder group,
 * and is limited by the fact that cylinder groups are at most one block.
 * Its size is derived from the size of blocks and the (struct cg) size,
 * by the number of remaining bits.
 */
#define     MAXBPG(fs) \
            (fragstoblks((fs), (NBBY * ((fs)->fs_bsize - (sizeof (struct cg))))))

/*
 * Turn file system block numbers into disk block addresses.
 * This maps file system blocks to device size blocks.
 */
#define fsbtodb(fs, b)   ((b) << (fs)->fs_fsbtodb)
#define     dbtofsb(fs, b)          ((b) >> (fs)->fs_fsbtodb)

/*
 * Cylinder group macros to locate things in cylinder groups.
 * They calc file system addresses of cylinder group data structures.
 */
#define     cgbase(fs, c)           ((daddr_t)((fs)->fs_fpg * (c)))
#define cgstart(fs, c) \
            (cgbase(fs, c) + (fs)->fs_cgoffset * ((c) & ~((fs)->fs_cgmask)))
#define     cgsblock(fs, c)         (cgstart(fs, c) + (fs)->fs_sblkno)    /* super blk */
#define     cgtod(fs, c) (cgstart(fs, c) + (fs)->fs_cblkno)    /* cg block */
#define     cgimin(fs, c)           (cgstart(fs, c) + (fs)->fs_iblkno)    /* inode blk */
#define     cgdmin(fs, c)           (cgstart(fs, c) + (fs)->fs_dblkno)    /* 1st data */

/*
 * Macros for handling inode numbers:
 *     inode number to file system block offset.
 *     inode number to cylinder group number.
 *     inode number to file system block address.
 */
#define     itoo(fs, x)   ((x) % INOPB(fs))
#define     itog(fs, x)   ((x) / (fs)->fs_ipg)
#define     itod(fs, x) \
            ((daddr_t)(cgimin(fs, itog(fs, x)) + \
            (blkstofrags((fs), (((x) % (fs)->fs_ipg) / INOPB(fs))))))
```

fs.h (cont.)

```
/*
 * Give cylinder group number for a file system block.
 * Give cylinder group block number for a file system block.
 */
#define     dtog(fs, d)  ((d) / (fs)->fs_fpg)
#define     dtogd(fs, d)((d) % (fs)->fs_fpg)

/*
 * Extract the bits for a block from a map.
 * Compute the cylinder and rotational position of a cyl block addr.
 */
#define blkmap(fs, map, loc) \
    (((map)[(loc) / NBBY] >> ((loc) % NBBY)) & (0xff >> (NBBY - (fs)->fs_frag)))
#define cbtocylno(fs, bno) \
    ((bno) * NSPF(fs) / (fs)->fs_spc)
#define cbtorpos(fs, bno) \
    ((bno) * NSPF(fs) % (fs)->fs_spc % (fs)->fs_nsect * NRPOS / (fs)->fs_nsect)

/*
 * The following macros optimize certain frequently calculated
 * quantities by using shifts and masks in place of divisions
 * modulos and multiplications.
 */
#define blkoff(fs, loc)             /* calculates (loc % fs->fs_bsize) */ \
            ((loc) & ~(fs)->fs_bmask)
#define fragoff(fs, loc) /* calculates (loc % fs->fs_fsize) */ \
            ((loc) & ~(fs)->fs_fmask)
#define lblkno(fs, loc)             /* calculates (loc / fs->fs_bsize) */ \
            ((loc) >> (fs)->fs_bshift)
#define numfrags(fs, loc)           /* calculates (loc / fs->fs_fsize) */ \
            ((loc) >> (fs)->fs_fshift)
#define blkroundup(fs, size)        /* calculates roundup(size, fs->fs_bsize) */ \
            (((size) + (fs)->fs_bsize - 1) & (fs)->fs_bmask)
#define fragroundup(fs, size)       /* calculates roundup(size, fs->fs_fsize) */ \
            (((size) + (fs)->fs_fsize - 1) & (fs)->fs_fmask)
#define fragstoblks(fs, frags)      /* calculates (frags / fs->fs_frag) */ \
            ((frags) >> (fs)->fs_fragshift)
#define blkstofrags(fs, blks)       /* calculates (blks * fs->fs_frag) */ \
            ((blks) << (fs)->fs_fragshift)
#define fragnum(fs, fsb)            /* calculates (fsb % fs->fs_frag) */ \
            ((fsb) & ((fs)->fs_frag - 1))
#define blknum(fs, fsb)             /* calculates rounddown(fsb, fs->fs_frag) */ \
            ((fsb) &~ ((fs)->fs_frag - 1))
```

fs.h (cont.)

```
/*
 * Determine the number of available frags given a
 * percentage to hold in reserve
 */
#define freespace(fs, percentreserved) \
        (blkstofrags((fs), (fs)->fs_cstotal.cs_nbfree) + \
        (fs)->fs_cstotal.cs_nffree - ((fs)->fs_dsize * (percentreserved) / 100))

/*
 * Determining the size of a file block in the file system.
 */
#define blksize(fs, ip, lbn) \
        (((lbn) >= NDADDR || (ip)->i_size >= ((lbn) + 1) << (fs)->fs_bshift) \
            ? (fs)->fs_bsize \
            : (fragroundup(fs, blkoff(fs, (ip)->i_size))))
#define dblksize(fs, dip, lbn) \
        (((lbn) >= NDADDR || (dip)->di_size >= ((lbn) + 1) << (fs)->fs_bshift) \
            ? (fs)->fs_bsize \
            : (fragroundup(fs, blkoff(fs, (dip)->di_size))))

/*
 * Number of disk sectors per block; assumes DEV_BSIZE byte sector size.
 */
#define     NSPB(fs)    ((fs)->fs_nspf << (fs)->fs_fragshift)
#define     NSPF(fs)    ((fs)->fs_nspf)

/*
 * INOPB is the number of inodes in a secondary storage block.
 */
#define     INOPB(fs)   ((fs)->fs_inopb)
#define     INOPF(fs)   ((fs)->fs_inopb >> (fs)->fs_fragshift)

/*
 * NINDIR is the number of indirects in a file system block.
 */
#define     NINDIR(fs)  ((fs)->fs_nindir)
```

hat.h

```
/*          @(#)hat.h  1.7 88/02/08 SMI      */
/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#ifndef _vm_hat_h
#define _vm_hat_h
/*
 * VM - Hardware Address Translation management.
 *
 * This file describes the machine independent interfaces to
 * the hardware address translation management routines.  Other
 * machine specific interfaces and structures are defined
 * in <machine/vm_hat.h>.  The hat layer manages the address
 * translation hardware as a cache driven by calls from the
 * higher levels of the VM system.
 */

#include <machine/vm_hat.h>

#ifdef KERNEL
/*
 * One time hat initialization
 */
void        hat_init();

/*
 * Operations on hat resources for an address space:
 *          - initialize any needed hat structures for the address space
 *          - free all hat resources now owned by this address space
 *
 * N.B. - The hat structure is guaranteed to be zeroed when created.
 * The hat layer can choose to define hat_alloc as a macro to avoid
 * a subroutine call if this is sufficient initialization.
 */
#ifndef hat_alloc
void        hat_alloc(/* as */);
#endif
void        hat_free(/* as */);
```

hat.h (cont.)

```
/*
 * Operations on a named address with in a segment:
 *          - load/lock the given page struct
 *          - load/lock the given page frame number
 *          - unlock the given address
 *
 * (Perhaps we need an interface to load several pages at once?)
 */
void        hat_memload(/* seg, addr, pp, prot, lock */);
void        hat_devload(/* seg, addr, pf, prot, lock */);
void        hat_unlock(/* seg, addr */);

/*
 * Operations over an address range:
 *          - change protections
 *          - change mapping to refer to a new segment
 *          - unload mapping
 */
void        hat_chgprot(/* seg, addr, len, prot */);
void        hat_newseg(/* seg, addr, len, nseg */);
void        hat_unload(/* seg, addr, len */);

/*
 * Operations that work on all active translation for a given page:
 *          - unload all translations to page
 *          - get hw stats from hardware into page struct and reset hw stats
 */
void        hat_pageunload(/* pp */);
void        hat_pagesync(/* pp */);

/*
 * Operations that return physical page numbers (ie - used by mapin):
 *          - return the pfn for kernel virtual address
 *          - return the pfn for arbitrary virtual address
 */
u_int       hat_getkpfnum(/* addr */);
/*
 * XXX - This one is not yet implemented - not yet needed
 * u_int hat_getpfnum(as, addr);
 */

#endif KERNEL
#endif !_vm_hat_h
```

inode.h

```
/*      @(#)inode.h 2.28 88/02/08 SMI; from UCB 4.24 83/07/01       */

/*
 * The I node is the focus of all local file activity in UNIX.
 * There is a unique inode allocated for each active file,
 * each current directory, each mounted-on file, each mapping,
 * and the root.  An inode is 'named' by its dev/inumber pair.
 * Data in icommon is read in from permanent inode on volume.
 */

#define     NDADDR  12                  /* direct addresses in inode */
#define     NIADDR  3                   /* indirect addresses in inode */

struct inode {
        struct      inode *i_chain[2];      /* must be first */
        struct      vnode i_vnode;          /* vnode associated with this inode */
        struct      vnode *i_devvp;         /* vnode for block I/O */
        u_short     i_flag;
        dev_t       i_dev;                  /* device where inode resides */
        ino_t       i_number;   /* i number, 1-to-1 with device address */
        off_t       i_diroff;   /* offset in dir, where we found last entry */
        struct      fs *i_fs;   /* file sys associated with this inode */
        struct      dquot *i_dquot;         /* quota structure controlling this file */
        short       i_owner;    /* proc index of process locking inode */
        short       i_count;    /* number of inode locks for i_owner */
        union {
                    daddr_t     if_nextr;   /* next byte read offset (read-ahead) */
                    struct      socket *is_socket;
        } i_un;
        struct      {
                    struct inode  *if_freef; /* free list forward */
                    struct inode **if_freeb; /* free list back */
        } i_fr;
        struct      icommon {
                    u_short     ic_mode;    /*  0: mode and type of file */
                    short       ic_nlink;   /*  2: number of links to file */
                    short       ic_uid;             /*  4: owner's user id */
                    short       ic_gid;             /*  6: owner's group id */
                    quad        ic_size;    /*  8: number of bytes in file */
```

inode.h (cont.)

```
#ifdef KERNEL
                struct timeval ic_atime;/* 16: time last accessed */
                struct timeval ic_mtime;/* 24: time last modified */
                struct timeval ic_ctime;/* 32: last time inode changed */
#else
                time_t      ic_atime;   /* 16: time last accessed */
                long        ic_atspare;
                time_t      ic_mtime;   /* 24: time last modified */
                long        ic_mtspare;
                time_t      ic_ctime;   /* 32: last time inode changed */
                long        ic_ctspare;
#endif
                daddr_t     ic_db[NDADDR];      /* 40: disk block addresses */
                daddr_t     ic_ib[NIADDR];      /* 88: indirect blocks */
                long        ic_flags;   /* 100: status, currently unused */
                long        ic_blocks;  /* 104: blocks actually held */
                long        ic_gen;             /* 108: generation number */
                long        ic_spare[4];/* 112: reserved, currently unused */
        } i_ic;
};
```

inode.h (cont.)

```
struct dinode {
        union {
                struct  icommon di_icom;
                char    di_size[128];
        } di_un;
};

#define     i_mode              i_ic.ic_mode
#define     i_nlink             i_ic.ic_nlink
#define     i_uid               i_ic.ic_uid
#define     i_gid               i_ic.ic_gid
/* ugh! -- must be fixed */
#if defined(vax) || defined(i386)
#define     i_size              i_ic.ic_size.val[0]
#endif
#if defined(mc68000) || defined(sparc)
#define     i_size              i_ic.ic_size.val[1]
#endif
#define     i_db                i_ic.ic_db
#define     i_ib                i_ic.ic_ib
#define     i_atime             i_ic.ic_atime
#define     i_mtime             i_ic.ic_mtime
#define     i_ctime             i_ic.ic_ctime
#define i_blocks        i_ic.ic_blocks
#define     i_rdev              i_ic.ic_db[0]
#define     i_gen               i_ic.ic_gen
#define     i_nextr             i_un.if_nextr
#define     i_socket    i_un.is_socket
#define     i_forw              i_chain[0]
#define     i_back              i_chain[1]
#define     i_freef             i_fr.if_freef
#define     i_freeb             i_fr.if_freeb
```

inode.h (cont.)

```
#define di_ic                   di_un.di_icom
#define     di_mode             di_ic.ic_mode
#define     di_nlink    di_ic.ic_nlink
#define     di_uid              di_ic.ic_uid
#define     di_gid              di_ic.ic_gid
#if defined(vax) || defined(i386)
#define     di_size             di_ic.ic_size.val[0]
#endif
#if defined(mc68000) || defined(sparc)
#define     di_size             di_ic.ic_size.val[1]
#endif
#define     di_db               di_ic.ic_db
#define     di_ib               di_ic.ic_ib
#define     di_atime    di_ic.ic_atime
#define     di_mtime    di_ic.ic_mtime
#define     di_ctime    di_ic.ic_ctime
#define     di_rdev             di_ic.ic_db[0]
#define     di_blocks   di_ic.ic_blocks
#define     di_gen              di_ic.ic_gen

/* flags */
#define     ILOCKED             0x001           /* inode is locked */
#define     IUPD                0x002           /* file has been modified */
#define     IACC                0x004           /* inode access time to be updated */
#define     IMOD                0x008           /* inode has been modified */
#define     IWANT               0x010           /* some process waiting on lock */
#define     ISYNC               0x020           /* do all allocation synchronously */
#define     ICHG                0x040           /* inode has been changed */
#define     ILWAIT              0x080           /* someone waiting on file lock */
#define     IREF                0x100           /* inode is being referenced */
#define     INOACC              0x200           /* no access time update in getpage */
#define     IMODTIME            0x400           /* mod time already set */

/* modes */
#define     IFMT                0170000         /* type of file */
#define     IFIFO               0010000         /* named pipe (fifo) */
#define     IFCHR               0020000         /* character special */
#define     IFDIR               0040000         /* directory */
#define     IFBLK               0060000         /* block special */
#define     IFREG               0100000         /* regular */
#define     IFLNK               0120000         /* symbolic link */
#define     IFSOCK              0140000         /* socket */
```

inode.h (cont.)

```
#define     ISUID           04000           /* set user id on execution */
#define     ISGID           02000           /* set group id on execution */
#define     ISVTX           01000           /* save swapped text even after use */
#define     IREAD           0400            /* read, write, execute permissions */
#define     IWRITE          0200
#define     IEXEC           0100

#ifdef KERNEL
struct inode *inode;                        /* the inode table itself */
struct inode *inodeNINODE;                  /* the end of the inode table */
int ninode;                                 /* number of slots in the table */

extern struct vnodeops ufs_vnodeops;        /* vnode operations for ufs */

extern ino_t dirpref();
extern daddr_t blkpref();

/*
 * A struct fbuf is used to get a mapping to part of a file using the
 * segkmap facilities.  After you get a mapping, you can fbrelse() it
 * (giving a seg code to pass back to segmap_release), you can fbwrite()
 * it (causes a synchronous write back using inode mapping information),
 * or you can fbiwrite it (causing indirect synchronous write back to
 * the block number given without using inode mapping information).
 */

struct fbuf {
        addr_t      fb_addr;
        u_int       fb_count;
};

extern int fbread(/* vp, off, len, rw, fbpp */);
extern void fbzero(/* vp, off, len, fbpp */);
extern int fbwrite(/* fb */);
extern int fbiwrite(/* fb, ip, bn */);
extern void fbrelse(/* fb, rw */);
```

inode.h (cont.)

```
/*
 * Convert between inode pointers and vnode pointers
 */
#define VTOI(VP)        ((struct inode *)(VP)->v_data)
#define ITOV(IP)        ((struct vnode *)&(IP)->i_vnode)

/*
 * Convert between vnode types and inode formats
 */
extern enum vtype       iftovt_tab[];
extern int              vttoif_tab[];
#define IFTOVT(M)       (iftovt_tab[((M) & IFMT) >> 13])
#define VTTOIF(T)       (vttoif_tab[(int)(T)])

#define MAKEIMODE(T, M)         (VTTOIF(T) | (M))

/*
 * Lock and unlock inodes.
 */
#define     ILOCK(ip) { \
            while (((ip)->i_flag & ILOCKED) && \
                (ip)->i_owner != u.u_procp - proc) { \
                        (ip)->i_flag |= IWANT; \
                        (void) sleep((caddr_t)(ip), PINOD); \
            } \
            (ip)->i_owner = u.u_procp - proc; \
            (ip)->i_count++; \
            (ip)->i_flag |= ILOCKED; \
            if (((ip)->i_vnode.v_flag & VISSWAP) != 0) \
                        u.u_procp->p_swlocks++; \
}
```

inode.h (cont.)

```
#define     IUNLOCK(ip) { \
            if (--(ip)->i_count < 0) \
                        panic("IUNLOCK"); \
            if (((ip)->i_vnode.v_flag & VISSWAP) != 0) \
                        u.u_procp->p_swlocks--; \
            if ((ip)->i_count == 0) { \
                        (ip)->i_flag &= ~ILOCKED; \
                        if ((ip)->i_flag & IWANT) { \
                                    (ip)->i_flag &= ~IWANT; \
                                    wakeup((caddr_t)(ip)); \
                        } \
            } \
}

#define IUPDAT(ip, waitfor) { \
            if (ip->i_flag & (IUPD|IACC|ICHG|IMOD)) \
                        iupdat(ip, waitfor); \
}

/*
 * Mark an inode with the current (unique) timestamp.
 */
struct timeval iuniqtime;

#define IMARK(ip) { \
            if (time.tv_sec > iuniqtime.tv_sec || \
                time.tv_usec > iuniqtime.tv_usec) { \
                        iuniqtime = time; \
            } else { \
                        iuniqtime.tv_usec++; \
            } \
            if ((ip)->i_flag & IACC) \
                        (ip)->i_atime = iuniqtime; \
            if ((ip)->i_flag & IUPD) { \
                        (ip)->i_mtime = iuniqtime; \
                        (ip)->i_flag |= IMODTIME; \
            } \
            if ((ip)->i_flag & ICHG) { \
                        ip->i_diroff = 0; \
                        (ip)->i_ctime = iuniqtime; \
            } \
}
```

inode.h (cont.)

```
#define ITIMES(ip) { \
        if ((ip)->i_flag & (IUPD|IACC|ICHG)) { \
                (ip)->i_flag |= IMOD; \
                IMARK(ip); \
                (ip)->i_flag &= ~(IACC|IUPD|ICHG); \
        } \
}

/*
 * Allocate the specified block in the inode
 * and make sure any in-core pages are initialized.
 */
#define     BMAPALLOC(ip, lbn, size) \
        bmap((ip), (lbn), (daddr_t *)NULL, (daddr_t *)NULL, (size), S_WRITE, 0)

#define ESAME       (-1)                /* trying to rename linked files (special) */

/*
 * Check that file is owned by current user or user is su.
 */
#define OWNER(CR, IP)           (((CR)->cr_uid == (IP)->i_uid)? 0: (suser()? 0: EPERM))

#define     UFS_HOLE            (daddr_t)-1         /* value used when no block allocated */

/*
 * enums
 */
enum de_op          { DE_CREATE, DE_LINK, DE_RENAME };  /* direnter ops */

/*
 * This overlays the fid structure (see vfs.h)
 */
struct ufid {
        u_short     ufid_len;
        ino_t       ufid_ino;
        long        ufid_gen;
};
#endif KERNEL
```

mbuf.h

```
/*      @(#)mbuf.h 2.16 88/02/08 SMI; from UCB 7.3 9/11/86 */
/*
 * Copyright (c) 1982 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef _sys_mbuf_h
#define _sys_mbuf_h

/*
 * Constants related to memory allocator.
 *
 * Standard 4.3 assumes that the size of a cluster mbuf is PAGESIZE,
 * thus linking (cluster) mbuf size to the system page size.  We
 * choose to break this linkage, instead setting the mbuf cluster
 * size to MCLBYTES, which must be a power of 2 no less than MSIZE.
 * However, when obtaining space from the system, allocation is still
 * performed in units of PAGESIZE.
 *
 * MBPOOLBYTES gives the amount of virtual space to reserve for mbuf
 * storage.  Experience shows that allocation can creep up over half
 * a megabyte, but some machines (notably Sun-2s) can't afford to
 * dedicate that much address space to mbufs.  Thus we case split on
 * the machine type.
 * XXX:  Need better way to handle this problem.
 */
#define MSIZE                   128                             /* size of an mbuf */

#define MMINOFF                 12                              /* mbuf header length */
#define MTAIL                   4
#define MMAXOFF                         (MSIZE-MTAIL)                   /* offset where data
ends */
#define MLEN                    (MSIZE-MMINOFF-MTAIL)   /* mbuf data length */
#define MCLBYTES                1024                            /* size of mbuf clusters */

#ifdef sun2
#define MBPOOLBYTES             ( 512 * 1024)
#else sun2
#define MBPOOLBYTES             (1024 * 1024)
#endif sun2
```

mbuf.h (cont.)

```
/*
 * Macros for type conversion
 */

/* bytes to mclusters, rounding up, and back */
#define     btomcl(x)   (((x) + MCLBYTES - 1) / MCLBYTES)
#define     mcltob(x)   ((x) * MCLBYTES)

/* mcluster index to mbuf address, and back */
#define     cltom(x) ((struct mbuf *)((int)mbutl + (mcltob(x))))
#define     mtocl(x) (((int)x - (int)mbutl) / MCLBYTES)

/* address in mbuf to mbuf head */
#define     dtom(x)                 ((struct mbuf *)((int)x & ~(MSIZE-1)))

/* mbuf head, to typed data */
#define     mtod(x,t)   ((t)((int)(x) + (x)->m_off))
```

mbuf.h (cont.)

```
/*
 * Standard 4.3 supports two kinds of mbufs: normal mbufs, with data
 * residing in the mbuf itself; and cluster mbufs with data stored
 * elsewhere (at a higher address than the corresponding mbuf).
 * The data area for cluster mbufs is allocated as a pool and is
 * managed by the mbuf system itself.  We add another kind of cluster
 * mbuf, whose data area is not part of the mbuf system.  Instead,
 * a routine wishing to use this kind of mbuf must provide a buffer
 * for the storage associated with the mbuf and a pointer to a routine
 * to free the buffer when the mbuf is freed.  The fields in mun_cl
 * below record the necessary bookkeeping information.
 */
struct mbuf {
        struct  mbuf *m_next;                   /* next buffer in chain */
        u_long  m_off;                          /* offset of data */
        short   m_len;                          /* amount of data in this mbuf */
        short   m_type;                         /* mbuf type (0 == free) */
        union {
                u_char  mun_dat[MLEN];          /* data storage */
                struct {
                        short   mun_cltype;             /* "cluster" type */
                        int     (*mun_clfun)();
                        int     mun_clarg;
                        int     (*mun_clswp)();
                } mun_cl;
        } m_un;
        struct  mbuf *m_act;                    /* link in higher-level mbuf list */
};
#define m_dat           m_un.mun_dat
#define m_cltype        m_un.mun_cl.mun_cltype
#define m_clfun         m_un.mun_cl.mun_clfun
#define m_clarg         m_un.mun_cl.mun_clarg
#define m_clswp         m_un.mun_cl.mun_clswp
```

mbuf.h (cont.)

```
/* mbuf types */
#define     MT_FREE         0       /* should be on free list */
#define     MT_DATA         1       /* dynamic (data) allocation */
#define     MT_HEADER       2       /* packet header */
#define     MT_SOCKET       3       /* socket structure */
#define     MT_PCB          4       /* protocol control block */
#define     MT_RTABLE       5       /* routing tables */
#define     MT_HTABLE       6       /* IMP host tables */
#define     MT_ATABLE       7       /* address resolution tables */
#define     MT_SONAME       8       /* socket name */
#define     MT_ZOMBIE       9       /* zombie proc status */
#define     MT_SOOPTS       10      /* socket options */
#define     MT_FTABLE       11      /* fragment reassembly header */
#define     MT_RIGHTS       12      /* access rights */
#define     MT_IFADDR       13      /* interface address */

/*
 * Values for m_cltype: applicable only for cluster mbufs
 */
#define     MCL_STATIC      1       /* data in mbuf cluster pool */
#define     MCL_LOANED      2       /* data allocated elsewhere and "loaned"
                                       to the mbuf */

/* flags to m_get */
#define     M_DONTWAIT      0
#define     M_WAIT          1

/* flags to m_pgalloc */
#define     MPG_MBUFS       0               /* put new mbufs on free list */
#define     MPG_CLUSTERS    1               /* put new clusters on free list */
#define     MPG_SPACE       2               /* don't free; caller wants space */

/* length to m_copy to copy all */
#define     M_COPYALL       1000000000

/*
 * m_pullup will pull up additional length if convenient;
 * should be enough to hold headers of second-level and higher protocols.
 */
#define     MPULL_EXTRA     32/*
 * Macros for mbuf manipulation.
 *
```

mbuf.h (cont.)

```
* Standard 4.3 defines some additional macros not given here
 * (MTOCL, MCLFREE), as they are useful only within uipc_mbuf.c
 * or can't cope with the existence of the MCL_LOANED cluster
 * mbuf type.
 *
 * MCLALLOC allocates MCL_STATIC clusters.  Note that it works
 * only with a count of 1 at the moment.  MCLGET connects such clusters
 * to a normal mbuf.  m->m_len is set to MCLBYTES upon success.
 */

#define     MGET(m, i, t) \
            { int ms = splimp(); \
              if ((m)=mfree) \
                        { if ((m)->m_type != MT_FREE) panic("mget"); (m)->m_type = t; \
                          mbstat.m_mtypes[MT_FREE]--; mbstat.m_mtypes[t]++; \
                          mfree = (m)->m_next; (m)->m_next = 0; \
                          (m)->m_off = MMINOFF; } \
              else \
                        (m) = m_more(i, t); \
              (void) splx(ms); }

#define     M_HASCL(m)          ((m)->m_off >= MSIZE)

#define     MCLALLOC(m, i) \
            { int ms = splimp(); \
              if (mclfree == 0) \
                        (void)m_clalloc(1, MPG_CLUSTERS, M_DONTWAIT); \
              if ((m)=mclfree) \
                 {++mclrefcnt[mtocl(m)];mbstat.m_clfree--;mclfree = (m)->m_next;} \
              splx(ms); }

/*
 * N.B.: no success/failure indication given.
 * Use mclget if you care.
 */
#define     MCLGET(m) \
            { struct mbuf *p; \
              MCLALLOC(p, 1); \
              if (p) { \
                        (m)->m_off = (int)p - (int)(m); \
                        (m)->m_len = MCLBYTES; \
                        (m)->m_cltype = MCL_STATIC; \
              } \
              else \
                        (m)->m_len = 0; \
            }
```

mbuf.h (cont.)

```
#define     MFREE(m, n) \
            { int ms = splimp(); \
              if ((m)->m_type == MT_FREE) panic("mfree"); \
              mbstat.m_mtypes[(m)->m_type]--; mbstat.m_mtypes[MT_FREE]++; \
              (m)->m_type = MT_FREE; \
              if (M_HASCL(m)) \
                        mclput(m); \
              (n) = (m)->m_next; (m)->m_next = mfree; \
              (m)->m_off = 0; (m)->m_act = 0; mfree = (m); \
              (void) splx(ms); \
              if (m_want) { \
                        m_want = 0; \
                        wakeup((caddr_t)&mfree); \
              } \
            }

/*
 * Mbuf statistics.
 */
struct mbstat {
            short       m_mbufs;  /* mbufs obtained from page pool */
            short       m_clusters; /* clusters obtained from page pool */
            short       m_clfree;  /* free clusters */
            short       m_drops;  /* times failed to find space */
            short       m_mtypes[256];        /* type specific mbuf allocations */
};

#ifdef      KERNEL
extern      struct mbuf mbutl[];                /* virtual address of net free mem */
extern      struct pte Mbmap[];                 /* page tables to map mbutl */
struct      mbstat mbstat;
struct      mbuf *mfree, *mclfree;
int         m_want;
struct      mbuf *m_get(),*m_getclr(),*m_free(),*m_more(),*m_copy(),*m_pullup();
struct      mbuf *mclgetx();
caddr_t     m_clalloc();
#endif

#endif      _sys_mbuf_h
```

mmu.h

```
/*      @(#)mmu.h 1.16 88/02/08 SMI      */
/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */
#ifndef _MMU_
#define _MMU_
/*
 * Sun-3 memory management.
 *
 * This file gives more mmu specific constants and defines
 * the interface to some pretty low level mmu routines.
 */
/*
 * Process Address space is 28 bits
 */
#define     ADDRESS_MASK      0x0fffffff   /* Mask of valid bits in an address */
/*
 * Hardware context and segment information
 * Mnemonic decoding:
 *          PMENT - Page Map ENTry
 *          PMGRP - Group of PMENTs (aka "segment")
 */
#define     MNCTXS                      8         /* Number of mapping contexts */
#define     NPMGRPPERCTX                0x800     /* 2048 */
#define     MNPMGRPS                    0x100     /* 256 */
#define     NPMENTPERPMGRP              0x10      /* 16 */
#define     NPMENTPERPMGRPSHIFT         4         /* log2(NPMENTPERPMGRP) */
/*
 * Useful defines for hat constants
 */
#define NCTXS           MNCTXS
#define NPMGRPS         MNPMGRPS
/*
 * More useful manifest constants
 */
#define     TOTALPMGRP                  (NCTXS * NPMGRPPERCTX)
#define     PMGRPSIZE                   (NPMENTPERPMGRP * PAGESIZE)
#define     MNPMENTS                    (NPMGRPS * NPMENTPERPMGRP)
#define     PMGRP_INVALID               (NPMGRPS - 1)
#define     CTXSIZE                     (PMGRPSIZE * NPMGRPPERCTX)
#define     CTXMASK                     (NCTXS - 1)
#define     PMGRPOFFSET                 (PMGRPSIZE - 1)
#define     PMGRPMASK                   (-PMGRPOFFSET)
#define     PMGRPSHIFT                  (PAGESHIFT + NPMENTPERPMGRPSHIFT)
```

mmu.h (cont.)

```
/*
 * XXX - Old names for some backwards compatibility
 */
#define     NBSG                PMGRPSIZE
#define     SGOFSET             PMGRPOFFSET
#define     SGSHIFT             PMGRPSHIFT

/*
 * 680x0 code register values.
 */
#define     FC_UD   1           /* user data */
#define     FC_UP   2           /* user program */
#define     FC_MAP  3           /* Sun-3 memory maps */
#define     FC_SD   5           /* supervisor data */
#define     FC_SP   6           /* supervisor program */
#define     FC_CPU  7           /* cpu space */

/*
 * FC_MAP base offsets, see architecture manual for definitions
 */
#define     ID_PROM             0x00000000
#define     PAGE_MAP            0x10000000
#define     SEGMENT_MAP         0x20000000
#define     CONTEXT_REG         0x30000000
#define     SYSTEM_ENABLE       0x40000000
#define     USER_DVMA_ENABLE    0x50000000
#define     BUS_ERROR_REG       0x60000000
#define     DIAGNOSTIC_REG      0x70000000
#define     CACHE_TAGS          0x80000000
#define     CACHE_DATA          0x90000000
#define     FLUSH_CACHE         0xa0000000
#define     BLOCK_COPY          0xb0000000
                                                /* Gap: 0xc, d, e */
#define     UART_BYPASS         0xf0000000

#define     IDPROMSIZE          0x20            /* size of id prom in bytes */

/*
 * We have to define the control mask to strip off the low 3 order
 * bits of the address maks in order to get long word aligned addresses.
 * This little tid bit was left out of the Sun-3 Architecture Manual.
 */
#define     CONTROL_MASK        0x0ffffffc
```

mmu.h (cont.)

```
/*
 * 68020 Cache Control Register
 */
#define CACHE_ENABLE        0x1     /* enable the cache */
#define CACHE_FREEZE        0x2     /* freeze the cache */
#define CACHE_CLRENTRY      0x4     /* clear entry specified by cache addr reg */
#define CACHE_CLEAR         0x8     /* clear entire cache */

/*
 * Sun-3 Virtual Address Cache (VAC)
 * The VAC h/w has a counter cycles bits 4 through bit 8.  We cycle
 * bit 9 through bit 15 for 64KB VAC.  For 128KB VAC, we cycle bit 9
 * through bit 16.  (There is only one size of VAC for Sun-3 260.
 * If there is more than one VAC size, VAC_FLUSH_HIGHBIT should be
 * a variable whose value is set at boot time.)
 */
#define VAC_FLUSH_BASE              0xa0000000
#define VAC_FLUSH_LOWBIT    9
#define VAC_FLUSH_HIGHBIT   15      /* 64KB VAC */
#define VAC_FLUSH_INCRMNT   512     /* 2 << VAC_LOW_BIT */
#define VAC_UNIT_ADDRBITS   0x0ffffe00
#define VAC_UNIT_MASK               0x000001ff

/*
 * Number of cycles to flush a context, segment, and page.
 * For a context flush or segment flush, we loop
 * 2 << (VAC_HIGH_BIT - VAC_LOW_BIT + 1) times, and for a page flush,
 * we loop 2 << (PHSHIFT - VAC_LOW_BIT) times.
 */
#define VAC_CTXFLUSH_COUNT          128
#define VAC_SEGFLUSH_COUNT          128
#define VAC_PAGEFLUSH_COUNT         16

/* contants to do a flush */
/* VAC_FLUSHALL is used for debugging only for SUN3_260 */
#define VAC_FLUSHALL                0
#define VAC_CTXFLUSH                0x01
#define VAC_SEGFLUSH                0x03
#define VAC_PAGEFLUSH               0x02
```

mmu.h (cont.)

```
/*
 * VAC Read/Write Cache Tags.
 * R/W tag is for one line in the cache, which has 16 bytes.
 * VAC_RWTAG_COUNT is 2 ^ (VAC_RWTAG_HIGHBIT - VAC_RWTAG_LOWBIT + 1) - 1
 */
#define VAC_RWTAG_BASE                  0x80000000
#define VAC_RWTAG_LOWBIT  4
#define VAC_RWTAG_HIGHBIT  15           /* 64KB VAC */
#define VAC_RWTAG_INCRMNT 16            /* 2 ^ VAC_RWTAG_LOWBIT */
#define VAC_RWTAG_COUNT                 4096

/*
 * Block Copy Read and Block Copy Write
 */
#define     BC_LINESIZE                 16
#define     BC_LINE_SHIFT               2
#define     BC_LINE_RESIDU              3
#define BC_BLOCK_CPCMD                  0xb0000000
#define BC_BLOCK_OFF                    0x0fffffff

/*
 * Masks for base addresses when doing VAC flush operations
 */
#define     VAC_PAGEMASK                0x0fffe000
#define     VAC_SEGMASK                 0x0ffe0000

/*
 * Context for kernel.  On a Sun-3 the kernel is in every address space,
 * but KCONTEXT is magic in that there is never any user context there.
 */
#define     KCONTEXT            0

#if defined(KERNEL) && !defined(LOCORE)
/*
 * Low level mmu-specific functions
 */
struct      ctx *mmu_getctx();
void        mmu_setctx(/* ctx */);
void        mmu_setpmg(/* base, pmg */);
void        mmu_settpmg(/* base, pmg */);
struct      pmgrp *mmu_getpmg(/* base */);
void        mmu_setpte(/* base, pte */);
void        mmu_getpte(/* base, ppte */);
void        mmu_getkpte(/* base, ppte */);
void        mmu_pmginval(/* pmg */);
```

mmu.h (cont.)

```
/*
 * Cache specific routines - ifdef'ed out if there is no chance
 * of running on a machine with a virtual address cache.
 */
#ifdef VAC
void        vac_init();
void        vac_flushall();
void        vac_ctxflush();
void        vac_segflush(/* base */);
void        vac_pageflush(/* base */);
void        vac_flush(/* base, len */);
#else VAC
#define     vac_init()
#define     vac_flushall()
#define     vac_ctxflush()
#define     vac_segflush(base)
#define     vac_pageflush(base)
#define     vac_flush(base, len)
#endif VAC

void        sun_setcontrol_long(/* space, addr */);
u_int       sun_getcontrol_long(/* space, addr, value */);
void        sun_setcontrol_byte(/* space, addr */);
u_int       sun_getcontrol_byte(/* space, addr, value */);
#endif defined(KERNEL) && !defined(LOCORE)

#endif !_MMU_
```

mount.h

```
/*      @(#)mount.h 1.18 88/02/08 SMI  */

/*
 * mount options
 */
#define M_RDONLY   0x01              /* mount fs read only */
#define M_NOSUID   0x02              /* mount fs with setuid not allowed */
#define     M_NEWTYPE       0x04            /* use type string instead of int */
#define     M_GRPID         0x08            /* Old BSD group-id on create */
#define M_REMOUNT 0x10               /* change options on an existing mount */
#define M_NOSUB             0x20            /* Disallow mounts beneath this mount */
#define M_MULTI             0x40            /* Do multi-component lookup on files */

#ifdef KERNEL
/*
 * File system types, these correspond to entries in fsconf
 */
#define     MOUNT_UFS       1
#define     MOUNT_NFS       2
#define     MOUNT_PC        3
#define     MOUNT_LO        4
#define     MOUNT_MAXTYPE   5
#endif KERNEL

struct ufs_args {
        char        *fspec;
};

#ifdef NFSCLIENT

struct nfs_args {
        struct sockaddr_in      *addr;          /* file server address */
        caddr_t                 fh;             /* File handle to be mounted */
        int                     flags;          /* flags */
        int                     wsize;          /* write size in bytes */
        int                     rsize;          /* read size in bytes */
        int                     timeo;          /* initial timeout in .1 secs */
        int                     retrans;    /* times to retry send */
        char                    *hostname;  /* server's hostname */
        int                     acregmin;   /* attr cache file min secs */
        int                     acregmax;   /* attr cache file max secs */
        int                     acdirmin;   /* attr cache dir min secs */
        int                     acdirmax;   /* attr cache dir max secs */
        char                    *netname;   /* server's netname */
};
```

mount.h (cont.)

```
/*
 * NFS mount option flags
 */
#define     NFSMNT_SOFT         0x001     /* soft mount (hard is default) */
#define     NFSMNT_WSIZE        0x002     /* set write size */
#define     NFSMNT_RSIZE        0x004     /* set read size */
#define     NFSMNT_TIMEO        0x008     /* set initial timeout */
#define     NFSMNT_RETRANS      0x010     /* set number of request retrys */
#define     NFSMNT_HOSTNAME               0x020     /* set hostname for error printf */
#define     NFSMNT_INT          0x040     /* allow interrupts on hard mount */
#define     NFSMNT_NOAC         0x080     /* don't cache attributes */
#define     NFSMNT_ACREGMIN 0x0100        /* set min secs for file attr cache */
#define     NFSMNT_ACREGMAX               0x0200     /* set max secs for file attr cache */
#define     NFSMNT_ACDIRMIN 0x0400        /* set min secs for dir attr cache */
#define     NFSMNT_ACDIRMAX               0x0800     /* set max secs for dir attr cache */
#define NFSMNT_SECURE       0x1000        /* secure mount */
#endif NFSCLIENT

#ifdef PCFS
struct pc_args {
        char        *fspec;
};
#endif PCFS

#ifdef LOFS
struct lo_args {
        char    *fsdir;
};
#endif LOFS

#ifdef RFS

struct token {
        int         t_id;         /* token id for differentiating multiple ckts    */
        char        t_uname[64]; /* full domain name of machine, 64 = MAXDNAME */
};

struct rfs_args {
        char    *rmtfs;                     /* name of service (fs) */
        struct token *token;    /* identifier of remote mach */
};

/*
 * RFS mount option flags
 */
#define RFS_RDONLY 0x001        /* read-only: passed with remote mount request */
#endif
```

page.h

```
/*      @(#)page.h 1.32 89/01/11 SMI    */
/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#ifndef _vm_page_h
#define _vm_page_h
/*
 * VM - Ram pages.
 *
 * Each physical page has a page structure, which is used to maintain
 * these pages as a cache.  A page can be found via a hashed lookup
 * based on the [vp, offset].  If a page has an [vp, offset] identity,
 * then it is entered on a doubly linked circular list off the
 * vnode using the vpnext/vpprev pointers.  If the p_free bit
 * is on, then the page is also on a doubly linked circular free
 * list using next/prev pointers.  If the p_intrans bit is on,
 * then the page is currently being read in or written back.
 * In this case, the next/prev pointers are used to link the
 * pages together for a consecutive IO request.  If the page
 * is in transit and the the page is coming in (pagein), then you
 * must wait for the IO to complete before you can attach to the page.
 *
 */
struct page {
        u_int   p_lock: 1,      /* locked for name manipulation */
                p_want: 1,      /* page wanted */
                p_free: 1,      /* on free list */
                p_intrans: 1,           /* data for [vp, offset] intransit */
                p_gone: 1,      /* page has been released */
                p_mod: 1,       /* software copy of modified bit */
                p_ref: 1,       /* software copy of reference bit */
                p_pagein: 1,            /* being paged in, data not valid */
                p_nc: 1,        /* do not cache page */
                p_age: 1;       /* on age free list */
        u_int   p_nio : 6;      /* # of outstanding io reqs needed */
        u_short p_keepcnt;      /* number of page 'keeps' */
        struct  vnode *p_vnode;         /* logical vnode this page is from */
        u_int   p_offset;       /* offset into vnode for this page */
        struct  page *p_hash;           /* hash by [vnode, offset] */
        struct  page *p_next;           /* next page in free/intrans lists */
```

page.h (cont.)

```
        struct  page *p_prev;                   /* prev page in free/intrans lists */
        struct  page *p_vpnext;                 /* next page in vnode list */
        struct  page *p_vpprev;                 /* prev page in vnode list */
        caddr_t p_mapping;          /* hat specific translation info */
};/*
 * Each segment of physical memory is described by a memseg struct. Within
 * a segment, memory is considered contiguous. The segments from a linked
 * list to describe all of physical memory. The list is ordered by increasing
 * physical addresses.
 */
struct memseg {
        struct page *pages, *epages;    /* [from, to) in page array */
        u_int pages_base, pages_end;     /* [from, to) in page numbers */
        struct memseg *next;            /* next segment in list */
};

#ifdef KERNEL
#define PAGE_HOLD(pp)           (pp)->p_keepcnt++
#define PAGE_RELE(pp)           page_rele(pp)

#define     PAGE_HASHSZ         page_hashsz

extern      int page_hashsz;
extern      struct page **page_hash;

/*
 * In the long term, we should generalize the next three values to
 * a tuple controlled at a higher level to allow for non-contiguous
 * memory layout.
 */
extern      struct      page *pages;                    /* array of all page structures */
extern      struct      page *epages;                   /* end of all pages */
#ifndef sun3x
extern      u_int       pages_base;         /* page # for pages[0] */
#endif !sun3x

#ifdef sun386
extern      struct      page *epages2;                  /* end of absolutely all pages */
extern      u_int       pages_base2;                    /* page # for compaq expanded mem */
#endif
```

page.h (cont.)

```
void        page_init(/* pp, num, base */);
void        page_reclaim(/* pp */);
struct      page *page_find(/* vp, off */);
struct      page *page_lookup(/* vp, off */);
int         page_enter(/* pp, vp, off */);
void        page_abort(/* pp */);
void        page_free(/* pp */);
void        page_unfree(/* pp */);
struct      page *page_get();
void        page_rele(/* pp */);
void        page_lock(/* pp */);
void        page_unlock(/* pp */);
void        page_hashout(/* pp */);
void        page_add(/* ppp, pp */);
void        page_sub(/* ppp, pp */);
void        page_sortadd(/* ppp, pp */);
void        page_wait(/* pp */);
u_int       page_pptonum(/* pp */);
struct      page *page_numtopp(/* pfnum */);
struct      page *page_numtookpp(/* pfnum */);
#endif KERNEL

/*
 * Page hash table is a power-of-two in size, externally chained
 * through the hash field.  PAGE_HASHAVELEN is the average length
 * desired for this chain, from which the size of the page_hash
 * table is derived at boot time and stored in the kernel variable
 * page_hashsz.  In the hash function it is given by PAGE_HASHSZ.
 * PAGE_HASHVPSHIFT is defined so that 1 << PAGE_HASHVPSHIFT is
 * the approximate size of a vnode struct.
 */
#define     PAGE_HASHAVELEN         4
#define     PAGE_HASHVPSHIFT        6
#define     PAGE_HASHFUNC(vp, off) \
            ((((off) >> PAGESHIFT) + ((int)(vp) >> PAGE_HASHVPSHIFT)) & \
                    (PAGE_HASHSZ - 1))
#endif !_vm_page_h


/*      @(#)pcb.h 5.7 88/02/08 SMI      */

/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#include <machine/reg.h>
```

pcb.h for Sun3s

```
/*
 * Sun software process control block
 */

#ifndef LOCORE
struct pcb {
        label_t     pcb_regs;   /* saved registers */
        int         pcb_sr;     /* program status word */
        struct      pte *pcb_p0br;          /* pseudo-P0BR for Sun */
        int         pcb_p0lr;   /* pseudo-P0LR for Sun */
        struct      pte *pcb_p1br;          /* pseudo-P1BR for Sun */
        int         pcb_p1lr;   /* pseudo-P1LR for Sun */
        int         pcb_szpt;   /* number of pages of user page table */
        int         *pcb_sswap;
        struct fp_status u_fp_status;       /* user visible fpp state */
/*
 * Struct for the internal state of the MC68881
 * This is the maximum allowed by the coprocessor interface for internal state.
 */
#define     FPIS_BUFSIZ         0x100
        struct fp_istate {
                unsigned char       fpis_vers;   /* version number */
                unsigned char       fpis_bufsiz; /* size of info in fpis_buf */
                unsigned short      fpis_reserved;          /* reserved word */
                unsigned char       fpis_buf[FPIS_BUFSIZ];          /* fpp internal state
buffer */
        } u_fp_istate;          /* internal fpp state */
        /* fpa only */
        short       u_fpa_flags; /* if zero, u_fpa_*'s are meaningless */
        struct fpa_status u_fpa_status; /* saved/restored on ctx switches */
        struct fpa_exception {
                caddr_t     fe_fmtptr; /* points to space holding bus error frame */
                int         fe_pc; /* pc of user code where the bus error occurs */
        } u_fpa_exception;
#define u_fpa_fmtptr            u_fpa_exception.fe_fmtptr
#define u_fpa_pc                u_fpa_exception.fe_pc
        /* end fpa only */
};
#endif
```

pcb.h for Sun3s (cont.)

```
#define     AST_SCHED           0x80000000          /* force a reschedule */
#define     AST_STEP 0x40000000             /* force a single step */
#define     TRACE_USER          0x20000000          /* user has requested tracing */
#define     TRACE_AST           0x10000000          /* AST has requested tracing */
#define     TRACE_PENDING       0x08000000          /* trace caught in supervisor mode */
#define     AST_CLR             0xf8000000
#define     PME_CLR             0
#define     AST_NONE            0

#define     aston()             {u.u_pcb.pcb_p0lr |= AST_SCHED;}
```

pcb.h for Sun4s

```
/*	@(#)pcb.h 1.5 88/02/08 SMI	*/

/*
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

#ifndef _PCB_
#define _PCB_

/*
 * Sun software process control block
 */

#include <machine/reg.h>

#define	MAXWIN	12			/* max number of windows currently supported */

/*
 * The system actually supports one more than the above number.
 * There is always one window reserved for trap handlers that
 * never has to be saved into the pcb struct.
 */

#ifndef LOCORE
struct pcb {
	label_t	pcb_regs;	/* saved pc and sp */
	int	pcb_psr;	/* processor status word */
	struct	pte *pcb_p0br;		/* pseudo-P0BR for Sun */
	int	pcb_p0lr;	/* pseudo-P0LR for Sun */
	struct	pte *pcb_p1br;		/* pseudo-P1BR for Sun */
	int	pcb_p1lr;	/* pseudo-P1LR for Sun */
	int	pcb_szpt;	/* number of pages of user page table */
	int	*pcb_sswap;
	int	pcb_uwm;	/* user window mask */
	struct	rwindow pcb_wbuf[MAXWIN]; /* user window save buffer */
	char	*pcb_spbuf[MAXWIN]; /* sp's for each wbuf */
	int	pcb_wbcnt; /* number of saved windows in pcb_wbuf */
	struct	fpu pcb_fpu;		/* fpu state */
	int	pcb_flags;	/* various state flags */
	int	pcb_wocnt; /* window overflow count */
	int	pcb_wucnt; /* window underflow count */
};
```

pcb.h for Sun4s (cont.)

```
#define pcb_pc                  pcb_regs.val[0]
#define pcb_sp                  pcb_regs.val[1]

#define     aston()             {u.u_pcb.pcb_p0lr |= AST_SCHED;}

#define astoff()        {u.u_pcb.pcb_p0lr &= -AST_SCHED;}

#endif !LOCORE

/* pcb_p0lr */
#define     AST_SCHED           0x80000000          /* force a reschedule */
#define     AST_CLR             0x80000000
#define     PME_CLR             0
#define     AST_NONE            0

/* pcb_flags */
#define CLEAN_WINDOWS           0x1                 /* keep user regs clean */
#define FIX_ALIGNMENT           0x2                 /* fix unaligned references */

#endif _PCB_
```

proc.h

The keeps track of proceses in a memory track of processes on the system by the proc structure. The proc structure is a memory resident structure that keeps all the information that is neede to keep track of a whether it is in currently in memory or swapped out to swap space. The proc structure can be found on at least one of three lists.
If the process is in use and not a zombie process its proc structure will be on the allproc list.
If the process is in the SZOMB state its proc structure will be on the zombproc list.
If the process table entry is not currently in use the corresponding proc structure will be on the freeproc list.

Also active processes that are not currently executable will be on a run queue or a sleep queue, whereas runnable stopped processes are on neither.

The queues are implemented though a set of doubly linked lists, using the p_link and the p_rlink of the proc stucture.

```
struct    proc {
          struct    proc *p_link;          /* linked list of running processes */
          struct    proc *p_rlink;
          struct    proc *p_nxt;           /* linked list of allocated proc slots */
          struct    proc **p_prev;         /* also zombies, and free procs */
          struct    as *p_as;   /* address space description */
          struct    segu_data *p_useg;
                                           /* information required to map the u area */
          char      p_usrpri;   /* user-priority based on p_cpu and p_nice */
          char      p_pri;                 /* priority, negative is high */
          char      p_cpu;                 /* (decayed) cpu usage solely for scheduling */
          char      p_stat;
          char      p_time;                /* seconds resident (for scheduling) */
          char      p_nice;                /* nice for cpu usage */
          char      p_slptime;  /* seconds since last block (sleep) */
          char      p_cursig;
          int       p_sig;                 /* signals pending to this process */
          int       p_sigmask;  /* current signal mask */
          int       p_sigignore;/* signals being ignored */
          int       p_sigcatch; /* signals being caught by user */
          int       p_flag;
          short     p_uid;                 /* user id, used to direct tty signals */
          short     p_suid;                /* saved (effective) user id from exec */
          short     p_sgid;                /* saved (effective) group id from exec */
          short     p_pgrp;                /* name of process group leader */
```

proc.h (cont.)

```
short       p_pid;                  /* unique process id */
short       p_ppid;                 /* process id of parent */
u_short     p_xstat;    /* Exit status for wait */
short       p_cpticks;  /* ticks of cpu time, used for p_pctcpu */
struct      ucred *p_cred;  /* Process credentials */
struct      rusage *p_ru;           /* mbuf holding exit information */
size_t      p_tsize;    /* size of text (clicks) */
size_t      p_dsize;    /* size of data space (clicks) */
size_t      p_ssize;    /* copy of stack size (clicks) */
size_t      p_rssize;   /* current resident set size in clicks */
size_t      p_maxrss;   /* copy of u.u_limit[MAXRSS] */
size_t      p_swrss;    /* resident set size before last swap */
caddr_t p_wchan;        /* event process is awaiting */
long        p_pctcpu;   /* (decayed) %cpu for this process */
struct      proc *p_pptr;           /* pointer to process structure of parent */
struct      proc *p_cptr;           /* pointer to youngest living child */
struct      proc *p_osptr;          /* pointer to older sibling processes */
struct      proc *p_ysptr;          /* pointer to younger siblings */
struct      proc *p_tptr;           /* pointer to process structure of tracer */
struct      itimerval p_realtimer;
short       p_idhash;   /* hashed based on p_pid for kill+exit+... */
short       p_swlocks;  /* number of swap vnode locks held */
```

protosw.h

```
/*      @(#)protosw.h 2.10 89/01/09 SMI; from UCB 7.1 6/4/86 */
/*
 * Copyright (c) 1982, 1986 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef _PROTOSW_
#define _PROTOSW_

/*
 * Protocol switch table.
 *
 * Each protocol has a handle initializing one of these structures,
 * which is used for protocol-protocol and system-protocol communication.
 *
 * A protocol is called through the pr_init entry before any other.
 * Thereafter it is called every 200ms through the pr_fasttimo entry and
 * every 500ms through the pr_slowtimo for timer based actions.
 * The system will call the pr_drain entry if it is low on space and
 * this should throw away any non-critical data.
 *
 * Protocols pass data between themselves as chains of mbufs using
 * the pr_input and pr_output hooks.  Pr_input passes data up (towards
 * UNIX) and pr_output passes it down (towards the imps); control
 * information passes up and down on pr_ctlinput and pr_ctloutput.
 * The protocol is responsible for the space occupied by any the
 * arguments to these entries and must dispose it.
 *
 * The userreq routine interfaces protocols to the system and is
 * described below.
 */
struct protosw {
        short   pr_type;            /* socket type used for */
        struct  domain *pr_domain;  /* domain protocol a member of */
        short   pr_protocol;                /* protocol number */
        short   pr_flags;           /* see below */
/* protocol-protocol hooks */
        int     (*pr_input)();              /* input to protocol (from below) */
        int     (*pr_output)();             /* output to protocol (from above) */
        int     (*pr_ctlinput)();   /* control input (from below) */
        int     (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
        int     (*pr_usrreq)();             /* user request: see list below */
/* utility hooks */
```

protosw.h (cont.)

```
        int     (*pr_init)();                   /* initialization hook */
        int     (*pr_fasttimo)();       /* fast timeout (200ms) */
        int     (*pr_slowtimo)();       /* slow timeout (500ms) */
        int     (*pr_drain)();                  /* flush any excess space possible */
};#define PR_SLOWHZ       2                       /* 2 slow timeouts per second */
#define PR_FASTHZ       5                       /* 5 fast timeouts per second */

/*
 * Values for pr_flags
 */
#define PR_ATOMIC           0x01                /* exchange atomic messages only */
#define PR_ADDR             0x02                /* addresses given with messages */
/* in the current implementation, PR_ADDR needs PR_ATOMIC to work */
#define PR_CONNREQUIRED 0x04                    /* connection required by protocol */
#define PR_WANTRCVD     0x08                    /* want PRU_RCVD calls */
#define PR_RIGHTS       0x10                    /* passes capabilities */
#define PR_OOB_ADDR     0x20            /* addresses given with OOB data */

/*
 * The arguments to usrreq are:
 *      (*protosw[].pr_usrreq)(up, req, m, nam, opt);
 * where up is a (struct socket *), req is one of these requests,
 * m is a optional mbuf chain containing a message,
 * nam is an optional mbuf chain containing an address,
 * and opt is a pointer to a socketopt structure or nil.
 * The protocol is responsible for disposal of the mbuf chain m,
 * the caller is responsible for any space held by nam and opt.
 * A non-zero return from usrreq gives an
 * UNIX error number which should be passed to higher level software.
 */
#define PRU_ATTACH              0       /* attach protocol to up */
#define PRU_DETACH              1       /* detach protocol from up */
#define PRU_BIND        2       /* bind socket to address */
#define PRU_LISTEN              3       /* listen for connection */
#define PRU_CONNECT             4       /* establish connection to peer */
#define PRU_ACCEPT              5       /* accept connection from peer */
#define PRU_DISCONNECT          6       /* disconnect from peer */
#define PRU_SHUTDOWN            7       /* won't send any more data */
#define PRU_RCVD                8       /* have taken data; more room now */
#define PRU_SEND                9       /* send this data */
#define PRU_ABORT               10      /* abort (fast DISCONNECT, DETATCH)
*/
```

protosw.h (cont.)

```
#define     PRU_CONTROL             11      /* control operations on protocol */
#define     PRU_SENSE               12      /* return status into m */
#define     PRU_RCVOOB              13      /* retrieve out of band data */
#define     PRU_SENDOOB             14      /* send out of band data */
#define     PRU_SOCKADDR            15      /* fetch socket's address */
#define     PRU_PEERADDR            16      /* fetch peer's address */
#define     PRU_CONNECT2            17      /* connect two sockets */
/* begin for protocols internal use */
#define     PRU_FASTTIMO            18      /* 200ms timeout */
#define     PRU_SLOWTIMO            19      /* 500ms timeout */
#define     PRU_PROTORCV            20      /* receive from below */
#define     PRU_PROTOSEND           21      /* send to below */
#define     PRU_NREQ                21

#ifdef PRUREQUESTS
char *prurequests[] = {
        "ATTACH",       "DETACH",       "BIND",         "LISTEN",
        "CONNECT",      "ACCEPT",       "DISCONNECT",   "SHUTDOWN",
        "RCVD",         "SEND",         "ABORT", "CONTROL",
        "SENSE", "RCVOOB",      "SENDOOB",      "SOCKADDR",
        "PEERADDR",     "CONNECT2",     "FASTTIMO",     "SLOWTIMO",
        "PROTORCV",     "PROTOSEND",
};
#endif
```

protosw.h (cont.)

```
/*
 * The arguments to the ctlinput routine are
 *          (*protosw[].pr_ctlinput)(cmd, arg);
 * where cmd is one of the commands below, and arg is
 * an optional argument (caddr_t).
 *
 * N.B. The IMP code, in particular, pressumes the values
 *      of some of the commands; change with extreme care.
 * TODO:
 *          spread out codes so new ICMP codes can be
 *          accomodated more easily
 */
#define     PRC_IFDOWN                  0           /* interface transition */
#define     PRC_ROUTEDEAD               1           /* select new route if possible */
#define     PRC_QUENCH                  4           /* some said to slow down */
#define     PRC_MSGSIZE                 5           /* message size forced drop */
#define     PRC_HOSTDEAD                6           /* normally from IMP */
#define     PRC_HOSTUNREACH             7           /* ditto */
#define     PRC_UNREACH_NET             8           /* no route to network */
#define     PRC_UNREACH_HOST            9           /* no route to host */
#define     PRC_UNREACH_PROTOCOL        10          /* dst says bad protocol */
#define     PRC_UNREACH_PORT            11          /* bad port # */
#define     PRC_UNREACH_NEEDFRAG        12          /* IP_DF caused drop */
#define     PRC_UNREACH_SRCFAIL         13          /* source route failed */
#define     PRC_REDIRECT_NET 14         /* net routing redirect */
#define     PRC_REDIRECT_HOST           15          /* host routing redirect */
#define     PRC_REDIRECT_TOSNET         16          /* redirect for type of service & net */
#define     PRC_REDIRECT_TOSHOST        17          /* redirect for tos & host */
#define     PRC_TIMXCEED_INTRANS        18          /* packet lifetime expired in transit */
#define     PRC_TIMXCEED_REASS          19          /* lifetime expired on reass q */
#define     PRC_PARAMPROB               20          /* header incorrect */
#define     PRC_GWDOWN                  21          /* gateway down */

#define     PRC_NCMDS                   22
```

protosw.h (cont.)

```
#ifdef PRCREQUESTS
char        *prcrequests[] = {
            "IFDOWN", "ROUTEDEAD", "#2", "#3",
            "QUENCH", "MSGSIZE", "HOSTDEAD", "HOSTUNREACH",
            "NET-UNREACH", "HOST-UNREACH", "PROTO-UNREACH", "PORT-UNREACH",
            "FRAG-UNREACH",  "SRCFAIL-UNREACH",  "NET-REDIRECT",  "HOST-REDI-
RECT",
            "TOSNET-REDIRECT", "TOSHOST-REDIRECT", "TX-INTRANS", "TX-REASS",
            "PARAMPROB"
};
#endif

/*
 * The arguments to ctloutput are:
 *          (*protosw[].pr_ctloutput)(req, so, level, optname, optval);
 * req is one of the actions listed below, so is a (struct socket *),
 * level is an indication of which protocol layer the option is intended.
 * optname is a protocol dependent socket option request,
 * optval is a pointer to a mbuf-chain pointer, for value-return results.
 * The protocol is responsible for disposal of the mbuf chain *optval
 * if supplied,
 * the caller is responsible for any space held by *optval, when returned.
 * A non-zero return from usrreq gives an
 * UNIX error number which should be passed to higher level software.
 */
#define     PRCO_GETOPT         0
#define     PRCO_SETOPT         1

#define     PRCO_NCMDS          2

#ifdef PRCOREQUESTS
char        *prcorequests[] = {
            "GETOPT", "SETOPT",
};
#endif

#ifdef KERNEL
extern      struct protosw *pffindproto(), *pffindtype();
#endif

#endif      _PROTOSW_
```

seg_map.h

```
/*      @(#)seg_map.h       1.12 88/02/08 SMI     */
/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#ifndef _vm_seg_map_h
#define _vm_seg_map_h

struct segmap_crargs {
        u_int       prot;
};

/*
 * Each smap struct represents a MAXBSIZE sized mapping to the
 * <sm_vp, sm_off> given in the structure.  The location of the
 * the structure in the array gives the virtual address of the
 * mapping.
 */
struct  smap {
        struct      vnode *sm_vp;                   /* vnode pointer (if mapped) */
        u_int       sm_off;                         /* file offset for mapping */
        /*
         * These next 4 entries can be coded as
         * u_shorts if we are tight on memory.
         */
        u_int       sm_refcnt;              /* reference count for uses */
        struct      smap *sm_hash;                  /* hash pointer */
        struct      smap *sm_next;                  /* next pointer */
        struct      smap *sm_prev;                  /* previous pointer */
};

/*
 * (Semi) private data maintained by the segmap driver per SEGMENT mapping
 */
struct  segmap_data {
        struct      smap *smd_sm;                   /* array of smap structures */
        struct      smap *smd_free;                 /* free list head pointer */
        u_char      smd_prot;               /* protections for all smap's */
        u_char      smd_want;               /* smap want flag */
        u_int       smd_hashsz;                     /* power-of-two hash table size */
        struct      smap **smd_hash;        /* pointer to hash table */
};
```

seg_map.h (cont.)

```
/*
 * These are flags used on release.  Some of these might get handled
 * by segment operations needed for msync (when we figure them out).
 * SM_ASYNC modifies SM_WRITE.  SM_DONTNEED modifies SM_FREE.  SM_FREE
 * and SM_INVAL are mutually exclusive.
 */
#define     SM_WRITE        0x01            /* write back the pages upon release */
#define     SM_ASYNC        0x02            /* do the write asynchronously */
#define     SM_FREE         0x04            /* put pages back on free list */
#define     SM_INVAL        0x08            /* invalidate page (no caching) */
#define     SM_DONTNEED     0x10            /* less likely to be needed soon */

#define MAXBSHIFT  13                   /* log2(DEVBSIZE) */
#define MAXBOFFSET          (MAXBSIZE - 1)
#define MAXBMASK  (-MAXBOFFSET)

/*
 * SMAP_HASHAVELEN is the average length desired for this chain, from
 * which the size of the smd_hash table is derived at segment create time.
 * SMAP_HASHVPSHIFT is defined so that 1 << SMAP_HASHVPSHIFT is the
 * approximate size of a vnode struct.
 */
#define     SMAP_HASHAVELEN                 4
#define     SMAP_HASHVPSHIFT        6

#define     SMAP_HASHFUNC(smd, vp, off) \
            ((((off) >> MAXBSHIFT) + ((int)(vp) >> SMAP_HASHVPSHIFT)) & \
                    ((smd)->smd_hashsz - 1))

#ifdef KERNEL
int         segmap_create(/* seg, argsp */);

/*
 * Special seg_map segment operations
 */
void        segmap_pagecreate(/* seg, addr, len, softlock */);
addr_t      segmap_getmap(/* seg, vp, off */);
int         segmap_release(/* seg, addr, flags */);

extern struct seg *segkmap;         /* the kernel generic mapping segment */
extern struct seg_ops segmap_ops;
#endif KERNEL
#endif !_vm_seg_map_h
```

seg_vn.h

```
/*	@(#)seg_vn.h	1.16 88/02/08 SMI	*/
/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#ifndef _vm_seg_vn_h
#define _vm_seg_vn_h
#include <vm/mp.h>

/*
 * Structure who's pointer is passed to the segvn_create routine
 */
struct segvn_crargs {
	struct	vnode *vp; /* vnode mapped from */
	u_int	offset;			/* starting offset of vnode for mapping */
	struct	ucred *cred;		/* creditials */
	u_char	type;			/* type of sharing done */
	u_char	prot;			/* protections */
	u_char	maxprot;	/* maximum protections */
	struct	anon_map *amp;		/* anon mapping to map to */
};

/*
 * The anon_map structure is used by the seg_vn driver to manage
 * unnamed (anonymous) memory.  When anonymous memory is shared,
 * then the different segvn_data structures will point to the
 * same anon_map structure.  Also, if a segment is unmapped
 * in the middle where an anon_map structure exists, the
 * newly created segment will also share the anon_map structure,
 * although the two segments will use different ranges of the
 * anon array.  When mappings are private (or shared with
 * a reference count of 1), an unmap operation will free up
 * a range of anon slots in the array given by the anon_map
 * structure.  Because of fragmentation due to this unmapping,
 * we have to store the size of anon array in the anon_map
 * structure so that we can free everything when the referernce
 * count goes to zero.
 */
struct anon_map {
	u_int	refcnt;			/* reference count on this structure */
	u_int	size;			/* size in bytes mapped by the anon array */
	struct	anon **anon;		/* pointer to an array of anon * pointers */
};
```

seg_vn.h (cont.)

```
/*
 * (Semi) private data maintained by the seg_vn driver per segment mapping
 */
struct      segvn_data {
            mon_t       lock;
            u_char      pageprot;   /* true if per page protections present */
            u_char      prot;               /* current segment prot if pageprot == 0 */
            u_char      maxprot;    /* maximum segment protections */
            u_char      type;               /* type of sharing done */
            struct      vnode *vp; /* vnode that segment mapping is to */
            u_int       offset;             /* starting offset of vnode for mapping */
            u_int       anon_index;         /* starting index into anon_map anon array */
            struct      anon_map *amp;      /* pointer to anon share structure, if needed */
            struct      vpage *vpage;       /* per-page information, if needed */
            struct      ucred *cred;        /* mapping creditials */
            u_int       swresv;             /* swap space reserved for this segment */
};

#ifdef KERNEL
int         segvn_create(/* seg, argsp */);

extern      struct seg_ops segvn_ops;

/*
 * Provided as short hand for creating user zfod segments
 */
extern      caddr_t zfod_argsp;
extern      caddr_t kzfod_argsp;
#endif KERNEL
#endif !_vm_seg_vn_h
```

socket.h

```
/*      @(#)socket.h 2.13 88/02/08 SMI; from UCB 7.1 6/4/86   */
/*
 * Copyright (c) 1982, 1985, 1986 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef  _SOCKET_
#define  _SOCKET_

/*
 * Definitions related to sockets: types, address families, options.
 */

/*
 * Types
 */
#define  SOCK_STREAM     1               /* stream socket */
#define  SOCK_DGRAM      2               /* datagram socket */
#define  SOCK_RAW        3               /* raw-protocol interface */
#define  SOCK_RDM        4               /* reliably-delivered message */
#define  SOCK_SEQPACKET  5               /* sequenced packet stream */

/*
 * Option flags per-socket.
 */
#define  SO_DEBUG        0x0001          /* turn on debugging info recording */
#define  SO_ACCEPTCONN   0x0002          /* socket has had listen() */
#define  SO_REUSEADDR    0x0004          /* allow local address reuse */
#define  SO_KEEPALIVE    0x0008          /* keep connections alive */
#define  SO_DONTROUTE    0x0010          /* just use interface addresses */
#define  SO_BROADCAST    0x0020          /* permit sending of broadcast msgs */
#define  SO_USELOOPBACK  0x0040          /* bypass hardware when possible */
#define  SO_LINGER       0x0080          /* linger on close if data present */
#define  SO_OOBINLINE    0x0100          /* leave received OOB data in line */
/*
 * N.B.: The following definition is present only for compatibility
 * with release 3.0.  It will disappear in later releases.
 */
#define  SO_DONTLINGER   (-SO_LINGER)    /* -SO_LINGER */
```

socket.h (cont.)

```
/*
 * Additional options, not kept in so_options.
 */
#define     SO_SNDBUF       0x1001          /* send buffer size */
#define     SO_RCVBUF       0x1002          /* receive buffer size */
#define     SO_SNDLOWAT     0x1003          /* send low-water mark */
#define     SO_RCVLOWAT     0x1004          /* receive low-water mark */
#define     SO_SNDTIMEO     0x1005          /* send timeout */
#define     SO_RCVTIMEO     0x1006          /* receive timeout */
#define     SO_ERROR0x1007              /* get error status and clear */
#define     SO_TYPE         0x1008          /* get socket type */

/*
 * Structure used for manipulating linger option.
 */
struct      linger {
            int         l_onoff;            /* option on/off */
            int         l_linger;           /* linger time */
};

/*
 * Level number for (get/set)sockopt() to apply to socket itself.
 */
#define     SOL_SOCKET      0xffff              /* options for socket level */
```

socket.h (cont.)

```
/*
 * Address families.
 */
#define     AF_UNSPEC       0               /* unspecified */
#define     AF_UNIX         1               /* local to host (pipes, portals) */
#define     AF_INET         2               /* internetwork: UDP, TCP, etc. */
#define     AF_IMPLINK      3               /* arpanet imp addresses */
#define     AF_PUP          4               /* pup protocols: e.g. BSP */
#define     AF_CHAOS        5               /* mit CHAOS protocols */
#define     AF_NS           6               /* XEROX NS protocols */
#define     AF_NBS          7               /* nbs protocols */
#define     AF_ECMA         8               /* european computer manufacturers */
#define     AF_DATAKIT      9               /* datakit protocols */
#define     AF_CCITT 10                 /* CCITT protocols, X.25 etc */
#define     AF_SNA          11              /* IBM SNA */
#define     AF_DECnet       12              /* DECnet */
#define     AF_DLI          13              /* Direct data link interface */
#define     AF_LAT          14              /* LAT */
#define     AF_HYLINK       15              /* NSC Hyperchannel */
#define     AF_APPLETALK    16              /* Apple Talk */

#define     AF_NIT          17              /* Network Interface Tap */
#define     AF_802          18              /* IEEE 802.2, also ISO 8802 */
#define     AF_OSI          19              /* umbrella for all families used
                                             * by OSI (e.g. protosw lookup) */
#define     AF_X25          20              /* CCITT X.25 in particular */
#define     AF_OSINET       21              /* AFI = 47, IDI = 4 */
#define     AF_GOSIP 22                 /* U.S. Government OSI */

#define     AF_MAX          21

/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
        u_short     sa_family;          /* address family */
        char        sa_data[14];            /* up to 14 bytes of direct address */
};
```

socket.h (cont.)

```
/*
 * Structure used by kernel to pass protocol
 * information in raw sockets.
 */
struct sockproto {
        u_short     sp_family;              /* address family */
        u_short     sp_protocol;                    /* protocol */
};

/*
 * Protocol families, same as address families for now.
 */
#define     PF_UNSPEC           AF_UNSPEC
#define     PF_UNIX             AF_UNIX
#define     PF_INET             AF_INET
#define     PF_IMPLINK          AF_IMPLINK
#define     PF_PUP              AF_PUP
#define     PF_CHAOS            AF_CHAOS
#define     PF_NS               AF_NS
#define     PF_NBS              AF_NBS
#define     PF_ECMA             AF_ECMA
#define     PF_DATAKIT          AF_DATAKIT
#define     PF_CCITT AF_CCITT
#define     PF_SNA              AF_SNA
#define     PF_DECnetAF_DECnet
#define     PF_DLI              AF_DLI
#define     PF_LAT              AF_LAT
#define     PF_HYLINK           AF_HYLINK
#define     PF_APPLETALK        AF_APPLETALK
#define     PF_NIT              AF_NIT
#define     PF_802              AF_802
#define     PF_OSI              AF_OSI
#define     PF_X25              AF_X25
#define     PF_OSINET           AF_OSINET
#define     PF_GOSIP AF_GOSIP

#define     PF_MAX              AF_MAX

/*
 * Maximum queue length specifiable by listen.
 */
#define     SOMAXCONN           5
```

socket.h (cont.)

```
/*
 * Message header for recvmsg and sendmsg calls.
 */
struct msghdr {
        caddr_t     msg_name;           /* optional address */
        int         msg_namelen;                /* size of address */
        struct      iovec *msg_iov;             /* scatter/gather array */
        int         msg_iovlen;         /* # elements in msg_iov */
        caddr_t     msg_accrights;              /* access rights sent/received */
        int         msg_accrightslen;
};

#define     MSG_OOB             0x1             /* process out-of-band data */
#define     MSG_PEEK            0x2             /* peek at incoming message */
#define     MSG_DONTROUTE       0x4             /* send without using routing tables */

#define     MSG_MAXIOVLEN       16

#endif      _SOCKET_
```

socketvar.h

```
/*      @(#)socketvar.h 2.15 88/02/08 SMI; from UCB 7.3 12/30/87 */
/*
 * Copyright (c) 1982, 1986 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef _SOCKETVAR_
#define _SOCKETVAR_

/*
 * Kernel structure per socket.
 * Contains send and receive buffer queues,
 * handle on protocol and pointer to protocol
 * private data and error information.
 */
struct socket {
        short   so_type;        /* generic type, see socket.h */
        short   so_options;     /* from socket call, see socket.h */
        short   so_linger;      /* time to linger while closing */
        short   so_state;       /* internal state flags SS_*, below */
        caddr_t so_pcb;                 /* protocol control block */
        struct  protosw *so_proto;/* protocol handle */
/*
 * Variables for connection queueing.
 * Socket where accepts occur is so_head in all subsidiary sockets.
 * If so_head is 0, socket is not related to an accept.
 * For head socket so_q0 queues partially completed connections,
 * while so_q is a queue of connections ready to be accepted.
 * If a connection is aborted and it has so_head set, then
 * it has to be pulled out of either so_q0 or so_q.
 * We allow connections to queue up based on current queue lengths
 * and limit on number of queued connections for this socket.
 */
        struct  socket *so_head;/* back pointer to accept socket */
        struct  socket *so_q0;          /* queue of partial connections */
        struct  socket *so_q;           /* queue of incoming connections */
        short   so_q0len;       /* partials on so_q0 */
        short   so_qlen;        /* number of connections on so_q */
        short   so_qlimit;      /* max number queued connections */
        short   so_timeo;       /* connection timeout */
        u_short so_error;       /* error affecting connection */
        short   so_pgrp;        /* pgrp for signals */
        u_long  so_oobmark;             /* chars to oob mark */
/*
```

socketvar.h (cont.)

```
 * Variables for socket buffering.
 */
        struct  sockbuf {
                u_long  sb_cc;                  /* actual chars in buffer */
                u_long  sb_hiwat;   /* max actual char count */
                u_long  sb_mbcnt;   /* chars of mbufs used */
                u_long  sb_mbmax;   /* max chars of mbufs to use */
                u_long  sb_lowat;   /* low water mark (not used yet) */
                struct  mbuf *sb_mb;            /* the mbuf chain */
                struct  proc *sb_sel;           /* process selecting read/write */
                short   sb_timeo;   /* timeout (not used yet) */
                short   sb_flags;   /* flags, see below */
        } so_rcv, so_snd;
#define SB_MAX          (64*1024)  /* max chars in sockbuf */
#define SB_LOCK         0x01                    /* lock on data queue (so_rcv only) */
#define SB_WANT         0x02                    /* someone is waiting to lock */
#define SB_WAIT         0x04                    /* someone is waiting for data/space */
#define SB_SEL          0x08                    /* buffer is selected */
#define SB_COLL         0x10                    /* collision selecting */
/*
 * Hooks for alternative wakeup strategies.
 * These are used by kernel subsystems wishing to access the socket
 * abstraction.  If so_wupfunc is nonnull, it is called in place of
 * wakeup any time that wakeup would otherwise be called with an
 * argument whose value is an address lying within a socket structure.
 */
        struct wupalt           *so_wupalt;
};

struct wupalt {
        int     (*wup_func)();                  /* function to call instead of wakeup */
        caddr_t wup_arg;                /* argument for so_wupfunc */
        /*
         * Other state information here, for example, for a stream
         * connected to a socket.
         */
};
```

socketvar.h (cont.)

```
/*
 * Socket state bits.
 */
#define     SS_NOFDREF                  0x001     /* no file table ref any more */
#define     SS_ISCONNECTED              0x002     /* socket connected to a peer */
#define     SS_ISCONNECTING             0x004     /* in process of connecting to peer */
#define     SS_ISDISCONNECTING          0x008     /* in process of disconnecting */
#define     SS_CANTSENDMORE             0x010     /* can't send more data to peer */
#define     SS_CANTRCVMORE              0x020     /* can't receive more data from peer */
#define     SS_RCVATMARK                0x040     /* at mark on input */

#define     SS_PRIV                     0x080     /* privileged for broadcast, raw... */
#define     SS_NBIO                     0x100     /* non-blocking ops */
#define     SS_ASYNC            0x200     /* async i/o notify */

/*
 * Macros for sockets and socket buffering.
 */

/* how much space is there in a socket buffer (so->so_snd or so->so_rcv) */
#define     sbspace(sb) \
    (MIN((int)((sb)->sb_hiwat - (sb)->sb_cc),\
             (int)((sb)->sb_mbmax - (sb)->sb_mbcnt)))

/* do we have to send all at once on a socket? */
#define     sosendallatonce(so) \
    ((so)->so_proto->pr_flags & PR_ATOMIC)

/* can we read something from so? */
#define     soreadable(so) \
    ((so)->so_rcv.sb_cc || ((so)->so_state & SS_CANTRCVMORE) || \
             (so)->so_qlen || (so)->so_error)

/* can we write something to so? */
#define     sowriteable(so) \
    (sbspace(&(so)->so_snd) > 0 && \
             (((so)->so_state&SS_ISCONNECTED) || \
               ((so)->so_proto->pr_flags&PR_CONNREQUIRED)==0) || \
     ((so)->so_state & SS_CANTSENDMORE) || \
     (so)->so_error)

/* adjust counters in sb reflecting allocation of m */
#define     sballoc(sb, m) { \
            (sb)->sb_cc += (m)->m_len; \
            (sb)->sb_mbcnt += MSIZE; \
            if ((m)->m_off > MMAXOFF) \
                        (sb)->sb_mbcnt += MCLBYTES; \
}
```

socketvar.h (cont.)

```
/* adjust counters in sb reflecting freeing of m */
#define     sbfree(sb, m) { \
            (sb)->sb_cc -= (m)->m_len; \
            (sb)->sb_mbcnt -= MSIZE; \
            if ((m)->m_off > MMAXOFF) \
                        (sb)->sb_mbcnt -= MCLBYTES; \
}

/* set lock on sockbuf sb */
#define     sblock(so, sb) { \
            while ((sb)->sb_flags & SB_LOCK) { \
                        (sb)->sb_flags |= SB_WANT; \
                        (void) sleep((caddr_t)&(sb)->sb_flags, PZERO+1); \
            } \
            (sb)->sb_flags |= SB_LOCK; \
}

/* release lock on sockbuf sb */
#define     sbunlock(so, sb) { \
            (sb)->sb_flags &= ~SB_LOCK; \
            if ((sb)->sb_flags & SB_WANT) { \
                        (sb)->sb_flags &= ~SB_WANT; \
                        if ((so)->so_wupalt) \
                                    (*(so)->so_wupalt->wup_func)(so, \
                                    (caddr_t)&(sb)->sb_flags, \
                                                (so)->so_wupalt->wup_arg);\
                        else \
                                    wakeup((caddr_t)&(sb)->sb_flags); \
            } \
}

#define     sorwakeup(so)           sowakeup((so), &(so)->so_rcv)
#define     sowwakeup(so)           sowakeup((so), &(so)->so_snd)

#ifdef KERNEL
struct      socket *sonewconn();
#endif

#endif      _SOCKETVAR_
```

ucred.h

```
/*          @(#)ucred.h          1.4 88/02/08 SMI      */

#ifndef _sys_ucred_h
#define _sys_ucred_h

#include <sys/label.h>
#include <sys/audit.h>

/*
 * User credential structure
 */
struct ucred {
        u_short         cr_ref;                 /* reference count */
        short           cr_uid;                 /* effective user id */
        short           cr_gid;                 /* effective group id */
        short           cr_auid;        /* user id, for auditing */
        short           cr_ruid;        /* real user id */
        short           cr_rgid;        /* real group id */
        int             cr_groups[NGROUPS]; /* groups, 0 terminated */
        blabel_t    cr_label;               /* Mandatory Access Control */
        audit_state_t   cr_audit;       /* audit_state */
};

#ifdef KERNEL
#define     crhold(cr)   (cr)->cr_ref++
void crfree();
struct ucred *crget();
struct ucred *crcopy();
struct ucred *crdup();
struct ucred *crgetcred();
#endif KERNEL

#endif !_sys_ucred_h
```

User.h

```
/*      @(#)user.h 2.44 88/02/08 SMI; from UCB 7.1 6/4/86       */

/*
 * Copyright (c) 1982, 1986 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef _user_h
#define _user_h

#ifdef sun386
#include <machine/tss.h>
#include <sys/filehdr.h>
#endif
#include <machine/pcb.h>
#include <machine/reg.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/exec.h>
#ifdef sun386
#include <sys/label.h>
#endif
#include <sys/proc.h>
#include <sys/ucred.h>

/*
 * Per process structure containing data that
 * isn't needed in core when the process is swapped out.
 * A redzone is set below the user structure as
 * a safe guard against kernel stack overflow.
 */

#define     SHSIZE              sizeof (struct exec)
#define     MAXCOMLEN           16                  /* <= MAXNAMLEN, >= sizeof (ac_comm)
*/

struct      user {
            char        u_stack[KERNSTACK];         /* the kernel stack */
            struct      pcb u_pcb;
            struct      proc *u_procp;              /* pointer to proc structure */
            int         *u_ar0;                     /* address of users saved R0 */
            char        u_comm[MAXCOMLEN + 1];
```

User.h (cont.)

```
/* syscall parameters, results and catches */
        int     u_arg[8];               /* arguments to current system call */
        int     *u_ap;                          /* pointer to arglist */
        label_t u_qsave;                /* for non-local gotos on interrupts */
        union {                                 /* syscall return values */
                struct  {
                        int     R_val1;
                        int     R_val2;
                } u_rv;
#define r_val1  u_rv.R_val1
#define r_val2  u_rv.R_val2
                off_t   r_off;
                time_t  r_time;
        } u_r;
#define u_rval1 u_r.r_val1
#define u_rval2 u_r.r_val2
        char    u_error;                /* return error code */
        char    u_eosys;                /* special action on end of syscall */

/* 1.1 - processes and protection */
#ifdef KERNEL
#define u_cred          u_procp->p_cred
#define u_uid           u_cred->cr_uid
#define u_gid           u_cred->cr_gid
#define u_groups        u_cred->cr_groups
#define u_ruid          u_cred->cr_ruid
#define u_rgid          u_cred->cr_rgid
#define u_auid          u_cred->cr_auid
#endif /*KERNEL*/

/* 1.2 - memory management */
#ifdef KERNEL
#define u_tsize         u_procp->p_tsize
#define u_dsize         u_procp->p_dsize
#define u_ssize         u_procp->p_ssize
#endif /*KERNEL*/
        label_t u_ssave;                /* label for swapping/forking */
```

User.h (cont.)

```
/* 1.3 - signal management */
        void        (*u_signal[NSIG])();  /* disposition of signals */
        int         u_sigmask[NSIG];      /* signals to be blocked */
        int         u_sigonstack;                 /* signals to take on sigstack */
        int         u_sigintr;            /* signals that interrupt syscalls */
        int         u_sigreset;           /* signals that reset the handler when taken */
        int         u_oldmask;            /* saved mask from before sigpause */
        int         u_code;                       /* "code" to trap */
        char        *u_addr;              /* "addr" to trap */
        struct      sigstack u_sigstack;  /* sp & on stack state variable */
#define u_onstack   u_sigstack.ss_onstack
#define u_sigsp             u_sigstack.ss_sp

/* 1.4 - descriptor management */
        /*
         * The next two fields are placeholders for pointers to the
         * u_ofile and u_pofile arrays.  When NOFILE goes over 64, these
         * arrays will be allocated dynamically.  For the moment, these
         * fields are just here so that kmem readers won't have to be
         * rebuilt when the remaining support for a larger NOFILE value
         * does go in.
         */
        struct      file **u_ofileX;
        char        *u_pofileX;
        struct      file *u_ofile[NOFILE]; /* file structures for open files */
        char        u_pofile[NOFILE];     /* per-process flags of open files */
        int         u_lastfile;           /* high-water mark of u_ofile */
#define UF_EXCLOSE          0x1                   /* auto-close on exec */
#define UF_FDLOCK  0x2                    /* file desc locked (SysV style) */
        struct      ucwd  *u_cwd;                 /* ascii current directory */
        struct      vnode *u_cdir;                /* current directory */
        struct      vnode *u_rdir;                /* root directory of current process */
        short       *u_ttyp;              /* pointer to pgrp in streamtab */
        dev_t       u_ttyd;                       /* controlling tty dev */
        struct      vnode *u_ttyvp;               /* vnode of controlling tty */
        short       u_cmask;              /* mask for file creation */

/* 1.5 - timing and statistics */
        struct      rusage u_ru;                  /* stats for this proc */
        struct      rusage u_cru;                 /* sum of stats for reaped children */
        struct      itimerval u_timer[3];
        int         u_XXX[3];
        struct timeval          u_start;
        short       u_acflag;
```

User.h (cont.)

```
        struct uprof {                                  /* profile arguments */
                short       *pr_base;   /* buffer base */
                u_int        pr_size;   /* buffer size */
                u_int        pr_off;    /* pc offset */
                u_int        pr_scale;  /* pc scaling */
        } u_prof;

/* 1.6 - resource controls */
        struct      rlimit u_rlimit[RLIM_NLIMITS];
/* BEGIN TRASH */
        union {
                struct exec Ux_A;       /* header of executable file */
                char ux_shell[SHSIZE];          /* #! and name of interpreter */
#ifdef sun386
                struct exec UX_C;       /* COFF file header */
#endif
        } u_exdata;
#ifdef sun386
        /*
         * The virtual address of the text and data is needed to exec
         * coff files.  Unfortunately, they won't fit into Ux_A above.
         */
        u_int       u_textvaddr;                        /* virtual address of text segment */
        u_int       u_datavaddr;                        /* virtual address of data segment */
        u_int       u_bssvaddr;             /* virtual address of bss segment */
#endif
#define     ux_mach             Ux_A.a_machtype
#define     ux_mag              Ux_A.a_magic
#define     ux_tsize    Ux_A.a_text
#define     ux_dsize    Ux_A.a_data
#define     ux_bsize    Ux_A.a_bss
#ifndef sun386
#define     ux_ssize    Ux_A.a_syms
#endif
#define     ux_entloc   Ux_A.a_entry
#ifdef sun386
#define ux_toffset      Ux_A.a_trsize
#define ux_doffset      Ux_A.a_drsize
#else
#define     ux_unused   Ux_A.a_trsize
#define     ux_relflg   Ux_A.a_drsize
#endif
#ifdef sun
        int         u_lofault;              /* catch faults in locore.s */
#endif sun
```

User.h (cont.)

```
* Current directory structure for auditing, pointed to by u_cwd.
 */
struct ucwd {
        u_short     cw_ref;                     /* reference count */
        u_short     cw_len;                     /* size of this whole thing */
        char        *cw_dir;            /* directory */
        char        *cw_root;           /* root directory */
};

#ifdef KERNEL
void cwincr();
void cwfree();
#endif KERNEL

/* u_eosys values */
#define     NORMALRETURN    0       /* normal return; adjusts PC, registers */
#define     JUSTRETURN      1       /* just return, leave registers alone */
#define     RESTARTSYS      2       /* back up PC, restart system call */

/* u_error codes */
#ifdef KERNEL
#include <sys/errno.h>
#else KERNEL
#include <errno.h>
#endif KERNEL

#ifdef KERNEL
#ifdef sun
#define     u           (*(struct user *)UADDR)
#else sun
extern      struct user u;
#endif sun

extern      struct user forkutl;

#endif KERNEL

#ifdef sun
#define U_FPA_USED      0x0100
#define U_FPA_INFORK        0x80
#define U_FPA_FDBITS    0x7F
#endif

#endif !_user_h
```

vfs.h

```
/*      @(#)vfs.h 2.23 88/02/08 SMI     */

/*
 * File system identifier. Should be unique (at least per machine).
 */
typedef struct {
        long val[2];                    /* file system id type */
} fsid_t;

/*
 * File identifier. Should be unique per filesystem on a single machine.
 */
#define MAXFIDSZ        16
#define freefid(fidp) \
   kmem_free((caddr_t)(fidp), sizeof (struct fid) - MAXFIDSZ + (fidp)->fid_len)

struct fid {
        u_short         fid_len;            /* length of data in bytes */
        char            fid_data[MAXFIDSZ]; /* data (variable length) */
};

/*
 * Structure per mounted file system.
 * Each mounted file system has an array of
 * operations and an instance record.
 * The file systems are put on a singly linked list.
 * If vfs_stats is non-NULL statistics are gathered, see vfs_stat.h
 */
struct vfs {
        struct vfs      *vfs_next;          /* next vfs in vfs list */
        struct vfsops   *vfs_op;            /* operations on vfs */
        struct vnode    *vfs_vnodecovered;  /* vnode we mounted on */
        int             vfs_flag;           /* flags */
        int             vfs_bsize;          /* native block size */
        fsid_t          vfs_fsid;           /* file system id */
        caddr_t     vfs_stats;              /* filesystem statistics */
        caddr_t         vfs_data;           /* private data */
};
```

vfs.h (cont.)

```
/*
 * vfs flags.
 * VFS_MLOCK lock the vfs so that name lookup cannot proceed past the vfs.
 * This keeps the subtree stable during mounts and unmounts.
 */
#define VFS_RDONLY 0x01                 /* read only vfs */
#define VFS_MLOCK  0x02                 /* lock vfs so that subtree is stable */
#define VFS_MWAIT  0x04                 /* someone is waiting for lock */
#define VFS_NOSUID 0x08                 /* turn off set-uid on exec */
#define VFS_GRPID  0x10                 /* Old BSD group-id on create */
#define VFS_NOSUB  0x20                 /* No mounts allowed beneath this fs */
#define VFS_REMOUNT         0x40                /* modify mount otions only */
#define VFS_MULTI  0x80                 /* Do multi-component lookup on files */

/*
 * Operations supported on virtual file system.
 */
struct vfsops {
        int     (*vfs_mount)();                 /* mount file system */
        int     (*vfs_unmount)();       /* unmount file system */
        int     (*vfs_root)();                  /* get root vnode */
        int     (*vfs_statfs)();        /* get fs statistics */
        int     (*vfs_sync)();                  /* flush fs buffers */
        int     (*vfs_vget)();                  /* get vnode from fid */
        int     (*vfs_mountroot)();     /* mount the root filesystem */
        int     (*vfs_swapvp)();        /* return vnode for swap */
};

#define VFS_MOUNT(VFSP, PATH, DATA) \
                                        (*(VFSP)->vfs_op->vfs_mount)(VFSP,      PATH,
DATA)
#define VFS_UNMOUNT(VFSP)    (*(VFSP)->vfs_op->vfs_unmount)(VFSP)
#define VFS_ROOT(VFSP, VPP)    (*(VFSP)->vfs_op->vfs_root)(VFSP,VPP)
#define VFS_STATFS(VFSP, SBP)           (*(VFSP)->vfs_op->vfs_statfs)(VFSP,SBP)
#define VFS_SYNC(VFSP)                  (*(VFSP)->vfs_op->vfs_sync)(VFSP)
#define VFS_VGET(VFSP, VPP, FIDP) (*(VFSP)->vfs_op->vfs_vget)(VFSP, VPP, FIDP)
#define VFS_MOUNTROOT(VFSP, VPP, NM) \
                                        (*(VFSP)->vfs_op->vfs_mountroot)(VFSP, VPP, NM)
#define VFS_SWAPVP(VFSP, VPP, NM) (*(VFSP)->vfs_op->vfs_swapvp)(VFSP, VPP, NM)
```

vfs.h (cont.)

```
/*
 * file system statistics
 */
struct statfs {
        long f_type;                    /* type of info, zero for now */
        long f_bsize;                           /* fundamental file system block size */
        long f_blocks;                          /* total blocks in file system */
        long f_bfree;                           /* free block in fs */
        long f_bavail;                          /* free blocks avail to non-superuser */
        long f_files;                   /* total file nodes in file system */
        long f_ffree;                           /* free file nodes in fs */
        fsid_t f_fsid;                          /* file system id */
        long f_spare[7];                /* spare for later */
};

#ifdef KERNEL
/*
 * Filesystem type switch table
 */
struct vfssw {
        char                    *vsw_name;              /* type name string */
        struct vfsops           *vsw_ops;  /* filesystem operations vector */
};

/*
 * public operations
 */
extern void vfs_mountroot();    /* mount the root */
extern int  vfs_add();          /* add a new vfs to mounted vfs list */
extern void vfs_remove();               /* remove a vfs from mounted vfs list */
extern int  vfs_lock();         /* lock a vfs */
extern void vfs_unlock();               /* unlock a vfs */
extern struct vfs *getvfs();            /* return vfs given fsid */
extern struct vfssw *getfstype();  /* find default filesystem type */
extern int vfs_getmajor();              /* get major device # for an fs type */
extern void vfs_putmajor();             /* free major device # for an fs type */
extern int vfs_getnum();                /* get device # for an fs type */
extern void vfs_putnum();               /* release device # for an fs type */
```

vfs.h (cont.)

```
#define VFS_INIT(VFSP, OP, DATA)        { \
        (VFSP)->vfs_next = (struct vfs *)0; \
        (VFSP)->vfs_op = (OP); \
        (VFSP)->vfs_flag = 0; \
        (VFSP)->vfs_stats = NULL; \
        (VFSP)->vfs_data = (DATA); \
}

/*
 * globals
 */
extern struct vfs *rootvfs;                 /* ptr to root vfs structure */
extern struct vfssw vfssw[];                /* table of filesystem types */
extern struct vfssw *vfsNVFS;               /* vfs switch table end marker */
#endif
```

vm_hat.h (Sun-3 version) *machine specific*

```
/*      @(#)vm_hat.h  1.28 89/01/12 SMI */
/*
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

/*
 * VM - Hardware Address Translation management.
 *
 * This file describes the contents of the machine specific
 * hat data structures and the machine specific hat procedures.
 * The machine independent interface is described in <vm/hat.h>.
 */

struct ctx {
        u_char      c_lock: 1;
        u_char      c_clean: 1;         /* this ctx has no lines in cache */
        u_char      : 6;
        u_char      c_num;                      /* hardware info (context number) */
        u_short     c_time;                     /* pseudo-time for ctx lru */
        struct      as *c_as;           /* back pointer to address space */
};

struct smgrp {
        u_int       smg_lock: 1;
        u_int       smg_num: 15;                /* hardware info (smgrp #) */
        u_short     smg_keepcnt;                /* 'keep' count */
        struct      as *smg_as;         /* address space using this smgrp */
        addr_t      smg_base;           /* base address within address space */
        struct      smgrp *smg_next;    /* next smgrp for this as */
        struct      smgrp *smg_prev;    /* previous smgrp for this as */
        struct      sment *smg_sme;             /* ptr to sment array for this smgrp */
};

struct sment {
        struct      pmgrp *sme_pmg;             /* what pmeg this maps */
        int         sme_valid;
};
```

vm_hat.h (Sun-3 version) (cont.)
machine specific

```
struct pmgrp {
        u_int       pmg_lock: 1;
        u_int       pmg_num: 15;            /* hardware info (pmgrp #) */
        u_short     pmg_keepcnt;            /* 'keep' count */
        struct      as *pmg_as;             /* address space using this pmgrp */
        addr_t      pmg_base;           /* base address within address space */
        struct      pmgrp *pmg_next;    /* next pmgrp for this as */
        struct      pmgrp *pmg_prev;    /* previous pmgrp for this as */
        struct      pment *pmg_pme;         /* ptr to pment array for this pmgrp */
};

struct pment {
        struct      page *pme_page;         /* what page this maps */
        u_short     pme_next;           /* other pments mapping same page */
        u_short     pme_intrep : 1;         /* interrupt replaceable flag */
        u_short     pme_nosync : 1;         /* ghost unload flag */
        u_short     pme_valid : 1;          /* pme loaded flag */
};

/*
 * These macros manipulate the next index of the pme struct.
 */
#define     PMENXT_PTR(x)           ((x) == PMENXT_NULL ? \
                                    (struct pment *)NULL : &pments[(x)])
#define     PMENXT_INDEX(x)         ((struct pment *)(x) == (struct pment *)NULL ? \
                                    PMENXT_NULL : (struct pment *)(x) - pments)
#define     PMENXT_NULL             0xFFFF

/*
 * The hat structure is the machine dependent hardware address translation
 * structure kept in the address space structure to show the translation.
 *
 * We maintain that any time an ctx structure exists,
 * that all pmgrps on the hat are loaded into the hardware.
 */
struct hat {
        struct      ctx *hat_ctx;           /* hardware ctx info (if any) */
        struct      smgrp *hat_smgrps;  /* smgrps belonging to this hat */
        struct      pmgrp *hat_pmgrps;  /* pmgrps belonging to this hat */
};
```

vm_hat.h (Sun-3 version) (cont.)
machine specific

```
/*
 * Flags to pass to hat_pteunload() and hat_ptesync().
 */
#define HAT_NOSYNC          0x00
#define     HAT_RMSYNC      0x01
#define     HAT_NCSYNC      0x02
#define     HAT_INVSYNC     0x04
#define     HAT_VADDR       0x08

/*
 * Flags to pass to hat_pteload() and segkmem_mapin().
 */
#define     PTELD_LOCK      0x01
#define     PTELD_INTREP    0x02
#define PTELD_NOSYNC        0x04
#define PTELD_IOCACHE       0x08

#ifdef KERNEL
/*
 * For the Sun MMU hat code, we don't need to do any initialization at
 * hat creation time since we know that the structure is guaranteed to be
 * all zeros.  So we can define the machine independent interface routine
 * hat_alloc() to be a nop here (<vm/hat.h> is set up to deal with this).
 */
#define     hat_alloc(as)

/*
 * These routines are all MACHINE-SPECIFIC interfaces to the hat routines.
 * These routines are called from machine specific places to do the
 * dirty work for things like boot initialization, mapin/mapout and
 * first level fault handling.
 */

/*
 * Boot time pmgrp initialization routines
 */

void        hat_pmginit();
void        hat_pmgreserve(/* seg, addr */);
```

vm_hat.h (Sun-3 version) (cont.)
machine specific

```
#ifdef sun4
#ifdef SUN4_460
void        hat_smginit();
void        hat_smgreserve(/* seg, addr */);
#endif

#else

#define                 hat_smginit()
#define                 hat_smgreserve(seg, addr)
#define                 hat_smgalloc(as, addr, pmg)

#endif

/*
 * Allocate a ctx for the specified as.
 */
void        hat_getctx(/* as */);

/*
 * Reserve hat resources for the address range. In this implementation, this
 * means allocating and locking the pmegs. This currently only works with
 * kernel addresses.
 */
void        hat_reserve(/* seg, addr, len */);
```

vm_hat.h (Sun-3 version) (cont.)
machine specific

```
/*
 * Called by UNIX during pagefault to insure a context is allocated
 * for this address space and that the ctx time is updated.  Also
 * used internally before all operations involving the translation
 * hardware if you need to be in the correct context for the operation.
 * Done inline for performance reasons.
 */
#ifndef sun2
#define hat_setup(x)    { register struct ctx *ctx = x->a_hat.hat_ctx; \
                                extern u_short ctx_time; \
                                if (ctx != kctx) { \
                                    if (ctx == NULL) { \
                                            hat_getctx(x); \
                                    } else { \
                                            ctx->c_time = ctx_time++; \
                                            mmu_setctx(ctx); \
                                    } \
                                } \
                            }
#else sun2
#define hat_setup(x)    { register struct ctx *ctx = x->a_hat.hat_ctx; \
                                extern u_short ctx_time; \
                                    if (ctx == NULL) { \
                                            hat_getctx(x); \
                                    } else { \
                                            ctx->c_time = ctx_time++; \
                                            mmu_setctx(ctx); \
                                    } \
                            }
#endif sun2
```

vm_hat.h (Sun-3 version) (cont.)
machine specific

```
/*
 * Load up a particular translation
 */
void        hat_pteload(/* seg, addr, pp, pte, lock */);

/*
 * Convert from mman protections to a protection value to be loaded in a pte
 */
u_int       hat_vtop_prot(/* vprot */);

/*
 * Construct a memory pte using the page pointer and mman protections specified
 */
void        hat_mempte(/* pp, vprot, ppte */);

#ifdef sun4c
/*
 * Flush all translations to a given physical page from the cache.
 */
void        hat_vacsync(/* pfnum */);
#endif sun4c

extern      struct pmgrp *pmgrp_invalid;
#endif KERNEL
```

vnode.h

```
/*
 * The vnode is the focus of all file activity in UNIX.
 * There is a unique vnode allocated for each active file, each current
 * directory, each mounted-on file, each mapped file, and the root.
 */

/*
 * vnode types. VNON means no type.
 */
enum vtype              { VNON, VREG, VDIR, VBLK, VCHR, VLNK, VSOCK, VBAD, VFIFO };

struct vnode {
        u_short                 v_flag;                         /* vnode flags (see below) */
        u_short                 v_count;                /* reference count */
        u_short                 v_shlockc;              /* count of shared locks */
        u_short                 v_exlockc;              /* count of exclusive locks */
        struct vfs      *v_vfsmountedhere;      /* ptr to vfs mounted here */
        struct vnodeops         *v_op;                          /* vnode operations */
        union {
                struct socket           *v_Socket; /* unix ipc */
                struct stdata           *v_Stream; /* stream */
                struct page *v_Pages;   /* vnode pages list */
        } v_s;
        struct vfs      *v_vfsp;                /* ptr to vfs we are in */
        enum vtype v_type;                              /* vnode type */
        dev_t                   v_rdev;                         /* device (VCHR, VBLK) */
        caddr_t                 v_data;                         /* private data for fs */
};
#define     v_stream    v_s.v_Stream
#define     v_socket    v_s.v_Socket
#define     v_pages             v_s.v_Pages

/*
 * vnode flags.
 */
#define     VROOT               0x01        /* root of its file system */
#define     VEXLOCK             0x02        /* exclusive lock */
#define     VSHLOCK             0x04        /* shared lock */
#define     VLWAIT              0x08        /* proc is waiting on shared or excl. lock */
#define     VNOCACHE            0x10        /* don't keep cache pages on vnode */
#define     VNOMAP              0x20        /* vnode cannot be mapped */
#define     VISSWAP             0x40        /* vnode is part of virtual swap device */
```

vnode.h (cont.)

```
 * Operations on vnodes.
 */
struct vnodeops {
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_lockctl)();
        int     (*vn_fid)();
        int     (*vn_getpage)();
        int     (*vn_putpage)();
        int     (*vn_map)();
        int     (*vn_dump)();
        int     (*vn_cmp)();
        int     (*vn_realvp)();
};

#ifdef KERNEL
```

vnode.h (cont.)

```
#define VOP_OPEN(VPP,F,C)                (*(*(VPP))->v_op->vn_open)(VPP, F, C)
#define VOP_CLOSE(VP,F,C,CR)             (*(VP)->v_op->vn_close)(VP,F,C,CR)
#define VOP_RDWR(VP,UIOP,RW,F,C)         (*(VP)->v_op->vn_rdwr)(VP,UIOP,RW,F,C)
#define VOP_IOCTL(VP,C,D,F,CR)                   (*(VP)->v_op->vn_ioctl)(VP,C,D,F,CR)
#define VOP_SELECT(VP,W,C)               (*(VP)->v_op->vn_select)(VP,W,C)
#define VOP_GETATTR(VP,VA,C)                     (*(VP)->v_op->vn_getattr)(VP,VA,C)
#define VOP_SETATTR(VP,VA,C)                     (*(VP)->v_op->vn_setattr)(VP,VA,C)
#define VOP_ACCESS(VP,M,C)               (*(VP)->v_op->vn_access)(VP,M,C)
#define VOP_LOOKUP(VP,NM,VPP,C,PN,F)     (*(VP)->v_op->vn_lookup)(VP,NM,VPP,C,PN,F)
#define VOP_CREATE(VP,NM,VA,E,M,VPP,C)           (*(VP)->v_op->vn_create) \
                                                         (VP,NM,VA,E,M,VPP,C)
#define VOP_REMOVE(VP,NM,C)                      (*(VP)->v_op->vn_remove)(VP,NM,C)
#define VOP_LINK(VP,TDVP,TNM,C)                  (*(TDVP)->v_op->vn_link)(VP,TDVP,TN
M,C)
#define VOP_RENAME(VP,NM,TDVP,TNM,C)             (*(VP)->v_op->vn_rename) \
                                                         (VP,NM,TDVP,TNM,C)
#define VOP_MKDIR(VP,NM,VA,VPP,C)        (*(VP)->v_op->vn_mkdir)(VP,NM,VA,VPP,C)
#define VOP_RMDIR(VP,NM,C)               (*(VP)->v_op->vn_rmdir)(VP,NM,C)
#define VOP_READDIR(VP,UIOP,C)                   (*(VP)->v_op->vn_readdir)(VP,UIOP,C)
#define VOP_SYMLINK(VP,LNM,VA,TNM,C)             (*(VP)->v_op->vn_symlink) \
                                                         (VP,LNM,VA,TNM,C)
#define VOP_READLINK(VP,UIOP,C)                  (*(VP)->v_op->vn_readlink)(VP,UIOP,C)
#define VOP_FSYNC(VP,C)                          (*(VP)->v_op->vn_fsync)(VP,C)
#define VOP_INACTIVE(VP,C)               (*(VP)->v_op->vn_inactive)(VP,C)
#define VOP_LOCKCTL(VP,LD,CMD,C,ID)      (*(VP)->v_op->vn_lockctl) \
                                                         (VP,LD,CMD,C,ID)
#define VOP_FID(VP, FIDPP)               (*(VP)->v_op->vn_fid)(VP, FIDPP)
#define VOP_GETPAGE(VP,OF,SZ,PR,PL,PS,SG,A,RW,C)\
                                                 (*(VP)->v_op->vn_getpage) \
                                                         (VP,OF,SZ,PR,PL,PS,SG,A,R
W,C)
#define VOP_PUTPAGE(VP,OF,SZ,FL,C)       (*(VP)->v_op->vn_putpage)(VP,OF,SZ,FL,C)
#define VOP_MAP(VP,OF,AS,A,SZ,P,MP,FL,C)\
                                                 (*(VP)->v_op->vn_map) \
                                                         (VP,OF,AS,A,SZ,P,MP,FL,C)
#define VOP_DUMP(VP, ADDR, BN, CT)       (*(VP)->v_op->vn_dump)(VP, ADDR, BN, CT)
#define VOP_CMP(VP1, VP2)                (*(VP1)->v_op->vn_cmp)(VP1, VP2)
#define VOP_REALVP(VP, VPP)              (*(VP)->v_op->vn_realvp)(VP, VPP)
```

vnode.h (cont.)

```
/*
 * Flags for above
 */
#define IO_UNIT                 0x01            /* do io as atomic unit for VOP_RDWR */
#define IO_APPEND   0x02                /* append write for VOP_RDWR */
#define IO_SYNC                 0x04            /* sync io for VOP_RDWR */
#define LOOKUP_DIR 0x08                 /* want parent dir. vnode, VOP_LOOKUP */

#endif KERNEL

* Vnode attributes.  A field value of -1
 * represents a field whose value is unavailable
 * (getattr) or which is not to be changed (setattr).
 */
struct vattr {
        enum vtype va_type;     /* vnode type (for create) */
        u_short                 va_mode;  /* files access mode and type */
        short                   va_uid;         /* owner user id */
        short                   va_gid;         /* owner group id */
        long                    va_fsid;    /* file system id (dev for now) */
        long                    va_nodeid; /* node id */
        short                   va_nlink;   /* number of references to file */
        u_long                  va_size;    /* file size in bytes (quad?) */
        long                    va_blocksize;           /* blocksize preferred for i/o */
        struct timeval          va_atime;  /* time of last access */
        struct timeval          va_mtime;  /* time of last modification */
        struct timeval          va_ctime;  /* time file "created */
        dev_t                   va_rdev;    /* device the file represents */
        long                    va_blocks; /* kbytes of disk space held by file */
};

/*
 *  Modes. Some values same as Ixxx entries from inode.h for now
 */
#define     VSUID     04000             /* set user id on execution */
#define     VSGID     02000             /* set group id on execution */
#define VSVTX         01000             /* save swapped text even after use */
#define     VREAD     0400              /* read, write, execute permissions */
#define     VWRITE    0200
#define     VEXEC     0100
```

vnode.h (cont.)

```
#ifdef KERNEL
/*
 * public vnode manipulation functions
 */
extern int vn_open();                   /* open vnode */
extern int vn_create();                 /* creat/mkdir vnode */
extern int vn_rdwr();                   /* read or write vnode */
extern int vn_close();                  /* close vnode */
extern void vn_rele();                  /* release vnode */
extern int vn_link();                   /* make hard link */
extern int vn_rename();                         /* rename (move) */
extern int vn_remove();                         /* remove/rmdir */
extern void vattr_null();               /* set attributes to null */
extern int getvnodefp();                /* get fp from vnode fd */

#define VN_HOLD(VP)             { \
        (VP)->v_count++; \
}

#define VN_RELE(VP) { \
        vn_rele(VP); \
}

#define VN_INIT(VP, VFSP, TYPE, DEV)    { \
        (VP)->v_flag = 0; \
        (VP)->v_count = 1; \
        (VP)->v_shlockc = (VP)->v_exlockc = 0; \
        (VP)->v_vfsp = (VFSP); \
        (VP)->v_type = (TYPE); \
        (VP)->v_rdev = (DEV); \
        (VP)->v_socket = 0; \
}
```

vnode.h (cont.)

```
* Compare two vnodes for equality.
 * This macro should always be used,
 * rather than calling VOP_CMP directly.
 */
#define VN_CMP(VP1,VP2)                 ((VP1) == (VP2) ? 1 : \
        ((VP1) && (VP2) && ((VP1)->v_op == (VP2)->v_op) ? VOP_CMP(VP1,VP2) : 0))

/*
 * Flags for above
 */
enum rm             { FILE, DIRECTORY };                    /* rmdir or rm (remove) */
enum symfollow      { NO_FOLLOW, FOLLOW_LINK };             /* follow symlinks (lookuppn)
*/
enum vcexcl         { NONEXCL, EXCL};           /* (non)excl create (create) */

/*
 * Global vnode data.
 */
extern struct vnode     *rootdir;           /* root (i.e. "/") vnode */
extern enum vtype       mftovt_tab[];
#define     MFMT            0170000                     /* type of file */
#define MFTOVT(M)   (mftovt_tab[((M) & MFMT) >> 13])

#endif KERNEL
```

Appendix C

Using kadb

- System must be booted using kadb
- kadb must reside in the root directory as /kadb
- Press L1–a keys simultaneously(to do a system break)
- If kadb is running you will see the following prompt

  kadb>
- If kadb is not running, pressing L1–a will give you

  >
- Type C to the PROM prompt to continue SunOS
- Halt the system using /etc/halt, then reboot the system using kadb by specifying kadb as the executable

  > b kadb
- After rebooting using kadb, doing a system break(pressing L1–a keys) will get you into kadb

Using kadb

Kadb can only be used if the system was booted using kadb. Kadb must reside in the root directory (/kadb) and is installed as part of the "debugger" installation option. Check to see if kadb is installed by listing the root directory.

Press the L1–a keys simultaneously to perform a system break. This will drop the system into kadb (if it is running) or into the PROM monitor (if kadb is not running). As long as you don't destroy the stack and registers, you can continue running SunOS. The L1 key may have a "Stop" label affixed to the keycap, but you can easily identify this key because it is upper leftmost key in the left hand function keypad.

If you execute the system break, and you get a PROM prompt >, then kadb is not running. In this case, type C to the PROM prompt to continue SunOS and then halt the system with **/etc/halt.** Then reboot the system specifying kadb as the executable (**>b kadb**). Kadb will then boot vmunix which must be located in the root directory (*/vmunix*). After rebooting, you must get the kadb prompt (**kadb>**) after doing a system break.

Using kadb (cont)

- To display the kernel's symbol table use the nm command

  # nm /vmunix

- Go into kadb and try using the address for _proc in the following way:

  kadb> proc_addr$<proc
  or
  kadb> proc$<proc
  kadb> u$<u
  kadb> proc,11/X
  kadb> psignal,22/x
  kadb> kernel_routine,10/o

- Other formats can be used:
  b for one byte octal
  o for one word octal
  d for one word decimal
  D for one longword decimal
  c for one byte character
  f for single precision
  F for double precision
  s for null terminated string

Using kadb (cont)

If you are unfamiliar with the symbolic names of the different kernel tables and structures, you can display the symbol table of the kernel (or any executable file) by using the nm command. The output of the nm command will be the names of the variables (including function names) and their respective virtual addresses within the executable file. If you do not want to display the entire symbol table, you can pipe the output of nm to grep in this manner:

% nm /vmunix | grep proc

and this will display only symbols that contain the string proc.

You then can use the address from the symbol table to display the contents of a proc structure from the process table, for example. Notice that if you do not know the actual address you can use the name of the symbol instead of the address as shown in the examples on the left. So, instead of an actual address we can enter proc (for a process table entry), u (for the user structure) and psignal (which is the name of a kernel routine).

The kadb command **proc,11/X** will display 17 (0x11) values of size X (4 bytes) starting at address proc. Other formats are available in addition to X, like: b for one byte octal, o for one word (16 bits) octal, d for one word decimal, D for one longword decimal, c for one byte character, f for single precision, F for double precision, and s for null terminated string (must be null terminated!).

Using kadb (cont)

- Formatting scripts can be used to display different kernel structures

  kadb> kernel_addr$<file
  kadb> kernel_addr$<inode
  kadb> kernel_addr$<seg
  The complete listing of different scripts can be found in /usr/lib/adb

- To return to SunOS, type :c at the kadb prompt

  kadb> :c

- Other kadb commands:

  kadb> $c
  kadb> $C
  kadb> psignal+4:b
  kadb> $b
  kadb> psig+4:b
  kadb> $b
  kadb> psig+4:d
  kadb> $b
  kadb> $r

Using kadb (cont)

There are formatting scripts (files that are located in */usr/lib/adb*) that can be used to display different structures if their addresses (or symbols representing their addresses are passed on to kadb or adb). If we want to display a file structure for example, we will pass an address corresponding to a file structure to kadb (or adb) and use the "file" script to display the file structure starting from the address that we passed. Kadb or adb does not check if the address is a valid file address, but what the command will tell kadb to do is to interpret the next so many bytes (the size of the struct file) as a file structure and to display its contents. Thus it is possible to pass an invalid address as that of a proc table entry and then to try to display the contents from the starting address. You will be able to tell if you have passed a wrong address when you get weird data in the structure that you have displayed.

To return to SunOS, simply type **:c** at the kadb prompt.

The kadb commands **$c** and **$C** respectively, will dump the stack, without and with local variable interpretation which is a common way to begin debugging. The next commands on the opposite page will alternately set or delete a breakpoint (**:b** to set, **:d** to delete) and then display the breakpoint status (**$b**). Note that all breakpoints must be set at function address plus 4 to allow the link instruction to execute before the break. Otherwise, we would break into a function that hasn't had its stack frame built yet. These commands also illustrate the symbolic nature of kadb. Psignal and psig are kernel routines. We could have done an **nm** command on */vmunix* to find the function addresses and used those in place of the symbols psignal and psig.

powers of two

0	1	
1	2	
2	4	
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1024	1k
11	2048	2k
12	4096	4k
13	8192	8k
14	16384	16k
15	32768	32k
16	65536	64k
17	131072	128k
18	262144	256k
19	524288	512k
20	1048576	1M
21	2097152	2M
22	4194304	4M
23	8388608	8M
24	16777216	16M
25	33554432	32M
26	67108864	64M
27	134217728	128M
28	268435456	256M
29	536870912	512M
30	1073741824	1G
31	2147483648	2G
32	4294967296	4G

Post-Class Survey

Sun Microsystems Educational Services thanks you for attending this class. Please take a few minutes to fill out this survey. Your opinions are valuable to us, and help us to insure that our course offerings provide the best possible training environment, course content, and instruction. Please ask your instructor to explain any question that you do not understand.

Course Name: _____ **Date:** ___/___/___

Student Name (optional): _____

FACILITIES

1) Did you attend this class at a Sun facility? No Yes

2) Did you have enough "hands on" time to complete your course work? No Undecided Yes

3) Did the machine(s) that you use function properly? No Undecided Yes

4) Did the lab environment promote learning? No Undecided Yes

5) Did the classroom environment promote learning? No Undecided Yes

Comments about the **facility** _____

_____

_____

Please rate your overall satisfaction with the **facility** 1 2 3 4 5 6 7 8 9 10
Least Most

COURSE CONTENT

1) When did you first learn of the course content?
☐ Before registering ☐ Before attending class ☐ The first day of class ☐ Other

2) Based on the course description, how well did the course content fulfill the course objectives?
☐ Not well ☐ Moderately well ☐ Exactly ☐ Exceeded the objectives

3) List any **specific** topics that you felt were:
a) presented in too much detail b) presented in too little detail
c) presented but did not apply d) not presented but should have been

Topic	Comment
_____	a b c d
_____	a b c d
_____	a b c d
_____	a b c d

4) Did the structure of the Student Guide promote learning? No Undecided Yes

5) Was the Student Guide easily readable? No Undecided Yes

6) Did the lab exercises reinforce the lecture material? No Undecided Yes

Comments about the **course content** _____

_____

_____

Please rate your overall satisfaction with the **content** 1 2 3 4 5 6 7 8 9 10
Least Most

INSTRUCTOR

1) Did the instructor state the course objectives? [No] [Undecided] [Yes]

2) Did the instructor use examples that reinforced the course material? [No] [Undecided] [Yes]

3) Did the instructor summarize the material so as to emphasize the important points? [No] [Undecided] [Yes]

4) Did the instructor manage the training environment in a way that promoted learning? [No] [Undecided] [Yes]

5) Did the instructor have a good understanding of the course content? [No] [Undecided] [Yes]

Comments about the **instructor** _____

_____

_____

Please rate your overall satisfaction with the **instructor** 1 2 3 4 5 6 7 8 9 10
Least Most

CLASS

Comments about the **class** _____

_____

_____

Please rate your overall satisfaction with the **class** 1 2 3 4 5 6 7 8 9 10
Least Most

Please use this space to make any other comments and/or suggestions that you may have. Feel free to note any exceptional service or effort performed by your instructor or any other member of the Educational Services staff. Again, we thank you for your patronage and we appreciate the time you took to provide this information.

_____

_____

_____

_____

_____

_____

_____

_____

HOTEL

If you stayed in a hotel during the class, name of hotel: _____

Comments about the **hotel** _____

_____

_____

Please rate your overall satisfaction with the **hotel** 1 2 3 4 5 6 7 8 9 10
Least Most