Engineer's Handbook
Uncle Art's Big Book of Irix

May 5, 1994

Contents

Engineer's Handbook
Uncle Art's Big Book of Irix

May 5, 1994

How to access the handbook command

- install the information tools software (only need to do this once or when the tools change)

```
% su
# inst -f dist.wpd:/sgi/infotools
# exit
% rehash
% echo "make sure /usr/local/bin is in path"
```

- handbook always accesses most current version of the data
- to find out what chapters exist

```
% handbook
```

- to find out what chapters exist

```
% handbook <chapter_name>
```

- to print a chapter on your default printer

```
% handbook -o - <chapter_name> | lp
```

- to get a copy of the actual data

```
% handbook -o <file_name> <chapter_name>
```

- see the man page for further handbook options

Uncle Art's Big Book of Irix
(Cypress only version)

access via: handbook -s dist.wpd:/sgi/doc/swdev/BigIrixBook *<chapter>*
probable data location: dist.wpd:/sgi/doc/swdev/BigIrixBook

Uncle Art's Big Book of IRIX

Document Version 1.2

Document Number

Written by Arthur Evans and Jeffrey B. Zurschmeide
Edited by
Production by
Engineering contributions by: Ana Maria De Avare, Nelson Bolyard, Paul Close, Ellen Desmond, Dave Higgen, Bill Kawakami, Sue Liu, Paul Mielke, Sandra Romero, Casey Schaufler, Aaron Schuman, Mike Thompson, Chris Wagner

Uncle Art's Big Book of IRIX
Document Version 1.2
Document Number

Silicon Graphics, Inc.
Mountain View, California

Contents

1. Introduction to the IRIX system

The IRIX operating system is a multi-user, multi-tasking operating system based on the UNIX system developed by AT&T. IRIX is in many respects identical to System V UNIX, as described by Maurice J. Bach in his book, *The Design of the UNIX Operating System*. IRIX also includes enhancements from the Berkley Software Distribution (BSD) version of UNIX, which is described in *The Design and Implementation of the 4.3BSD UNIX Operating System*, by Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. Both of these books will be referenced throughout this book.

The major differences between IRIX and System V UNIX are:

- IRIX supports multiple file system types. The native IRIX file system is the Extent File System (EFS) which provides improved performance over the standard AT&T file system. Chapter 3 of this book describes both the Extent File System and the file system abstraction that allows IRIX to support multiple file system types.
- IRIX supports the BSD *socket* abstraction for interprocess communication. This is described in Leffler, et al., Chapter 10. A few differences between the BSD implementation and the IRIX implementation are discussed in Chapter 7.
- IRIX features BSD UNIX derived networking facilities, including support for the DARPA Internet network protocols. These facilities are described in Leffler, et al., Chapters 11 and 12.
- IRIX supports the Sun Network File System (NFS). NFS is described in *TCP/IP and NFS: Internetworking in a UNIX Environment* by Michael Santifaller.

1.1 The IRIX Environment

Two of the basic concepts in the IRIX system are *file* and *process*. A file contains data. A process is an instance of an executable file being run.

1.1.1 Processes

As noted above, a process is an instance of a program being executed. Under IRIX, a large number of processes may run simultaneously. The IRIX system manages the system resources—CPU, memory, and peripheral devices—in such a way that many processes can share these resources. The system allows many process to use one CPU by switching between processes. The system allows many processes to run in limited amount of physical memory by implementing a *virtual memory* scheme. Memory addresses in compiled programs are interpreted as virtual addresses, which are mapped to physical memory addresses by the system. In this way, a process is prevented from accidentally accessing another process's memory. IRIX features *demand paged* memory management, which means that the system can cache pages of text and data which are not immediately needed on disk. This allows the system to run programs which are larger than physical memory.

Each process has a *user file descriptor table* which contains references to "open" files. System calls which perform file I/O take an integer *file descriptor* as an argument, which is used to find the correct entry in the user file descriptor table. Usually, a process will start out with three open files, *standard input*, *standard output*, and *standard error output*, which by convention are represented by file descriptors 0, 1 and 2. In a normal interactive process, standard input, standard output, and standard error output all refer to the user's terminal.

The IRIX system provides a number of *system calls* which provide for the creation and termination of processes, the manipulation of process address space, interprocess communication, and other important operations. The *fork* system call creates a new process by making a copy of the calling process. This new process is referred to as the child of the calling, or parent, process. Unless otherwise specified, the child inherits the parent's file descriptors. This property has important implications, as will be seen in the section on the shell, below.

1.1.2 Files

As far as IRIX is concerned, file data consist of unformatted streams of bytes. User programs may impose their own structure on data that they read or write, but the operating system imposes no structure of its own. A file can be simply a set of data stored on disk, in which case it is referred to as a *regular file*. A file can also be an I/O device, such as a tape drive or terminal (such a file is called a *device special file*). Certain other types of files exist, and will be described as they come up.

Files are organized into a hierarchical *file system*. This file system may be represented as a tree, with nodes representing files. All non-leaf nodes are *directories*, special files which contain references to other files. The root of the tree is called the *root directory*, and is represented by a slash (*/*). Any file in the hierarchy can be identified by its *path name*. Path names are constructed from the names of all the individual nodes on the path between the root directory and the file being identified. Thus, in the file system pictured in figure 1-1, "/dev/tape," "/etc," and "/usr/people/arthur/poem" are all valid path names. In this example, the files pictured as folders are directories, **/dev/tape** is a device special file representing a tape drive, and **/etc/passwd** and **/usr/people/arthur/poem** are regular files.

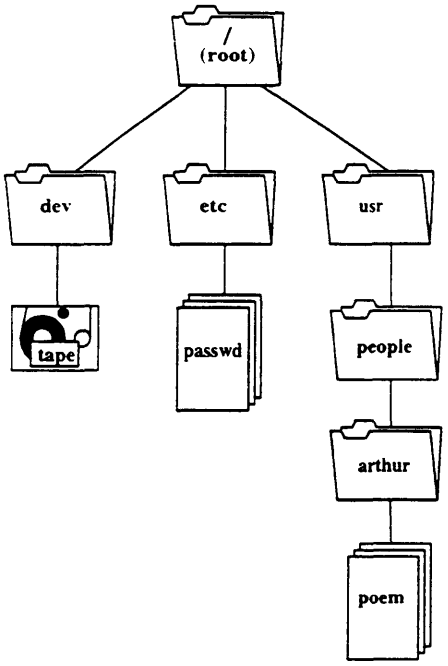

Figure 1-1. Example File System

In addition to the type of path name described above, called an *absolute path name*, files can be referred to by *relative path names*. Each IRIX process has a *current directory*, and path names which do not begin with a slash are interpreted relative to the current directory. The special file names "." and ".." are used to refer to the current directory, and the parent of the current directory, respectively. Therefore, if a process's current directory was **/usr**, it would interpret the path names **people/arthur** or **./people/arthur** as equivalent to **/usr/people/arthur**, and interpret the path name **../etc/passwd** as equivalent to **/etc/passwd**.

IRIX files are protected by a *discretionary access control* system. Under this system a file's owner may set the file's *access modes* to either prevent or allow access by other users. IRIX recognizes three kinds of access: read, write, and execute. Permission for each of these kinds of access can be

controlled for each of three classes of users: the owner of the file, users in the same *group* as the file's owner, and all other users.

While directories use the same access modes as other files, they are treated slightly differently. A process must have execute permission for a directory in order to make that directory its current directory, or to access files in that directory (the process must still have access permission for the files themselves). Write permission for a directory allows a process to create or delete files in that directory—it is not necessary to own a file, or have write permission for it, to delete it.

User programs access the file system through *system calls*. Some of the most common system calls for the file system are *creat*, *open*, *close*, *read*, *write* and *mknod*. *creat* creates a new file, and *open* opens a file for reading or writing, creating it if necessary. Both *creat* and *open* return an integer *file descriptor*, which is used to identify the file to other system calls. *read* reads data from an open file, and *write* writes data to an open file. *close* closes an open file, so that its file descriptor can be reused. *mknod* is used to create special files, such as device files and directories.

Pipes

One kind of special file deserving mention at this point is the *pipe*, which provides one means of interprocess communication. Pipes can be read from and written to like other files; however, data written to a pipe can only be read once. Data written to the pipe is held in a circular buffer until read. Therefore the file acts like a pipeline, carrying data from one process to another. There are two kinds of pipes, *named pipes* and *unnamed pipes*. The named pipe exists within the regular file system hierarchy. That is to say, it has a pathname and access permissions, and may be accessed with the *open* system call. Named pipes are created with the *mknod* system call, like other special files. Unnamed pipes have no place in the file system hierarchy. They are created with the *pipe* system call, which returns two file descriptors, representing the two "ends" of the pipe. By creating an unnamed pipe and then *fork*ing a child process, which will also be able to reference the pipe's file descriptors, a process creates a channel of communication with the child process. Communication between two child processes can be accomplished in a similar manner. An unnamed pipe ceases to exist when the last file descriptor referencing it is *close*d.

1.2 The Kernel

The *kernel* is the center of the IRIX system. The kernel is a collection of routines which perform the following functions:

- divide processor time fairly among all processes,
- mediate access to data objects by processes, ensuring that data ownership is maintained,
- provide a consistent programmatic interface to system hardware,
- provide a file-hierarchic view of mass storage devices, and
- provide complex protocols for network communication.

The kernel routines do not run as separate processes—they are run as part of the individual processes. When a process is executing a kernel routine, it operates in *kernel mode*, meaning it can access both kernel and user virtual memory, and use certain privileged instructions. When a process is not in kernel mode, it is in *user mode*, and can only access its own virtual memory. Kernel mode and user mode are implemented at the hardware level. A process may make the transition into kernel mode voluntarily, by requesting a service from the kernel (that is, by making a system call). A process may also enter kernel mode if the kernel has to service an interrupt or exception while the process is running. This book will use the convention of attributing various actions to "the kernel," but it should be remembered that these actions are carried out by the current process, operating in kernel mode.

1.2.1 Interrupts and Exceptions

IRIX uses a single mechanism to deal with the various occurrences that can interrupt the execution of a process. These occurrences may be broken down into two categories: *interrupts* and *exceptions*. Interrupts are asynchronous with the execution of the current process, and are not necessarily caused by anything that the process did—for example, an interrupt could signal the completion of an asynchronous I/O operation requested by another process. Exceptions, on the other hand, are caused by the process itself, when it attempts an invalid operation, such as dividing by zero or accessing non-existent memory locations. When either an interrupt or an exception is received, the kernel first preserves the *context* of the

current process (the information necessary to restart the process where it left off), then determines the cause of interrupt or exception and calls the appropriate handler routine. The only difference between the treatment of interrupts and exceptions is that exceptions occur, by their nature, in the middle of instructions, so that after returning from an exception, the system must attempt to restart the instruction that caused the exception. Interrupts are handled between instructions, so there is no need to restart an instruction.

1.2.2 Processor Execution Levels

The IRIX system prioritizes interrupts so that crucial, time-critical interrupts are not interfered with by less crucial interrupts. Interrupt routines use a privileged instruction to set the *processor execution level* such that lower-priority interrupts are blocked out. Under IRIX, the highest priority interrupts are hardware errors (for example, bus errors).

1.3 The Shell

So far the system has been discussed in terms of its architecture and programmatic interface. The interface that the user sees is not a part of the kernel operating system, but rather a separate user program, referred to as the *shell*. Since the shell is an ordinary program, users can write and compile their own shells, thereby customizing their user interfaces. The IRIX system offers two shells to chose from, the Bourne shell (named after its author, Steven Bourne) and the C shell (so named because its syntax resembles that of the C programming language). For the sake of simplicity, this book will only cover the Bourne shell, so references to "the shell" should be understood to refer to the Bourne shell.

Commands to the shell are divided into two categories, built-in commands and executable files. Built-in commands, as the name implies, are built into the shell. They include commands for changing the shell's working directory and setting various shell parameters. The shell also provides variables, branching and looping mechanisms, and a facility for defining functions. Together, these built-ins allow the shell to be used as a programming language as well as a command interpreter. Executable files may be specified with a full pathname, or by a simple filename. In the latter

case, the shell searches through a series of directories to find the file. If the shell finds a file with the correct name, it tries to execute the file; if it cannot find the file it reports an error. For example, given the command:

```
$ who
```

The shell would find the program **/bin/who**, which prints a list of the users logged on to the system (the dollar sign is the shell's *prompt*, showing that it is ready for a command). The series of directories searched by the shell is referred to as the *execution search path*, and is defined by a shell variable, so that the users can configure the path to their own needs. This path commonly contains the directories **/bin**, **/usr/bin**, and **/usr/bsd**, and **/usr/sbin**.

The shell also allows the user to redirect a command's input and output. For example, the command:

```
$ psroff < document
```

Runs the program *psroff*, which formats text to be printed on a PostScript printer, taking standard input from the file **document**, instead of the terminal. Standard output and standard error output can also be redirected:

```
$ diff prog.c prog.c.old > diff.out
$ cc prog.c 2> errors
```

In the first example, the program *diff* is being used to compare the contents of two files, and its standard output is being redirected to the file **diff.out**. In the second example, the C language compiler, *cc*, is being run on a C program file, **prog.c**, and its standard error output is being redirected to the file **errors**. In addition to redirecting input and output to files, the shell can form "pipelines" by connecting the standard output of one process to the standard input of the next process with a pipe:

```
$ ls | lp
```

In this example, the standard output of the *ls* program, which produces a list of the files in the current directory, is being piped to the *lp* command, which sends output to a line printer.

1.4 IRIX Documentation

A large part of the IRIX documentation consists of ''manual pages'' each documenting a single command, routine, or file format. The manual pages that are referenced in this book are in several different places: the *IRIX Programmer's Reference Manual* (3 volumes), the *IRIX User's Reference Manual* (2 volumes), the *IRIX System Administrator's Reference Manual*, and *Writing Device Drivers For Silicon Graphics Computer Systems*. The manual pages are organized in sections according to content. The sections, and the manuals in which they are located, are shown in the table below.

Section		Manual
1	Commands	*User's Reference*, Vol. 1&2
1M	System Maintainence Commands	*System Administrator's Reference*
2	System Calls	*Programmer's Reference*, Vol. 1
3	Library Routines	*Programmer's Reference*, Vol. 2
4	File Formats	*Programmer's Reference*, Vol. 3
5	Miscellaneous Facilities	*Programmer's Reference*, Vol. 3
6	Demos and Games	*User's Reference*, Vol. 2
7	Special Files	*System Administrator's Reference*
K	Kernel Functions	*Writing Device Drivers*

Manual pages are referred to throughout the text by their name and section number: for example, *passwd*(1) refers to the manual page for the *passwd* command, in section 1 in the *IRIX User's Reference Manual.*

Other important documents include the *IRIS-4D System Administrator's Guide*, which contains information on system administration tasks, and the *IRIS-4D Programmer's Guide* (2 volumes) which contains information on frequently-used libraries and development tools. In addition to containing manual pages for certain kernel functions, *Writing Device Drivers for Silicon Graphics Computer Systems*, contains information on the I/O subsystem.

1.5 Typographical Conventions

In this book, the the names of a kernel data structures and variables are set in the `courier` typeface (for example: "The process table is an array of `proc` structures."). Courier is also used for examples and algorithms set apart from the main body text.

Italic text is used for numerous purposes. Italics are used to indicate a system call, library routine, or anything else that has a manual page associated with it (for example, *passwd*(1) above). Italics are also used to indicate the first use of a term, and to indicate the names of kernel routines. Very occasionally, italics are used to place special emphasis on a word or phrase.

2. Overview of the Kernel

The previous chapter introduced the concept of the kernel, and some of its functions. This chapter will describe in more detail the various components of the kernel and the relationships between them.

Figure 2-1 is a logical diagram of the modules in the kernel.

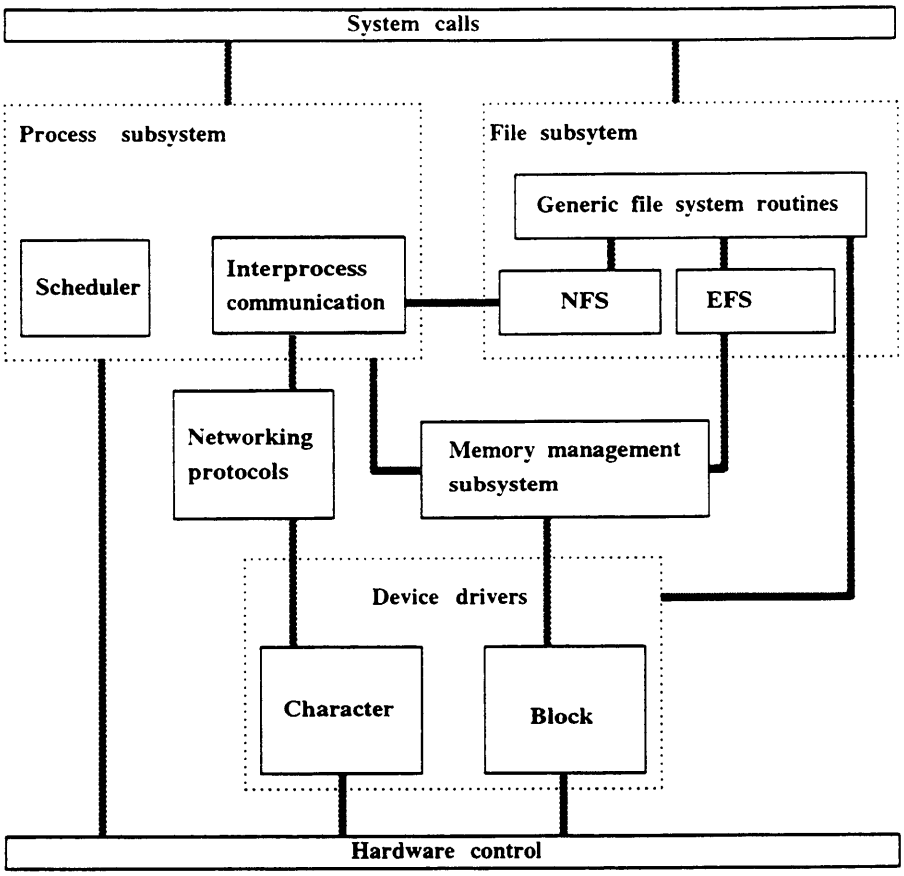

Figure 2-1. Relationship of kernel modules

At the top of the diagram are the system calls, the user entry points into the kernel. At the bottom of the diagram are the lowest-level hardware control routines. In between these layers, the kernel routines have been divided into groups: the process subsystem, the file subsystem, the memory management subsystem, networking protocols, and device drivers.

Device drivers are modules that interact with peripheral devices, such as disk drives, tape drives, and serial terminals. These devices are divided up into *block* and *character* devices. Block device drivers are random access and will only read or write fixed sized "blocks" of data. Character device drivers, on the other hand, support variable-size I/O requests and may or

may not support random access.

I/O on block devices is buffered through the *integrated data cache*. The integrated data cache is so named because the functions it serves were originally served by two separate modules—the buffer cache, for buffering regular I/O requests, and the page cache, for managing virtual memory. The integrated data cache is part of the memory management subsystem.

The process subsystem deals with the control of processes: process creation, termination, and scheduling, interprocess communication, and the manipulation of process address space. The process subsystem interacts with the integrated buffer cache in managing memory and virtual address space, and the process subsystem interacts with both the integrated data cache and the file subsystem to load a new program into memory (with the *exec* system call), since the executable file must be paged into memory from the file system. The scheduler module is responsible for fairly allocating processor time between processes. In this simplified diagram, interprocess communication has been shown as one module, but there are in fact two sets of interprocess communication primitives available under IRIX: the System V style primitives, and BSD sockets. User programs interact with the networking protocols through the socket abstraction.

The file subsystem manages most user I/O. Diagram 2-1 shows two file system types: EFS, the Extent File System, which IRIX uses to access file systems that reside on local hard disks, and NFS, the Sun Network File System, which provides for access to file systems on remote machines, which need not run IRIX as long as they support the NFS protocol. Regular file I/O is buffered through the integrated data cache. In the case of EFS, performing I/O involves interacting with the device driver for the hard disk. In the case of NFS, performing I/O involves interacting with the networking protocols through the socket abstraction. The networking protocols, in turn, interact with the network hardware device drivers. Users can also access devices directly through the file subsystem, by opening device special files.

2.1 Process Subsystem

The process subsystem relies on a number of data structures. The *process table* is an array of `proc` structures, each of which contains information about a single process. Each `proc` structure may be linked onto one or more of several lists maintained by the kernel. There is a *free list*, from

which the kernel allocates unused `proc` structures. Processes which are not on the free list are on the *active list*. In addition to these lists, there is a list of runnable processes, a double-linked list of sleeping processes, and a list of exiting processes.

The virtual address space of a process is divided into *regions*, which are contiguous areas of virtual address space. Each region is described by a `region` structure, which contains information about what kind of region it is, how many processes reference it, and so on. The link between the `region` structures and the `proc` structure is the `pregion` structure, or per-process region structure. Each `proc` structure contains a pointer to a linked list of `pregion` structures, each of which points to a single `region` structure. Each process also has a `user` structure, or *user block* (also called the *user area* or simply *u area*). The user block contains information about the process which can be swapped with the process, as opposed to the information in the `proc` structure, which must remain in core at all times. Each process also has a *kernel stack*, which the kernel uses when that process is running. The kernel stack is associated with the user block, and unless otherwise noted, whatever is said of the user block is true of the kernel stack. So a process consists of a `proc` structure, a user block and kernel stack, a linked list of `pregion` structures and a number of `region` structures.

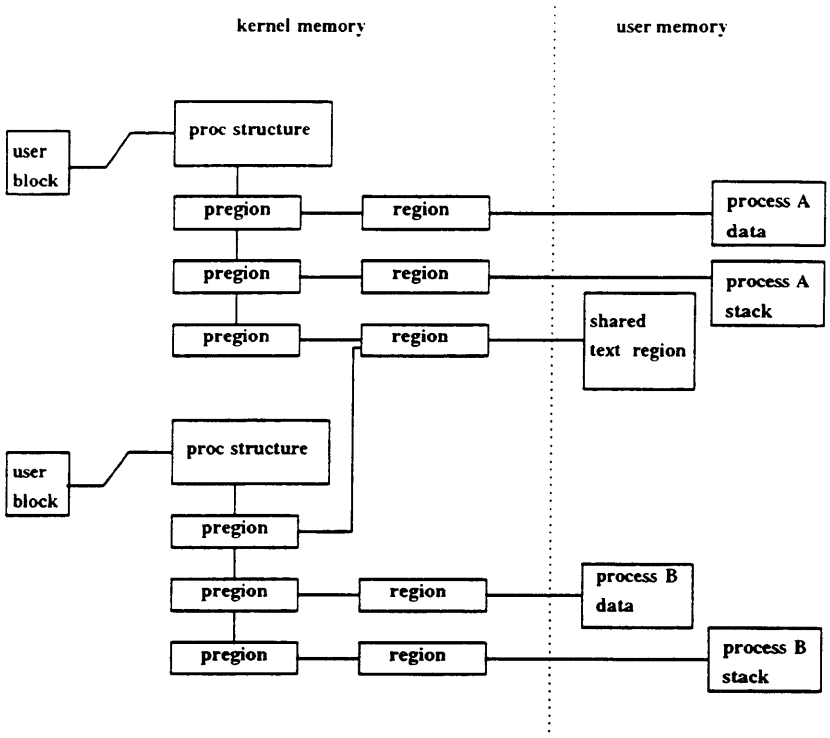

Figure 2-2. Process data structures

Figure 2-2 shows the data structures for two processes with one shared region. To simplify the figure, only the two `proc` structures are shown, not the entire process table. In the figure, the processes are sharing a text region, so each process has a `pregion` structure which points to the `region` structure controlling the text region. This could happen as the result of a *fork* system call—the child must be given its own data and stack regions, so it will not overwrite its parent's data. However, since program text is read-only, there is no need for the child to duplicate the text region.

The structures on the left of the dotted line exist within the kernel's virtual address space. The structures on the right of the dotted line exist in memory

managed by the integrated data cache, and are subject to swapping. The "region" blocks on the right of the dotted line are simply collections of pages in the integrated data cache. The pages of memory which make up a region need not be physically contiguous: they are mapped to contiguous virtual addresses by the memory management subsystem.

2.1.1 System Calls

How system calls are accomplished deserves some mention at this point. To make a system call, a process loads the system call number into one of the processor's registers, then generates a special kind of exception to make the switch into kernel mode. This invokes an exception handler called *syscall*, which executes the appropriate system call. Of course, the programmer does not need to know system call numbers. The standard library contains "front-end" routines for all the system calls, so that the programmer can call them from a C program. The C language calling conventions for all of the system calls are documented in the *IRIX Programmer's Reference Manual, Volume 1*. The arguments to the system call are copied from user address space into the user block. There are a number of variables in the user block which are simply used for storing arguments to system calls, so that kernel routines need not pass them on the stack. For example, during the *read* system call, which reads data from a file into a buffer in the user's address space, the address of the buffer is stored in the user block variable `u_base`, the number of bytes to read is stored in `u_count`, and the logical offset in the file is stored in `u_offset`.

When the system encounters an error during the execution of a system call, it sets the global variable `errno` to indicate the cause of the error. These error numbers are listed on the *intro*(2) manual page. The *perror*(3C) library routine may be used to print out a message describing the error.

2.1.2 Process Regions

The virtual address space of a process is divided up into regions. Each process has at least three regions: a text region, containing machine instructions; a data region; and a stack region. In addition to text, data, and stack, a process can have one or more *shared memory* regions, which it can share with other processes (shared memory regions are covered in Bach, Chapter 11). A final sort of region is the *mapped file* region, which

represents a file which has been mapped into memory through the *mmap*(2) system call.

2.1.3 Creating Processes

Processes are created through the *fork*(2) system call. The *fork* system call is executed, the kernel allocates a new `proc` structure (off the `proc` structure free list) for the new process, and assigns it a unique process ID number. The new process inherits the parent process's file descriptors. The new process also has a set of regions which appear to be an exact copy of the old process's regions at the time of the *fork* call.

IRIX links parent and child processes together on a list called the parent-child-sibling chain. Each process has a pointer to its parent process, and if it has children, a pointer to a child process. If the process has more than one child, the first child has a pointer to a second child, and so forth. Each parent process is considered to be the head of its own parent-child-sibling chain.

2.1.4 Sessions and Process Groups

IRIX also links related processes together on *session* and *process group* lists. A session is a set of processes related to a single login session. A process group is a set of processes which constitute a single logical "job." For example, if a user issues the following command:

```
ls | lp
```

The shell will start two programs, *ls*, which lists the files in the current directory, and *lp*, which sends output to the line printer. Since these two programs were issued together, they are considered a single logical job. Therefore, they are both put in the same process group. This allows them to be manipulated as a unit, where appropriate.

2.1.5 Signals

Processes can be notified of various conditions through the use of *signals*. Signals can be generated either by the kernel, in response to various errors (attempts to access illegal addresses, bus errors, etc.), by other processes, using the *kill*(2) system call, or by the process itself, which can request a signal at a pre-arranged time through the *alarm*(2) system call.

By default, a process will terminate when it receives a signal. A process can specify alternate behavior through the *signal*(2) system call. The process can opt to either ignore a given signal, or to call a specified subroutine when the signal is received. The *proc* structure contains fields defining the process's response to different signals, and a pointer to a queue of pending signals.

2.1.6 Job Control

A special set of signals is defined for *job control*. These signals can be used to suspend execution of an active process, and to resume execution of a suspended process. The shell uses these signals to control its child processes. For example, when the shell receives a certain character (usually control-Z), it suspends the process currently running in the foreground. Shell commands can then be used to continue the process in the background, or to resume its execution in the foreground.

2.2 File Subsystem

The most important kernel data structures for the file subsystem are the *file table*, the *inode table*, and the *file system switch table*. On the user side, each process has a *user file descriptor table* in its user block. The inode table, or inode pool, is the center of all file activity. The inode table is an array of `inode` structures, which contain information about files. Each `inode` uniquely identifies a file, and contains such information as the type of file (for example, regular, directory, or device), the access modes, and the length of the file. Inode structures also contain pointers to other `inode` structures which allow them to be linked onto lists. The kernel maintains a *free list*, containing all `inodes` which are not currently in use, and a

number of *hash lists*, onto which inodes may be linked to simplify the process of searching for a specific `inode`.

The file table is an array of `file` structures, which contain information specific to each open file. Each file structure contains:

- a set of flags which show the mode with which the file was opened (read, write, or both),
- an offset into the file (often referred to as the read/write pointer), and,
- a pointer to an *CWinode* structure.

The reason that this information is separate from the `inode` is that many processes can access a given file at the same time. If two processes *open* the same file at the same time, each process is allocated its own entry in the file table, but both file table entries point to the same `inode`. There is a free list of `file` structures analogous to the `inode` free list, so that the kernel can quickly locate unused entries to allocate.

The file descriptor, described in Chapter 1, is simply an index into the user file descriptor table in the process's user block. Each entry in the user file descriptor table is a pointer to an entry in the kernel file table. Figure 2-3 shows the relationship between the user file descriptor table, file table, and inode table. In this example, two of the process's file descriptors refer to different `file` structures, but refer to the same *CWinode* (and hence, the same file). All of the structures in this figure are in kernel virtual address space.

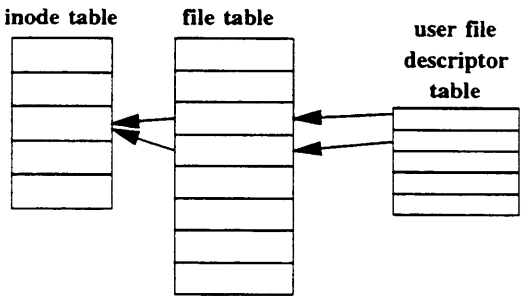

Figure 2-3. File data structures

IRIX uses two principal file systems: the Extent File System (EFS) and the Network File System (NFS). The Extent File System is a disk file system,

and gets its name from the fact that data is stored in variable-length "extents" on disk. The Network File System allows IRIX users to transparently access file systems on remote machines over a network.

2.2.1 The File System Switch

As mentioned in Chapter 1, IRIX supports multiple file system types. For this reason, each `inode` has a field identifying its file system type, a pointer to a separate file system specific data structure, and a pointer to a *file system switch structure*. The file system switch (`fstypsw`) structure contains a set of pointers to file system specific routines. When the kernel has to perform a file system specific operation on an `inode`, such as read file data, it makes an indirect call through the file system switch. The file system switch structures make up an array called the *file system switch table*, which contains one file system switch structure for each type of file system supported. Figure 2-4 shows these data structures for a sample inode.

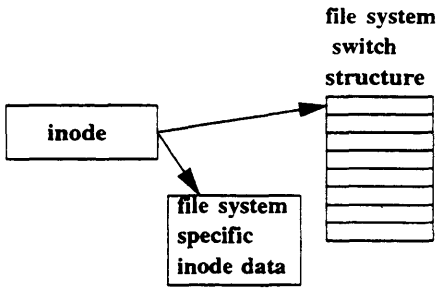

Figure 2-4. File system switch and related data structures

2.2.2 Using Multiple File Systems

So far, we have only discussed the case in which all of a system's files exist on a single file system. In practice, most IRIX systems have a logical file system consisting of several physical file systems. New file systems are grafted on to the file system tree with the *mount*(2) system call. The *mount* system call takes as arguments a device which contains the new file system, and a directory to be "mounted on." After the call returns, references to the

mounted on directory access the root directory of the mounted file system. The original contents of the mounted on directory are hidden until the mounted file system is unmounted with the *umount*(2) system call. Mounted file systems are recorded in a kernel data structure called the *mount table*. A field in the ''mounted-on'' inode contains a pointer to the mount table entry for the mounted file system, and flags are set in the mounted-on inode and the root inode of the mounted file system to indicate their special status.

The first file system to be mounted when the system is initialized is called the *root* file system, since it contains the root directory of the logical file system.

2.3 Memory Management

Memory under IRIX is divided up into 4Kbyte *pages*. The page is the basic unit used by the memory management subsystem. All virtual addresses under IRIX take the form of a *virtual page number* and an offset into the page. A virtual page number represents a logical page, either in a process's virtual address space, or in the kernel's virtual address space. Virtual pages are usually represented by *page descriptor entries*. Each process region contains one or more pages of page descriptor entries, with each page descriptor entry corresponding to a single virtual page within the region.

The memory management subsystem maps virtual page numbers to *physical page numbers*, representing actual locations in memory. The system maintains a pool of physical pages, represented by the *page frame data table*. The page frame data table is an array of `pfdat` structures, each representing a page of physical memory. The maintains several lists of `pfdat` structures which represent available pages—these list are collectively referred to as the *free page pool*.

Virtual pages do not necessarily have physical pages associated with them. For example, the data on a given virtual page can be temporarily written out to disk, or ''paged out,'' in order to reclaim system memory. And the text of a process is only ''paged in'' (read in from the file system) as it is needed. In the case of virtual pages which exist on disk (either because they have been swapped out, or because they contain program text that has not yet been paged in), the page descriptor entry (or `pde`) which describes the virtual page contains the information necessary to locate the page on disk.

2.3.1 Virtual Addresses

IRIX uses four types of virtual addresses, referred to as *kuseg*, *k0seg*, *k1seg*, and *k2seg*. The kuseg addresses are user virtual addresses. A process, running in user mode, can only access user addresses. Furthermore, the interpretation of kuseg addresses is different for each process, so that each process has its own virtual address space. Addresses in k0seg, k1seg, and k2seg are kernel virtual addresses. These addresses are only accessible while a process is in kernel mode.

The different types of kernel virtual address are distinguished by whether or not they utilize the hardware cache(s), and whether they are mapped to physical addresses directly, or through the memory management hardware. k0seg and k1seg addresses are direct mapped, so that a given k0seg address always refers to the same physical address. k2seg addresses are mapped through the memory management hardware, so that a given k2seg address may refer to different physical addresses at different times. k0seg and k1seg addresses actually access the exact same physical address space—the difference being that k0seg addresses use the hardware cache(s), while k1seg addresses do not. Thus, k0seg addresses are used for kernel text and static data structures, so that these may be accessed most efficiently. k1seg addresses are used, for example, for I/O registers, which should not be cached. Since the structures in k0seg and k1seg are static, the pages on which they reside are not part of the system page pool. k2seg addresses are generally used for dynamically allocated kernel data structures. For this purpose, pages are borrowed from the page pool, and assigned k2seg addresses.

All the pages in a process have kuseg virtual addresses, which refer to pages in the system page pool.

2.3.2 Page Faults

Virtual pages which are not resident in memory are brought into memory through *page faults*. When a process attempts to access a page which is not in memory, it receives an exception called a page fault. The page fault handler determines where the page is stored on the file system (by examining the page's `pde`), allocates a page of memory from the free page pool, and reads the data into memory. This kind of page fault is called a *validity fault*. There is another kind of page fault, called a *protection fault*,

which occurs when a process tries to access a page it does not have permission to access.

2.3.3 Integrated Data Cache

As mentioned above, some pages, such as pages of program text, are copies of pages from the file system. Pages that represent file system data are placed on hash lists, so they can be located quickly. If two processes are running the same executable file, they may be using the same set of pages. When one of them accesses a given page for the first time, it is faulted into memory (necessitating a slow I/O operation). If the second process tries to access the same page somewhat later, it may find the page in memory. The page fault handler, mentioned earlier, checks the appropriate hash list to see if a given page is already in memory before it tries to read it from the file system.

2.4 Input/Output Subsystem

Input and output are handled through device drivers. When a process attempts to *read* or *write* a device special file, the appropriate device driver is called to handle the operation. Each device driver handles a particular type of device. Many physical devices may be handled by the same device driver. For example, if a workstation has several hard drives of the same type, they all use the same device driver. If there are two types of disk in use (for example, ESDI and SCSI), each type will have a separate device driver. As mentioned before, device drivers fall into two categories: character and block device drivers.

2.4.1 Kernel-Driver Interface

IRIX can be made to support new types of devices by writing new device driver modules. The driver module must define a set of routines which can be invoked by the kernel to perform various operations on the device. These routines form the *kernel-driver interface*. *Writing Device Drivers for Silicon Graphics Computer Systems* contains details about the kernel-driver interface and information about writing device drivers.

2.4.2 Device Special Files

Device special files are accessed like other files in the file system. Each device special file has a *major device number* and a *minor device number* associated with it. The major device number identifies the type of device (that is, what device driver it uses), and the minor device number identifies the device unit (for example, which disk drive). The kernel maintains character and block *device switch tables*, not unlike the file system switch table. Each device switch table entry contain pointers to the kernel-driver interface routines for a given device driver. These entries are indexed by major device number. So, when a process *open*s a character device, the kernel makes an indirect call through the character device switch table in order to call the device-specific open routine.

2.5 Interprocess Communication Mechanisms

IRIX supports a number of mechanisms for communication between processes. The lowest-level mechanism is the *spinlock*, used for synchronization between processes. The implementation of spinlocks in IRIX is not common to other UNIX systems. IRIX also supports System V UNIX style interprocess communication mechanisms, semaphores, message queues, and shared memory. Semaphores are usually used for synchronization, like spinlocks. Message queues are used to exchange discrete data objects between processes. Shared memory, as the name implies, is a facility which allows two or more processes to access the same area in memory. Finally, IRIX supports BSD UNIX sockets. Sockets may be used to exchange discrete data objects (''datagrams'') among multiple processes, or to create full-duplex communications channels between processes. The IRIX networking facilities are built on top of the socket abstraction.

2.5.1 Spinlocks

Spinlocks are simple test and set locks used for low-level process synchronization. IRIX provides routines to allocate and initialize spinlocks, to lock and unlock them, and to test their status. Spinlocks are ''busy-wait'' locks, meaning that a process will keep trying to acquire the lock

continuously until it succeeds. IRIX also has conditional locking routines, which only lock a spinlock if they can do so immediately.

Multi-processor machines have special hardware to support the spinlock primitive. On other machines, spinlocks are implemented using Dijkstra's algorithm for cooperating sequential processes.[1]

The IRIX kernel uses its own versions of the spinlock routines to insure consistency of kernel data structures in a multiprocessor environment.

2.5.2 Semaphores

Semaphores are synchronization objects similar to spinlocks. However, they are somewhat more flexible and have involve greater overhead. Semaphores are objects with an integer value. The two basic operations used on semaphores are *psema* and *vsema* operations. The *psema* operation decrements the value of the semaphore if it is greater than zero. If it is not greater than zero, the *psema* operation sleeps until it is greater than zero, and then decrements it. The *vsema* operation increments the value of a semaphore. Both psema and vsema are *atomic* operations—they are implemented so that no other process may modify the semaphore while these operations are being performed. (These operations are actually generalizations of the semaphore operations used in the Dijkstra algorithm mentioned above.)

Typically, when a semaphore is used to protect a resource, it is initialized to a value of 1. The first process that tries to use the resource performs a psema operation on the semaphore, reducing its value to zero (the process is said to "acquire" the semaphore). If a second process then attempts to acquire the semaphore, by performing a psema operation, it is put to sleep. When the first process is finished with the resource, it "releases" the semaphore by performing a vsema operation. This increments the value of the semaphore to 1, and wakes up the second process.

Like spinlocks, semaphores are used in the kernel for synchronization, and there are both kernel and user versions of the routines.

---

1. Dijkstra, E. W., "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, Academic Press, New York, 1968.

2.5.3 Message Queues and Shared Memory

Message queues and shared memory both allow processes to exchange data. With message queues, data is exchanged explicitly, through system calls. The *msgsnd* system call is used to send messages, and the *msgrcv* system call is used to receive messages. The message queue bears some resemblance to a pipe, in that it allows processes to exchange data in a sequential fashion. However, unlike pipe data, messages are discrete units. The *msgrcv* call reads exactly one message, unlike the *read* call, which simply reads a specified amount of data. Also, processes can define different "types" of messages, and can request the first message of a given type off the queue.

Shared memory allows processes to exchange data in a more implicit fashion. The shared memory routines allow multiple processes to map the same region of memory into their virtual address space. These processes may then access this memory like any other memory in their address space.

Message queues and shared memory are not used in the kernel.

2.5.4 Sockets

Sockets are data objects representing the endpoints of communications channels. The combine some of the functionality of pipes and message queues, in that they can be used to exchange either unformatted streams of data or discrete messages (sockets which exchange streams of data are referred to as *stream sockets*, while sockets that exchange discrete messages are referred to as *datagram sockets*).

The *socket* system call creates a new socket. The socket descriptor returned by *socket* is identical to a file descriptor. It identifies an entry in the user file descriptor table, which points to a file table entry, which points to an `inode` structure. This `inode` structure has a special file system type which identifies it as a socket inode.

There are two basic ways of sending data through sockets. I/O can be performed using the regular *read* and *write* system calls, or through a set of special-purpose system calls. Stream sockets are "connected" in pairs, forming full-duplex communication channels which may be written to using the *write* system call and read from using the *read* system call, much like a pipe (except that a pipe is a one-way channel). In addition to *read* and

write, there is a special set of system calls for performing I/O on sockets. Some of these system calls are used only on stream sockets, while others can be used for any type of socket.

2.6 Multiprocessor Systems

IRIX is designed to run on machines with more than one CPU. Tasks under IRIX are not delegated by a master processor; rather, each processor runs independently. When a processor is idle, it selects a new process to run from a system-wide list of runnable processes. A process can only be running on one processor at a time, for obvious reasons.

Having multiple processors complicates many of the kernel algorithms. Some protocol must be followed to ensure that only one processor is accessing a given data structure at one time. Different data structures are protected in different ways. Some process data structures are only modified by the process itself, and since the process can only be current on one processor at a time, these structures are only modified in a single threaded fashion. Some code is "bound" to a single processor such that it will only run on on that one processor. Data structures used by this code will only be accessed by that processor. The last and most general way that data structures are protected is through the use of spinlocks and semaphores, as mentioned above. Device drivers may either be bound to run on a single processor, or semaphored for multi-processor operation.

2.6.1 Kernel Locking Protocols

The IRIX kernel uses a large set of locking routines to ensure the consistency of data structures. In general, spinlocks are used to protect resources which will only be held for a very short period of times—for example, a list of free structures might be protected by a spinlock, since processes will only need to hold the lock for long enough to remove a structure from the list. On the other hand, for resources which may be locked for comparatively long periods of time—for example, `inodes`, which may be held for the duration of an I/O operation—a semaphore is used for protection. Processes waiting for a semaphore will be put to sleep, and free up the processor, while processes waiting for a spinlock will monopolize the processor until they succeed.

In addition to the regular locking routines (*psema* and *vsema* for semaphores, and the corresponding *spsema* and *svsema* for spinlocks), IRIX uses a number of other primitives for manipulating spinlocks and semaphores. There are conditional routines, such as *cpsema*, which acquires a semaphore only if this can be done without sleeping. *cpsema*'s complement, *cvsema* increments a semaphore only if it is negative (that is, only if there is a process currently waiting on it).

There are also routines which atomically exchange locks. That is, they release one lock and acquire the other in a single operation. There are routines for exchanging one semaphore for another, one spinlock for another, a spinlock for a semaphore, and so forth. Finally, there are locking routines which allow the process to specify a processor execution level to operate at while holding the lock. These routines are used to bracket particularly critical regions of code, to block out interrupts while the lock is being held.

3. The File Subsystem

This chapter discusses the way files are stored under IRIX. As mentioned previously, the inode is the focus of file activity under IRIX. Each file on an IRIX system is associated with a unique inode. These inodes are file system, or "disk" inodes, and are uniquely identified by a pair of numbers: the inode number and the device number. The inode number is unique within a given file system, and the device number identifies the file system on which the inode resides. Thus, on a system with many file systems mounted, there may be several inodes with a given inode number, but only one inode with a given inode/device number pair. When a file is *open*ed, its file system inode is read into an in-core `inode` structure.[1]

Note that although a file is associated with only one inode, any number of path names may refer to that inode, so that a file may have many names. Each path name which refers to a given file is said to be a "link" to that file.

IRIX is capable of dealing with a number of different file system types transparently. This chapter describes the programmatic interface to the file subsystem, and the kernel routines and data structures used with all file system types. This chapter also covers two file system types: the Extent File System, and the Common File System.

The primary file system type under IRIX is the Extent File System, which provides for file systems stored on disk. The Extent File System (EFS) is so called because it stores files on disk in "extents," variable-sized groups of contiguous disk blocks.

---

1. As in the rest of the book, `courier` type is used to designate data structure names. Thus, "`inode`" refers to the kernel data structure, while "inode" is refers to the abstract unit which the structure represents (thus, for example: "the inode data is read into the in-core `inode`"). Hopefully, this practice clarifies more than it obscures.

The Common File System is sometimes referred to as a pseudo file system type, because it does not provide any permanent storage. Rather, it provides routines to deal with pipes and other unusual "file" objects. It also provides a number of utility routines which may be utilized when implementing a new file system type, provided that the new file system type follows certain conventions.

3.1 File System Interface

The following section discusses the most frequently used system calls for the file subsystem. Each system call is more thoroughly described in the *IRIX Programmer's Reference Manual*, Vol. 1.

3.1.1 Opening a File

The *open*(2) system call is used to gain access to a file. It returns a file descriptor which can be used for future I/O. The user can specify whether *open*ing a non-existent file should cause an error, or simply cause the file to be created. The calling sequence for *open* is:

```
int open(const char *path, int oflag[, int mode]);
```

The `path` argument is the path name of the file to be opened; `oflag` contains a number of bitfields which can be set to indicate how the file should be opened (for reading, writing, or both), whether the file should be created if it does not already exist, and several other options. The optional `mode` argument determines the permission mode for a newly created file, and is therefore only relevant when the "create" flag is set. The return value is a valid file descriptor if the call succeeds, -1 if it fails.

Files can also be created with the *creat*(2) system call.

3.1.2 Reading File Data

The *read*(2) system call is used for reading data from a file. The data is copied into the user's address space at a specified location. The calling sequence for *read* is:

```
int read(int fd, void *buf, unsigned nbyte);
```

The `fd` argument should be a valid file descriptor, such as that returned by *open*. The `nbyte` argument indicates the number of bytes to read, and the `buf` argument indicates the destination for the data to be copied to. Normally, *read* returns the number of bytes actually read. This may not equal `nbyte` if there are fewer than `nbyte` bytes left in the file. Usually a return value of zero indicates that the end of the file has been reached (when reading from a communications channel, such as a pipe, socket, or communications device, a return value of zero may simply indicate that there is no data available at the moment). A return value of -1 indicates an error. The data is usually read from the current position in the file, as defined by the read/write pointer in the `file` structure associated with *fd*. Some devices, however, are incapable of seeking (for example, serial terminals), and in these cases the read/write pointer is ignored.

3.1.3 Writing File Data

The *write*(2) system call is the logical complement of the *read* system call. It copies data from user space to a file. The calling sequence for the *write* system call is identical to that for the *read* system call:

```
int write(int fd, void *buf, unsigned nbyte);
```

write returns the number of bytes written, or -1 if an error is encountered. The data is written beginning at the current position in the file (as defined by the file's read/write pointer) unless the device is incapable of seeking.

3.1.4 Adjusting the Read/Write Pointer

As mentioned above, the read/write pointer determines where I/O will occur in a file. Random access can be achieved by adjusting the read/write pointer with the *lseek*(2) system call. The calling sequence for *lseek* is:

```
off_t lseek(int fd, off_t offset, int whence);
```

The file to be operated on is indicated by its file descriptor, `fd`. The `offset` argument is an offset to be applied to the file, and the `whence` argument indicates how it should be applied. The `whence` argument can be set to one of three symbolic constants: If `whence` is **SEEK_SET**, the pointer is set to `offset` bytes; if `whence` is **SEEK_CUR**, the pointer is set to its current location plus `offset` bytes; if `whence` is **SEEK_END**, the pointer is set to the size of the file plus `offset` bytes. *lseek* returns the new value for the read/write pointer, or -1 if an error is encountered.

3.1.5 Closing a File

Open files can be closed, logically enough, with the *close*(2) system call. The calling sequence for *close* is:

```
int close(int fd);
```

close returns a value of 0 if successful, -1 if an error is encountered.

3.1.6 Making a New Directory

New directories can be created with the *mkdir*(2) system call. The calling sequence for *mkdir* is:

```
int mkdir(const char *path, mode_t mode);
```

Where `path` is the path name for the new directory and `mode` contains the access modes for the new directory. To create a new directory, a process must have write permission for the parent directory. *mkdir* returns a value of 0 if it succeeds, and a value of -1 if it encounters an error.

3.1.7 Removing a Directory

Directories can be removed using the *rmdir*(2) system call. The calling sequence for *rmdir* is:

```
int rmdir(const char *path);
```

Where `path` is the path name for the directory to be removed. The process must have write permission for the parent directory. *rmdir* returns a value of 0 if it succeeds, and a value of -1 if it encounters an error.

3.1.8 Reading Directories

The *getdents*(2) system call allows processes to read directories without having to know the directory format of the file system on which the directory resides. *getdents* behaves much like *read*, except that instead of returning unformatted data, it returns a series of `dirent` structures, each containing information pertaining to a single directory entry (the format of a `dirent` structure is declared in the */usr/include/sys/dirent.h* header file). The calling sequence for *getdents* is:

```
int getdents(int fd, char *buf, unsigned nbyte);
```

The arguments and return value are the same as for the *read* system call.

3.1.9 Changing Current Directory

A process can change its current directory with the *chdir*(2) system call. The calling sequence for *chdir* is:

```
int chdir(const char *path);
```

The `path` argument contains the path name of the new directory. If successful, *chdir* sets the current directory (a variable in the user block) and returns a value of 0. In case of error, *chdir* returns a value of -1.

3.1.10 Changing the Root Directory

In addition to their current directory, processes keep track of their root directory. While this is usually the same as the root directory of the logical file system, a process can change its effective root directory with the *chroot*(2) system call. After making a *chroot* system call, a process cannot access any files above its new root directory. A process must have the effective user ID of the super-user in order to change its root directory. The calling sequence for *chroot* is:

```
int chroot(const char *path);
```

The meaning of the argument and return value are the same as for *chdir*.

3.1.11 Changing File Ownership

The *chown*(2) system call can be used to change the ownership of a file. For a process to *chown* a file, the user id of the process must be the same as the user id of the file's owner, or the user id of the superuser (user id 0). The calling sequence for *chown* is:

```
int chown(const char *path, uid_t uid, gid_t gid);
```

If successful, *chown* changes the user id of the indicated file to `uid` and the group id to `gid`, and returns a value of 0. If unsuccessful, *chown* returns a value of -1. The *fchown*(2) system call is nearly identical to *chown*, except that the file is specified by file descriptor instead of by path name.

3.1.12 Changing File Access Modes

The access modes of a file may be changed with the *chmod* system call. Like *chown*, *chmod* may only be performed by the owner of the file or the superuser. The calling sequence for *chmod* is:

```
int chmod(const char *path, mode_t mode);
```

chmod returns a value of 0 on success, -1 if it encounters an error. The *fchmod*(2) system call is nearly identical to *chmod*, except that the file is specified by file descriptor instead of by path name.

3.1.13 Getting File Status

The *stat* system call can be used to obtain information about a file. A process need not have read permission on a file to get information about it, but it must have search (execute) permission on the directory that contains the file. The information is placed in a special *stat* structure, declared in the */usr/include/sys/stat.h* header file. The information contained in the stat structure includes: the device that the file's parent directory resides on, the file's inode number, access modes, user id, group id, and size, the number of links to the file, and three time values: the last time the file was accessed, the last time the file was modified, and the last time the file's inode was modified. The calling sequence for *stat* is:

```
int stat(const char *path, struct stat *buf);
```

On successful completion, *stat* places the file status information in `buf`, and returns a value of 0. In case of an error, `stat` returns a value of -1. Again, *stat* has a parallel, *fstat*, which takes a file descriptor instead of a path name.

3.1.14 Creating Pipes

Unnamed pipes can be created with the *pipe*(2) system call, as was mentioned in Chapter 2. The calling sequence for *pipe* is:

```
int pipe(int *fds);
```

The `fds` argument should be a pointer to enough space for two integers—usually, it is simply declared as an array containing two integers. When *pipe* completes successfully, `fds[0]` and `fds[1]` are file descriptors for the pipe. `fds[0]` is opened for reading, and `fds[1]` is opened for writing.

3.1.15 Duplicating File Descriptors

The *dup*(2) system call duplicates a file descriptor—that is, a produces a second file descriptor which references the same file structure as the original. Any time a new file descriptor is allocated, it uses the first available slot in the user file descriptor table. This is useful for input

redirection. Take the example of a shell process redirecting the standard input of a child process to come from a file, instead of from the terminal. The process *dup*s file descriptor 0 (standard input), *close*s file descriptor 0, then *open*s a new file. The new file is assigned file descriptor 0, which has just been freed. If the process then forks, the child process (like the parent) will be reading standard input from the file. The parent then *close*s file descriptor 0, and *dup*s the file descriptor that was returned by the original *dup* call, above. Thus the parent gets back its original standard input. The calling sequence for *dup* is:

```
int dup(int fd);
```

dup returns a non-negative number (the new file descriptor) if it succeeds. It returns a value of -1 if it encounters an error.

3.1.16 Linking Files

The *link*(2) system call can be used to add a new "link" to an existing file: that is, to add a new path name which refers to the same file. The calling sequence for *link* is:

```
int link(const char *path1, const char *path2);
```

`path1` is the path name of the existing file to be linked to, and `path2` is the path name of the new link. In order to execute *link*, a process must have execute permission for all directory components of both path names, and have write permission for the directory containing the new link. In addition, both `path1` and `path2` must be on the same file system. *link* returns a value of 0 if successful, or a value of -1 if an error is encountered.

3.1.17 Unlinking Files

The *unlink*(2) system call removes a link to an existing file. If the link being removed is the last link to the file, the file is removed. The calling sequence for *unlink* is:

```
int unlink(const char *path);
```

To *unlink* a file, a process must have write permission on the directory containing the file. *unlink* returns a value of 0 if successful, or a value of -1

if an error is encountered.

3.1.18 Mounting File Systems

The *mount*(2) system call adds a new physical file system to the logical file system. The calling sequence for *mount* is:

```
int mount(const char *spec, const char *dir[, int mflag, int fstyp]);
```

The `spec` argument is the path name of a block device special file, and the `dir` argument is the directory on which to mount the new file system. If the *mount* call succeeds, future references to `dir` access the root directory of the file system on `spec`. The original contents of `dir` become inaccessible until the file system has been unmounted again (with the *umount*(2) system call). *mount* returns a value of 0 if successful, or a value of -1 if it encounters an error.

3.1.19 Unmounting File Systems

The *umount*(2) system call is used to unmount a file system, removing it from the logical file system. The calling sequence for *umount* is:

```
int umount(const char *file);
```

The `file` argument can be either the path name of the device special file containing the file system, or the path name of the directory on which the file system is mounted. *umount* returns a value of 0 if successful, or a value of -1 if an error is encountered.

3.1.20 Creating Symbolic Links

The *symlink*(2) system call creates a symbolic link to a file somewhere in the logical file system. The symbolic link is simply a special file containing a path name to another file. The calling sequence for *symlink* is identical to the sequence for *link*:

```
int symlink(const char *path1, const char *path2);
```

symlink returns a value of 0 if it succeeds, or a value of -1 if an error is

encountered.

3.1.21 Other System Calls

Other system calls which affect the file system include *access*(2), for testing the access modes of a file; *fcntl*(2), for setting flags associated with open files and performing file and record locking; *mknod*(2), for creating named pipes and other special files; *mmap*(2), for mapping pages of files into user address space (covered in chapter 5); and *rename*(2), for moving files.

3.2 File System Data Structures

The following section discusses the major file system data structures, the inode table, the file system switch, and the mount table. Some of the general routines used to manipulate these data structures are also covered.

3.2.1 The Inode Table

When IRIX accesses a file, it reads the file's inode information into an `inode` structure in core memory. The in-core `inode` structures make up the inode table, or *inode pool*, described in Chapter 2. An in-core `inode` must be allocated from the pool for each open file or pipe. In addition, `inodes` must be allocated for each directory which is either the current directory or root directory of some process, for each mount point, and for the root directory of each file system. The in-core `inode` structures have the following fields:

- inode number,
- the device the inode resides on,
- type of file system the inode resides on,
- the type of file (regular file, named pipe, symbolic link, or a character or block device special file),
- the file's permission modes,

- number of links to the file (the number of path names which refer to this inode),
- the owner's user id,
- the owner's group id,
- the number of bytes in the file,
- the last time the file was accessed,
- the last time the file was modified,
- the last time the inode was changed,
- a reference count (indicating how many processes are using the inode),
- Pointers to other `inodes`. The kernel uses these pointers to link the `inode` structures onto hash queues and/or the free list.
- a pointer to the mount-table entry for the file system the inode resides on,
- a pointer to the mount-table entry for the file system that is mounted on the inode,
- a semaphore used for locking the inode,

The complete declaration for the `inode` structure can be found in the */usr/include/sys/inode.h* header file.

Inodes are cached according to a least-recently used algorithm. The kernel maintains a *free list* of in-core `inode` structures which are not in use. The free list is a circular, double-linked list, with its beginning and end marked by the *free list head*. When the kernel needs a free inode structure, it takes one off the head of the free list. When it is finished with an `inode`, it returns it to the tail of the free list. When the system is booted, all the `inode` structures are on the free list.

When the kernel receives a request for a given file system inode, it does not immediately allocate an `inode` from the free list. First, it checks to see whether the file system inode is already represented by an in-core `inode`. To make it easier to find a given `inode` in the in-core inode pool, `inodes` are sorted onto *hash queues*. The hash queues are also double linked circular lists. The kernel maintains a number of these queues, and which queue an `inode` is placed on is determined by a hashing function based on the its device and inode numbers. Thus, when processing a request for a given file system inode, the kernel first determines the value of the

hashing function for that device/inode number pair, and checks the appropriate hash queue. If the inode is not represented on the hash queue, it is not in memory, so the kernel takes a free `inode` from the head of the free list, reads the inode information from disk into the `inode` structure, and places the `inode` on the correct hash queue.

Each `inode` is protected by a semaphore. The `inode` semaphore is only held while a process is in the course of a system call involving that `inode`: `inodes` are never locked across system calls. Likewise, both the free list and the hash list are protected by semaphores. Before a process traverses or manipulates either of these lists, it must acquire the appropriate semaphore. Like the inode semaphores, the hash list and free list semaphores are never held across system calls. They are only held for short periods of time while the process searches or manipulates the list. These practices ensure the consistency of the inode pool.

At any one time, an `inode` is one of four states:

- referenced: there are active references to the `inode` (it represents either an open file, the text file of a running program, a memory mapped file, a current directory, a root directory, or a mounted-on directory). The inode has a reference count greater than zero, is on a hash list, and is not on the free list.
- checked out: the `inode` is being used to represent some entity which does not have a normal file system inode number: an unnamed pipe is an example of one such entity. In this case, the `inode` is not on the free list or the hash list.
- cached: the `inode` is not currently in use, but contains inode data for a file on disk. The `inode` is not locked, and is linked onto both the free list and the hash list.
- free: the `inode` is not currently in use, and does not contain any useful data. All inodes start out in this state, and `inodes` enter this state after being checked out or used by a file system which does not support caching. The inode is unlocked, is linked onto the free list, and is not on any hash list.

Processes must lock `inodes` before operating on them. In all cases, `inodes` are locked when they are being changed from one state to another (for example, when changing a cached inode to a referenced inode). In addition, inodes which are "referenced" or "checked out" are locked by processes performing I/O and other operations on them.

3.2.2 Accessing Inodes

Inodes are commonly accessed through the *iget* routine. The iget routine takes as input a device number and an inode number, and returns a pointer to a locked, in-core `inode` structure. The kernel uses device number and inode number to compute a hash value, and searches the appropriate hash queue. If an `inode` structure representing the requested file system inode is already present on the hash queue (that is, it is cached), the routine locks the `inode` by acquiring its semaphore. Since the it might sleep while acquiring the `inode`'s semaphore, and the `inode` might be re-used in the meantime, the routine must check after locking the `inode` to see if it is still represents the desired device/inode number pair the process is looking for. If not, the routine must release the `inode` and start over again. If, however, the `inode` has not been re-used, *iget* increments its reference count, and returns the `inode` to the calling routine.

```
algorithm iget
inputs: device and inode number
output: locked inode structure

{
        determine which hash list to search
        lock hash list
        if (inode is on hash list)
        {
                unlock hash list
                lock inode
                if (inode is on free list)
                {
                        lock free list
                        remove inode from free list
                        unlock free list
                }
                increment inode's reference count
                return inode
        }

        locate an inode on free list
        lock inode
        remove inode from free list
        if (inode is on a hash list)
                remove from old hash list
        place inode on new hash list
        unlock hash list
        read in inode data from disk
        return inode
}
```

Figure 3-1. Accessing an Inode

If the inode is not represented on the hash queue, the kernel gets an `inode` off the free list and fills in the inode data by reading it in from the file system. This is done by calling the file system dependent *iread* routine.

Once the file system inode has been read into an in-core `inode` structure, the calling routine can access the data in the file through file system dependent routines. In the case of the *open* system call, a pointer to the inode will be stored in a file table entry, and a pointer to the file table entry will be stored in the *open*ing process's user file descriptor table. Future *read* or *write* system calls will follow these pointers back to the inode. `inodes` are never locked across system calls, so an `inode` must be locked near the beginning of any system call that will modify the `inode` or the file it

identifies. The inode must then be unlocked before the system call returns. Since the *iget* routine returns a locked inode, the calling routine must unlock the inode before returning.

3.2.3 Releasing Inodes

The *iput* routine is called to release an inode that is no longer needed. For example, *iput* is called when a process *closes* a file. When releasing an inode the kernel checks the inode's reference count. If the reference count is one (i.e., the releasing process is the only process referencing the inode), then file system dependent code is called to write the inode date out to the file system, and the inode is put on the free list and unlocked. If the reference count is greater than one, the kernel merely decrements the reference count and unlocks the inode.

```
algorithm iput
input: locked inode structure
output: none

{
        if (reference count == 1)
                write out inode
                if (the file system inode resides
                    on does not use inode caching)
                        lock hash list
                        remove inode from hash list
                        unlock hash list
                lock free list
                put inode on free list
                unlock free list

        else
                decrement reference count
        unlock inode
}
```

Figure 3-2. Releasing an inode

Note that some file system types do not use the inode cache. For example, with a remote file system, caching inodes might cause consistency problems. Therefore, when releasing an inode, the *iput* routine checks whether the file system in question uses inode caching. If the file system does use caching, then the inode is left on its hash list, where it may be

located by another process looking for the same file. If the file system does not use caching, the `inode` is removed from its hash list. In this way, the next process that requires the inode will not find it on the hash list, and will be forced to read the inode data in from the file system.

3.2.4 The File Table

The file table may be thought of as an intermediate level of indirection between the user file descriptor table and the inode table. The file table consists of an array of `file` structures, each representing an open file. One `file` structure is allocated each time a process executes a *open*, *creat*, or *pipe* system call. Note that there may be several *open*s on the same file—for example, two processes may independently *open* a file, in which case a separate file structure is allocated for each process. However, more than one process can refer to the same file structure. For example, if a process *fork*s, its child process will inherit its file descriptors, and will therefore refer to the same file table entries. The `file` structure contains the following fields:

- a pointer to an `inode`
- a reference count
- the current offset into the file (also called the read/write pointer)
- file status flags (set by the *open* and *fcntl* system calls)

Allocation of file structures is simplified by the use of a *free list*, which contains all the currently unused file table entries. The entire file table is protected by a spinlock, which processes must acquire before accessing the file table. When a process *open*s a file, it acquires the process table spinlock, takes a `file` structure off the free list, fills it in with the appropriate data, and releases the spinlock. The first free slot in the user file descriptor table (in the process's user block) is then filled in with a pointer to the `file` structure. The *open* system call returns an integer file descriptor, indicating which slot in the user file descriptor table was used.

The reference count in the `file` structure is used to keep track of the number of file descriptors which refer to the `file` structure. When a new file descriptor is allocated to refer to an existing `file` structure, the file structure's reference count is incremented (this happens when a file descriptor is duplicated explicitly, through the *dup* system call, or implicitly,

during the course of a *fork* system call). When a file descriptor is *close*d, the corresponding `file` structure has its reference count decremented. When a `file` structure's reference count reaches zero, the file structure is placed on the free list, and the *iput* routine is called to release the `inode` that the `file` structure referred to.

3.2.5 The Mount Table

The use of multiple physical file systems as a single logical file system is facilitated by the *mount table*. The mount table is an array of `mount` structures, or *mount table entries*, each of which contains information about a single file system. A single mount table entry contains the following information:

- the status of the file system,
- the type of file system,
- the device the file system resides on,
- a pointer to the mounted-on `inode`,
- a pointer to the root `inode` of the file system,
- a pointer to some file-system specific data,
- a semaphore for locking the mount table entry.

In addition to the semaphores in each mount table entry, there is a single semaphore, the mount table lock, used to single-thread mounting and unmounting procedures. When the system is initialized, the mount table entry for the root file system is set up. Since the root file system is not mounted on any other file system, the pointer to the mounted-on inode in its mount table entry is a null pointer.

When *mount*ing a file system, the kernel does several things to ensure consistency. First, it must lock the mounted-on `inode`, so it can be modified. Second, it must acquire the mount table lock, so no other process can modify the mount table at the same time. Then it must lock the mount table entry to be used for the new file system, and finally, the root `inode` of the new file system. The IMOUNT flag is set in the mounted-on `inode`, and the IISROOT flag is set in the root `inode` of the new file system. Each mounted-on `inode` contains a pointer to the mount table entry for the file system that is mounted on it, and every `inode` contains a pointer to the

mount table entry for the file system it resides on. These flags and pointers make it possible for IRIX to resolve path names that traverse mount points. A file system specific mount routine is called during the mounting process, to initialize any special data structures the file system may need. Figure 3-3 depicts the data structures involved with mounting a file system, showing the connections between the mounted-on `inode`, the root `inode` of the new file system, and the mount table.

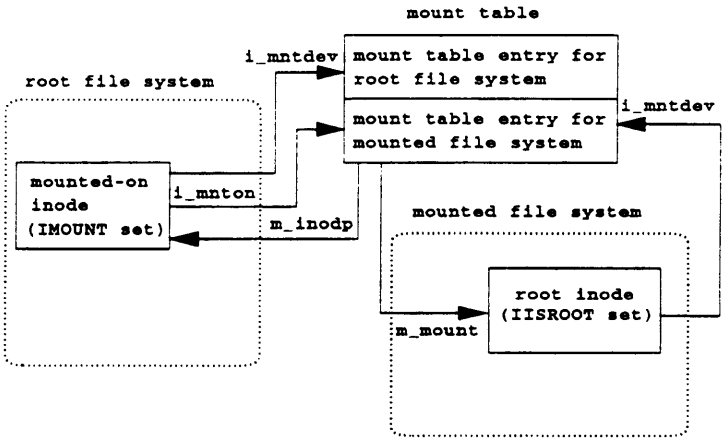

Figure 3-3. Mount table and associated inodes

The procedure for unmounting a file system is complicated by the fact that other processes could be referencing files on the file system. If there are any active references to files on the file system, it may not be unmounted. Furthermore, the unmounting process must make sure that no process tries to reference a file on the file system while it is working. Therefore, after locking the mounted-on `inode`, the mount table and the mount table entry, the process locks the inode free list, preventing any new `inodes` from being allocated, and flushes the inode cache of `inodes` referring to the system by calling the *iflush* routine. *iflush* examines all the `inodes` in the inode table. If it finds cached `inodes` belonging to the file system being unmounted, it uncaches them by removing them from their hash lists. It also readies each `inode` that it uncaches for recycling by deallocating the file system specific portion, and zeroing out the file system type and mount device fields of the `inode`. If it encounters an `inode` belonging to the file system which is currently referenced (besides the root `inode`, which should have exactly one reference—from the mount table), it returns a code indicating that the file system is busy, and *umount* returns an error to the

calling process.

Once the inode cache has been flushed, *umount* sets the MOFFLINE flag in the mount table entry status word and releases the free list lock. Next it locks the root `inode` of the mounted file system, makes sure that any outstanding I/O requests are written to the file system, frees the root `inode`, and writes out the file system data (the superblock and bitmap). Finally, the process clears out the mount table entry, clears the mounted-on flag in the (formerly) mounted-on `inode`, and releases all the remaining locks it holds.

The structure of the mount table entry is declared in the */usr/include/sys/mount.h* header file.

3.2.6 The File System Switch

The in-core `inode` is a "generic" structure which can be used with many types of file systems. So, for example, the Extent File System and the Network File System use the same pool of inode structures. To facilitate the use of different types of file systems, the in-core `inode` has a field indicating the type of file system that the file it represents resides on, a pointer to a structure containing file system dependent inode data (such as the extent descriptors for an EFS inode), and a pointer to a file system switch structure.

The file system switch structure contains function pointers to file system dependent functions (such as *iread*, which is used to read in inode data from the file system). When the kernel has to execute a file system dependent function, it does so indirectly, by calling the correct entry in the file system switch structure that the `inode` points to. The kernel maintains an array of these file system switch structures, one for each type of file system in use, referred to as "the file system switch table," or simply, "the file system switch." The file system switch structure, `fstypsw`, is defined in the */usr/include/sys/conf.h* header file. Some of the routines that are accessed through the file system switch are:

fs_iput Called by *iput* when releasing an `inode`, it synchronizes the disk inode with the in-core `inode`. Called by *iget* when recycling an `inode`, it deallocates the file system specific portion of the `inode`.

fs_iread Reads inode data from the file system into the in-core `inode`.

fs_iupdat The complement of *fs_iread*, this routine writes out the in-core `inode` to the file system.

fs_readi This routine is called when a process performs a *read* system call. It gets the appropriate data and copies it out to an address supplied by the user.

fs_writei This routine is the complement of *fs_readi*. Called when a process performs a *write* system call.

fs_namei This routine is called by the *namei* routine when resolving path names. It tries to find a path name component in a specified directory, and if it can, returns a locked `inode` corresponding to that component. *fs_namei* is also called at the end of the *namei* routine to perform a specified operation—creating, opening, or removing a file.

fs_mount Gets a file system ready for access, initializes its mount table entry and any per-file system data structures it may need.

fs_umount Removes a file system from the mount table, purges references to the file system from system caches, and makes sure all pending write requests are written out to the file system.

fs_openi This routine performs initialization for special files (devices and pipes). It is called when a process *open*s a named pipe or device special file, or when a process executes a *pipe* system call.

fs_closei This routine is the complement of *fs_openi*, and is called when a process closes a device special file or pipe.

fs_update "Update" the file system—for a disk file system, this means to synchronize the on-disk representation of the file system with the in-core data structures.

fs_statfs Returns file system statistics—free space, number of free inodes, and so on.

fs_access Checks whether a process has access to a given file.

fs_getdents Called when the user makes a *getdents* system call. Returns a number of file system independent directory entry structures.

fs_readmap Used by the memory management code when paging data in from the file system.

fs_setattr Used by *chown* and *chmod* to set the attributes of a file.

fs_fcntl Used to pass various *fcntl* flags to the file system.

fs_bmap Map I/O request to file system block number(s).

The file system specific routine corresponding to a given file system switch entry usually has the same name as the file system switch entry, but with a different prefix. For example, the Common File System routine *com_readi* is accessed through the *fs_readi* switch entry. Likewise, the Extent File System version of *fs_bmap* is named *efs_bmap*.

A few words are in order here about the Common File System. The Common File System routines may be used by other file systems—for example, EFS uses some of the common file system routines. In addition, Common File System routines are used for some special "files" such as in-core pipes, which require an `inode` but are not part of any "file system" per se. The next section examines the file system switch routines in more detail, with particular attention to the facilities provided by the Common File System.

3.3 File System Switch Operations

The Common File System provides routines for dealing with special files, such as devices, and pipes. The Common File System also provides utility routines which may be used by other file system types. The Extent File System, for example, relies upon the Common File System routines *com_readi*, *com_writei*, and *com_namei*. The following section covers the functions of some of the more important file system switch routines, with special attention to the Common File System routines and their interaction with other file system specific routines.

3.3.1 Reading regular file data

As noted above, *read* requests on regular files end up calling the appropriate *readi* routine through the file system switch table. The *readi* routine is passed a pointer to a locked, in-core `inode` structure. The arguments to the *read* system call are placed in the u-block, as follows:

`u_base`	destination address
`u_offset`	byte offset in file
`u_count`	number of bytes to read

So, for example, if a read request is done on a file on an EFS file system with the call:

```
read(fd, buf, 128);
```

The kernel copies the value of `buf` (a pointer into the process's virtual address space) into `u_base` and sets `u_count` equal to 128. It uses the file descriptor to locate the file table entry, and from the file table entry it finds the `inode` pointer and tries to lock the `inode` by acquiring its semaphore. When it has locked the `inode`, it copies the current offset in the file into `u_offset`. (It must copy the offset *after* locking the `inode`, because the process will sleep if it cannot acquire the semaphore at once, and another process using the same file table entry might modify the offset while the first process is asleep.) At this point, the file system specific *readi* routine is called.

As mentioned above, EFS actually uses the Common File System routine, *com_readi*. When dealing with a regular file, *com_readi* calls a file system specific routine *bmap*, to map the read request to physical block numbers on the device. The *bmap* routine is passed the `inode` pointer, a flag indicating whether it is reading or writing, and two `bmapval` structures (defined in */usr/include/sys/fstyp.h*). The first `bmapval` structure is filled in with the physical location of the requested data on the device (we will deal with the second `bmapval` structure in a moment). Both `bmapval` structures are then passed to the *chunkread* routine, which looks for the requested data in the integrated data cache, reading it in from disk if it is not in the data cache. The second `bmapval` structure contains the physical location of the logical block(s) immediately following the block(s) requested. This allows the *chunkread* routine to perform "read ahead" by starting an asynchronous read on these blocks, thus pre-loading the buffer

cache with the blocks that the process is likely to want next. Obviously, read ahead is only valuable if the process is accessing a file sequentially. The *chunkread* routine returns a buffer header, and the *com_readi* routine copies the appropriate data from the buffer to the location specified in the *read* call. For large *read* requests, *com_readi* may have to repeat the *bmap* - *chunkread* - copy out process many times to satisfy the request.

3.3.2 Writing Regular File Data

The procedure for writing regular file data, *com_writei*, is similar to the procedure for reading. First, the file system specific *bmap* routine is called to map the request. If a process tries to write out a logical block of the file for which no physical storage has been allocated, the *bmap* routine is responsible for allocating storage space.

After calling *bmap*, the *com_writei* passes the two `bmapval` structures to the *getchunk* routine. The *getchunk* routine searches for the appropriate buffer in the integrated data cache. If the buffer is found in the cache, *getchunk* returns it. If the buffer is not in the cache, *getchunk* simply returns a free buffer. *com_writei* then determines whether the write request will completely overwrite the buffer. If so, there is no need to worry about the data currently in the buffer. But if not, *com_writei* must call the *pread* routine to read the data in from the file system. (*pread* does not always need to read the data in. If the buffer was found in the cache, it probably contains valid data. In this case it is marked as valid, and *pread* simply returns. If the buffer was allocated off the free list, however, it will be marked as invalid, and *pread* will request the data from the I/O subsystem, and wait for the request to finish before returning the buffer.) The mechanics of the integrated data cache are covered in Chapter 5.

Once *com_writei* has the appropriate buffer, it copies the data from user address space into the buffer. It then calls either *pdwrite*, or *pwrite* to write the buffer. *pdwrite*, which is called most of the time, is a ''delayed write''—it simply marks the buffer so that it will be written at some future time, and releases it. *pwrite* actually passes the buffer to the I/O subsystem, and waits for the I/O to complete before returning. *pwrite* is only called if the file's synchronous write flag is set (this flag may be set either when the file is *open*ed, or through the *fcntl* system call).

3.3.3 Reading Directories

Each file system has its own directory format, however, for most UNIX type file systems reading a directory works much like reading a regular file. The main difference is that *com_readi* uses a different set of routines to interface with the integrated data cache. This is because the integrated data cache divides data up into two categories: regular file data, and everything else. These types of data are cached differently. The difference between the caching mechanisms is covered in Chapter 5.

A process could simply read a directory using the *read* system call. However, since different file system types may have different directory formats, IRIX provides a consistent directory interface through the *getdents*(2) system call. The *getdents* system call is much like the *read* system call, except that it fills the user's buffer with a number of `dirent` structures, each containing information about a single directory entry. This service is provided by the file system specific *getdents* routine.

3.3.4 Writing Directories

The actual writing of directory data works much like the writing of regular file data, except that it uses different routines to interact with the integrated data cache.

However, it is not practical to use the *write* system call to write directory entries. Instead, there are several system calls for manipulating directories. *creat*, *mkdir*, *mknod*, *link open*, *unlink*, *rmdir*, and *rename* all manipulate directory entries. All of these system calls end up calling the *namei* routine, described below.

3.3.5 Path Name Lookup

The *namei* routine is used to locate the file corresponding to a path name. *namei* is utilized by all system calls that specify a path name—for example, *chdir*, *open*, and *link*. By default, *namei* finds an existing file and returns a locked `inode` structure, representing it. To specify behavior other than the default, *namei* can be passed an `argnamei` structure, filled in with a command flag and appropriate arguments. The possible values for the command flag, and the functions they specify, are listed below:

Flag value	Function
NI_DEL	unlink specified file
NI_CREAT	open file, creating it if it doesn't exist
NI_XCREAT	create this file, return error if it already exists
NI_LINK	make a link
NI_MKDIR	make a directory
NI_RMDIR	remove a directory
NI_MKNOD	make a special file
NI_SYMLINK	make a symbolic link
NI_RENAME	rename a file

For commands which involve two files, such as NI_LINK or NI_RENAME, the source file (the file being renamed or linked to) is specified in the `argnamei` structure. For files being created, access modes, ownership and (for device special files) device number are specified in the `argnamei` structure. The `argnamei` structure is declared in the */usr/include/sys/nami.h* header file.

namei keeps the path name it is trying to resolve in a buffer. It breaks the path name up into components, or individual directory entries. It maintains pointers to the current component, and the next component in the path name. *namei* also keeps track of its current directory (not to be confused with the *process's* current directory) as it works its way through the path name. *namei* begins its search with a directory inode. If the path name is relative, the search begins at the process's current directory, and if the path name is absolute, the search begins at the root directory (the process's user block contains pointers to the in-core inodes representing the current directory and the root directory, so these need not be looked up). This directory is *namei*'s current directory. *namei* then iteratively carries out the following steps for each non-terminal component in the path name: first, it checks the current directory for access permission. It passes the current directory's inode, the current path name component, and the `argnamei` structure to the file system specific *fs_namei* routine. The *fs_namei* routine tries to find the inode matching the component, and passes back a success code and the inode for the component, if it was found. This new inode should represent a directory, since it is a non-terminal component of the path name. If it is not a directory, *namei* returns an error. If it is a directory, *namei* makes it the current directory and proceeds to look up the next path name component.

In this way *namei* locates the parent directory of the target file, and calls *fs_namei* one final time, with an argument specifying the requested operation—creating a new link, removing an existing link, or simply locking the inode. The *fs_namei* routine takes care of manipulating the directory entries, and allocating an inode for newly created files (an example of how this is done under EFS is presented in section 3.4.7, "File Creation").

There are, however, several complications that *namei* has to deal with. It has to deal with mount points, and it has to deal with symbolic links. When *namei* encounters a "mounted-on" `inode`, it follows a pointer in the `inode` structure to the mount-table entry for the mounted file system. This mount-table entry has a pointer to the root `inode` of the mounted file system, and *namei* makes this `inode` the starting directory for its next call to *fs_namei*. Similarly, *namei* must be able to traverse a mount point in the other direction. For example, if the current directory, */usr* is the root directory of a mounted file system, "../etc" should be interpreted as the */etc* directory of the root file system. Therefore, *namei* accords special treatment to the ".." path name component. If *namei*'s working directory is the process's root directory, then the ".." component is ignored; if *namei*'s working directory is the root directory of a mounted file system, then *namei* can follow the pointer to the mount table entry, and from there back to the parent of the mounted-on *inode*. (An *inode* representing the root directory of a file system will have the IISROOT flag set. If the file system is a mounted file system, as opposed to the root file system, its mount table entry will have a pointer to the mounted-on `inode`).

Symbolic links present another problem for *namei*. When a symbolic link is encountered, it is opened, and the path name it contains is read into the path name buffer in front of the next component of the path name. If the new path name starts with a "/," (that is, it is an absolute path name), *namei* changes its current directory to the process's root directory—otherwise, *namei*'s current directory remains unchanged. *namei* then begins resolving the new path name, one component at a time. To guard against endless loops, *namei* keeps track of the number of symbolic links it has encountered, and if this number exceeds a configurable maximum (normally 8), *namei* gives up and returns an error.

File system types which utilize the Common File System's *namei* routine must support several extra file system switch routines for examining and modifying directories. The advantage of using the *com_namei* routine is that the Common File System maintains a *path name component cache* to

speed up path name conversion. This cache works much like other caches maintained by the kernel. The kernel has a pool of `ncblock` structures, each linked onto a least-recently-used (LRU) list and one of a number of hash lists. Each `ncblock` structure contains information on a single path name component, which may be uniquely identified by three pieces of information: the device on which its parent directory resides, the inode number of its parent directory, and its name. The hashing function which determines which hash list an `ncblock` structure will be placed on is based on these three pieces of information. Each time the Common File System searches for a path name component, it searches the cache first. If the path name component is found in the cache, it is moved to the tail of the LRU list. If the path name component is not in the cache, a file system specific lookup routine is called. If this routine finds the component, an `ncblock` structure is taken off the head of the LRU list, filled in with the information for the new component, and placed on the tail of the LRU list. The `ncblock` structure is also moved to a new hash list, unless its new hash value happens to be the same as its previous one. Since cache operations take little time, the entire cache is protected by a single spinlock. The cache routines acquire the lock before searching or modifying the cache, thus ensuring cache consistency.

3.3.6 Pipes

The Common File System code is used for both named pipes and unnamed pipes. When a *pipe* system call is made, an `inode` is allocated off the free list (''checked out''). A `pipe_inode` structure is allocated as the file system specific portion of the `inode`. This structure contains a pointer to a 10 Kbyte buffer used for storing data written to the pipe, a read pointer and a write pointer, and four semaphores. Two file structures are allocated for the two ends of the pipe. Both `file` structures point to the `inode` representing the pipe, and each `file` structure is in turn pointed to by a new entry in the user's file descriptor table. Figure 3-4 shows the data structures associated with a pipe created by the *pipe* system call.

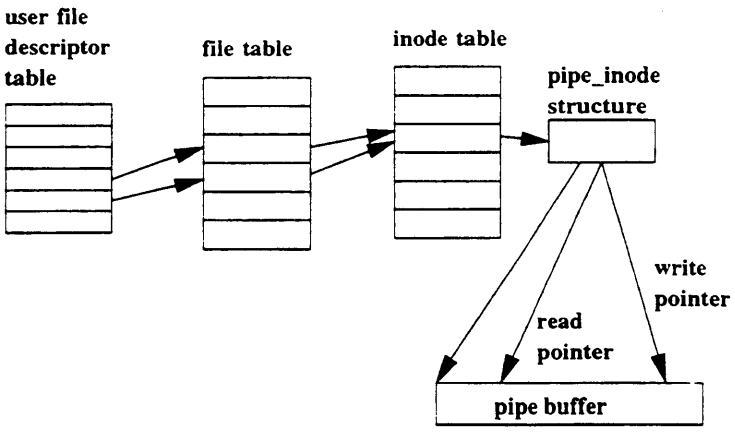

Figure 3-4. Data structures associated with a pipe

Two of the semaphores in the `pipe_inode` structure are used to keep track of the number of processes which have the pipe open for reading and writing. These semaphores are called the *reader count* and the *writer count*. When a pipe is created with the *pipe* system call, both of these semaphores are incremented (with a *vsema* operation). The other two semaphores in the `pipe_inode` are the *empty* semaphore and the *full* semaphore. These semaphores are used by routines waiting to read data, and waiting to write data, respectively.

When a named pipe is *open*ed, it is treated much like any other file. The path name is passed to the *namei* routine, which eventually calls *iget* to access the inode. If the inode is already in core, *iget* simply locks it, increments its reference count and returns it. If it is not in core, *iget* allocates an `inode` structure off the free list, and calls the file system specific *iread* routine to fill in the inode data. The *iread* routine gets various information from the file system (such as the file type, access modes, and ownership), then, finding that the file is a pipe, calls a Common File System routine to allocate the `pipe_inode` structure, as above.

The buffer used by the pipe is not allocated until the first read or write request on the pipe. The buffer is used in a "circular" fashion: as data is written to the buffer, the write pointer is incremented by the number of bytes written, and when the write pointer reaches the end of the buffer, it wraps

around to the beginning of the buffer. The write pointer may not, however, wrap around past the read pointer—if enough data is written to the pipe that the write pointer would wrap past the read pointer, the pipe is full, and the writing process will block until the pipe empties (in some cases, mentioned below, a write on a full pipe will return immediately, either with an error or with a short write count). As data is read, the read pointer is advanced by the number of bytes read, but the read pointer is not allowed to pass the write pointer. When the read pointer is equal to the write pointer, the pipe is empty. When the pipe is initialized, the read and write pointers are both zero.

The behavior of *read* and *write* system calls on pipes depends upon the file status flags (set by the *open* or *fcntl* system calls). Normally, if process does a read call on an empty pipe, it will simply wait until there is data available by waiting on the empty semaphore. The process must unlock the `inode` at the same time (otherwise no process would be able to lock the `inode` to write to it), so the atomic *vpsema*(K) operation is used to release the `inode` semaphore and perform a *psema* operation on the empty semaphore. When the *vpsema* operation returns, the pipe is no longer empty, so the process relocks the `inode` and reads the data. The process will not wait if one of the following is true: there are no writers on the pipe, or if the O_NONBLOCK or O_NDELAY flags is set in the `file` structure. If there are writers on the pipe, but the O_NONBLOCK flag is set, the *read* call will return an error. Otherwise, it will simply return a short read (that is, it will return fewer bytes than requested, possibly none). The *write* system call behaves similarly, blocking when the pipe is full. However, if a process attempts to *write* to a pipe which has no readers, the writing process will receiving a SIGPIPE signal and the *write* call will return an error.

The `pipe_inode` structure is declared in the */usr/include/sys/fs/pipe_inode.h* header file.

3.4 The Extent File System

An Extent File System file consists of an on-disk inode, and zero or more "extents"—variable length, contiguous groups of disk blocks. An instance of the Extent File System on disk consists of the following components:

- the superblock, which holds important information about the file system,

- the bitmap, which is used to keep track of which disk blocks in the file system are in use,
- a variable number of cylinder groups, each consisting of a set number of inode blocks followed by a set number of data blocks.

In addition to these components, the file system contains a variable amount of unused space, for alignment reasons, and a copy of the superblock, in case the original should become corrupted.

The layout of a regular file system is pictured below:

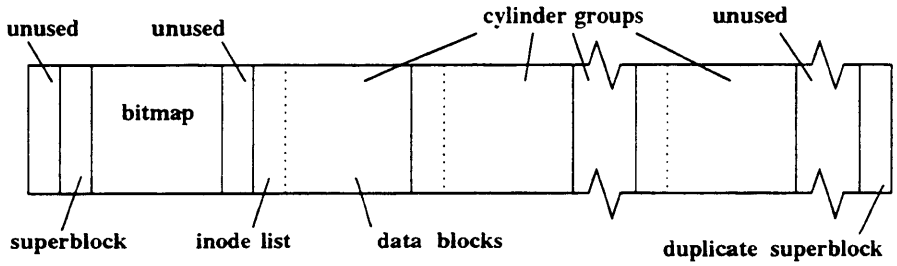

Figure 3-5. Layout of a regular file system

The first block, or "boot block" is historically left free for system-boot procedures. The second block of the file system is the superblock. This is followed by the bitmap, which contains one bit for each data block in the file system. After the bitmap there is (usually) some unused space, followed by the cylinder groups. After the last cylinder group, there is usually more alignment space, followed by the replicated superblock, which is the last block of the file system.

3.4.1 The Superblock

As mentioned above, the superblock contains important information about the file system. When a file system is mounted, the superblock is read into memory. The following information is read in from disk:

- the size of the file system (in blocks),
- the offset of the first cylinder group (in blocks),
- the number of blocks in a cylinder group,

- the number of blocks devoted to inodes in a cylinder group,
- a flag indicating whether the file system is ''dirty'' or ''clean,''
- the last time the superblock was written to disk,
- the name of the file system,
- the size of the bitmap in bytes,
- the total number of free disk blocks,
- the total number of free inodes,
- the offset to the beginning of the bitmap (in blocks?),
- the location of the replicated superblock.

In addition to this information, the superblock contains a number of fields which are initialized at mount time. For example, there are flags indicating whether the file system is mounted read-only, and whether the superblock has been modified since it was last written out to disk. When an EFS file system is mounted, the superblock is read into memory, these flags are initialized, and memory is dynamically allocated for a *cylinder group summary table*, which maintains a count of the number of free inodes and data block in a given cylinder group. The cylinder group summary table is built at mount time, and is kept up to date by routines which allocate and deallocate resources. The structure of the superblock and the cylinder group summary structures is declared in the */usr/include/sys/fs/efs_sb.h* header file.

3.4.2 The Bitmap

The bitmap is used to find free data blocks for allocation. Each bit in the bitmap represents a block in some cylinder group (therefore, some of the bits in the bitmap represent blocks used for inodes, and these bits are never consulted). The bits representing unallocated blocks are set. When searching for a chunk of free space, the kernel can call a bitmap routine, which will search the bitmap for a series of set bits of the appropriate length. Rather than keep the whole bitmap in memory at all time, IRIX accesses it through the buffer cache, using the *bread* routine.

3.4.3 On-Disk Inodes

The on-disk EFS inode contains the following information:

- the type of file
- the file's permission modes,
- number of links to the file
- the owner's uid,
- the owner's gid,
- the number of bytes in the file,
- the last time the file was accessed,
- the last time the file was modified,
- the last time the inode was changed,
- the number of extents,
- space for twelve extent descriptors, or (for device special files) the major and minor device numbers.

Except for the information about the file's extents, the information in an EFS disk inode is a subset of the information in an in-core inode.

The EFS inode as it appears on disk is exactly 128 bytes long. Disk blocks, or "basic blocks" as they are sometimes called, are 512 bytes long, so 4 inodes can be stored in a single disk block. The structure of an EFS disk inode is declared in the */usr/include/sys/fs/efs_ino.h* header file.

3.4.4 Regular file structure

As has been mentioned, IRIX files are viewed as unformatted streams of bytes. In the case of a regular file, these bytes are stored on disk. In the Extent File System, a regular file's data is stored in one or more "extents," variable-sized groups of contiguous disk blocks. The EFS inode has space for 12 extent descriptors, each containing the following information:

- offset in the file system of the first block in the extent,

- the number of blocks in the extent,
- the logical offset into the file at which the extent starts.

Each extent can be up to 248 blocks long, so most files can be contained in 12 extents. However, if more than 12 extents are needed, the kernel will allocate a new extent, copy all of the inode's extent descriptors into this new, "indirect" extent, and change the inode's first extent descriptor to point to the indirect extent. To extend the file, more extent descriptors can be stored in the indirect extent. If the indirect extent fills up, a second indirect extent can be allocated, and pointed to by the inode's second extent descriptor. In this manner very large files can be addressed.

When an EFS inode is read in (by the *efs_iread* routine), all of the extent descriptors are read into memory as part of the file system dependent inode data. (The EFS version of the file system dependent inode data structure is declared in the */usr/include/sys/fs/efs_inode.h* header file.) If the inode has any indirect extents, the kernel dynamically allocates enough memory to hold all the inode's extent descriptors and reads in the indirect extents. To locate a given block in the file, the *efs_bmap* routine does a binary search through the array of extent descriptors. Note that a binary search requires that all logical blocks in the file be included in one of the extents, which means that they must be represented by actual disk blocks. Thus, IRIX does not support "holes" in files.

3.4.5 EFS Directory Structure

Basically, a directory is a file whose data consists of a number of filename/inode number pairs describing each file in the directory. In practice, it is somewhat more complicated because of the use of variable-length names. The first four bytes of each directory block are occupied by a header, which contains the following information:

- a magic number to identify the block as a directory block (two bytes)
- an offset to the beginning of the first used directory entry (one byte)
- the number of slots in the directory (one byte)

After the header are a series of one-byte offsets, each indicating the beginning of a variable-length directory entry. Note that since it actually takes nine bits to represent all the possible offsets in a 512 byte directory block (9^2=512), the offsets are compacted by shifting them right one bit,

eliminating the least-significant bit. Because of the way the offsets are stored, each directory entry must begin at an even byte offset. An offset with a value of zero is an unused slot, and may be reused for a new directory entry. The offset to the first used directory entry in the directory block header is also stored in the compacted format.

The directory entries contain the following information:

- the inode number (four bytes)
- the length of the filename (one byte)
- the filename (from one to 255 bytes)

Directory entries are added at the end of the directory block, so that the directory entries grow backward while the offsets grow forward. When a new directory entry is added, the kernel first makes sure that there is enough space in the directory block. It then searches through the offsets, looking for an unused one. If there are no unused offsets, a new offset is allocated after the last one currently in use, and the number of slots in the directory block header is incremented. The offset of the new directory entry is calculated by taking the offset of the first used directory entry (from the block header) and subtracting the length of the new directory entry (if necessary, an extra byte is subtracted to locate the new directory entry at an even offset). The first used offset and the offset allocated for the new directory entry are then updated to point to the new directory entry's offset, and the directory entry itself is written.

If there is not enough room in the current block for a new directory entry, the kernel tries again in the next directory block, and so on until it has checked every block in the directory. If the kernel reaches the last block in the directory without finding space for the new entry, it will attempt to allocate new blocks for the directory in the same way it would for a regular file, first attempting to expand the current extent, then attempting to allocate a new extent.

When a directory entry is removed, the kernel zeros out the offset corresponding to the directory entry and compacts the remaining directory entries in the block. If the directory entry is the first used slot in the block, then the first used offset is merely set to start of the next directory entry, and the old directory entry is zeroed out. If the removed entry is not in the first used slot, then all of the directory entries before the removed entry are moved up to fill in its slot, and the offsets and the first used slot offset are updated accordingly.

The directory block and directory entry structures for EFS are declared in the */usr/include/sys/fs/efs_dir.h* header file.

3.4.6 Disk Block Allocation

When a file is extended beyond the last logical block for which there is a disk block allocated in some extent, IRIX must allocate more disk space to hold the new data. This occurs when the *efs_bmap* routine is called upon to map a write request to a logical block for which no corresponding disk block has been allocated. The first thing that the kernel must do is to acquire the file system's semaphore to prevent other processes from allocating blocks at the same time. Next, the kernel attempts to enlarge the current extent by allocating a set of adjacent disk blocks. It checks the status of these blocks by consulting the bitmap. Each basic block in the file system is represented by a bit in the bitmap. If the bit corresponding to a given basic block is set, the block is unallocated.

If it proves impossible to enlarge the current extent, the file system will attempt to allocate a new extent. There are three reasons the attempt to enlarge the current extent might fail:

1. the disk blocks immediately following the extent are already allocated to another extent,
2. the extent is at the end of the cylinder group, or
3. the extent has reached the maximum allowable size.

When IRIX allocates a new extent, it tries to allocate one large enough to hold all the data being written, and preferably locate it in the same cylinder group as the last extent in the file, to keep the file's data close together. If the extent being allocated is the first extent in the file, IRIX attempts to place it in the same cylinder group that the file's inode is located in.

3.4.7 File Creation

When a new file is created (with the *open*, *creat*, *mkdir*, or *mknod* system calls), an inode must be allocated to represent it. As mentioned above, all of these calls utilize the *namei* routine. When the *namei* routine has located the parent directory of the file to be created, it calls the file system specific *namei* routine once more to perform the requested operation. In the case of

EFS, the *com_namei* routine is used. *com_namei* it checks to see whether there is already an entry for the new file in the parent directory. If so, the inode for the file is simply returned. Otherwise, the file system specific *ialloc* routine is called to allocate a new inode.

When called to allocate a new inode, *efs_ialloc* first acquires the file system semaphore, then searches for an unallocated inode by scanning the inode portions of the file system's cylinder groups. It begins by trying to place the new inode in the same cylinder group as the parent directory's inode. If there are no free inodes in this cylinder group, *efs_ialloc* will try to find another suitable cylinder group to place the inode in. Once it has found a cylinder group, *efs_ialloc* reads inodes off of the disk, one buffer full at a time, using *bread*, and tries to find an unallocated inode (one with a file type of zero). *efs_ialloc* does not use *iget* to scan for free inodes so as to avoid flushing the inode cache. However, once a free inode has been found, it must be read into an in-core `inode` structure using *iget*. Before this can be done, the routine must release the buffer containing the inode, so *iget* can access it, and release the file system semaphore to prevent deadlocking.

When *iget* returns the `inode`, *efs_ialloc* re-acquires the file system semaphore, and checks if the file type of the new inode is still zero, to make sure that no other process has allocated it in the meantime. If the inode has been allocated out from under the process, it starts over again searching for a free inode.

Once an EFS inode and its corresponding in-core `inode` structure have been allocated, *efs_ialloc* initializes the `inode` with the correct ownership information, then returns the `inode` to the calling routine, which enters the inode number into the parent directory. Eventually, control returns to the system call routine. In the case of *open* and *creat*, the calling routine allocates a `file` structure for the new file, and returns a file descriptor for it. In the case of *mknod* or *mkdir*, the system call releases the `inode` and simply returns a success code.

The way that IRIX chooses a suitable cylinder group when allocating an inode depends upon the type of file being created. For a regular file, symbolic link, or directory, it tries to select a cylinder group with a certain number of free data blocks, so that extents may be allocated for the file in the same cylinder group as the inode. When selecting a cylinder group, IRIX checks for free inodes and data blocks by consulting the cylinder group summary table. For other types of files, IRIX only looks for a cylinder group with a free inode. Regardless of what type of file is being created, *efs_ialloc* checks cylinder groups starting with those closest to the cylinder group that

the parent directory's inode was in, and moving outward in either direction, towards higher and lower cylinder group numbers. If no cylinder group can be found that meets the desired criteria, IRIX will select the first cylinder group with a free inode.

3.5 Chapter Summary

The IRIX file subsystem supports multiple physical file systems, of different file system types, and gives them the appearance of a single logical file system with a hierarchical arrangement. File operations are focussed on the inode table, which contains entries for all active files. The integration of multiple physical file systems is made possible by mount table entries, each of which is connected to the inode table through pointers to the mounted-on `inode` and the root `inode` of the mounted file system. Each `inode` also contains a pointer to the mount table entry for the file system which it resides on, and each mounted-on `inode` also contains a pointer to the mount table entry for the file system which is mounted on it.

The file system switch, which allows indirect function calls to file system specific routines, makes it possible to have multiple file system types operate transparently to the user. Each in-core `inode` has a pointer to the appropriate file system switch entry, which in turn has pointers to file system specific routines, allowing routines to perform file system specific operations on the `inode` without having to know what type of file system the `inode` belongs to.

namei, which might be called the workhorse routine of the file system, converts path names to inodes, opens files, creates files, deletes files, and manipulates directories.

The native file system, EFS, provides storage and retrieval of data by storing data in contiguous blocks, and by trying to maintain locality between the inode and its data extents, and between parent directories and their children. Allocation of resources is sped up by a bitmap of allocated data blocks, and by cylinder group summary structures which track the number of free data blocks and inodes in each cylinder group.

4. The Process Subsystem

The IRIX process represents a ''thread'' of execution. This abstract entity is defined by a collection of data. The virtual address space of a process, the contents of its user structure and proc table entry, and the values contained in machine registers when the process is running all constitute the *context* of the process.

In order to support multiple processes, IRIX implements a process scheduling algorithm which assures a fairly equitable division of processor time between all processes. This algorithm is said to be non-preemptive. That is, the running process can not be preempted by another process (although it can be preempted by the kernel). The running process may yield to another process ''voluntarily,'' by making a system call which will cause it to sleep (such as an I/O request), in which case another process will be selected to run, or the running process may be preempted by the kernel in order to handle an exception, in which case execution will return to the process after the exception handler has finished its business. The kernel also enforces a limit on the amount of time a process can monopolize the processor. When this specified time has elapsed, an exception is generated, and the exception handler selects a new process to run and executes a *context switch.*

4.1 Process System Calls

4.1.1 Creating New Processes

The *fork* system call is used to create new processes. The calling sequence for *fork* is:

```
int fork(void);
```

fork takes no arguments. If it succeeds, *fork* returns the child's process ID to the parent. The child process begins execution as if it were returning from the *fork* call, but receives a return value of zero. If an error is encountered, the *fork* call returns a value of -1 to the parent process (and no child is created).

4.1.2 Executing Programs

There are a number of system calls for executing a program, but they are all fundamentally similar. The calls overlay the text of the calling process with the text of a new program. The calling sequences for two representative system calls, *execl* and *execv*, are given below.

```
int execl(char *path, *arg0, arg1, ..., argn, (char *)0)
int execv(char *path, *argv[]);
```

execl and *execv* both start a new program, identified by its path name (`path`). They differ in the way they handle the program's arguments. *execl* takes a variable number of arguments, `arg0` to `argn`, each of which is used as a single argument to the new program. A null pointer argument indicates the end of the variable-length argument list. *execv* takes two arguments, `path` and `argv`, the latter being an array of arguments. An unsuccessful *exec* call will return a value of -1. A successful *exec* call will not return at all, since the text of the calling process is replaced by the text of the new process.

4.1.3 Resizing the Data Region

The *brk* system call is used to change the size of a process's data region. This is done by adjusting the *break value*, which is the address of the first location beyond the end of the data region. The calling sequence for *brk* is:

```
int brk(void *endds)
```

The `endds` argument specifies the new break value for the process. *brk* returns a value of 0 if successful, or a value of -1 if an error is encountered.

4.1.4 Sending Signals

The *kill* system call is used to send signals to processes. The calling sequence for *kill* is:

```
int kill(pid_t pid, int sig)
```

Where `pid` is the process ID of the process that the signal is intended for, and `sig` is identifies the signal number to be sent (symbolic constants are defined for the various signals, for example SIGCLD, death of child process, is sent to a parent process when its child dies, and SIGSEGV, segmentation violation, is sent to a process when it tries to access an illegal memory location). *kill* returns a value of 0 if successful, or a value of -1 in an error is encountered.

4.1.5 Catching Signals

The *signal* system call is used to specify a process's behavior when receiving a signal. A program can specify a a function to be called when a specific signal is received. The calling sequence for *signal* is:

```
void (*signal(int sig, void (*func)(int, ...)))(int, ...);
```

The `sig` argument specifies a signal, and the `func` argument is a pointer to the signal-handling function. Two symbolic constant "functions" are defined, and may be used as the second argument to *signal*. SIG_IGN causes IRIX to ignore the specified signal, and SIG_DFL restores the default behavior (terminating when a signal is received). If successful, *signal* returns the previous value of `func`. If an error is encountered, *signal* returns a value of -1.

4.1.6 Terminating a Process

The *exit* system call terminates the calling process. When the *exit* system call finishes, the process is a "zombie" process, no longer active, but still taking up a slot in the process table. The zombie child remains in the process table until the parent "reaps" it by invoking the *wait* system call. For obvious reasons, the *exit* system call never returns. When the system call finishes, the processor tries to schedule a new process.

```
void exit(int status)
```

The `status` argument specifies the value to be returned to the parent process through the *wait* system call, below.

4.1.7 Waiting for a Process to Terminate

The *wait* system call is used to collect information about a terminated child process. If the *wait*ing process already has a zombie child, then the *wait* system call returns immediately. If the process does not have any zombie children, then the *wait* system call will block until a child process exits. The *wait* system call will return before a child exits if the *wait*ing process receives a signal.

```
int wait(int *statptr)
```

The `statptr` argument can be used to obtain information about the terminated child process. If `statptr` is a valid integer pointer, *wait* will store status information in the location pointed to by `statptr`. This information includes the reason that the child was terminated, and if the child process terminated itself by calling the *exit* system call, the status value that the child passed to *exit*. The *wait* system call returns the process ID of the terminated child process. If *wait* is interrupted by a signal, or the `statptr` argument is invalid, or the calling process has no children, *wait* returns a value of -1 and sets the global variable *errno* to indicate the reason for the failure.

4.1.8 Other system calls

Other system calls for the process subsystem include *nice* and *schedctl*, which adjust the scheduling priority of a process, and *mmap*, which maps a file into the process's address space.

4.2 Process Data Structures

The most important data structures to consider for the purposes of discussing processes under IRIX are the process table, the user block and kernel stack, and the `pregion` and `region` structures. As has been mentioned before, all the information necessary for scheduling a process (along with other information that cannot be swapped out) is contained in a process table entry. The process table entry contains fields identifying the user block and kernel stack for the process, and a pointer to a linked list of `pregion` structures. The user block contains information about the process which is not necessary for scheduling, and the `pregion` and `region` structures define the process's virtual address space.

4.2.1 The Process Table

The process table is an array of `proc` structures. A `proc` structure is allocated for each active process, so the number of processes which can be active at one time is limited by the size of the process table (the process table is compiled into the kernel, so the kernel must be reconfigured to change the size of the process table). Each proc structure contains the following information:

- process status (for example: running, ready to run, exiting),
- several values used for scheduling (priority, ''nice'' value, user level priority and cpu usage),
- process ID and parent process ID,
- pointers to other `proc` structures (used for linking the `proc` structure on to various lists),

- Structures describing the pages used for the user block and kernel stack
- a pointer to a list of `pregion` structures
- a session identifier
- a number of semaphores and spinlocks protecting various fields

The structure of the process table is declared in the */usr/include/sys/proc.h* header file.

4.2.2 Process States and Transitions

As mentioned in Chapter 2, each `proc` structure may be threaded onto a number of linked lists, depending on what state the `proc` structure is in. The kernel maintains the following lists:

- the active list, containing all active processes,
- the free list, containing all the `proc` structures which are not being used for active processes,
- the run queue, containing processes which are ready to be run (a subset of the active list),
- the sleeping process list, containing processes which are sleeping (these processes are active, but not runnable),
- the exiting process list, containing processes which have issued the *exit* system call, but have not yet been *wait*ed for.

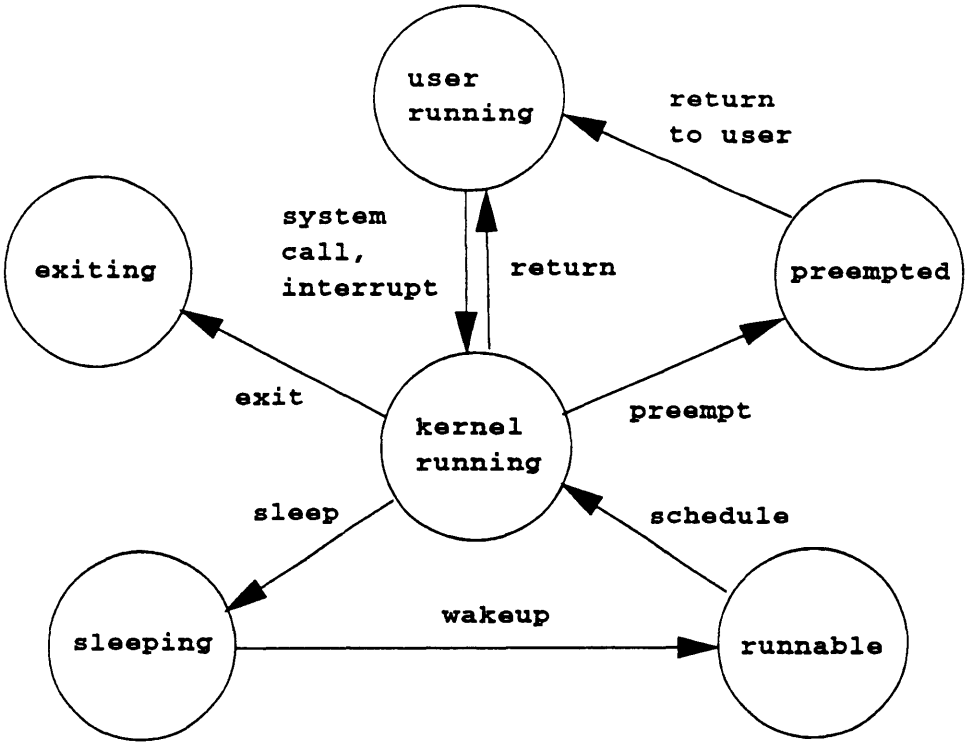

Figure 4-1. Process State Transitions

Figure 4-1 shows the transitions between the various process states an active process may be in. To demonstrate how these transitions work in practice, take the example of a process spawning a child process.

The process starts out in the "user running" state, and changes into the "kernel running" state by making a *fork* system call. When a process makes a *fork* system call, a process table entry is allocated off the free list. While the new process is being set up, it is in an intermediate "newly created" state. When the new process is set up, it is placed in the "runable" state. At this point, the *fork* call returns, and the parent process returns to "user running." Eventually, the new process will be selected to run, and will switch to "kernel running" state, and eventually return to "user running."

At some later point, the new process can execute the *exit* system call, causing it to re-enter "kernel running" state. The process then enters "exiting," or "zombie" state until its parent process performs a *wait*

system call. When the parent performs the *wait* call, the *wait* routine removes the zombie process from the process table, and places its process table entry on the free list.

A number of spinlocks and semaphores are used to ensure the consistency of the process table without unnecessarily restricting access to the table. The free list is protected by a single spinlock—this is all that is necessary, since free list accesses are fairly quick.

Accesses to the active list may take longer, and therefore a more complex access mechanism is used to minimize delays. To simplify the task of locating a given process by its process ID, active processes are linked onto hash lists, using a hashing function based on the process IDs.

The locking protocol used with the active list is called the *shared read lock*—it allows multiple processes to read the list at one time, but requires writing processes to acquire exclusive access to the table before updating it. Two kernel variables, `pactcnt` and `pupdcnt`, keep track of the number of processes reading the list, and the number of processes waiting to update it, respectively. These fields are protected by a spinlock, `pactlck`, which must be acquired before modifying these fields or the active list itself. A semaphore, `pupdwait`, is used to queue processes waiting to write the list.

Routines which modify the active list first acquire `pactlck`, then check if there are any readers on the list (`pactcnt` > 0). If there are, the routine must increment the writer count (`pupdcnt`), relase `pactlck`, and do a `psema` operation on the `pupdwait` semaphore, sleeping until another process performs a `vsema`. A process sleeping on `pupdwait` is woken up when the last reading process is done with the list, and must then re-acquire `pactlck` in order to modify the list.

Routines which read the active list start out by acquiring `pactlck`, incrementing the reader count (`pactcnt`), and releasing `pactlck`. The routine may then scan the active list—even though the lock is not held, the active list may not be modified while the reader count is greater than zero. When the routine finishes, it once again locks `pactlck`, and decrements the reader count. If it was the last reader (`pactcnt` <= 0), it checks the writer count, and if it is greater than zero, performs a *vsema* operation for each waiting writer. The process then unlocks `pactlck`, allowing other processes to access the list.

In addition to these global locking mechanisms, there are a number of locking mechanisms used to protect various fields in the individual process table entries. A spinlock, `p_siglck`, is used to protect fields used for signal handling. It is also used during context switches to protect the `p_sonproc` field, which indicates which processor the process is currently running on. The `p_sema` semaphore is used to protect the parent-child-sibling chain, and is important for process creation. The `p_parlck` spinlock is used to protect the parent-process ID field. Finally, there is a semaphore, `p_wait`, used by the *wait* system call.

4.2.3 Run Queue

Processes which are ready to run are kept on the *run queue*. Although referred to as a single "queue," the run queue is in fact implemented as two separate priority queues, one for high priority processes and one for normal priority processes. These queues are sorted according to priority. When a process becomes runnable, it is added to the appropriate queue at the appropriate level. The priorities of normal processes are recalculated once per second, and the normal priority run queue is resorted at this time. Special system processes are linked on the high priority queue. These processes operate at fixed priority levels, so the queue does not need to be resorted.

The run queue is protected by a spinlock, which must be held while accessing or modifying the run queue. The run queue structure is defined in the file */usr/include/sys/runq.h*.

4.2.4 Sessions and Process Groups

IRIX processes are grouped into *sessions* and *process groups*. A session represents all of the processes associated with a given login session. The first process in a session—usually the shell—is the session leader. When the session leader exits, all of the processes in the foreground process group are sent a signal (SIGHUP).

4.2.5 The User Block

The user block contains a tremendous amount of information about the process. Some of the important fields in the user block are:

- pointers to `inodes` representing the process's current directory and root directory
- the process's real and effective user and group IDs
- a pointer to the process's proc structure
- the user file descriptor table
- various fields used for passing arguments between kernel routines
- various fields used for signal handling, memory management, and process accounting

The current process's user block is mapped into kernel virtual memory at a known location, identified by the global kernel variable `u`. This mapping is set up in the processor's memory management hardware.[1] On a multiprocessor system, each processor has its own memory management hardware, and this mapping is made on a per-processor basis. Therefore, each processor can find the user block of its current process at the same virtual address, `u`. In each case, the processor's memory management hardware maps this virtual address to the physical address at which the current process's user block resides.

The user block does not have any locking mechanisms associated with it, since it is only manipulated when the process is being created, or when the process is running. In both cases, the user block is protected by the process table locking mechanisms.

---

1. The memory management system uses a hardware cache of virtual-to-physical address mappings called the Translation Look-aside Buffer, or TLB. When the kernel executes a context switch, it "wires" the pages for the user block and kernel stack of the new process into the TLB. "Wired" entries stay in the TLB until explicitly removed, so "wiring" an entry into the TLB creates a constant hardware level virtual-to-physical address mapping. As an extensive discussion of hardware issues is beyond the scope of this book, see *MIPS RISC Architecture* for more information on the TLB and related hardware issues.

4.2.6 Process Regions

The virtual address space of a process is defined by a set of *per-process region structures*, or `pregions`. At a minimum, a process has three `pregions`, representing the process's text, data, and stack. In addition to these, a process may have `pregions` representing shared memory segments or memory mapped files. (Shared memory is covered in Bach, Chapter 10. Memory mapped files are covered later in this chapter.)

The virtual memory layout for a process with just three regions (text, data, and stack) is shown below.

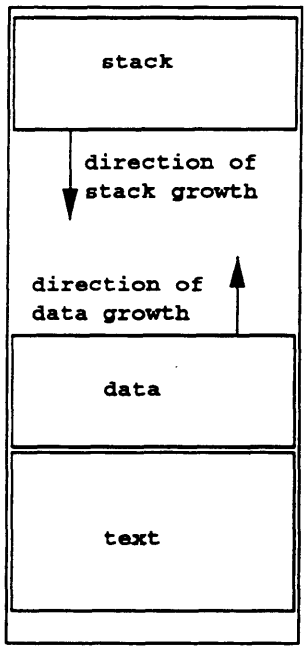

Figure 4-2. Process Virtual Address Space

The text region of the process, containing the machine instructions that comprise the program, is located near the bottom of the process's virtual address space (the space below the beginning of the text region is used by the operating system). The text region is static in size, and the contents cannot usually be altered by program.

The data region, representing both statically and dynamically allocated memory, is above the text region. The size of the data region may be altered by the process, either directly, through the *brk* system call, or indirectly,

through one of the library routines provided for memory allocation (for example, *malloc*(3X)). The data region grows upwards (towards larger virtual addresses).

The stack region is located at the top of the process's virtual address space. Like the data region, the stack region can be resized. Unlike the data region, however, the stack region is resized automatically by the kernel. When a process attempts to access a virtual page beyond the end of the stack region, the process incurs a memory fault, and the memory fault handler identifies the problem and attempts to allocate more memory for the stack (for details on the memory fault handler, see Chapter 5, "Memory Management"). The stack grows downward, towards smaller virtual addresses. Theoretically, a process could run out of virtual address space when expanding the data or stack regions. However, since each process has a two gigabyte virtual address space, processes will usually run out of available memory and swap space long before they run out of virtual address space.

Access to a process's `pregion` list is protected by a semaphore, `p_preglock`, in the process's `proc` structure. This lock must be acquired before manipulating the `pregion` list—for example, adding a new shared data segment.

A `pregion` structure contains the following fields:

- a pointer to a global `region` structure
- the virtual address of the region
- the type of region (for example, text, data)
- a set of flags

The `region` structure that the `pregion` refers to may be shared with other processes. For example, two instances of the same program in memory may use the same text region. Though both processes use the same `region`, they do not necessarily access it at the same virtual address. For example, if two processes attach to the same shared memory region, they may locate the region at completely different virtual addresses. The mapping of regions into process virtual address space is one of the main function of the `pregion` structure. The `region` structure is used by the memory management subsystem to locate the individual pages of memory referenced by the process. The `region` structure is treated in more detail later in this chapter and in Chapter 5, "Memory Management." The `pregion` and `region` structures are declared in the */usr/include/sys/region.h* header file.

4.3 Process Context

The *context* of a process is all of the information needed to run that process. In *The Design of the UNIX Operating System*, Maurice Bach identifies three discrete components of a process's context: *user-level context*, *register context*, and *system-level context*. The user-level context consists of the memory which can be accessed by the process in user mode, that is, anything in the virtual address space of the process. The register context consists of the values of crucial machine registers, such as the stack pointer and the program counter. The system-level context consists of all the structures which define the process but which may not be accessed by the process in user mode: for example, the process table entry, user block, the `region` and `pregion` structures, and the kernel stack.

When the current context needs to be saved, such as when an interrupt is received, or the process undergoes a context switch, the register context is pushed onto the kernel stack, followed by a new kernel stack frame. If another interrupt is received before the interrupt handler returns, another layer will be pushed onto the kernel stack. Bach refers to these layers as *context layers*. When a kernel is operating at a given processor execution level, it will only handle interrupts of a higher level. Thus, the maximum number of context layers is determined by the number of processor execution levels.

Take as an example the case of a process making a system call. The process pushes the arguments to the system call onto the process's stack, stores the system call number in a certain machine register, and executes an instruction which causes an exception, switching it to kernel mode. The process's register context is pushed onto the kernel stack, along with a new kernel stack frame. Then the system call handler is invoked. The system call handler looks up the system call number in a table, where it finds the address of the desired system call routine, and the number of arguments expected. The arguments are copied from the process's stack into the user block, and the desired system call routine is invoked. This routine reads its arguments from fields in the user block and writes its return value to a field in the user block. The system call handler copies the return value from the user block into the appropriate machine register before returning. The system call handler also checks for errors in the system call, and if it finds one, copies the error number from the user block into the appropriate machine register. When the system call handler returns, the process's register context is restored and it returns to user mode. As far as the

program is concerned, this entire procedure is identical to a simple subroutine call.

4.4 Process Scheduling

On both single-processor and multi-processor systems, the scheduling procedure is essentially the same. Whenever a processor becomes eligible for a job, it executes the process dispatcher routine, *disp*, which searches the run queue for a new process to run. If the dispatcher cannot locate a process to run, the processor enters an idle loop, waiting for something to do. If the dispatcher locates a suitable process, the processor begins execution of the new process. The process will continue running until:

1. the process exits, or,
2. the process goes to sleep waiting for a resource, or,
3. the process exceeds a given time slice.

If the processor receives an interrupt, it may suspend execution of the process temporarily in order to deal with the interrupt. It does this without leaving the context of the process. When one of the three conditions listed above occurs, the processor executes a *context switch*, and tries to locate a new process to run, as described above.

4.4.1 Process Priority

The order in which processes are scheduled is determined by their priorities. The priority of a normal process is a function of a system constant value, a per-process variable, (the process's *nice* value) and the process's recent CPU usage. The lower a process's numerical priority value, the sooner it will be chosen to run. When normal processes become runnable, they are linked on to the normal-priority run queue. The high priority run queue is reserved for special system processes, which operate at fixed priority values below those of normal processes.

The easiest way to affect the scheduling process is to adjust a process's nice value. Nice values can range from 0 to 39, with the default value being 20. Higher nice values result in higher priority values, so processes with high

nice values tend to run less often. A process can change its nice value using the *nice*(2) system call. Any process can increase its nice value, but it must have superuser privilege to lower its nice value. A process with superuser privilege can also alter the nice value of another process using the *schedctl*(2) system call.

CPU usage is recorded by the clock interrupt handler, which is called every time a hardware clock interrupt is generated (for most IRIX machines, clock interrupts occur at an interval of 10 milliseconds). At each clock tick, the running process has its CPU usage incremented. Once a second, the priorities of all active processes are recalculated. At this time, each process's CPU usage is divided by two, and the new CPU usage value is added to the nice value and the system constant to produce a new priority. Therefore, processes which have run recently receive relatively high priority values, making them less eligible for processor time. This ensures that one process cannot monopolize the processor while other processes are waiting to execute. After recalculating all the process priorities, the kernel resorts the normal-priority run queue.

While this scheduling system ensures a more-or-less equitable distribution of processor time, it causes processes to be run at unpredictable intervals, causing problems for real-time applications. It is possible to a assign a process a *real-time*, or *non-degrading*, priority value, using the *schedctl* system call. The process then operates at the assigned priority value, instead of a calculated priority value. Non-degrading priorities may be assigned in the range of 30 to 255. Normal interactive processes have priority values in the range of 40 to 127. Therefore, processes with non-degrading priorities below 40 will be selected to run in preference to any normal interactive processes. Processes with non-degrading priorities above 127 will only be selected to run when no normally scheduled processes are runnable. If two or more processes are set at the same non-degrading priority, they will be scheduled in a round-robin fashion. Non-degrading priorities may only be assigned by processes with super-user privilege, although any process may cancel its own non-degrading priority. Non-degrading priority processes are linked onto the normal-priority queue along with normally scheduled processes.

4.4.2 The Dispatcher

The process selection algorithm, implemented by the *disp* routine, is fairly simple. While holding the run queue lock, it searches the run queue for a process to run, checking first the high-priority queue and then the normal-priority queue. When the dispatcher finds a process on one of these queues, it must make sure that the process is actually runnable. There are several reasons that a process might be on the run queue even though it is not, in fact, runnable. These include:

1. The process has been swapped out, as explained below.
2. The process is in the middle of a context switch after running on another processor. The process has been placed on the run queue, but the other processor has not let go of the process's kernel stack yet. This is explained further under "Context Switches," below.
3. The process is bound to another processor.

As the *disp* routine examines each process, it holds the process's signal lock (`p_siglck`). This prevents the process from being swapped out while it is being examined. At this point there is a race condition. If the dispatcher acquires the signal lock first, and chooses the process to run, then when the swapper acquires the signal lock, it will find the process running, and will try to find another process to swap. If the swapper acquires the signal lock first, it will swap the process out, so that when the dispatcher acquires the signal lock it will find the process no longer in core, and therefore, not runnable. The swapper process, *sched*, is covered in more detail in Chapter 5.

Once the *disp* routine has located a process to run, it takes the process off the run queue, marks it as running on the processor, and releases the process's signal lock. *disp* returns a pointer to the selected process's `proc` structure, or a null pointer if no suitable process can be found.

4.4.3 Context Switches

There are three basic routines called to handle context switches. These routines are *swtch*, *qswtch*, and *pswtch*. The *swtch* routine saves the context of the current process, and then calls the dispatcher routine to select a new process to run. The *qswtch* routine places the current process back on the run queue, and then calls the *swtch* routine. The *pswtch* routine is called

when a process is exiting, and it deallocates most of the exiting process's resources before selecting a new process to run.

The *swtch* routine is called when a process goes to sleep, or from *qswtch*, which is called when a process exceeds its time slice.

If, after switching out a process, the processor cannot find a new process to run, it enters the *idle* routine, waiting for a process to become eligible to run. The processor will idle until one of two things happens:

1. an interrupt occurs which has the effect of waking up a process which was sleeping (for example, an interrupt signaling the completion of an I/O request), or,
2. another processor places a process on the run queue (for example, it has just executed a *fork* system call, creating a new process).

In both cases, the new or newly runnable process is placed on the run queue, and the *kickidle* routine is called to notify all idle processors that there are runnable processes.

4.5 Process Creation and Termination

A process can spawn a "child" process using the *fork* system call, and terminate itself with the *exit* system call. The peculiarity of the *fork* system call is that it returns twice—once in the parent process and once in the child process. The child process begins its operation by returning from a system call it didn't make, and ends its existence by making a system call which doesn't return. The exit status of the child process can be retrieved by the parent using the *wait* system call. Since a child's return status must be saved until its parent *wait*s for it, a child process which has *exit*ed is kept in the process table until it is *wait*ed for. These processes which have exited but not yet been removed from the process table are called *zombie processes*.

4.5.1 Creating a New Process

The *fork* system call creates a nearly exact, logical copy of the parent process. Nearly exact, since the processes have at least one value different—the return value of the *fork* call. Logical, because the actual regions and data pages may not be duplicated. Usually, the new process will refer to the same text `region` as the old process. As for the data and stack, new `regions` will be allocated for the new processes, but these `regions` will continue to refer to the old data pages. These pages will be marked "copy-on-write." This means that each page will only be copied when one process or the other attempts to modify it.

```
algorithm fork
inputs: none
outputs: returns child process ID in parent process,
    zero in child process.

{
   allocate proc structure off free list, place on active list
   if (couldn't allocate a proc structure) return error
   acquire p_sema on new proc structure
   copy data from parent's proc structure to child's
   proc structure
    add child process to the parent-child chain
   add child process to parent's session and process group

   for (each file descriptor)
      increment reference count on file structure
   increment reference count on current directory inode
   increment reference count on root directory inode

   duplicate parent's user block
   change u_procp in child's user block to point to
   child process
   copy regions
   put process on run queue
   if (couldn't duplicate process) return error
   set child process return value to zero
   relase cp->p_sema
   return child's process ID to parent
}
```

Figure 4-3. Algorithm for creating a new process

Figure 4-3 shows the *fork* algorithm. First the routine must allocate a `proc` structure off the free list and place it on the active list. The routine

must hold the free list lock and the active list lock while manipulating these lists. If the free list is empty, a new process cannot be created, and the *fork* system call returns a value of -1 to the calling process. Once a `proc` structure has been allocated, the routine locks the structure by acquiring the structure's `p_sema`. The routine then copies data from the parent's `proc` structure to the child's `proc` structure—real and effective user IDs, signal handling masks, and other inherited traits.

The `proc` structure is then added to the parent-child-sibling chain. First, the routine acquires the parent's `p_sema`. It then sets the child's `p_sibling` pointer to point to the parent's previous child, locks the child's `p_parlck`, sets the child's `p_parent` field to point to the parent structure, releases the child's `p_parlck`, and finally sets the parent's `p_child` field to point to the new child process before releasing the parent's `p_sema` semaphore.

The child is then added to the parent's session and process group. While manipulating these lists, the routine holds the global process-group and session semaphore, `pglobal`.

4.5.2 Terminating a Process

Processes may terminate their execution by using the *exit* system call. The *exit* routine releases most of the resources associated with a process. The *exit* algorithm is outlined in Figure 4-4. This algorithm is complicated by a number of considerations. If the exiting process is a session leader, all of the processes in the foreground process group should receive a signal. If the process has any children, these must be passed on to the *init* process, which inherits all child processes. If any of these children are zombie processes, the exiting process must send *init* a signal, to indicate to *init* that it should *wait* for the zombie processes. In addition, the process must check whether its parent process is ignoring the SIGCLD signal, which indicates the exiting of a child process. Normally, information such as the process's exit value is stored in its proc structure, which is not reused until the parent process retrieves the information through the *wait* system call. However, if the parent is ignoring SIGCLD, it is assumed that the parent will never *wait* for its child processes. In this case, therefore, the exiting process's `proc` structure is reused immediately. There would race condition in the *exit* routine if a process changed its handling of the SIGCLD signal at the same time that one of its children was exiting. To avoid problems, the exiting process holds the parent's signal lock while examining its signal handling. If

the parent process changes its signal handling status after this point, the signal handling code takes care of zombie children. When a process changes its signal handling to ignore SIGCLD, the signal handling code automatically frees any zombie child processes.

There is also a potential race condition when a process is going through the *exit* code at the same time as one of its children. To prevent problems, the exiting process holds its parent lock during the critical portion of code. The process's parent lock must be held when changing the process's parent pointer, so holding this lock prevents the parent process from giving the child away to *init*.

The *exit*ing process does three things to ensure that the parent process is notified of its child's demise. First, it unsets the parent's wait-search flag (SWSRCH). This has the effect of catching a parent process which is in the *wait* routine in the process of searching its list of children for zombie processes. The child process must hold the parent's signal lock while changing the SWSRCH flag. Second, the *exit*ing process performs a conditional vsema operation on the parent's wait semaphore, which will awaken the parent if it is in the *wait* routine, sleeping on the wait semaphore. Lastly, the *exit*ing process sends its parent a SIGCLD signal. This provides asynchronous notification to the parent that one of its children has *exit*ed and should be *wait*ed for.

```
algorithm exit
inputs: exit value
outputs: none
{
   acquire signal lock (p_siglck)
   clear pending signals
   set process status to "exiting" (p_flag |= SEXIT)
   set signal handling to ignore all signals
   release signal lock
   if (process is session leader)
      send SIGHUP to all members of foreground process group
   close all files
   (if unblock on exec/exit is set, do unblocking now)
   iput inodes for current and root directories.
   deallocate interprocess communication stuff
   acquire address space lock
   detach regions
   release address space lock
   leave process group and session (lock pglobal)
   give all child processes to init
   if (one or more child is a "zombie")
      cvsema init's p_wait
      send init SIGCLD
   acquire parent's p_sema (semaphore locking parent-child-sibling chain)
   acquire parent's signal lock
   if (parent is ignoring SIGCLD)
      release parent's signal lock
      remove child from active process hash table
   else /* parent should wait for this process */
      release parent's signal lock
   acquire process's parent lock
   lock process's signal lock
   change status to "zombie"
   release signal lock
   lock parent's signal lock
   unset parent's wait-search flag
   release parent's signal lock
   cvsema parent's p_wait semaphore
   send parent SIGCLD
   release process's parent lock
   switch to idle stack
   free upage/kernel stack
   if (parent is ignoring SIGCLD)
      free proc structure
   release parent's p_sema
}
```

Figure 4-4. Terminating a Process

4.5.3 Awaiting Process Termination

A parent process may await the termination of a child process, or collect the status of a child process which has already terminated, using the *wait* system call. The algorithm for the *wait* system call is outlined in Figure 4-5. The routine first checks to see if any of its children are already in the zombie state. If the process has a terminated child, then the routine saves the child's status information, frees the child's `proc` structure, and returns immediately. If the process has no children at all, then the routine returns immediately with an error. If the process has one or more children and none are terminated, the routine does a *psema* operation on the wait semaphore (in the process's `proc` structure), causing the process to sleep until one of its children performs a *vsema* operation, as part of the *exit* procedure outlined above.

When the *wait* routine finds a terminated child, it records the child's process ID, and if it has been passed a valid pointer to a status structure, it copies the child's status information into the structure.

```
algorithm wait
inputs: pointer to status structure
outputs: process ID (and status) of terminated child
{
   while (we haven't found a child to wait for)
      acquire signal lock
      set wait-search flag bit
      release signal lock
      lock parent-child-sibling chain
      search chain for zombie child
      if (found zombie child)
         record process ID of child
         if (pointer to status structure is valid)
         copy status of child to status structure
         free child's process table entry
         break /* to end of while loop */
      if (process has no children)
         return -1
      release lock on parent-child-sibling chain
      acquire signal lock
      if (wait-search flag has been unset)
         release signal lock
         continue /* A child process changed status during  */
                  /* scan.  Return to top of while loop.    */
      unset wait-search flag
      release signal lock; psema wait semaphore
      /* Process sleeps until it receives a signal */
      /* or a child exits.                         */
      if (interrupted by signal)
         return -1
      /* otherwise return to top of while loop, */
      /* to scan for terminated child           */
   acquire signal lock
   unset wait-search flag
   release signal lock
   relase p_sema
   if (we actually found a terminated child process)
      return process ID of terminated child
   else
      return -1
}
```

Figure 4-5. Awaiting Process Termination

4.6 Signals

Signals provide a mechanism by which processes may be notified asynchronously of various conditions. These conditions include:

1. the death of a child process,
2. unexpected or unrecoverable errors encountered during system calls,
3. the expiration of a timer, set by the process using the *alarm*(2) system call,
4. exceptions caused by illegal instructions (for example, attempts to use privileged instructions or access invalid virtual addresses)
5. hardware errors

Signals may also be sent by other processes (using the *kill*(2) system call), or generated by the user, by pressing a certain key. For example, the Control-C combination is usually used to generate an "interrupt" signal, used to terminate a process prematurely.

When a process receives a signal, the kernel records the fact in the pending signal field in the process's `proc` structure. This field has one bit for each type of signal, which can be set to indicate that that type of signal has been received. No count is kept of the number of signals received. The proc structure also contains a mask indicating which signals the process is ignoring. If the process is ignoring a given signal, the kernel simply doesn't deliver that signal. If the process is sleeping when it receives a signal, it may be awakened, provided that it is sleeping at an interruptible priority.

The kernel handles a process's signals just before returning from kernel to user mode. Thus if a process incurs an exception which causes a signal to be sent to the process, the process goes into kernel mode to handle the exception, then handles the signal before returning to user mode. Similarly, if a process makes a system call which generates a signal, the signal is handled before the system call returns. Most signals cause the process to exit by default. Some of these also cause the program to dump a *core image file*, a kind of snapshot of the running process which can be helpful in debugging programs (more information on core image files can be found on the *core*(4) manual page). Most other signals are ignored by default. The exceptions are the job control signals, SIGSTOP and SIGCONT. SIGSTOP causes the process to suspend execution, until it receives a SIGCONT signal.

If the process opts to catch a given signal, by specifying a signal-handling function for it using the *signal* system call, the function pointer is stored in a table in the process's user block. There is one entry in this table for each type of signal, so changing the handling of one signal has no effect on the handling of any other signal.

4.6.1 Handling Signals

When the kernel recognizes that a process has received a signal, it checks to see how the signal should be handled. If the process is ignoring the signal, it will simply be cleared. The signal would not have been delivered if the process was ignoring it at the time it was sent, however, a process can receive a signal while it is in the middle of calling *signal* to ignore the same signal. In this case, the signal would be posted normally, but by the time the signal was recognized (just before the *signal* system call returned), the process would be set to ignore it.

If the process is not ignoring the signal, the kernel checks whether the process has set up a signal handling routine for it. If so, the kernel sets up the process to execute the signal handling function. First, it makes a record of the address of the signal handler, stored in a field in the user block, and then clears the field, restoring the default behavior for the signal (the reason for this will be explained shortly). Once the kernel has the address of the signal handler, it creates a new stack frame on the top of the process's stack, and copies into it the return address from the process's kernel stack. The return address on the kernel stack is then changed to point to the beginning of the signal handler function. When the process returns to user mode, it ''returns'' to the signal handler, instead of the routine it was in when the signal was received. When the signal handler finishes, it returns to the original return address--that is, the address that the process would have returned to had there been no signal. If the process was in kernel mode because of a system call, for example, then the signal handler will return to the next instruction after the system call.

As mentioned above, the kernel resets a process's signal handling status when it catches a signal. If the kernel did not do this, a steady stream of signals would cause nested calls to the signal handler, possibly overflowing the stack. Therefore, for the process to continue catching a signal it must call *signal* each time it catches a signal. This is commonly done in the signal handling routine itself. This solution is not foolproof, however, as a process may receive a signal in between the time the signal handler is

called, and the time that the *signal* system call is actually performed. In this case, the signal will have its default effect, usually causing the process to exit.

4.7 Manipulating Process Address Space

As mentioned before, a process's address space is divided into regions. This section deals with the manipulation of process regions. First, we will outline some basic algorithms— allocating and freeing `region` structures, attaching a region to a process, detaching a region from a process, changing the size of a region, loading a file into a region, and duplicating a region.

Like many other kernel data structures, `region` structures are allocated from a free list. When `regions` are allocated by the *allocreg* routine, they are placed on the active region list and assigned a type (for example text, data, stack) and a pointer to an in-core `inode`. For text regions, this `inode` will represent the file from which the program's text is to be read. In this case, the `inode`'s reference count will be incremented. In the case of data and stack regions, which do not correspond to any disk file, the inode pointer in the `region` structure will be null. Each `region` structure contains a locking semaphore, which is used in a similar way to the inode locking semaphore. The *allocreg* routine returns a locked `region`, and the routines for attaching and freeing regions expect locked `regions`.

The *attachreg* routine is called to attach a region to a process. It takes four arguments: a locked `region` structure, a `proc` structure, the virtual address at which the region should be attached, and the type of region. *attachreg* allocates a per-process region structure (`pregion`) and fills in the data for the new region (type, virtual address, and a pointer to the `region` structure). It then adds the new `pregion` structure to the process's `pregion` list, and increments the reference count in the `region` structure.

The *growreg* routine takes as arguments a pointer to a `pregion`,the number of pages to add to or remove from the region, and an argument specifying how newly allocated pages are to be treated. *growreg* returns 0 if the request is a no-op (a request to grow by zero pages), 1 if the request is successful, and -1 if the request fails. *growreg* records the new size of the region, and frees or allocates pages and page tables as necessary. When called to increase the size of a region, *growreg* may fail because the request

would cause the process or region to exceed the maximum allowable size, or because it would cause the region to overlap another region.

The *attachreg* and *allocreg* routines have counterparts in the *detachreg* and *freereg* routines. The *detachreg* routine removes the region from the process's virtual address space, deallocating the `pregion` structure. It also decrements the `region`'s reference count, and if the reference count drops to zero, calls *freereg* to deallocate the `region`.

During the *fork* system call, regions are duplicated using the *dupreg* routine. Depending on the type of region it is passed, *dupreg* may copy the region, or simply return the original region. In the case of read-only text regions, there is no point in duplicating the region, so *dupreg* simply returns the original region. For writable regions, *dupreg* allocates a new `region` structure and associated structures (page tables and pages). This procedure is covered in detail in the following chapter, "Memory Management."

There are two routines for loading files into regions. For most executable files, the *mapreg* routine is used. This routine does not actually read the file data into memory, but sets up the kernel data structures to allow the file to be read in on demand, a page at a time (this process is covered in Chapter 5). The *loadreg* routine exists for the sake of backwards compatibility with versions of the operating system that did not support demand paging. *loadreg* reads the file data into the region. This allows to run executable files which were not designed to run in a demand paging environment. Both *loadreg* and *mapreg* take the following arguments:

1. a pointer to a `pregion` structure,
2. the virtual address at which the text should start in the region (which may be different from the region's attach address),
3. a pointer to a locked in-core `inode` representing the executable file,
4. the offset in the file at which the program code begins, and
5. the size of the program code (in bytes).

Both routines grow the region to the appropriate size using the *growreg* routine. The *loadreg* routine then loads the file data into the region, using the file-system dependent *readi* routine. The *mapreg* routine sets up the data structures necessary for demand paging. These structures are described in Chapter 5.

4.7.1 Executing Another Program

The algorithm for executing a new program exercises most of the region-manipulation routines described above. In the simple case, the *exec* algorithm is executed using an executable file which is a compiled binary, containing header information and machine instructions (the format of executable file headers is detailed on the *a.out*(4) manual page). The kernel opens the executable file and reads the header information. The header provides such data as the size of the program's text and the offset in the file at which the text begins.

The kernel then unlocks the `inode` for the executable file, and proceeds to detach all the process's old regions, freeing them if necessary. It then allocates and attaches new regions (algorithms *allocreg* and *attachreg*) and sets them up as specified by the header information. There are two special cases related to the text region. First, if a program is *exec*ing itself, and the text region is read-only, the kernel does not bother to detach the text region (debuggers may make the text region writable in order to set breakpoints in a program—in this case, a new copy of the text would be loaded). Second, the kernel searches the active region list to see if a copy of the text is already loaded (that is, there is a `region` with type "text" referring to the in-core `inode` for the executable file). If it finds such a region, it simply attaches the region to the process. The stack region also receives special treatment, since the first few pages of stack have already been allocated and filled in with the new program's arguments and environment variables. When the new stack region is created, then, it will use these previously allocated pages instead of allocating new ones.

So far, it has been assumed that the executable file being *exec*ed is a compiled program. However, it is also possible to *exec* a file which contains instructions to be executed by a command interpreter, such as the shell. Such files are referred must begin with the string **#!**, followed by the path name of the interpreter (which must be a compiled program). For example, a file containing commands to be executed by the Bourne shell would begin with the string:

```
#! /bin/sh
```

If the kernel fails in its attempt to load header information from an executable file, it checks to see whether it is an interpreter file. If so, it reads the path name of the interpreter, and tries to *exec* the interpreter, passing the

original file name to it as an argument.

```
algorithm exec
inputs: path name of executable file to be run
optional inputs: arguments to program, environment variables
outputs: none
{
   get locked inode for executable file (algorithm namei)
   get setuid/setgid information for file
   load header information from file
   if (couldn't load header information)
      if (file is interpreter file)
         load header information from interpreter
         if (couldn't load header information)
            return error
      else /* file can't be executed for some reason */
         return error
   allocate memory for new stack
   copy arguments (and environment) to new stack area
   unlock inode
   unlock locked regions
   acquire process's address space lock
   for each region
      lock inode corresponding to region (if any)
      lock region
      detach region
      iput region's inode (if any)
   load regions from executable file or interpreter
   set up stack region (using previously allocated pages)
   release address space lock
}
```

Figure 4-6. Executing a New Program

4.7.2 Changing the Size of a Process

User programs which need more memory can increase the size of their data region using the *brk* or *sbrk* system calls. Both *brk* and *sbrk* invoke the *growreg* routine to increase the size of the data region. As noted above, *growreg* may fail if the request would make the region too large, or would cause the region to overlap another part of the process's virtual address space.

Programmers do not need to use *brk* or *sbrk* directly to allocate memory. provides a set of library routines for managing dynamic memory allocation.

These routines are described on the *malloc*(3X) manual page.

The stack region also grows, but this procedure is automatic and transparent to the user program. When a process overflows its stack region, it incurs a memory fault. The fault handler determines that the fault was caused by a stack overflow, and invokes the *growreg* routine to allocate more stack space.

4.7.3 Mapping Files into Process Address Space

It is possible for user programs to map files into a process's virtual address space using the *mmap* system call. The arguments to *mmap* specify which portion of the file should be mapped into the process's address space, and the virtual address at which the mapping should begin. The *mmap* system call actually creates a new region, a *mapped file region*, and attaches it to the process. The file is mapped in using the *mapreg* routine, and it is treated much the same as a text region, except that a mapped file region can be either read-only or read-write. In the case of writable mapped file regions, provisions must be made to ensure data consistency. These aspects of mapped files are covered in Chapter 5.

5. Memory Management

Memory on a multiuser computer is a limited resource. An operating system such as IRIX cannot maintain every process in main memory at all times. To do so would require an unreasonable amount of memory. Therefore, the available memory must be managed to allow the greatest number of processes to run with the greatest efficiency. Like a juggler who controls many balls with two hands, IRIX controls many processes by placing portions of text and data in memory as they are required, possibly removing them to secondary storage when they are no longer needed. There are programs, such as the IRIX kernel itself, which are always wholly resident in memory. Other programs are placed, in whole or part, in memory as needed. The algorithm by which IRIX places programs in memory is called *paging* or *page swapping*. The various regions that make up a process are divided into *pages* of 4Kbytes. Each page can be independently read into memory or written out to secondary storage (the *swap device*). Pages are read into memory only when needed, so that if only ten percent of a program's code is executed, only ten percent need be read into memory. This system is referred to as *demand paging*. Demand paging systems have obvious advantages over *swapping* systems, which must read an entire program into memory before executing it. Demand paging systems are not only more efficient in their use of resources, they actually allow the user to run programs which are larger than physical memory.

The IRIS hardware provides mechanisms for mapping virtual pages to physical addresses, and controlling access to physical memory. The hardware maintains a cache of virtual to physical address mappings, called the *translation look-aside buffer*, or TLB. For the most part, the TLB entries are loaded from user page tables. These entries are transient, and may be replaced at any time (when a new entry is loaded, it is dropped into a random slot in the TLB, replacing an old entry). A few TLB entries are semi-permanent, or "wired." These entries stay in the TLB until explicitly removed, and are used by the kernel for mapping certain pages into virtual memory (for example, the user block of the running process is "wired" into

a certain virtual address).

The IRIS hardware provides two privilege modes, *user* and *supervisor*. These correspond to the user and kernel modes discussed earlier. Thus, when a program performs a system call, it executes a special command which causes it to enter supervisor mode. In user mode, the memory management hardware will only allow a process to access its own address space—that is, only the pages mapped by its own page tables. This is because there is a *context* field in the TLB that must match the process's ID for the hardware to consider it a valid mapping. Kernel pages have the *global* bit set in their TLB entries. This bit means that the page may be accessed by any process, but only if the process is in supervisor mode.

The IRIX system keeps a pool of physical pages, which may be dynamically allocated to store data. Each process region has a set of page tables associated with it, representing a set of virtual pages. Each page table entry represents a virtual page in the process's address space, which may be, for example, in core, on the filesystem, or on the swap device. If a page is in core, then the page table entry will contain the page number of a physical page from the aforementioned page pool.

Free memory is maintained by a daemon, the *page stealer*, which removes virtual pages from core. IRIX keeps track of the number of times a given virtual page is accessed, and uses this information to determine which pages can safely be removed.

When a process attempts to access an invalid or protected virtual page, it will incur an exception. These exceptions, or *page faults*, fall into two categories: *validity faults* and *protection faults*. A *validity fault* occurs when a process tries to reference an invalid page. There are three usual reasons for a *validity fault*:

1. The referenced page is part of the process's address space, and is in memory, but has been marked as invalid to detect a reference. In this case, the kernel simply notes that the page has been referenced, and marks it as valid.

2. The referenced page is part of the process's virtual address space, but is not in memory (that is, there is no physical page representing the virtual page). In this case, the kernel loads the page into memory. If the page needs to be read in from disk, the process will be put to sleep until the I/O is complete.

3. The referenced page is not part of the process's virtual address space. In this case, the kernel sends the process a signal to indicate that it has tried to access an illegal address.

A *protection fault* is caused when a process attempts to modify a page that is locked or protected. There are several reasons that this might occur:

1. The process attempts to write a page which is marked copy-on-write (for example, the data region of a process which has just *fork*ed).
2. The process attempts to write a page which has been marked read-only to detect a modification (the reasons for this are described later).
3. The process attempts to write a page that is marked read-only (for example, a page of of the process's text region).
4. The process attempts to access a page of kernel data. In the first case, the fault handler makes a new, private copy of the page for the process that incurred the fault. In the other two cases, the kernel sends the process a signal to indicate that it has attempted to access an invalid address.

5.1 Memory Management Data Structures

IRIX uses many data structures in its memory management system. The main data structures used are:

1. page frame data table
2. page descriptor entries
3. swap-use table

The page frame data table is an array of `pfdat` structures, each corresponding to a single page of physical memory. Each `pfdat` indicates what state the corresponding page is in—for example, whether it is locked, whether it is the target of a pending I/O operation, and so forth. Only pages which may be dynamically allocated are represented in the page frame data table. Pages used for kernel text and static kernel data structures, for example, are not represented in the page frame data table.

The pages that are represented in the page frame data table are said to be in the *system page pool*. Pages are dynamically allocated from this pool to

hold user and kernel data. Each page in the system page pool will have a corresponding `pfdat` structure in the page frame data table. The kernel code uses macros for locating the `pfdat` that corresponds to a given physical page, and vice versa. Physical pages are identified by their *page frame number*, or *physical page number*. IRIX creates that page frame data table at boot time, after determining the number of pages available for the system page pool.

The kernel maintains a free list and a number of hash lists of `pfdat` structures. These lists work much like the free and hash lists discussed for `inodes` in Chapter 3. Only pages representing file data are hashed. The hashing function is based on two values: the in-core `inode` associated with the page, and its logical page number (that is, its offset into the file, in pages).

The fields in a `pfdat` structure include:

1. A set of flags, detailing the status of the page—whether the page is on the free list and/or the hash lists, whether there is pending I/O on the page and whether there is a copy of the page on the swap device. There is also a flag that indicates that the page is being manipulated. This is used to prevent two processes from manipulating the same page at the same time.
2. The current number of processes using the page. The reference count equals the number of running processes (as determined by page table entries) that are using the page.
3. The in-core `inode` of the file containing the page (if there is a copy of the page on the file system), or an index into the swap table, indicating which swap file the page is stored in (if there is a copy of the page on the swap device).
4. The pages logical page number within the file (if there is a copy of the page on the file system or swap device).
5. Pointers to other `pfdat` structures, used for linking the `pfdat` onto the free list and hash lists.

The `pfdat` structure is defined in the */usr/include/sys/pfdat.h* header file.

All user virtual addresses and some kernel virtual addresses are mapped by means of *page tables*. A page table is nothing more than a set of page descriptor entries, or `pdes` each mapping a page of virtual memory to a physical page represented by a `pfdat`. Page tables are themselves stored

on pages dynamically allocated from the system page pool. The memory addressed by one page of page descriptor entries is referred to as a *segment*. A segment corresponds to 2 megabytes of memory. (A page descriptor entry is eight bytes long, therefore 512 pdes fit into a 4K page. 512 4K pages are equal to 2 megabytes of memory.)

Each region of memory maintains a page table that maps the parts of a process to the actual physical memory locations where the information is stored. In this way, the virtual addresses that a process uses are translated into physical addresses that the hardware uses to access the memory. The format of the page table is somewhat convoluted: the region structure contains a pointer to an array of pdes. These pdes describe the set of pages that make up the page table. That is, each pde in the array describes a page containing a number of pdes which describe the region's data pages. Therefore, each pde in the list pointed to by the region structure maps one segment (2 megabytes) of memory.

Figure 5-1, below, shows a process region with one of its associated page tables.

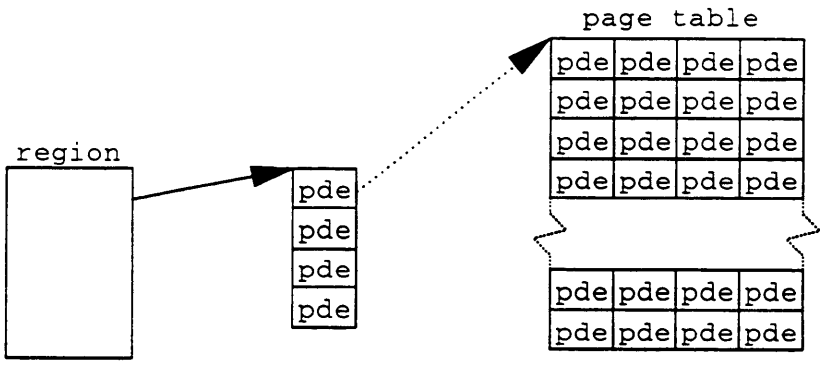

Figure 5-1. Region structure

The page descriptor entry consists of two components, the *page table entry* and the *disk block descriptor*. The page table entry is the hardware-specific portion of the page descriptor entry. It contains information about the page in the format required by the memory management hardware. The disk block descriptor contains additional information significant to the kernel.

Each page table entry contains the physical address of a page, permission information stored in bits indicating whether processes can read or write the page, and certain bits that are set to indicate various states relevant to the

paging system. The paging bits are called:

- valid and referenced (also called hardware valid)
- software valid
- modify
- copy on write
- age bits
- global
- global swappable

The kernel manipulates all these bits as the system runs. When a page is read in, or a new process starts, the kernel changes the fields in these data structures to reflect the usage of each page of memory. The format of the page table entry structure is different for different processors. The various formats are defined in the header file */usr/include/sys/immu.h*.

The *valid and referenced* bit indicates that the page is valid. If this bit is cleared, any process attempting to access the page will incur a validity fault. This bit is used in conjunction with the *age* bits to determine if the page is eligible for swapping.

The *software valid* bit indicates whether the page is actually valid. It is examined when a process incurs a validity fault on a page (because the valid and referenced bit is clear). If the software valid bit is set, the page is actually valid, and the valid and referenced bit has simply been cleared to detect a reference. If the software valid bit is clear, the page is not valid, and must be loaded into memory.

The *modify* bit indicates whether the page is writable or not. Any attempt to write a page for which the modify bit is cleared will result in a protection fault.

The *copy on write* bit indicates that the page must be copied the next time a process needs to modify it. This bit is set when a *fork* system call is made. Because two processes will be using the same page of memory, if one writes new information, the other will see incorrect information the next time it accesses that page. So, when a write operation is about to take place, the writing process will be forced to copy the page and disassociate itself from the original page. When the copy on write bit is set, the modify bit is cleared, so that the next attempt to write the page will cause a fault.

The three *age* bits indicate how recently the page has been used. These bits are treated as a single field, with a value from zero to seven. When a page is accessed, the age field is set to seven. This value is decremented periodically by a system process. When the age value drops below a configurable threshold value, the page becomes eligible for swapping.

The *global* and *global swappable* bits are related to kernel memory. The global bit indicates that a page is part of kernel, rather than user, virtual address space. Global pages may be accessed by any process, but only while operating in kernel mode. The global swappable bit indicates that a page of kernel memory may be swapped out. If the global swappable bit is clear, the page must be kept in memory at all times.

The disk block descriptor contains two important fields: `dbd_type`, which determines how the system treats the page when it is being swapped in or out, and `dbd_pgno`, which holds the location of the page on disk. If there is a copy of the page on the file system, the value of `dbd_type` is DBD_FILE and the value of `dbd_pgno` is the logical page number of the page within the file. Recall from the previous chapter that if a region is associated with a file, the region structure contains a pointer to the file's in-core `inode`. Therefore, between the region structure and the disk block descriptor, the kernel has all the data it needs to locate a page on disk. If there is a valid copy of the page on the swap device, `dbd_type` is DBD_SWAP, and the value of `dbd_pgno` is the offset of the page in the swap file. Another field in the disk block descriptor, `dbd_swpi`, contains an offset into the swap-use table, indicating which swap file the page is stored in. Using these two data, the kernel can locate a swapped page on disk.

Other possible values for `dbd_type` include DBD_NONE, and DBD_DZERO. DBD_NONE indicates that there is no valid copy of the page, either on the swap device or the file system. DBD_NONE pages are usually working pages of data or stack. DBD_DZERO indicates a blank page. When a DBD_DZERO page is faulted in, a page of memory is allocated and zeroed out, and its type is changed to DBD_NONE. Demand zero pages are used to represent uninitialized data, and stack pages which have not yet been accessed.

The swap-use table keeps track of swap space. Each entry describes a single area of secondary storage—a "swap file." The swap-use table entry contains a use count for each page in the swap file, indicating how many processes are using the page in question. If the use count of a page is zero, no process is using it and it may be safely overwritten.

5.2 Integrated Data Cache

The integrated data cache can be thought of as two interrelated caches: the page cache and the buffer cache. The difference between these caches lies in the type of data they handle. It was mentioned previously that `pfdat` structures may be placed on a hash list. Only `pfdats` which represent pages of regular file data are hashed. This includes not only pages from executable files, but also pages accessed using normal file system I/O functions (*open*(2), *read*(2), and so forth), and pages of memory mapped files. These pages are said to be in the *page cache.*

Other forms of data are cached slightly differently. The kernel maintains a table of `buf` structures, or *buffer headers*. Each buffer header is associated with one or more blocks of data from a given device. These buffer headers are hashed like the `pfdat` structures, except that the hashing functions is based on the device number and physical block number of the associated data (as opposed to in-core `inode` number and logical page number). Examples of data that might be hashed in this buffer cache include chunks of EFS bitmaps, blocks from a tape device, and so forth. In short, all filesystem data except regular file data. This data is actually stored on pages from the system page pool.

Buffer headers are used by the I/O subsystem as handles for pages of data being read from or written to peripheral devices. When dealing with a page of regular file data, a buffer header is simply "borrowed" from the buffer cache (that is, the first available buffer is taken and removed from any hash list it may be on). This "borrowed" buffer header is then associated with the page for the duration of the I/O operation.

5.3 Duplicating Processes Regions

The *fork* system call creates an exact duplicate of the calling process. The new process is called the *child* process and runs exactly as the parent process does. On receipt of the *fork* call the IRIX kernel copies the process's regions, using the *dupreg* routine, as mentioned in Chapter 4. For shared read-only regions, *dupreg* simply increments the region reference count of the shared regions.

For regions with read and write permission such as data and stack, the kernel makes a copy of the `region` structure and page tables and then examines each parent table entry: if a page is valid for the parent process, it increments the reference count in the pfdat table entry, indicating that the number of regions that share the page has grown sets the page's copy on write bit, and clears the page's modify bit. If there is a copy of the page on the swap device, the kernel increments the swap-use table reference count for the page.

When a page is made copy on write during a *fork* system call, it can be referenced through both the parent and child regions, which share the page until one process writes to it. If either process writes the page, it incurs a protection fault on that page (because the page's modify bit has been cleared). The protection fault handler will determine that the page's copy on write bit is set, and then check the reference count on the page. If the reference count is one, then the other process must have called *exit* or *exec*, and is no longer using the page. In this case, the protection fault handler will simply clear the copy on write bit, and make the page writable. If the reference count is greater than one, the protection fault handler will make a private, writable copy of the page for the process incurring the fault, and decrement the reference count on the original page. The physical copying of the page is thus deferred until it becomes necessary.

5.4 Running New Programs

The memory management subsystem works closely with the process subsystem when *exec*ing a new program, since an entirely new process image has to be created—which requires the manipulation of `pregion` and `region` structures, page tables, and all the associated memory management data structures.

Logically, the *exec* routine does the following: detaches all of the process's old memory regions, attaches new regions for the new program's text, data, and stack, and allocates page tables. The *exec* routine also sets up the disk block descriptors in the new page tables so that the fault hander will know how to fault the pages in (for example, dbds in the text region will be marked DBD_FILE, to indicate that they should be read in from the filesystem).

IRIX does perform some optimizations to speed up the *exec* routine. In particular, if one of the old process regions is not shared with any other process, IRIX doesn't detach the region. Instead, it shrinks the region to zero, thus getting rid of any pages and page tables associated with the region, then resizes the region to the size required by the new process. In this way, the kernel saves the steps of detaching and deallocating the old region, and allocating and attaching a new one. These operations are particularly expensive on a multiprocessor machine, where the kernel must lock the region list in order to locate a new region, and move it from the free list to the active list.

5.5 Maintaining Free Pages

The page stealer, *vhand*, is a special system process that maintains a supply of free pages for the system. The page stealer swaps out memory pages that have not been accessed by any process for a given amount of time. The page stealer also trims processes which are using too much memory. The number of pages a process has in memory at any one point is called its *resident set size*. When process's resident set size exceeds a certain limit, the page stealer swaps out pages from that process until the process's resident set size is reduced below the limit.

The kernel creates the page stealer during system initialization and wakes it up periodically to check for memory hogs and low free memory. Like several other system daemons, the page stealer runs in kernel mode, so that it can manipulate kernel data structures, and it runs at high priority, so that it will take precedence over all ordinary user programs. When the page stealer is woken, it first scans the active process list for processes which have exceeded the allowable resident set size. If it finds any, it trims their resident sets by swapping out pages. Then, if the amount of free memory is below a certain point, the page stealer goes through the process of "aging" pages and swapping out pages which haven't been used recently. To do this, the page stealer examines every active unlocked region, skipping locked regions in expectation of examining them during its next pass through the region list. For each region, the page stealer calls the routine *ageregion*, which decrements the age fields of all valid pages. Whenever a page is accessed, its age field is set to all ones (for a value of 7).

There are three states for a page in memory:

1. The page is valid and referenced.
2. The page is aging and is not yet eligible for swapping.
3. The page is eligible for swapping.

The pages in the first two states have been recently accessed, and are therefore presumably likely to be needed again in the near future. These pages are said to be in a process's *working set*. When a page is in the first state, its valid and referenced bit and its software valid bit are set, and its age field is set to 7.

A page in the second state has the valid and referenced bit cleared, and the value of its age field is less than seven, but greater than a certain threshold value. (This value of is a tuneable parameter, which may be changed by the system administrator using the *syssgi*(2) system call.)

When the value of the age field drops to the threshold value or below, it enters the third state, eligible for swapping.

The *ageregion* routine clears the valid and referenced bit and decrements the age field. Thus, when a page in the first state is aged, it goes into the second state. If it is aged several more times without being referenced, *ageregion* will place it in the third state, eligible for swapping.

If two or more processes share a region, they all update the same reference bits of the pfdat. Pages can thus be part of the working set of more than one process, but the page stealer ignores this. If a page has been accessed recently enough to be in the working set of one or more processes, it will not be removed from memory; if it has not been accessed, it does not matter how many processes reference it, it will be made eligible for removal. Also, the page stealer makes no attempt to remove equal numbers of pages from every region of memory. All decisions are based on the last reference of a page.

The page stealer starts aging pages when the number of pages on the free list falls below a certain point. This number is a tunable parameter, which may be changed by the system administrator using the *syssgi* system call. The page stealer then removes pages that have not been used until the number of free pages rises above another configurable number. These thresholds are used to prevent the page stealer from running constantly. If the page stealer were to use only one threshold, it would remove enough pages to turn itself

off and the kernel would then begin faulting pages back into memory. The number of free pages would soon drop below the threshold again. The page stealer would constantly be orbiting around the threshold number of free pages and would cause considerable system overhead. By swapping out pages until the number of free pages exceeds a high threshold, the supply of free pages will remain good for some time. This limits the time that the page stealer is running and reduces system overhead.

When the page stealer has to select pages for swapping, it does so by selecting pages from low priority processes first. It does this by building a sorted binary tree, where each node of the tree contains a process ID, a priority number, and pointers to two other nodes. This tree is sorted by priority, and the page stealer "walks" it in order, from low priorities to high priorities, stealing eligible pages from each process until the amount of free memory exceeds the upper threshold. Thus, the page stealer does not need to swap pages from high priority processes unless memory use is very heavy.

When the page stealer decides to swap a page, it examines the page's disk block descriptor to determine how to swap it out:

1. If there is no copy of the page on the swap device or file system (DBD_NONE), the page stealer places the page on a list of pages to be swapped and the page stealer goes on to other regions. This effectively completes the swap except for the actual mechanics of copying. The page state goes from DBD_NONE to DBD_SWAP.

2. If there is already an exact copy of the page on a swap device (DBD_SWAP), the kernel places the page on the free list and updates the disk block descriptor and the swap-use table. In this case the page is already marked DBD_SWAP, and this does not change.

3. If there is a copy of the page on the file system (DBD_FILE), the page stealer will free the page and update the data structures to retrieve the page from the file system if it is needed. In this case the page state remains DBD_FILE.

When a page is removed from memory, the software valid bit in its page table entry is cleared, so that the kernel knows that it is truly invalid (as opposed to a page for which the valid and referenced bit has been cleared to detect a reference). The `pfdat` that represents the page is placed on the free list, but is effectively locked until the page has actually been swapped. (The `pfdat` is marked busy until the data on the page has been written out.

This prevents the page from being reused before the data on it has been written to disk.)

For an example of the above rules, consider a page that exists only on a swap device and is copied into memory after a process incurs a validity fault. IRIX does not automatically remove the swap copy. If the page is not used for some time, it will be swapped again. If the page was not modified while it was in memory, the memory copy will be identical to the swap copy and the page can simply be placed on the free list and the swap-use table updated. If the page was written and changed, its association with the copy on the swap device has been broken. In this case, a new copy of the page must be written to the swap device.

Not every page of a process is swapped out when the page stealer identifies removable pages. If a page is currently in use, it is not swapped out. The page stealer makes no attempt to remove pages evenly between processes. It is part of the nature of the system that only rarely does one process hog memory to the detriment of others.

IRIX makes every transfer between memory and swap as large as possible. If the hardware cannot transfer all the queued pages in one operation, the software must iteratively transfer the pages in blocks. The exact rate of data transfer and the physical actions taken depend on the individual machine and usage. For instance, since IRIX memory is organized in pages, the data to be swapped out is not likely to be contiguous in physical memory. IRIX gathers the page addresses of data to be swapped out, and the disk driver uses the collection of page addresses to set up the operation. IRIX waits for each swap operation to complete before swapping out other data. If no swap device contains enough contiguous space to hold the block of pages, the kernel swaps out one page at a time. This is clearly more costly in terms of system overhead. There is also more fragmentation of swap space under the IRIX paging scheme than in a process swapping scheme, because IRIX swaps out blocks of pages but swaps in only one page at a time.

IRIX does not write the entire virtual address space of a process to a swap device. Instead it copies physical memory pages assigned to a process to the allocated space on the swap device, ignoring portions of the program that have never been read into main memory. When IRIX copies the pages back into memory, it reassigns the pages to the correct kernel virtual addresses. This scheme is transparent outside the kernel. The physical addresses of memory are thus completely malleable with no visible change to the user.

The page stealer doesn't keep the region locked any longer than it has to, so the region lock is released once the page stealer is through manipulating its page tables.

In theory, all non-kernel pages, and some dynamically allocated kernel pages, may be swapped out by the page stealer. In addition, if available memory becomes very low, the system may swap out entire processes, including their user blocks and kernel stacks. This is performed by a different daemon, called the memory scheduler (not to be confused with the process scheduling routine described in Chapter 4). The memory scheduler is described in more detail in the section on "Process Swapping," below.

When IRIX swaps a page out, the *software valid* bit in the page table entry is turned off and the reference count in its pfdat is decremented. If no more processes reference the page, the page is added to the free list, to be saved until it can be used again. If the count is positive, at least one process is still using the page, but the kernel still swaps the page out. But the pfdat entry is not added to the free list. Finally, the kernel allocates space on the swap device to receive the page, places the address of the allocated space in the disk block descriptor, and adds 1 to the swap use table count for the page. If a process needs the page while it is on the free list, IRIX can pull the page from the free list instead of having to read it from the swap device. Still, the page is always swapped if it is on the swap list.

It is important to note that if the page stealer is unable to allocate swap space to store a page, then the process owning the page will be killed. This may result in the "innocent" processes being killed, but it only happens under extreme circumstances.

5.5.1 Summary of Page Swapping

There are two phases involved when a page is swapped. First, the page stealer finds the page has not been used recently and places it on a list of pages that can be swapped. Next, IRIX copies the page to a swap device when convenient, zeros the *valid* bit in the page table entry, decrements the pfdat reference count, and places the pfdat entry at the end of the free list if its reference count is 0. The contents of the physical page in memory are unchanged until the page is reassigned. This allows the page to be retrieved if it is needed before it is reused.

5.5.2 Process Swapping

Under some conditions, the demand for memory will become great enough that it is worthwhile to swap out entire processes. As mentioned above, the memory scheduler is responsible for these transactions. When memory is needed, the memory scheduler, *sched*, will find processes which are sleeping and swap out their user blocks, kernel stacks, and page tables. The memory scheduler will swap processes back in when memory become available. If memory continues to be congested, the memory scheduler will make sure that processes don't spend to much time swapped out, by swapping more sleeping processes out, and swapping processes back in on a first-in, first-out basis.

5.6 Page Faults

IRIX has two types of page faults: validity faults and protection faults. Since these mechanisms are closely related to address translation, we should briefly review how address translation works under IRIX. As mentioned in Chapter 2, there are four different classes of virtual addresses: k0seg, k1seg, k2seg, and kuseg. The first two classes are direct-mapped, meaning that the the addresses map directly to virtual addresses (k0seg and k1seg addresses may be translated to physical addresses by subtracting a constant). k2seg and kuseg addresses are mapped via the memory management hardware (the TLB). Addresses in k2seg may be mapped using either "wired" TLB entries, or page tables. K2seg addresses are used for such data structures as page tables, user blocks, and so forth. The kernel maintains a special region structure, the *system region*, which holds page tables mapping k2seg pages. All kuseg addresses are mapped using page tables.

As was mentioned in Chapter 4, the TLB acts as a cache of virtual to physical address translations. However, the exact mechanics of the TLB are not crucial to this discussion, and it is sufficient to pretend that the memory management hardware consults the page tables directly.

5.6.1 Validity Faults

A validity fault occurs when the system tries to access a virtual page which has been marked as invalid (that is, valid and referenced bit has been cleared). There are several reasons that this might happen:

1. The page table entry has been marked invalid to detect a reference. In this case, the valid and referenced bit is cleared, but the software valid bit is set, indicating that the page table entry contains a valid virtual to physical page mapping.
2. The page table entry has been marked invalid because the page is not in core. In this case, the software valid bit has been cleared, to indicate that there is no valid physical page mapped by the page table entry. Pages which fall into this category are either demand zero, on the filesystem, or swapped (DBD_DZERO, DBD_FILE, or DBD_SWAP).
3. The page table entry represents an unassigned page. This may occur because page tables entries are allocated one segment at a time (a page worth of page tables, mapping 2 megabytes of memory). If a process region doesn't end conveniently on a segment boundary, the remainder of its last segment may be filled with this type of page table entry, with the software valid bit cleared and the dbd flagged DBD_NONE.

When a validity fault occurs, the fault handler obtains the faulting address from the IRIS hardware and finds the corresponding page table entry. The fault handler then locks the region containing the page table entry to prevent other processors from manipulating it, and determines why the page was marked invalid.

If the software valid bit in the page table entry is set, then the page has merely been marked invalid to detect a reference, as described in the section on page swapping, above. In this case, the fault handler simply resets the age bits and the valid and referenced bit, and returns.

If the page table entry represents an unassigned page, then the fault handler simply returns an error, and the process is sent a *segmentation violation* signal.

If neither of the above cases is true, then the fault handler may have to do some actual work. The page is demand zero, or else there is a copy of it on disk, either on the swap device or on the file system. If the page is demand zero, there are no two ways of going about it—a page must be allocated off the free list and zeroed out, and the page table entry and disk block

descriptor filled in to reflect the fact that a page has been allocated. If, however, the page is marked DBD_SWAP or DBD_FILE, there is a chance that it is still in the page cache, on the free list awaiting reuse. So the fault handler checks the appropriate hash list, and if it finds the page, simply removes it from the free list, sets the software valid bit in the page table entry, and returns.

If the page is DBD_FILE or DBD_SWAP and can't be found in the page cache, then the fault handler will really have to go to disk. A page is allocated and locked down, the "no shrink" flag is set in the region structure to prevent the region from shrinking while file I/O is taking place, and the I/O operation is initiated. The faulting process is put to sleep until the I/O is finished, and the fault handler returns (at which point the kernel must find another process to run). The newly allocated page is placed on the appropriate hash list *before* the I/O operation begins with a special I/O wait flag set in its `pfdat` structure. This way, if another process tries to fault in the page, the fault handler will find that the page is on the hash list, but is waiting for I/O. The fault handler can then put the process to sleep until the page is ready. When the file I/O completes, all the processes waiting for it will be woken up.

The kernel does not always have to read in a page to handle a validity fault, even though the disk block descriptor indicates that the page has been swapped. It is possible that the kernel had not yet reassigned the page after writing the contents of out to the swap device, or that another process had faulted the same page into another physical page. In either case, IRIX finds the page in the page cache by searching the appropriate hash list. It reassigns the page table entry to point to the copy of the page just found in memory, increments the page reference count, and removes the page from the free list, if necessary. For example, a process faults on a page. Searching the hash list, the kernel finds a copy of the correct page. It resets the page table entry for the virtual address to point to the valid page, sets the software valid bit, and returns.

5.6.2 Protection Faults

The second kind of memory fault that a process can incur is a *protection* fault, meaning that the process tried to write to an out-of-range virtual address, or to a page with the modify bit set in its page table entry. There are several reasons for a page to have its modify bit set:

1. The page has been marked copy-on-write.
2. The page has had its modify bit set to catch attempts at modifying it. This is used, for example, when there is a copy of the page on the swap device. If the page is modified, it will no longer be identical to the copy on the swap device, so the association with the swap copy must be broken. This is performed by the protection fault handler.
3. The page is actually read-only (for example, a page in a text region).

The IRIS hardware supplies the protection fault handler with the virtual address where the fault occurred, and the fault handler finds the appropriate region and page table entry. It locks the region so that the page stealer cannot swap the page while the protection fault is handled.

If the region in which the protection fault occurred is marked read-only, then the protection fault handler simply sends a signal to the faulting process, to indicate that it has tried to write to a read-only page.

If the fault handler determines that the fault was caused because the copy on write bit was set, and if the page is shared with other processes, the kernel allocates a new page and copies the contents of the old page to it; the other processes retain their references to the old page. After copying the page and updating the page table entry with the new page number, the kernel decrements the reference count of the old pfdat table entry. For example, assume two processes share page 375. Process B attempts to write the page but fails because the copy on write bit is set. The protection fault handler allocates page 400, copies the contents of page 375 to the new page, decrements the reference count of page 375 because process B will now be using page 400, and updates the page table entry accessed by process B to point to page 400.

If the copy on write bit is set but no other processes share the page, there is no need to make the copy. This may occur because a process has *fork*ed, and then *exit*ed or *exec*ed another program, thus disassociating itself from its old process image. In this case the protection fault handler simply clears the copy on write bit. If there is a copy of the page on the swap device, it disassociates the page from the swap copy, since the page is being modified and will no longer be identical to the swap copy. Then, it decrements the swap-use count for the page, and if the count drops to 0, frees the swap space.

As noted, the modify bit is also used to detect modification in order to break the association between a page and a swap copy. For this purpose, the

modify bit is set when pages are swapped. If the protection fault handler finds that the page is marked DBD_SWAP, it will disassociate the page from the swap copy by changing it to DBD_NONE, and decrementing the use count on the swap copy. If the use count on the swap copy drops to zero, swap space is freed.

5.7 Chapter Summary

This chapter has outlines the methods used under IRIX for memory management. The implementation of demand paging allows processes to execute even though their entire virtual address space is not loaded into memory; therefore the virtual size of a process can exceed the amount of physical memory available in the system. When the kernel runs low on free pages, the page stealer goes through the active pages of every region, marks pages eligible for swapping if they have aged sufficiently, and eventually copies them out to a swap device. When a process addresses a virtual page that is currently swapped out, it incurs a validity fault. The kernel invokes the validity fault handler to assign a new physical page to the region and copies the contents of the virtual page to main memory.

With the implementation of the demand paging algorithm, several features improve system performance. First, the kernel uses the copy on write bit for forking processes, eliminating the need to make physical copies of pages in most cases. Second, the kernel can read in pages of an executable file from the file system, eliminating the need for *exec* to read the file into memory immediately. This helps performance because such pages may never be needed during the lifetime of a process, and it eliminates the extra thrashing that would be caused if the page stealer were to swap such pages from memory before they were used.

6. The Input/Output Subsystem

The IRIX input/output subsystem is very similar to that used by System V. Except as specified below, the IRIX IO subsystem conforms to the model described in Chapter 10 of Bach's *Design of the UNIX Operating System.*

6.1 Device Drivers for Multiprocessor Machines

The driver kernel interface used by IRIX is essentially the same as the driver kernel interface described by Bach. However, writing a device driver for IRIX is more complicated because IRIX machines may have more than one processor. There are two ways to make a normal device driver work on a multiprocessor machine. The driver may be bound to a particular processor, or it may use locks and semaphores to maintain data integrity.

When a device driver is bound to a particular processor, the device driver code will only run on that processor. This allows a device driver to work on a multiprocessor machine without modification. If a process running on another processor tries to access the device, it will be put to sleep until it can be scheduled on the appropriate processor.

The other way to make a device driver work on a multiprocessor system is to implement a locking protocol like that used in other parts of the kernel, using spinlocks and semaphores to make sure that critical code regions are single threaded.

7. Interprocess Communication

IRIX supports System V style interprocess communication mechanisms (semaphores, message queues, and shared memory) as well as BSD sockets. The System V facilities are described in Bach, Chapter 11. Sockets are described in Leffler, et al., Chapter 10.

In addition to these facilities, IRIX provides low-level synchronization primitives called spinlocks.

7.1 Spinlocks

Spinlocks are simple busy-wait locks. A process attempting to acquire a spinlock continuously tries to acquire the lock until it succeeds.

On multiprocessor IRIX machines, spinlocks are implemented in hardware. These hardware spinlocks are used throughout the kernel to protect vital data structures and code regions, including the data structures for the other IPC facilities. This allows the System V facilities, as described in Bach, to function normally in a multiprocessor environment.

On sincle processor IRIX machines, spinlocks are implemented in software, using the same algorithm normally used for semaphores.

7.2 Sockets

The IRIX implementation of BSD sockets corresponds in most details to the implementation described in *The Design and Implementation of the 4.3 BSD UNIX Operating System.* The main difference between the two is the manner in which sockets are integrated into kernel. In BSD, extra data is

added to the kernel open file table. Each file structure has a flag indicating whether or not it represents a socket, and a pointer to a `socket` data structure. Under IRIX, sockets are implemented as a file system type (as described in Chapter 3 of this book). Each socket is associated with an in-core inode, and this generic inode contains a pointer to the `socket` structure (recall from Chapter 3 that each `inode` structure contains a pointer to a file system dependant data structure). Thus, operations on sockets are switched through the file system switch table, just like regular file system operations.

8. Networking

IRIX networking functions exactly like BSD networking, as described in Leffler, et al., Chapters 11 and 12. The only differences lie in the implementation of sockets, as described in the previous chapter. On multiprocessor systems, the networking device drivers are constrained to run on a specific processor. Therefore, they do not need to implement any special locking protocol to operate in the multiprocessor environment.

IRIX also implements the Sun Network File System (NFS). This system is described in *TCP/IP and NFS: Internetworking in a UNIX Environment*, by Michael Santifaller. IRIX implements NFS as a file system type, so that operations on NFS mounted file systems pass through the File System Switch, as described in Chapter 3.

Index

F

G

Q

R

S

U

V

W