AIX/6000 Internals and Architecture

J. Ranade Workstation Series

BAMBARA, ALLEN • *PowerBuilder: A Guide for Developing Client/Server Applications*, 0-07-005413-4

CERVONE • *AIX/6000 System Guide*, 0-07-024129-5

CHAKRAVARTY • *Power RISC System/6000: Concepts, Facilities, and Architecture*, 0-07-011047-6

CHAKRAVARTY, CANNON • *PowerPC: Concepts, Architecture, and Design*, 0-07-011192-8

CHAPMAN • *OS/2 Power User's Reference: From OS/2 2.0 Through Warp*, 0-07-912218-3

DEROEST • *AIX for RS/6000: System and Administration Guide*, 0-07-036439-7

GRAHAM • *Solaris 2.X: Internals and Architecture*, 0-07-911876-3

HENRY, GRAHAM • *Solaris 2.X System Administrator's Guide*, 0-07-029368-6

JOHNSTON • *OS/2 Connectivity and Networking: A Guide to Communication Manager/2*, 0-07-032696-7

JOHNSTON • *OS/2 Productivity Tool Kit*, 0-07-912029-6

LAMB • *MicroFocus Workbench and Toolset Developer's Guide*, 0-07-036123-3

LEININGER • *AIX/6000 Developer's Tool Kit*, 0-07-911992-1

LEININGER • *HP-UX Developer's Tool Kit*, 0-07-912174-8

LEININGER • *Solaris Developer's Tool Kit*, 0-07-911851-8

LEININGER • *UNIX Developer's Tool Kit*, 0-07-911646-9

LOCKHART • *OSF DCE: Guide to Developing Distributed Applications*, 0-07-911481-4

PETERSON • *DCE: A Guide to Developing Portable Applications*, 0-07-911801-1

RANADE, ZAMIR • *C++ Primer for C Programmers, Second Edition,* 0-07-051487-9

ROBERTSON, KOOP • *Integrating Windows and Netware*, 0-07-912126-8

SANCHEZ, CANTON • *Graphics Programming Solutions*, 0-07-911464-4

SANCHEZ, CANTON • *High Resolution Video Graphics*, 0-07-911646-9

SANCHEZ, CANTON • *PC Programmer's Handbook, Second Edition,* 0-07-054948-6

SANCHEZ, CANTON • *Solutions Handbook for PC Programmers, Second Edition,* 0-07-912249-3

WALKER, SCHWALLER • *CPI-C Programming in C: An Application Developer's Guide to APPC*, 0-07-911733-3

WIGGINS • *The Internet for Everyone: A Guide for Users and Providers*, 0-07-067019-8

AIX/6000 Internals and Architecture

David A. Kelly

McGraw-Hill

New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging-in-Publication Data

Kelly, David A. (David Allen)
AIX/6000 internals and architecture / David A. Kelly.
p. cm.—(J. Ranade workstation series)
Includes index.
ISBN 0-07-034061-7
1. AIX (Computer file) 2. Operating systems (Computers) 3. IBM RS/6000 Workstation. I. Title. II. Series.
QA76.76.063K452 1996
005.4'469—dc20 95-25794
CIP

McGraw-Hill
A Division of The McGraw-Hill Companies

1 2 3 4 5 6 7 8 9 0 AGM/AGM 9 0 0 9 8 7 6

ISBN 0-07-034061-7

The sponsoring editor for this book was Jerry Papke, the editing supervisor was Fred Bernardi, and the production supervisor was Pamela Pelton. It was set in Century Schoolbook by Renee Lipton of McGraw-Hill's Professional Book Group composition unit.

Printed and bound by Quebecor/Martinsburg.

This book is printed on acid-free paper.

To my parents

Contents

Preface

AIX (Advanced Interactive eXecutive) is IBM's primary contribution to the open operating system market. Its popularity has grown steadily since its introduction in 1986. Combined with the outstanding performance of the RISC System/6000 line of servers and workstations, AIX Version 3 has been honored with industry praise. Since AIX is offered on the new POWER chip generation of personal computers, the future of this operating system is solid.

The kernel of the AIX Version 3 operating system is based on the System V, Release 2 version of AT&T's UNIX. It also includes BSD (Berkeley) enhancements, as well as many new features introduced by IBM. Some of these features include the Journaled File System (JFS) and support for logical disk partitioning (the Logical Volume Manager). IBM has also implemented file I/O routines through the operating system's virtual memory management subsystem. Because of these enhancements, study of the AIX kernel must be approached from a unique perspective of combining traditional UNIX concepts with new techniques.

This book provides detailed information about the AIX Version 3 kernel and its subsystems. It gives systems programmers an understanding of kernel components and extensions, as well as techniques for synchronizing access to global kernel data structures. It supplies valuable information for applications programmers who wish to write code that makes better use of kernel services and system resources. Finally, it helps system administrators understand the concepts behind system parameters.

This book is organized into the following chapters:

Chapter 1, How to Study the AIX Kernel, introduces the AIX Version 3 kernel and its subsystems. It explains how one can learn about the internal workings of the operating system without having access to the source code. This is accomplished by examining system header files and using various tools, such as debuggers and performance-monitoring facilities. It also provides a brief overview of C language concepts that must be understood in order to read the header files.

Chapter 2, An Overview of the AIX 3.2 Operating System, gives a description of the operating system and its applications. It explains the implementation of system administration utilities such as the system management interface tool (SMIT), the object data manager (ODM), the logical volume manager (LVM), and the queueing system. It describes characteristics of the AIX kernel, such as how it is preemptable, pageable, and dynamically extendable. It also includes the history of AIX.

Chapter 3, AIX 3.2 Programs and Processes, illustrates the structure of an AIX program as defined by the XCOFF model. It also describes different types of processes and details their virtual memory images. Dynamic binding is explained as part of a discussion of program creation in AIX. Finally, the different techniques used by AIX Version 3.1 and AIX Version 3.2 for dynamically allocated memory are explained.

Chapter 4, AIX 3.2 Memory Management, provides information on how AIX manages physical and virtual memory. It introduces the concepts of persistent storage and working storage. It details virtual memory management data structures such as the page frame table, segment control blocks, and external page tables. It explains how the virtual memory manager translates effective addresses into virtual addresses and real addresses, as well as how the page stealer allocates memory to popular program and file pages.

Chapter 5, The Process Management Subsystem, lists and describes process attributes. It details the life cycle of a process, process states, and process scheduling. It also explains how AIX performs context switches and how interrupts are handled. Real-time programming is introduced.

Chapter 6, The Journaled File System, describes in detail IBM's variation of the Berkeley fast file system. This chapter illustrates the layout of the JFS and explains how it logs changes to its own data structures, thus reducing the likelihood of file system corruption when a system crash occurs. JFS components, such as files, directories, inodes, and super blocks are examined.

Chapter 7, AIX 3.2 Disk File I/O, explains how the AIX kernel handles local disk file I/O. The kernel's file table and in-core inode table are described, as well as lock lists and memory mapped files. Many of the file I/O system calls, such as open(), read(), write(), and lseek() are discussed from the kernel's perspective.

Chapter 8, Virtual File Systems, examines the kernel components used to support various types of physical file systems. Vnodes, vfs and gfs structures, as well as vnodeops and vfsops are described. The vmount table is also discussed.

Chapter 9, The Device I/O Subsystem, lists and describes the components of device I/O. It explains the role of the file system and how AIX implements special block and character device file names. The device switch table is

introduced and a detailed discussion of device driver top and bottom halves is provided.

Chapter 10, Interprocess Communication, describes how the kernel implements various means of sharing information between processes. Signals, pipes, shared memory, semaphores, and message queues are defined from an application's perspective and from the kernel's perspective.

Acknowledgements

So many people have contributed to this book. Some have given their technical support while others have provided the encouragement I needed to undertake and complete this project. They all have my sincere thanks.

I would like to express my deepest gratitude to Jay for his guidance and patience, as well as to Fred Bernardi, my editor, for putting this book together as quickly as possible, despite the fact that I kept missing deadlines.

I would also like to thank Elizabeth Johnstone, my business partner for the past four years for helping me find the time to work on this book, as well as Tim Armbrust for convincing me to write it.

Many thanks go to the people who have enhanced my understanding of AIX over the years. The list includes Gary Wilson, Frank Edwards, Mike Heinrich, Bobby Higgins, and Harvey and Jackie Metté, as well as many of my former coworkers at IBM, whose names must go unmentioned. You know who you are.

I must also thank the hundreds of students I have had the pleasure of teaching over the past thirteen years. Their questions helped expand my technical knowledge and provided many of the "war stories" you'll read in this book.

Special thanks go to Bill Scaling and all the nice folks at ProAmerica for letting me borrow their RISC System/6000 in times of need.

Finally, special thanks to my wife, Carolyn, for putting up with many lost evenings and weekends these past few months.

David A. Kelly

Chapter

1

How to Study the AIX Kernel

"AIX internals" is the study of the AIX kernel and kernel subsystems. This book provides a starting point for such a study. This chapter describes what is involved in the study of the AIX Version 3.2 kernel.

1.1 Layers of the AIX Operating System

UNIX is frequently described by its functional layers. The same can be said for AIX. Figure 1.1 describes the components of the AIX 3.2 system. At the center of the illustration is the system hardware. This would represent the RISC System/6000, PowerPC, or any computer designed to support the AIX operating system. While the hardware is not part of AIX, the goal of any operating system is to provide the link between the applications and the hardware, so it is shown here.

The kernel is the operating system itself. All operating systems have two primary responsibilities: to facilitate I/O, forming the link between the appli-

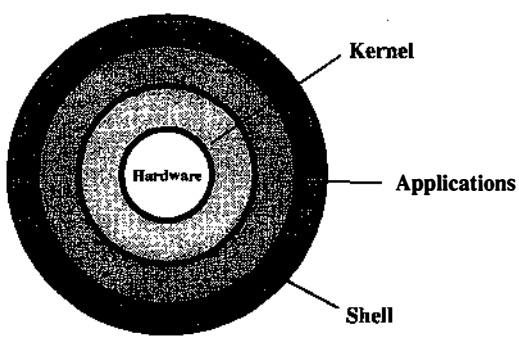

Figure 1.1 Layers of the AIX operating system.

cations and the system hardware, as described above, and to provide support for executing programs.

> **Author's Note:** AIX, like all UNIX-based operating systems, makes a distinction between an executable file and a program that is executing. The term "program" refers to an executable file stored on disk or some other medium. When a program is loaded into memory and executed, it is called a "process." To illustrate the difference between a program and a process, a system might have a program named "/usr/local/bin/games/ds9." This file contains the program. If eight users on the system decide to play the game at the same time, there will be eight processes executing "ds9." Each process has its own identity, attributes, and private data. Details on programs and processes are provided in Chap. 3.

The AIX kernel supports device I/O and file I/O. Device I/O allows an application to open a device, then interface with the device by reading or writing data. An example of device I/O would be opening a tape drive, then writing data to the drive. The user's terminal interface also represents device I/O.

File I/O allows an application to store data for later retrieval. To this end, the kernel supports file systems that logically arrange files into directories. AIX 3.2 goes a step further by supporting different types of file systems, referred to as virtual file systems. Examples of virtual file systems include local disk file systems, remote file systems (file sharing via a client-server scenario), and CD-ROM file systems.

As is true with all UNIX-based systems, AIX implements device I/O through the File I/O Subsystem. This means that each device available to the system has a file representation in a file system. By convention, the abstract files that represent devices are found in the /dev directory. An application opens a device file in the same way as any other file. For instance, when an application opens the file "/dev/lp0," then writes data to the file, it is actually writing data to a printer.

> **Author's Note:** AIX 3.2 usually only allows the root user to access device files directly as described above.

AIX process management involves the loading and executing of programs, process scheduling for the CPU, memory management, and interprocess communication (IPC). The AIX 3.2 kernel includes routines for performing tasks associated with process management, as well as tables that hold information about active processes.

Figure 1.1 shows the application layer surrounding the kernel layer. The application layer represents user processes, which include any programs executed by system users. Applications are programs written by your local staff, programs purchased from vendors, and programs that come with the AIX operating system.

The shell is a special application that provides the user's interface to the system. AIX 3.2 includes the three most popular UNIX shells; the Bourne shell, the C shell, and the Korn shell. The Korn shell is the default shell for AIX 3.2 users. The shell serves as a command interpreter and includes a pro-

gramming language suitable for automating user and system administrative tasks. The shell is not discussed in this book.

> **Author's Note:** Figure 1.1 illustrates the layers of the AIX operating system. The circle drawing used is found in many UNIX books and training courses. My version differs slightly from others I have seen in that I prefer to show the shell outside of the application layer. Most other versions of the "circle drawing" show the shell surrounding the kernel, with the application layer on the outside. Actually, since the shell is an application itself, the outer two layers should be considered equal. My justification for placing the application layer within the shell layer is that applications use system calls to request services of the kernel. System calls are not made by users of the shell prompt.

1.2 The AIX 3.2 Kernel

Figure 1.2 provides another view of the AIX 3.2 kernel and its relationship to the application layer of the system. As mentioned earlier, UNIX-based systems assign file names to physical and logical devices. The system's physical memory is called /dev/mem. A portion of the physical memory is allocated to the kernel. The kernel's memory space is represented by the file /dev/kmem and is called "system memory." All remaining memory from /dev/mem is available for applications and is called "user memory."

> **Author's Note:** The kernel's memory space is treated as virtual memory since the AIX kernel uses paging space. Details on the virtual memory characteristics of the kernel are provided in later chapters of this book. The concepts are introduced here only as an overview.

The kernel contains code and data used to facilitate I/O and manage processes. The data are found in various system tables. For instance, the ker-

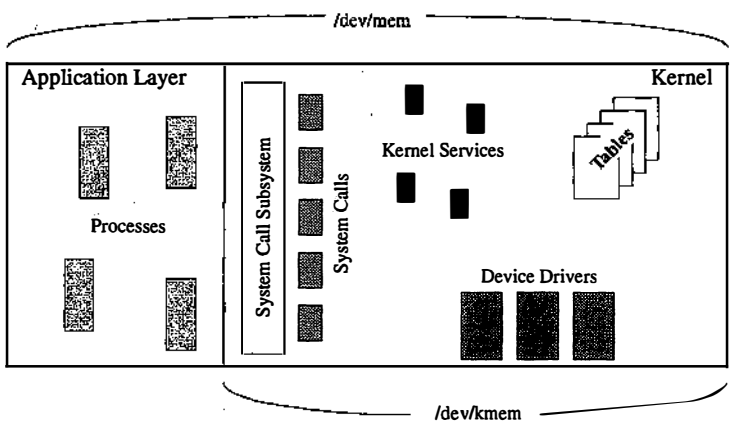

Figure 1.2 The AIX 3.2 kernel.

nel maintains a process table, which has one entry for each active process. Other tables include the file table, the in-core inode table, and the device switch table. These tables and their structures are the primary focus of this book.

The kernel uses device drivers to interface with the peripheral hardware of the system. Device drivers include routines for opening and communicating with the devices. Device drivers also handle interrupts from controllers when devices need the system's attention.

System memory is off limits to applications. In other words, applications cannot directly access data stored in the kernel. Applications request information and services from the kernel by issuing system calls. A system call is a routine found within the kernel. When executed, a system call runs on behalf of the calling process. AIX 3.2 includes a large number of system calls, which represent the kernel's application programming interface (API).

The AIX 3.2 kernel includes routines which are available to system calls and device drivers. These routines provide various kernel-level services. IBM calls these routines "kernel services." Kernel services are not available to application programs. In fact, application programmers need not know about the kernel services routines, but systems programmers (those writing device drivers, system calls, and other types of kernel extensions) must know how to use the kernel services. The InfoExplorer on-line documentation system includes descriptions of all AIX 3.2 kernel services.

How is a kernel created?

With many UNIX-based systems, the kernel must be rebuilt to load new device drivers or change system parameters. The rebuilding process involves modifying various configuration files, then recompiling and relinking these files to the kernel image file. A make file is usually included with the system to automate much of the kernel rebuild. The problem is that the system must be rebooted every time a new device driver is added.

The AIX operating system includes a dynamic kernel which need not be rebuilt to implement new device drivers, system calls, or other kernel extensions. Tunable kernel parameters can also be changed "on the fly."

Kernel source code

Most open systems vendors, including IBM, consider the kernel code of their operating system as proprietary. This is due to the fact that much of the kernel code is licensed from its original source, such as AT&T. The license agreements stipulate that the licensee must not divulge proprietary information about the kernel. Most AIX customers do not have access to the source code.

If one is lucky enough to have access to the source code for the various kernel subsystems, learning the internals is simply a matter of reading and properly interpreting the code. While the source code for AIX is available, it is very expensive. It is likely that the source code will not be at one's disposal.

Therefore, the approach taken by this book assumes that source code is not available. All concepts are discussed using resources commonly available on the standard system.

> **Author's Note:** As of this writing, IBM offers a source code license for AIX Version 3.1 at a price that would prohibit most customers from purchasing it simply to "find out how it really works." IBM does not offer the source code for AIX Version 3.2.

1.3 What Can I Do without the Source Code?

Without source code, one can still study the kernel by reading the documentation and examining the system header files that come with the system. Various utilities are available for displaying kernel structures and debugging kernel extensions. Even with these tools, however, some aspects of the kernel remain undocumented and unreachable from the user's perspective. This book relies on information obtained from those who developed the AIX kernel to fill in many of the gaps in the IBM published documentation.

AIX header files

Header files are C source code files that are included by the preprocessor when compiling programs. These header files (often called "include" files) usually define variables and structures, macros, and type definitions. System header files are included since some are required for system calls. They provide a wealth of information about the kernel.

The system header files are an optional install module as part of the application development toolkit (ADT). In other words, the system header files come with the system but are loaded only if specified during the installation process. If installed, they are mostly located under the /usr/include/sys directory. Figure 1.3 illustrates the directory tree for AIX 3.2 header files.

Some important header files include:

/usr/include/sys/types.h. This file includes many typedefs. Typedefs are definitions of new data types derived from standard types. (See Sec. 1.4 for an example of a typedef.) They make it easier to port application code from one UNIX-based system to another. Typedefs are illustrated throughout the header files discussed in this book.

/usr/include/sys/limits.h. This file defines many of the system limits, such as the maximum number of user IDs or the maximum number of characters in a file's path name. It also defines the system page size as 4096 bytes.

/usr/include/sys/param.h. This file defines many symbolic constants that are used throughout the system. Some of the definitions are made to provide compatibility with other UNIX-based systems. This file also defines many useful macros.

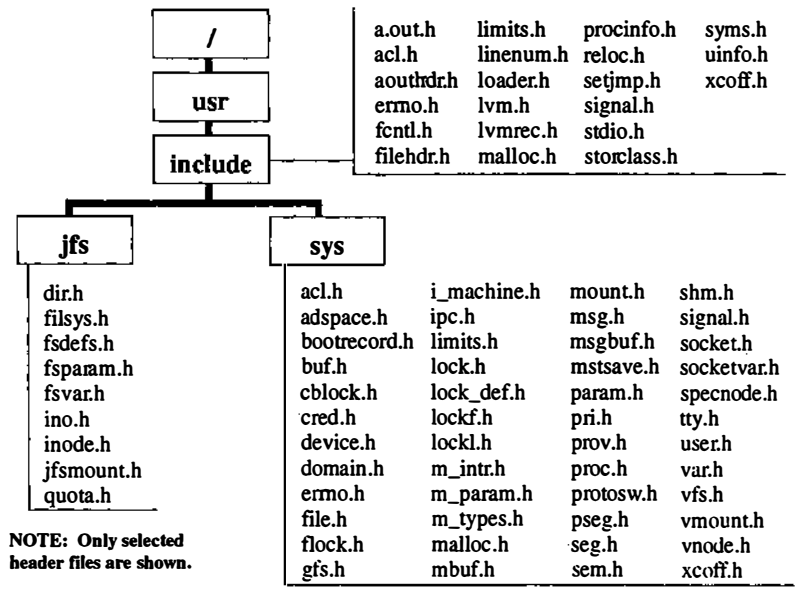

Figure 1.3 AIX 3.2 header files.

/usr/include/sys/m_param.h. This file defines parameters that are specific to a machine type, such as the number of general-purpose registers and floating-point registers.

> **Author's Note:** Any header file whose name begins with "m_" contains machine-specific definitions.

Not all header files are shipped with the AIX 3.2 system. IBM considers the header files that describe structures of the virtual memory manager (VMM) and logical volume manager (LVM) as proprietary. They will be mentioned in this book, but specific code examples are not given.

Tools

AIX 3.2 includes many tools designed to help programmers and system administrators examine and debug the kernel and other operating system components. The tools also provide a means of learning about the kernel and can be used to reinforce the concepts presented in this book. This section introduces some of these tools but is not intended to provide detailed information on their usage. Consult the InfoExplorer on-line documentation that comes with the AIX operating system for more details on using these tools.

Crash. One of the most useful tools for examining kernel structures and kernel data is the crash utility. It is an interactive facility that allows the user to

examine the image of the system's current kernel or to examine the image of a system dump. A system dump is the kernel image which is preserved by AIX 3.2 when the kernel panics and the system crashes. The crash utility is often used to try to determine the reason for a system crash.

Trace. The trace command, when activated, records kernel events with an extremely fine granularity of detail. The trace utility records the events to a memory buffer, which is written to a disk-based log file. Another command, trcrpt, is used to format the trace log into a report. Many kernel routines already contain trace hooks, so running trace only involves the collection and recording of data which are already available. It is possible for systems programmers to add their own hooks to the device drivers and other kernel extensions they create. The biggest problem with trace is that it collects a large amount of data very quickly. Sorting through a trace report can be very tedious. Fortunately, trace has options for specifying the desired events on which to report.

> **Author's Note:** The trace facility is often a last resort for isolating a performance problem or other system anomaly. It is not uncommon to run trace for as little as 30 seconds and collect over 10 megabytes worth of kernel event information! It is very useful, though, for determining exactly what the kernel is doing in a chronological fashion.

The low-level kernel debugger. While the capabilities of this tool are similar to those of crash, it is used primarily to debug kernel extensions. The kernel debugger can be run only with the system in maintenance mode.

fsdb. The file system debugger, fsdb, is a utility found on most UNIX-based systems. It allows the user to examine file system structures and trace links between those structures. It also allows the user to change file system control data, and should be used with extreme caution. Only root can run fsdb.

svmon. When it comes to looking at the virtual memory manager information, svmon provides a great deal of help. This tool was developed exclusively for AIX by a team of IBM programmers in Austin who were responsible for creating performance management utilities. The svmon command allows the user to look at virtual memory segments, the kernel's page frame table, and other components of the VMM.

Other tools that can aid in the understanding of the AIX 3.2 kernel are too countless to mention here but are described throughout this book.

IBM documentation

Most of the documentation for AIX 3.2 is provided on-line via the InfoExplorer hypertext retrieval system. It includes the Command Reference Guide for user-level commands, the Technical Reference Guide for library subroutines and system calls, general guides for communications and graphics programming, as well as documentation for system management. The

InfoExplorer data set can be installed on hard drive or is available on CD-ROM. The CD-ROM version of InfoExplorer, while a little slower when accessing data, contains all of the AIX 3.2 documentation and saves disk space. In addition to InfoExplorer, all AIX 3.2 documentation is available in hard copy form from IBM.

Another documentation option for AIX is the series of books produced by an IBM organization called the International Technical Support Center (ITSC). The books are called "red books" because of the color of their covers. They provide valuable "how to" information, which complements the traditional reference style of the InfoExplorer documentation. Ask your IBM representative for information on the ITSC red books.

Using InfoExplorer

The InfoExplorer program supports a graphical mode and a character (ASCII) mode. The graphical mode, which provides navigation windows and mouse support, is much easier to use. The program is invoked via the "info" command.

The InfoExplorer main navigation screen includes buttons to access information via "Tasks," "Commands," "Programming Reference," and "List of Books." A search option is also provided. Users wanting information on the topics discussed in this book will most likely want to access the "Programming Reference" or "List of Books" options. Under the "List of Books" option, the book called "Programming the Base Operating System" includes manual pages for all of the library routines and system calls. The book called "General Programming Guide" contains information about system memory allocation and the process image.

Perhaps the most useful InfoExplorer option is the search facility. A simple search involves scanning all InfoExplorer data bases for matches of the user-supplied string(s). The complex search option provides the ability to look for text matches that meet specified criteria, such as exact combinations of strings.

The best feature of InfoExplorer is its hypertext links. This allows the user to jump from one document to another by clicking on a jump point (any text surrounded by a box). This is useful in those "See also..." situations.

1.4 How to Read a Header File

This section is appropriate for the reader who is not experienced with the C programming language. It provides an overview to C header files, and how they define structures, unions, and pointers. It is not intended to teach C programming.

The C preprocessor

The first phase of compiling a C program is the preprocessor phase. The C preprocessor processes every line of source code that begins with the "#" symbol. Figure 1.4 gives examples of preprocessor directives. The #include preprocessor directive is used to copy the code found in the specified header file into the applications prior to calling the compiler. Header files contain definitions of structures and other variable types that can be shared by many applications. For instance, an application that manages an employee data base might include a header file that defines an emprec (employee record) structure. This structure might also be used by an application that processes payroll taxes. By placing the definition of the emprec structure in a header file that is included by both applications, there is no need to define the structure within each application. This saves time and avoids errors.

The "<>" symbols surrounding a file name tell the C preprocessor that the file can be found in the /usr/include directory. Many of the system header files, which describe kernel structures, are found in the directory /usr/include/sys. Therefore, it is common to see #include statements that reference header files such as <sys/proc.h>, which includes the /usr/include/sys/proc.h header file.

If a #include statement references a file whose name appears in double quotes, such as #include "mydefs.h," the preprocessor attempts to locate the file based on the relative or absolute path name specified within the quotes. In the example #include "mydefs.h," the preprocessor looks for the header file in the current directory.

The #define preprocessor directive defines a symbolic constant or a macro function. A symbolic constant is a string of characters, usually capitalized to make it easily recognized. The preprocessor performs a "search and replace" operation for symbolic constants defined with a value. For the example shown in Fig. 1.4, the preprocessor would replace all occurrences of MAXSIZE throughout the code with the value 2000. This makes changing the value easier.

Sometimes, a symbolic constant is defined without an explicit value. This signifies a condition to the preprocessor. The directives #ifdef and #endif are used to provide conditional inclusion of text. Figure 1.5 provides an example.

```
#include <stdio.h>
#include <sys/proc.h>
#include "mydefs.h"
#define MAXSIZE 2000
#define __AIX3.2__
```

Figure 1.4 C preprocessor directives.

```
...
#ifdef DEBUG
    printf("Debug mode is turned on.\n");
#endif
...
```

Figure 1.5 The #ifdef preprocessor directive.

Here, if the symbolic constant DEBUG is defined, either within the file or from the command line (see the -D option to the xlc command for AIX 3.2), the printf() line is included in the compilation. Otherwise, the printf() line is omitted. A #ifndef directive is available to test for a symbolic constant that is not defined.

> **Author's Note:** It's important to be aware of #ifdef directives when tracing system header files. Since much of the code has been ported from other systems (e.g., the NFS header files from Sun MicroSystems), #ifdef directives exist for different implementations.

Structures

A C structure is the definition of a record type. It is a collection of variables, called members, that hold information relative to the instance of the structure. An array of structures constitutes a table. As mentioned earlier, the kernel has many tables for storing data related to the activities of the system.

Figure 1.6 gives an example of a structure named stu_rec. It represents a record of a student who is or was registered at a university. It includes members (referred to as fields for those familiar with data base concepts) for the student's first name, last name, and sex. The stu_rec structure is defined in the stu_db.h header file. The stu_app.c application includes the stu_db.h header file, then defines an array called stu_db[], which is an array of stu_rec structures. The size of the array depends on whether the BIG_MACH symbolic constant is defined at compilation time. The BIG_MACH symbolic constant refers to a system with a large amount of physical memory.

Type definitions

The typedef statement in C allows a programmer to derive a new data type by building upon existing data types. For instance, C does not include a date data type, but one can be constructed by using a structure that contains three integer members (month, day, year). Figure 1.7 shows the example application updated to include a date_t data type for the student's birth date. Note that the stu_db.h header file includes the mytypes.h header file.

stu_db.h

```
#ifdef BIG_MACH
#define SIZE 2000
#else
#define SIZE 200
#endif

struct stu_rec
{
  char s_fname[15];
  char s_lname[15];
  char s_sex;
};

...
```

stu_app.c

```
#include "stu_db.h"

main()
{
   struct stu_rec stu_db[SIZE];
   ...
```

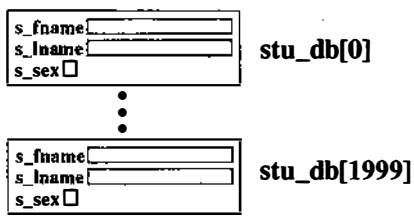

Figure 1.6 C structures.

mytypes.h

```
struct date
{
  int month;
  int day;
  int year;
}
typedef  date date_t;
```

stu_db.h

```
...
#include "mytypes.h"
...
struct stu_rec
{
  char s_fname[15];
  char s_lname[15];
  char s_sex;
  date_t bdate;
};
...
```

stu_app.c

```
#include "stu_db.h"

main()
{
   struct stu_rec stu_db[SIZE];
   ...
```

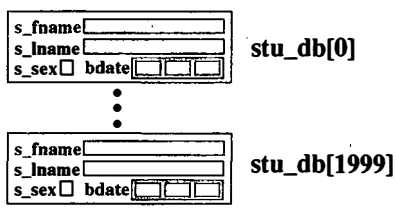

Figure 1.7 Type definitions.

Typedefs are often used to aid in porting code. For instance, each AIX process has a process ID number, which is an integer data type. The system includes a derived data type called pid_t, which is an integer. The pid_t data type is defined in the /usr/include/sys/types.h header file. Applications that declare variables for process ID numbers should declare them as pid_t data types instead of integers. By doing this, if the application is ever ported to another UNIX-based system, where, for instance, the process ID number is defined as a short, the pid_t data type in that system's types.h header file would be declared as a short.

Pointers

A pointer is a variable that holds a memory address. Pointers are often used to hold the addresses of other variables. Pointers are a very complex topic in C. We'll keep our discussion focused on the use of pointers to construct linked lists and form relationships between tables.

A pointer is declared according to the type of data to which it will point. For instance, an integer pointer is designed to point to some integer data. Pointers can be declared to point to structures. Figure 1.8 adds a new field to the stu_rec structure. It is a pointer to a class structure. The application creates an array of class structures, call classes. This table lists the classes for which a student is registered. Note that the stu_rec structure has a single pointer to a class structure. But a student could be registered for many classes at once. The class structure includes a cl_next pointer and a cl_prev pointer, which point to other class structures (other entries in the classes table).

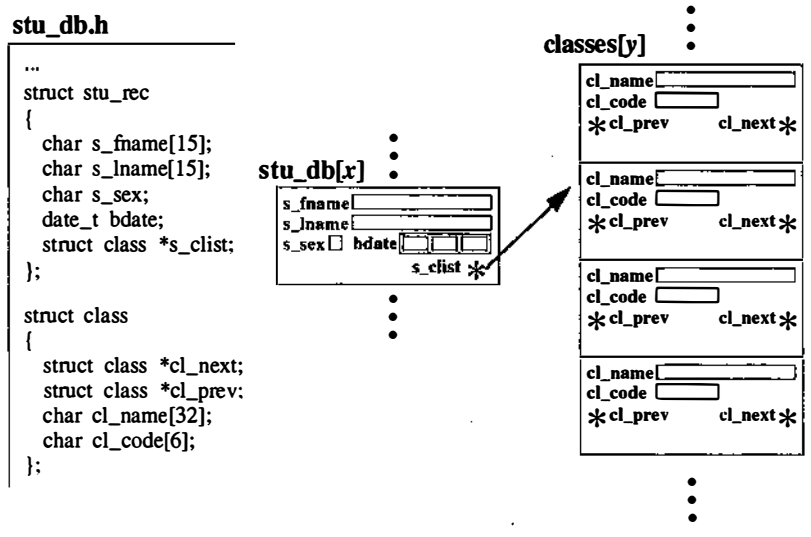

Figure 1.8 Pointers to structures.

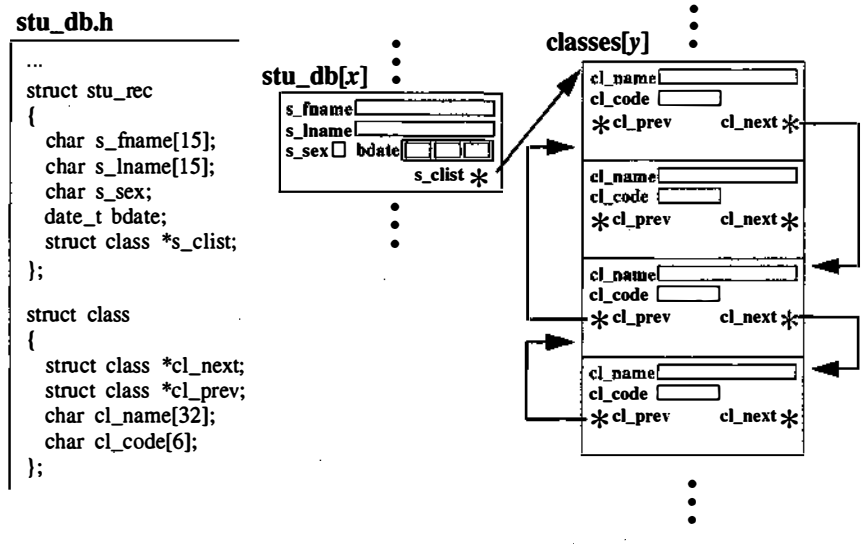

Figure 1.9 Linked lists.

This is known as a doubly linked list. Figure 1.9 illustrates how the doubly linked list works.

Unions

A union is a variable that holds a single value in one of many possible objects of different sizes and types. A union can be used in a data base record to hold data for one of many fields where the fields are mutually exclusive. For instance, the student data base example could be modified to keep track of a student's current classes, the date the student graduated, or the reason a student dropped out of school. These three conditions are mutually exclusive, so a union saves space by overlapping the definitions of the three fields. Figure 1.10 illustrates the use of a union.

In the example, if a student is currently enrolled, the s_data union uses the _s_classes pointer to point to the linked list of class records. If a student has graduated, the s_data union uses the _s_graddate field to hold the date of graduation. If the student has dropped out of school, the s_data union uses the _s_dropout pointer to point to an entry in the dropout table. Note the use of the s_stat field and its defined values to control how the union members are accessed.

The size of a union is set by the compiler to the size of the largest data type within the union.

stu_db.h

```
  ...
struct stu_rec
{
  char s_stat;
  char s_fname[15];
  char s_lname[15];
  char s_sex;
  date_t bdate;
  union {
      struct class *_s_classes;
      date_t _s_graddate;
      struct dropout _s_dropout;
      } s_data;
/*  Values for s_stat  */
#define STUCUR 0x00 /* enrolled */
#define STUGRAD 0x01 /* graduated */
#define STUDROP 0x02 /* dropped out */
 };
```

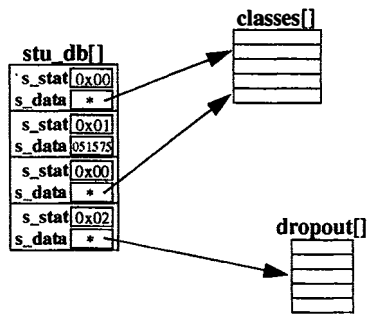

Figure 1.10 Unions.

Macros

A macro is a function defined and substituted by the preprocessor. Macros exist to simplify access to union and structure members. They can also perform evaluations of expressions. Figure 1.11 illustrates a macro that returns the year of a student's graduation.

Examining system header files

Armed with the ability to read and trace C header files, one can learn much about the internals of the AIX 3.2 kernel. For instance, each process has the ability to open files. As will be discussed in Chap. 7, the files that a process opens are tracked in the process's user area. A process's user area is defined as a user structure in the /usr/include/sys/user.h header file. Pointers to

stu_db.h

```
...
/*  stu is a pointer to a stu_rec structure     */
#define GRADYEAR(stu)                  \
     (((stu)->s_stat & STUGRAD) ? (stu)->s_data._s_graddate.year : 0 )
...
```

Figure 1.11 Macros.

opened files are maintained in the u_ufd[] array, which is defined at the end of the user structure. The size of the array is a symbolic constant called OPEN_MAX. The OPEN_MAX symbolic constant is not defined in the user.h header file, however. Looking at the other header files included by user.h, one should notice a header file named limits.h. The /usr/include/sys/limits.h file contains a definition of OPEN_MAX as 2000. This means that an AIX 3.2 process can open up to 2000 files at a time.

One of the best tools for tracing header files is the grep command. It can be used to locate the definition of variables and symbolic constants. When using grep, remember to search all directories where system header files are found.

1.5 The Journey Begins

I've described how one studies internals without having access to the source code. You are now prepared to begin your voyage through the AIX 3.2 kernel. The next chapter provides a high-level view of the system and discusses its major subsystems. Those familiar with other UNIX kernels might be tempted to skip the next chapter, but the AIX 3.2 operating system has a few interesting differences, which one must understand before delving deeper into the technical characteristics of the kernel.

Chapter

2

An Overview of the AIX 3.2 Operating System

The Advanced Interactive eXecutive (AIX) operating system is IBM's open system offering. Its origins can be traced back to the Interactive eXecutive (IX) operating system for an IBM mainframe series in the early 1980s. The current version of AIX, Version 3.2, runs on the IBM RISC System/6000, as well as the IBM PowerPC line of computers. AIX is also the foundation of Bull's BOS operating system.

> **Author's Note:** At the writing of this book, AIX Version 4.1 is available in limited distribution. I anticipate that most of IBM's AIX customers will move to Version 4.1 in late 1995 and early 1996.

AIX 3.2 is based on AT&T's UNIX System V Release 2, with enhancements from Berkeley (BSD) 4.2 and 4.3. It also includes subsystems inherited from previous versions of AIX, as well as components developed exclusively by IBM. As a founding member of the Open Software Foundation (OSF), IBM has incorporated OSF/1 features, such as Motif and the Distributed Computing Environment (DCE) into software offerings for AIX. IBM has also contributed to OSF/1. The Logical Volume Manager (LVM), created for AIX 3.1, is part of the OSF/1 definition.

IBM has made compliance with the IEEE's POSIX (Portable Operating System Interface Standards) and X/Open's XPG3 (X/Open Portability Guide) standards a top priority. Although a few issues are debated regarding full compliance with these standards, programs written to POSIX and XPG3 standards should be easily ported to AIX 3.2.

> **Author's Note:** One topic of debate regarding standards compliance is how AIX 3.2 allocates paging space when the malloc() subroutine is called. See "The AIX 3.2 malloc() Subroutine and Paging Space" in Chap. 4.

This chapter provides a brief history of the AIX operating system, as well as an overview of its internal (kernel layer) and external (application and system management layer) components.

> **Author's Note:** When I joined IBM's Product Education staff in 1989, an old UNIX friend of mine asked me, "Tell the truth, IBM doesn't really want to sell UNIX. They'd rather lock customers into proprietary systems like the AS/400." My response, while I could hardly speak for the folks at the top of the IBM decision-making chain, was that IBM probably would rather sell proprietary systems. Let's face it, when you control the standards, you control the market. But IBM realizes that customers want open systems. There is little choice but to be in the UNIX marketplace. It is clear that if IBM chooses to be in the open systems arena, they will strive to offer the best product out there.
>
> Of course, you'll think I'm biased toward AIX. You're probably right, to some degree. But I have worked with all of the most popular UNIX-based operating systems, and I think AIX has a lot going for it.
>
> I've taught AIX System Administration, Programming, Performance Management, and Internals to quite a few UNIX professionals over the past six years, and I've often found some degree of skepticism about those aspects of AIX that are different from other UNIX-based systems. When given time to work with the system's utilities and features, however, most of my students end up appreciating the differences that AIX offers. Many of the strongest AIX supporters today are those people with a high level of experience using other UNIX-based operating systems.

2.1 A Brief History of AIX

As mentioned earlier, IBM's first venture into offering a UNIX-based operating system was a product called IX (Interactive eXecutive), which ran on a mainframe system. That was followed, in 1985, by AIX for the RT. The RT was IBM's implementation of the Reduced Instruction Set Computing (RISC) processor architecture for a workstation. The RT (RISC Technology) came in desktop and deskside models and offered peripherals for the commercial marketplace, such as external disk drive bays. It also offered 2D and 3D graphics adapters for engineering and scientific applications.

The last version of AIX released for the RT was Version 2. It included TCP/IP and NFS for networking, an IBM-proprietary application for network file sharing called Distributed Services (DS), interfaces for managing devices and file systems, and a variety of applications for mainframe connectivity.

The RT and AIX Version 2 never really penetrated the engineering and scientific marketplace, which at the time was dominated by Sun workstations. The RT lacked power and speed. Although a floating-point adapter card was available as an option, the system never really shook the RT/PC name that it was given early on. Still, IBM learned a great deal from their first venture into the open systems workstation arena.

In 1989, IBM introduced AIX for the PS/2. It ran only on the top-of-the-line models of that time, the Model 80 and the Model 90. It included many fea-

tures of AIX Version 2. In the same year, IBM offered AIX/370, which ran on the 370 mainframe series. It used AIX on a PS/2 as the control system, and included the Transparent Computing Facility (TCF), which provided users a consistent work environment via AIX access from any workstation attached to the network.

AIX Version 3 and the RISC System/6000 were introduced together in February 1990. The RISC System/6000 hardware shows IBM's commitment to the engineering and scientific workstation marketplace. Various models, all based on what IBM calls the second-generation RISC chip, include desktop, deskside, and rack-mounted systems. IBM continues to upgrade the RISC chip capabilities, and new models are introduced regularly. Figure 2.1 provides an overview of the original RISC System/6000 models.

In 1992, IBM added the Model 220 to the RISC System/6000 family. Shortly thereafter, other 2XX models were introduced. Originally, the Model 220 was designated as the diskless workstation. While it had a RISC processor and its own memory, it had no fixed disk drives. Diskless workstations have become popular in the open systems marketplace as a low-cost solution for providing RISC processing for individual users. When a diskless workstation is powered up, it broadcasts a message across the network looking for its boot server system. The boot server responds by downloading boot code, as well as a root file system, to the diskless workstation. Other file systems are then mounted from any one of many possible NFS servers.

The Model 220 was the first RISC System/6000 to include an integrated SCSI (Small Computer System Interface) controller for disk, tape, and CD-ROM support, as well as integrated Ethernet and optional Token Ring cards. This meant that controller card slots need not be used for these adapters. IBM now includes these integrated cards on many of the newer models.

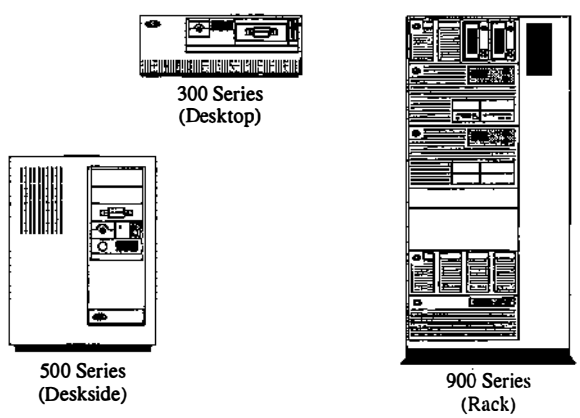

Figure 2.1 RISC System/6000 models.

1986	AIX Version 2 for the RT
1989	AIX for the PS/2
1990	AIX Version 3.1 for the RISC System/6000
1992	AIX Version 3.2 for the RISC System/6000

Figure 2.2 The history of AIX.

The Model 220 can include an optional 2-gigabyte fixed disk. This can be used in a diskless workstation environment to support local paging space, since paging across a network is not one of the most pleasant experiences where performance and network throughput are concerned. IBM calls this configuration a "dataless" workstation.

AIX 3.1 represented quite a change from AIX Version 2. A considerable amount of work was done to improve the system management tools, as well as the kernel subsystems described throughout this book. AIX 3.2 was introduced in 1992. It added support for diskless workstations, which included rearranging the file tree. See Sec. 2.3 for more information on the system management features of AIX 3.2.

Figure 2.2 summarizes the history of the AIX operating system.

2.2 A User's Perspective of AIX 3.2

To the user, AIX looks and acts like most other versions of UNIX-based systems. This is thanks to standards like POSIX, which strive to define common syntax, function, and output from commands and application programming interfaces (APIs).

> **Author's Note:** I have had numerous students over the years who, while experienced with other operating systems, are new to UNIX. A common question they have is "If UNIX is supposed to be the same on all different types of systems, why are there so many UNIX vendors, each claiming that their version of UNIX is superior?" After a brief explanation of the history of UNIX and how each vendor participates in the open systems marketplace, I compare using a UNIX-based operating system to renting an automobile. When you rent an automobile, it's usually not the same make or model as you normally drive, but you're able to drive it because things like the gas and brake pedals and the steering wheel are all in the same place and do the same things you're used to on your own car. However, it's likely that the radio controls, the air conditioning controls, and the switch that turns on the headlights are different from those on your own car. Yet

you're able to figure them out quickly (usually). UNIX is like that. A user who is familiar with UNIX knows what the ls command does, for instance. However, not all UNIX-based systems produce output from the ls command with the same format. Some systems print the names of all files in the current directory in a single column. Other systems, like AIX, print the names of all files in the current directory in multiple columns. It's easy to move from one UNIX-based system to another, but the user should expect subtle differences for things that do not fall under the control of standards.

AIX supports the three most popular UNIX shells, the Bourne shell, the C shell, and the Korn shell. IBM also offers AIXwindows as a graphical user interface (GUI). AIXwindows include the X Window System (Version 11 Release 5), the Motif Window Manager (from the OSF), and the Xdesktop application (from IXI, LTD). The X Window System provides a display server, which handles I/O from the terminal and routes it to the appropriate window client. The Motif Window Manager controls the look and feel of the windows and their attributes. Other window managers are available for the X Window System. Xdesktop is an application that provides a graphical view of the file system, using icons to represent files and directories. Users can customize the desktop by adding their own icons.

AIX 3.2 supports a high function terminal (HFT) device driver. This driver is multiplexed to handle I/O for a graphical display, a keyboard, a mouse, and a tablet device. One of the most useful features of the HFT is virtual terminals. A virtual terminal is a full-screen session, as opposed to a window. An initial virtual terminal is implemented for a user's login session. The user can start additional virtual terminals by issuing the open command (see the manual page for the open command). The open command takes, as an argument, the name of the program to be executed in the new terminal session. The program executed is usually a shell. Each virtual terminal is represented as a device named /dev/hft/n, where n is the number of the virtual terminal. The user switches between virtual terminals by pressing the <ALT>-<CTRL> (or <ALT>-<ACTION>) key combination. Figure 2.3 illustrates the use of virtual terminals.

2.3 AIX 3.2 System Administration

Experienced UNIX users quickly become comfortable with AIX commands. Experienced UNIX system administrators find many enhancements to the traditional system management paradigm. This section lists and describes many of the system administration improvements made to AIX.

The system management interface tool

The most striking system administration improvement made by IBM was the creation of the system management interface tool (SMIT). While many UNIX-

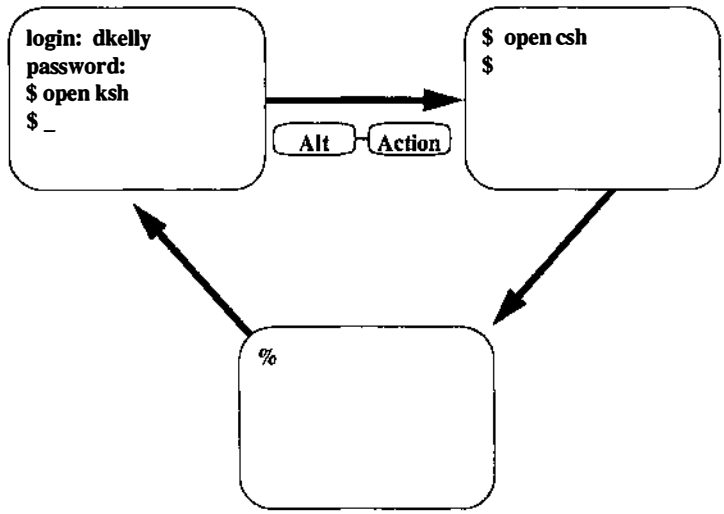

Figure 2.3 Virtual terminals.

based systems have menu-driven administration tools, SMIT provides a complete and consistent interface for virtually all system management components. The philosophy of SMIT is that it doesn't matter if one is adding a new tape drive or a new user to the system. The interface should look and work the same. Figure 2.4 shows the main menu of SMIT.

Actually, SMIT does not perform the system management tasks. It only constructs a command string that is executed in a shell environment. An adminis-

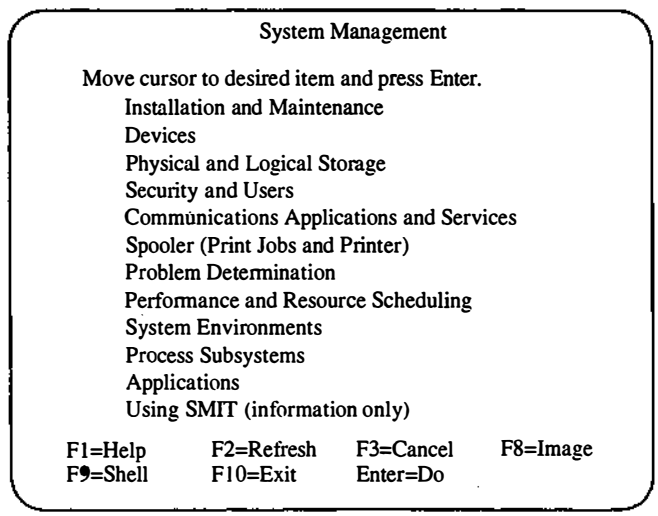

Figure 2.4 The system management interface tool.

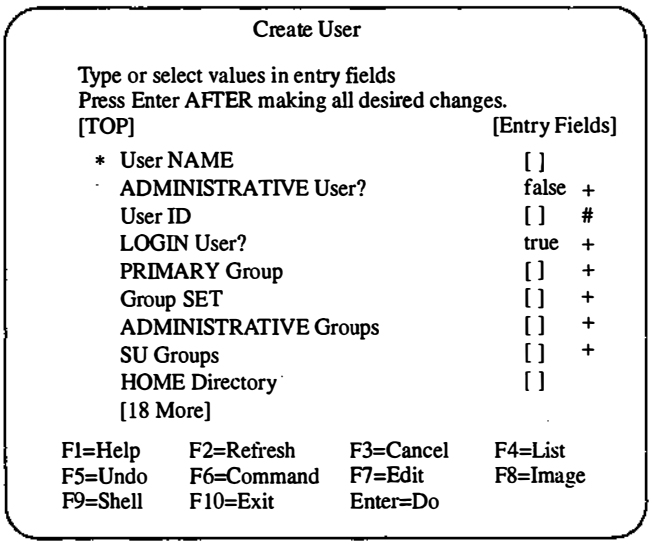

Figure 2.5 A SMIT dialog screen.

trator can execute the commands, called "high-level commands," directly, without using SMIT. Each high-level command maps to a specific system management task. There are four primary task prefixes for the commands: "mk..." to make, or add, a new object, "ls..." to list existing objects, "ch..." to change an existing object, and "rm..." to remove an existing object. For instance, the high-level command "mkuser" adds a new user account to the system, while the high-level command "mkvg" adds a new volume group of disk drives to the system (see an explanation of the logical volume manager later in this section). Each high-level command maps to a SMIT dialog screen. A dialog screen prompts the administrator for information which becomes parameters for the corresponding high-level command. Figure 2.5 provides an example of the SMIT dialog screen for the mkuser high-level command.

The object data manager

AIX's object data manager (ODM) is an application that uses an object-oriented data base to store information about devices, installed software, system management daemons, and the SMIT screens. Some network configuration data are also kept by the ODM. The data base files reside in the /etc/objrepos and /usr/lib/objrepos directories. Table 2.1 lists some of the files and what they hold.

Figure 2.6 shows how SMIT, the high-level commands, and the ODM work together to facilitate system management. High-level commands query or modify the system configuration by updating or listing information found in the ODM data bases or various ASCII files. An arrow is shown from the ODM

TABLE 2.1 ODM Files

ODM File	Contains...
/etc/objrepos/PdDv	all supported devices
/etc/objrepos/PdAt	default attributes for all supported devices
/etc/objrepos/CuDv	customized devices (defined)
/etc/objrepos/CuAt	customized device attributes
/etc/objrepos/ConfigRules	rules for the sequence of device configuration
/etc/objrepos/lpp	all installed software products and updates
/etc/objrepos/history	historical data for software product updates
/etc/objrepos/inventory	file names for installed software products
/etc/objrepos/sm_menu_opt	SMIT menu screen options
/etc/objrepos/sm_name_hdr	SMIT name selector screen data
/etc/objrepos/sm_cmd_hdr	SMIT dialog screen data
/etc/objrepos/sm_cmd_opt	SMIT dialog screen field data

to SMIT to illustrate that the data used to construct the SMIT screens are found within the ODM. A system administrator can customize SMIT screens by changing the data in the ODM.

Author's Note: The ODM has probably been the most controversial component of AIX. I initially learned about the ODM while working for IBM in 1989, prior to the announcement of AIX 3.1. The talk at that time was that the ODM data base files would replace all traditional UNIX ASCII files, including files like /etc/passwd. My first thought was "how could AIX be anything like UNIX without the /etc/passwd file!" I found out later that the role of the ODM was a hot point of

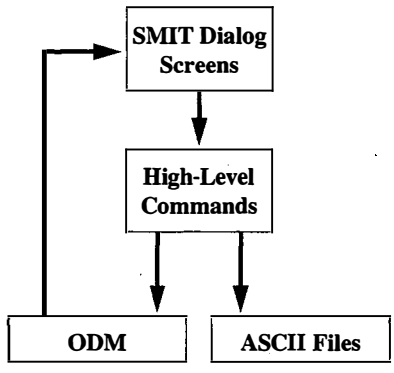

Figure 2.6 SMIT and high-level commands.

debate between various groups of IBM designers and developers. There were those who believed that the ODM should control all system configuration information, while others stressed the importance of keeping some ties to tradition. In the end, the decision was made not to include any user management information in the ODM data bases.

As with SMIT, one of the goals of the ODM is to provide a consistent view of the system configuration. This is accomplished by reducing all information to objects, which can be treated in a common fashion. For instance, all supported devices are stored as objects in the PdDv (predefined devices) file. The default values for the attributes of each device are stored as objects in the PdAt (predefined attributes) file. While the attributes of a SCSI disk drive are different from the attributes of a parallel printer, Fig. 2.7 illustrates how the ODM links the objects. The ODM includes user-level commands that allow an administrator to work directly with the ODM data base files. The ODM also includes a library named /usr/lib/libodm.a that provides an API for programmers.

Author's Note: Some might be tempted to use the ODM tools supplied with AIX to create their own data base applications. I strongly discourage this. IBM provides the ODM to manage system configuration data. It is not intended to be used as a data base engine for user applications. Aspects of the ODM might change from one release of AIX to another, possibly leaving one with a data base application that no longer works.

AIX 3.2 device management

AIX uses two ODM files to define devices. The PdDv (predefined devices) file contains descriptions of all supported devices. The CuDv (customized devices)

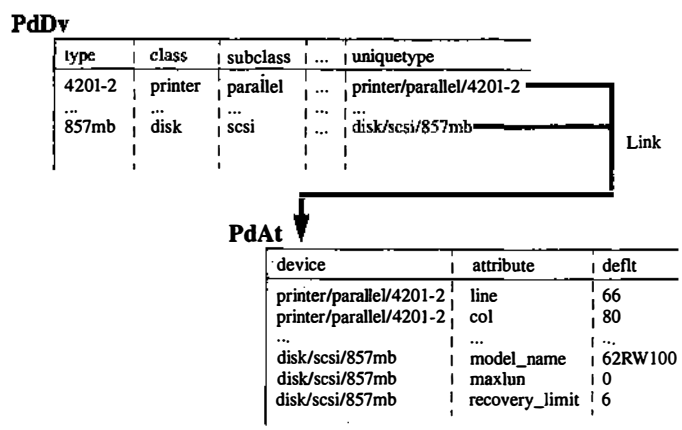

Figure 2.7 The ODM and device attributes.

file contains information about the devices that the system has (or thinks it has). AIX classifies all devices as being in one of three possible states: predefined, defined, and configured. A predefined device is one that is supported but not present on the system. In other words, a predefined device is in the PdDv file but not in the CuDv file. When an administrator uses SMIT or the corresponding high-level command to add a device, an entry for that device is made in the CuDv file. The device is now said to be in a defined state. The defined device, however, is not yet available for use. First, the device must be set to a configured state. A device is configured when its device driver has been loaded into the kernel. Figure 2.8 illustrates device states. More information on device configuration is provided in Chap. 9.

The AIX 3.2 queuing system

AIX 3.2 implements printer management via a queuing system that is unlike the System V spooler or the BSD queuing system. It includes the concept of virtual printers to provide separate user interfaces to a single printer that supports multiple emulation modes. The AIX queuing system supports queues not only for printers but for any type of device or resource for which serialization is desired.

Users submit queue requests by issuing one of four possible commands. The enq command queues a job request. The qprt command queues print job requests and has options for controlling the attributes of the virtual printer. AIX also supports the lp (System V) command and the lpr (BSD) command for compatibility. When a user submits a queue request, a file is created, describing the request, in the /var/spool/lpd/qdir directory. When the resource for which the queue applies is available, the request is fulfilled.

At the heart of the AIX queuing system is the qdaemon program. This program monitors the availability of resources for which queues are imple-

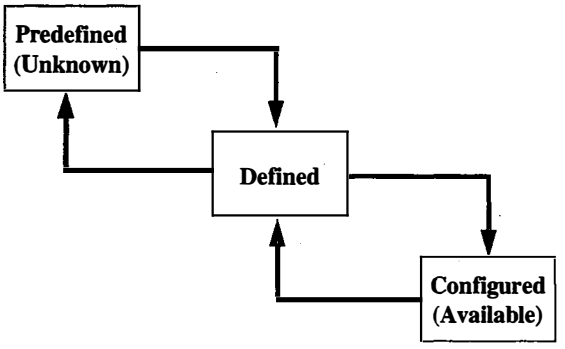

Figure 2.8 Device states.

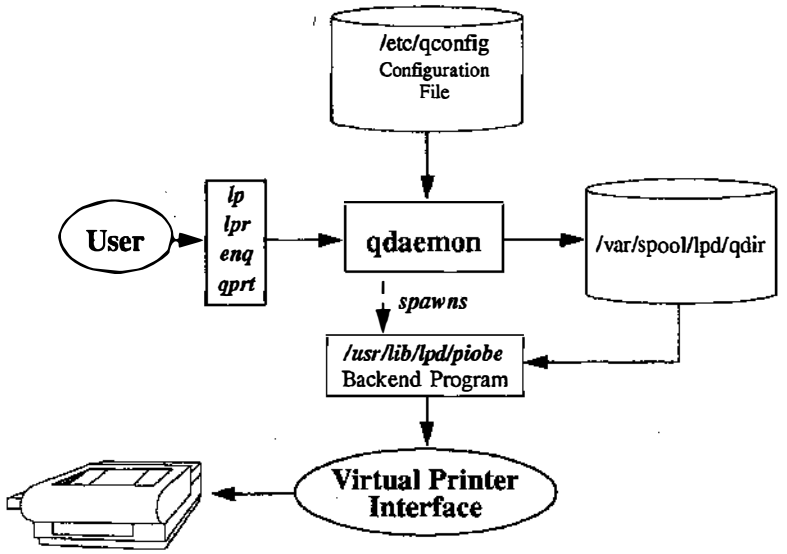

Figure 2.9 The AIX 3.2 queuing system.

mented. When a resource is available and a job request exists in the /var/spool/lpd/qdir directory for the resource, the qdaemon spawns a backend program to process the request. The backend program for local printers is /usr/lib/lpd/piobe. Figure 2.9 illustrates the AIX 3.2 queuing system.

Software management

IBM uses three terms to describe installed software: base operating system (BOS), licensed program product (LPP), and optional program product (OPP). The BOS consists of all software that makes up the core of the AIX operating system. It includes the kernel, all essential applications, and configuration files. An LPP is an application that the customer has purchased for AIX. It is not bundled with the BOS. Examples of LPPs include AIXwindows, the Fortran compiler, and the GraPHIGS application. An OPP is an application that comes with AIX, at no additional charge, and can be selected for installation or skipped. Examples of OPPs include the system accounting facility, the C shell, the INed editor, and the text processing applications (nroff, troff, etc.).

Information about the BOS, LPPs, and OPPs is stored by the ODM in a series of files called the vital product data base. This includes the software release levels, update history, inventories of files, and product serial numbers.

> **Author's Note:** AIX makes no distinction between LPPs and OPPs. The ODM calls them both LPPs.

The journaled file system

AIX Version 3 introduced the journaled file system (JFS). Its details are described in Chap. 6. The JFS is the default file system type for local disk files. It offers two advantages over other types of UNIX disk file system. First, the JFS logs activity associated with the file system control structures. This allows it to reconstruct a file system to a known state in the event of a system crash. Second, a JFS file system can be extended while it is in use. This allows the system administrator to increase the size of the file system without disrupting user activity.

The logical volume manager

One of the most popular features of AIX Version 3 is the logical volume manager (LVM). It aids the management of disk space by providing a logical view of disk drives. A logical volume is a specified amount of disk space designated to hold a journaled file system or some raw partition, such as paging space. The disk space allocated to a logical volume need not be contiguous, or even all on the same physical disk drive. As with a journaled file system, a logical volume can be extended dynamically.

This section provides information on the implementation of the LVM. It begins by introducing some important terms.

Physical volume (PV). A physical volume is a disk drive. It can be an internal drive or an external drive. It can be a SCSI drive or some other type of drive. Its size is not important.

Volume group (VG). A volume group is a collection of one or more physical volumes. The physical volumes in a volume group are configured together for some logical reason. In fact, a volume group can be thought of as a collection of drives grouped as one logical drive. The system requires at least one volume group, called the root volume group (or rootvg), which contains the root file system.

A volume group can contain up to 32 physical volumes. A physical volume must belong entirely to a single volume group. In other words, a disk drive cannot be split between two volume groups. The extent of a volume group becomes the boundaries for its contents, as described shortly. New physical volumes can be added dynamically to a volume group.

AIX uses a command called varyonvg to vary on a volume group. A volume group must be "varied on" before its contents are available. The varyoffvg command is used to take a volume group off-line, thus making its contents inaccessible.

Volume group descriptor area (VGDA). Each physical volume of a volume group has a structure stored in a reserved set of sectors called the volume group

descriptor area. This structure maintains information about the entire volume group. Multiple copies are kept to assure their integrity.

Physical partition (PP). The LVM carves all physical volumes in a volume group into equal-sized partitions. The partitions are allocated from contiguous space. Most AIX systems use a default physical partition size of 4 megabytes, but system administrators can specify a different physical partition size when the volume group is created. The Model 320, one of the original RISC System/6000 models with a smaller disk capacity, uses a default physical partition size of 2 megabytes. A physical partition is the smallest amount of disk space that can be allocated to a logical volume.

Figure 2.10 illustrates a sample volume group and its components described thus far.

Logical volume (LV). As mentioned earlier, a logical volume is a container for a journaled file system or some other entity, such as paging space or a dump device. A logical volume is a collection of one or more physical partitions. The physical partitions that make up a logical volume need not be contiguous or even all on the same physical volume, but they must all be allocated from physical volumes that are in the same volume group. In other words, while a logical volume can span multiple physical volumes, it cannot span multiple volume groups.

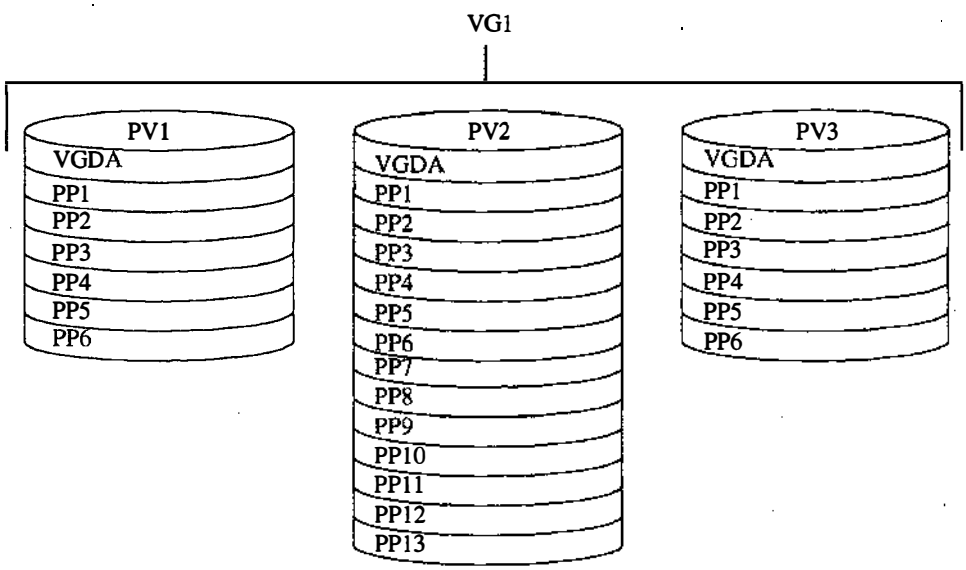

Figure 2.10 The logical volume manager.

A JFS must be stored in a logical volume. The relationship between a logical volume and a JFS is one-to-one, in that only one JFS is ever found in a single logical volume.

Logical partition (LP). The final LVM term is logical partition. A logical partition is similar to a physical partition except that it is referenced as a subdivision of a logical volume where a physical partition is referenced as a subdivision of a physical volume. A logical volume numbers its logical partitions from 1 to *n*, where *n* is the last partition in the logical volume. This is done without regard to the physical location of each logical partition. Figure 2.11 illustrates a volume group with a set of logical volumes defined. In the example, the third logical partition of logical volume three is found in the fifth physical partition of physical volume two.

One of the most important features of the LVM is mirroring. The LVM allows the system administrator to mirror critical logical volumes to improve availability and reliability. When mirroring is implemented, the LVM maintains one (single mirroring) or two (double mirroring) spare copies of each physical partition of a mirrored logical volume. While the administrator can choose on which physical volumes the mirrors are stored, it is advisable to allocate each copy of a mirrored physical partition on a different physical volume. Figure 2.12 illustrates a single mirror for LV3. Although mirroring is

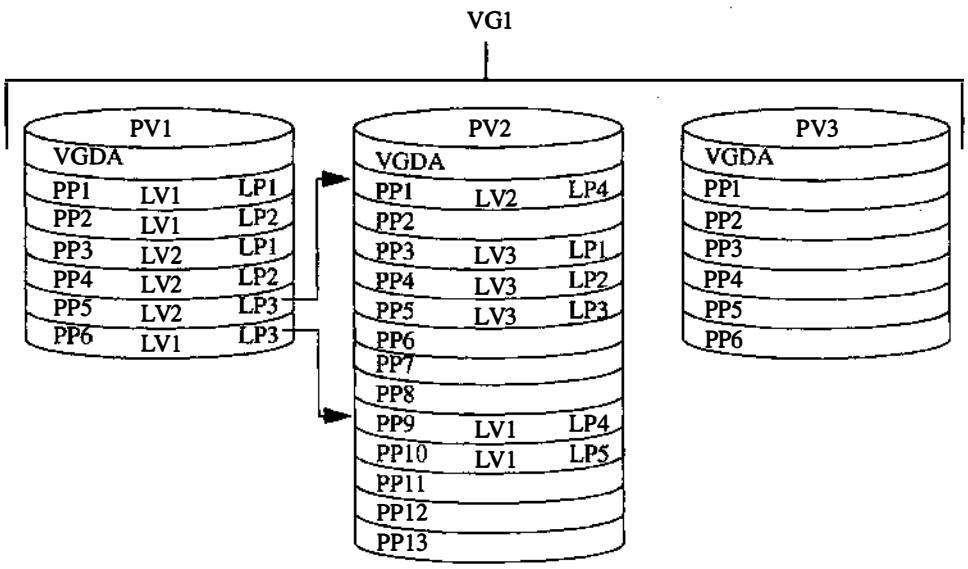

Figure 2.11 Logical volumes.

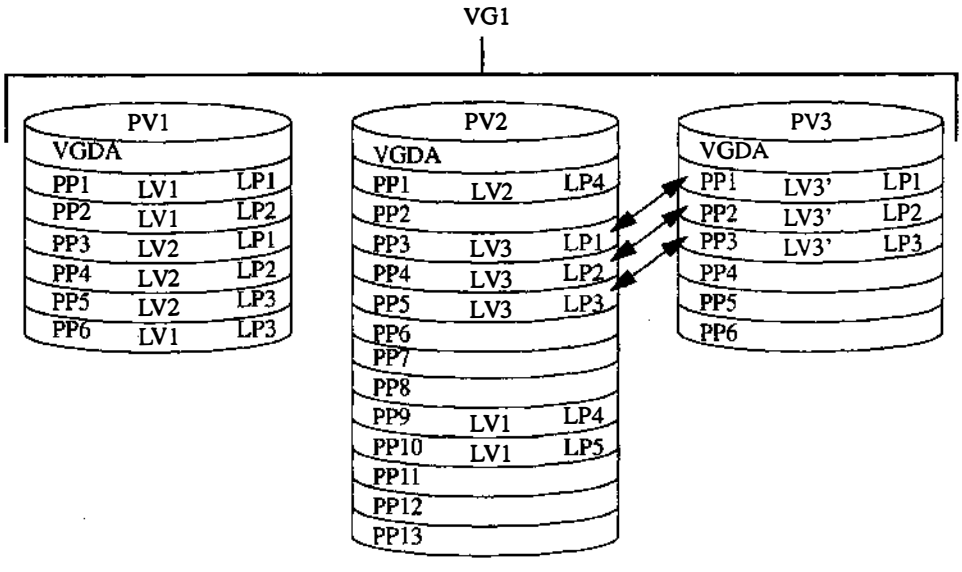

Figure 2.12 LVM mirrors.

established by the administrator for an entire logical volume, the internals of the LVM implement mirroring on a partition-by-partition basis.

There are two reasons for mirroring important logical volumes. First, keeping spare copies on separate physical volumes helps assure that the logical volume is still accessible, even if one of the physical volumes fails. In the example in Fig. 2.12, LV3 will still be available even if PV3 fails. If PV2 fails, LV1 and LV2 will be inaccessible, but LV3 will still be accessible. It is important for a system administrator to plan the physical layout and mirroring strategies for critical logical volumes.

The second advantage to mirroring is that, while it takes longer for the operating system to write multiple copies of data to the mirrors, read time is improved because the LVM decides which copy provides the fastest access time based on estimated disk seek distance. This means that mirroring logical volumes that have more read operations than write operations can actually improve the disk I/O performance. The rule of thumb for most read/write file systems is that 80 percent of the I/O operations involve reads and 20 percent of the I/O operations involve writes.

Of course, mirroring isn't for every logical volume. The disadvantage to mirroring is that it requires two or three times as much disk space. Mirroring is designed for fault tolerance of critical data, such as on-line transaction processing and other data base or commercial applications. Stand-alone workstations in a development shop, for instance, might find little benefit in mirror, at least not enough to offset the additional disk space requirements.

Another feature of the LVM is bad block relocation. If an I/O request encounters a bad disk block, the disk hardware usually handles it, remapping the bad block to a spare block location. (Most disk drives maintain a pool of spare blocks used for remapping bad blocks.) If the disk hardware does not provide bad block relocation, the LVM handles it at the software level. The LVM relocates the bad block to another block within the file system. The LVM implements bad block relocation only when it is not provided by the disk hardware.

> **Author's Note:** To the best of my knowledge, all of the disk drive models offered by IBM for the RISC System/6000 support hardware-level bad block relocation. Check the documentation of third party disk drive systems to determine whether or not they support hardware-level bad block relocation.

If a bad block is detected during a write operation (the LVM supports a write verify option to confirm all writes), the block is remapped and the data are written to the new location. If a bad block is detected during a read operation, if the logical volume is mirrored, the LVM reads from another copy. It then remaps the bad block and copies the data from the mirror to the new location. If the logical volume is not mirrored, the read request fails and returns an error to the application.

2.4 AIX Application Programming

AIX 3.2 supports a family of compilers that are designed specifically to produce code optimized to run on the RISC System/6000 processors. Languages that are part of the XL family include ANSI C, C++, Fortran, and Pascal. Details of the XL compilers and the AIX 3.2 compilation process are found in Chap. 3.

Other languages available for AIX include COBOL and ADA. Assembler programming for the RISC System/6000 is also supported.

The AIX 3.2 kernel API includes system calls that are compliant with POSIX and X/PG3 standards. System calls are also included for compatibility with System V and BSD applications.

The application development toolkit (ADT), an AIX LPP, contains many of the traditional UNIX programming utilities. It includes lint, make, source code control system, and the dbx debugger. IBM also provides CASE tool applications for AIX.

2.5 The Design of the AIX 3.2 Kernel

The AIX 3.2 kernel was introduced in Chap. 1. Recall that the kernel is the code and data that provides I/O and process management services for applications. The kernel uses device drivers to interface with system hardware. Applications request services of the kernel by issuing system calls. The mem-

ory used by the kernel has the device name /dev/kmem. All remaining real memory is allocated to applications and is called user memory.

There are four important features of the AIX 3.2 kernel: it is preemptable, it is dynamically extendable, it supports real-time applications, and most of it is pageable. This section describes the importance of these features.

The preemptable kernel

A preemptable kernel means that a process can be preempted while it is running a system call. Preemption occurs when the kernel dispatcher decides that the current running process is no longer the most favored process. The current running process is preempted to allow another process to run. Older UNIX systems had nonpreemptable kernels. This meant that the dispatcher had to wait until the current process finished its system call before preempting it to allow another process to run. Figure 2.13 illustrates the preemptable AIX 3.2 kernel.

In the example below, Process A is the current running process. Process B, which has a more favored priority than Process A, is in a sleep state, waiting for a hardware event to occur. Process A is in the middle of a system call when the event for which Process B is waiting occurs. Hardware events interrupt all system processing so that the event can be handled by the device driver. The interrupt handler in the device driver wakes the sleeping Process B and notifies the kernel's dispatcher. The dispatcher preempts Process A so that Process B can run. Process A will resume the system call the next time the dispatcher chooses it to run. Details on how the kernel dispatches processes are provided in Chap. 5.

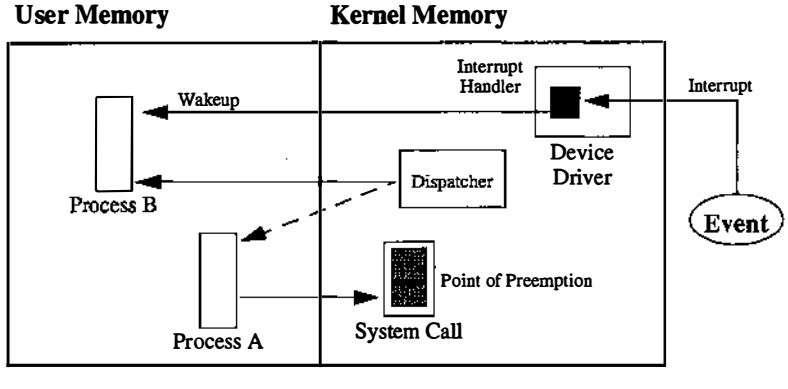

Figure 2.13 The preemptable AIX 3.2 kernel.

Author's Note: I stress the difference in the terms "preemption" and "interruption." Preemption is what happens to a running process when the dispatcher has selected a more important process to run. Interrupts occur when a hardware device needs the operating system's attention. While interrupts do cause the system to stop running the current process while the interrupt is handled, unless the dispatcher selects another process to run after the interrupt is complete, the current process resumes running.

Dynamic kernel extensions

The fact that the AIX 3.2 kernel can be dynamically extended was mentioned briefly in Chap. 1. There are four types of AIX 3.2 kernel extensions. They are device drivers, system calls, virtual file systems, and STREAMS modules, all of which can be added to the kernel without rebuilding the kernel image or rebooting the system. Each of the kernel extensions mentioned is described in various chapters of this book. InfoExplorer includes a book called "Kernel Extensions and Device Support Programming Concepts," which is required reading for anyone creating device drivers, system calls, or virtual file systems. There is limited documentation on writing STREAMS modules.

The pageable kernel

Traditionally, UNIX kernels are pinned in memory, which means that the pages of the kernel remain in real memory. The advantage of pinning the kernel is that access to the kernel code and data is faster. The disadvantage is that less memory is available for applications' user code. As kernels have grown larger on many UNIX-based systems, pageable kernels have become more common. Pageable kernels use disk paging space to hold those pages of the kernel that are not frequently referenced. This allows kernels to provide additional services without extracting the penalty of the increased kernel size. It also frees up real memory for applications.

Most of the AIX 3.2 kernel is pageable. System call code as well as many of the kernel's tables are pageable. Figure 2.14 illustrates the pageable portion of the kernel. Note that some of the kernel's memory is still pinned. For instance, the interrupt handler portion of device drivers must be pinned. Interrupt handlers must run quickly to avoid blocking other system activities for too long; therefore, page faults are not allowed within interrupt handler code. Any kernel data structures accessed by interrupt handlers must also be pinned. Besides, imagine what would happen if the system paged out the interrupt handler code for the disk drive devices!

Real-time programming

The nonpreemptable kernel described earlier in this section illustrates one of the reasons that UNIX, historically, is not considered a real-time operating

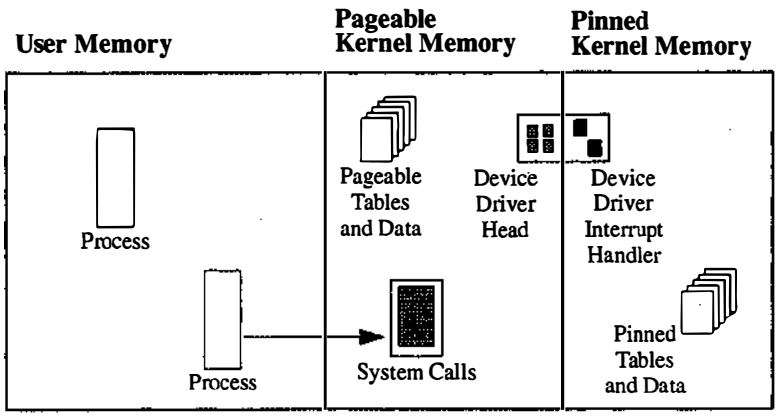

Figure 2.14 The pageable AIX 3.2 kernel.

system. A real-time operating system is one that can guarantee the maximum amount of time it takes for an application running on the system to react to some event. The time between the occurrence of the event and the dispatching of the process that handles the event is called the "context latency." Real-time systems can guarantee a maximum context latency.

> **Author's Note:** I often see "real time" mistakenly described as "fast." A real-time system need not guarantee a "fast" context latency. The true definition of real time is that the context latency falls within the expectations of the application users. For instance, a banking system might define real time as guaranteeing that a customer's account is updating within 15 minutes of an ATM transaction. One of the largest markets for real-time programming is the stock trading and securities industry.

AIX 3.2 provides real-time options through many of its characteristics. For instance, the preemptable kernel is an essential part of real time. Nonpreemptable kernels allow a process to finish a system call before dispatching another process. This is unacceptable for real-time applications.

Other features of the AIX 3.2 kernel that support real-time programming include system calls that allow real-time programs to set their priority levels, faster context switch time (a context switch, detailed later in this book, is the action of switching between running processes), and enhanced system timers.

> **Author's Note:** Actually, IBM describes AIX as providing "near real time" support.

2.6 Kernel Subsystems

Figure 2.15 illustrates the two major subsystems of the AIX 3.2 kernel; the I/O subsystem and the process management subsystem. Each subsystem has

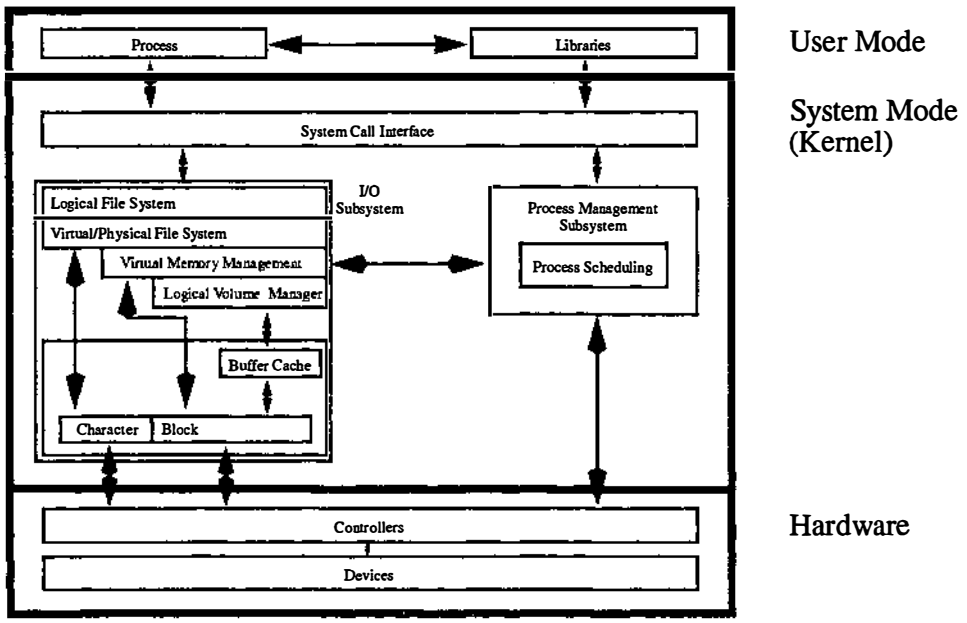

Figure 2.15 AIX 3.2 kernel subsystems.

its own subsystems. For instance, the I/O subsystem includes the file I/O subsystem, the virtual memory manager (VMM) and the logical volume manager (LVM). The process management subsystem includes the kernel loader, which loads and executes programs, the process scheduling subsystem, and support for interprocess communications (IPC). The line from the process management subsystem to the VMM indicates that the VMM handles the memory requirements of processes.

As mentioned before, system calls provide the application's interface to the kernel. System calls and library routines are described in detail in Chap. 3.

Device I/O takes one of two forms: block I/O or character I/O. Block devices are those devices that perform I/O using a buffer strategy. The I/O occurs in blocks. Disk drives are block devices. Character devices are those devices that perform I/O one character at a time. Examples of character devices include printers and terminals. Character devices are said to perform "raw" I/O. Conversely, block devices are said to perform "cooked" I/O. Many block devices have corresponding "raw" modes that allow processes to access the device using character I/O.

Device types are determined by issuing the ls -l command on the /dev directory. The first column of each line indicates the device type. A "b" indicates a

```
$ ls -l /dev
crw-rw----  1  root  system  14,0  May 15 09:31  console
...         ...  ...   ...     ...   ...          ...
brw-r-----  1  root  system  10,0  May 15 09:31  hd1
brw-r-----  1  root  system  10,1  May 15 09:31  hd2
...         ...  ...   ...     ...   ...          ...
brw-r-----  1  root  system  15,0  May 15 09:31  hdisk0
brw-r-----  1  root  system  15,1  May 15 09:31  hdisk1
...         ...  ...   ...     ...   ...          ...
crw-r-----  1  root  system  10,0  May 15 09:31  rhd1
crw-r-----  1  root  system  10,1  May 15 09:31  rhd2
...         ...  ...   ...     ...   ...          ...
crw-r-----  1  root  system  15,0  May 15 09:31  rhdisk0
crw-r-----  1  root  system  15,1  May 15 09:31  rhdisk1
...         ...  ...   ...     ...   ...          ...
crw-rw----  1  root  system  12,0  May 15 09:31  tty0
crw-rw----  1  root  system  12,1  May 15 09:31  tty1
```

Figure 2.16 AIX 3.2 device special files.

block device, while a "c" indicates a character device. Note in the example shown in Fig. 2.16 that AIX 3.2 refers to fixed (hard) disk drives as "hdisk..." for block mode and "rhdisk..." for character mode. The same is true for logical volumes where the block mode name is "hd..." and the character mode name is "rhd...."

Chapter

3

AIX 3.2 Programs and Processes

This chapter presents an overview of programs and processes by defining each and describing their attributes.

3.1 Programs

A program is an executable image stored on disk or some other medium. In other words, a program is an executable file. Programs include the applications that come with AIX, such as the vi editor, the cat program, the various shells, etc. Programs also include applications purchased for the system, as well as applications written by in-house programmers.

3.2 Processes and Process Types

A process is the executing image of a program. To illustrate, if three users on the same system are all running the vi editor, there are three processes running the same program. Figure 3.1 illustrates this example. The kernel controls how each process shares the system resources, such as the CPU, memory, disks, etc., with the other processes. A kernel component known as the dispatcher decides which process will control the CPU. This time-sharing concept, called process scheduling, is discussed in detail in Chap. 5.

Processes have attributes that distinguish them from one another. Each process has a unique process ID number (PID). Other process attributes include user identification numbers, group sets, current directories, and resource usage statistics. Each process also includes the instructions and data of the program they are executing. The precise image of a process is described shortly.

There are different types of processes.

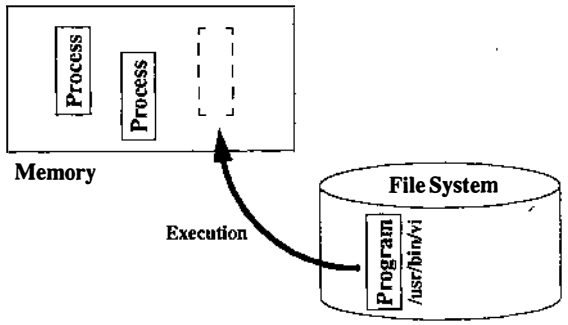

Figure 3.1 Programs and processes.

User processes. User processes are the most common type of process. They perform work on behalf of a user. Examples include the shells, editors, commands, utilities, and any application invoked by a user. They are created by other user processes. This defines a parent-child relationship between user processes. User processes are in user mode when running code that is part of the program executing within the process. When the program executing within a user process makes a system call, a mode switch occurs, transferring execution to the system call code in kernel memory. The kernel is said to be executing on behalf of the user process and is running in system mode. A process's CPU time is the sum of the process's user mode time plus system mode time.

> **Author's Note:** CPU time is not the same as total run time (or response time) since processes are not always running. Sometimes they sleep (wait on an event) or are preempted by a more favored process.

When it comes to scheduling, there are two classes of user processes: ordinary user processes and fixed-priority user processes. An ordinary user process has a priority value that changes over the lifetime of the process (see Chap. 5 for a detailed discussion of process scheduling). A fixed-priority process has a priority value that remains constant. A setpri() system call issued by the process causes its priority to become fixed. Real-time programs usually use the setpri() system call to assign themselves to a highly favored priority level, thus assuring that they will preempt ordinary user processes.

Kernel processes. Kernel processes are processes that spend all of their time in system mode, running kernel code. They can be created only by system calls or device drivers. They perform tasks on behalf of system calls, device drivers, or other kernel entities. They are scheduled and selected for dispatch the same way as user processes and have many of the same attributes as user processes.

Author's Note: AIX 3.2 displays kernel processes as "kproc"s when the ps-ek command is issued. See Fig. 3.3 for an example.

Daemon processes. Daemon processes represent another type of user process. A daemon process runs in user mode and system mode, like other user processes; however, a daemon process has no controlling terminal session. It will continue to run even when no one is logged into the system. Daemons are frequently used by system administrators to perform repetitive tasks, such as collecting system performance information or doing file system backups in the middle of the night. Daemons are also used by the system and many of the network facilities. Examples of daemons include cron, syncd, inetd, telnetd, and swapper. All of these daemon processes are started at boot time and usually run for the duration of the system. Daemon processes are identified in the output of the ps -ef command by the "-" symbol in the TTY column, since they have no controlling terminals.

Figure 3.2 illustrates the various types of processes described. Figure 3.3 shows an example of the ps command, displaying all of the process types.

3.3 Program Creation in AIX

AIX includes a family of compilers that have separate language-specific front ends but a common backend. This is known as the XL family of compilers. The four XL compilers are xlc for ANSI C, xlC for C++, xlf for Fortran, and xlp for Pascal. AIX Version 3 includes the xlc compiler. All other compilers must be purchased separately.

Each XL compiler front end compiles its particular language's source code into an intermediate language that is common to all XL compilers. The common

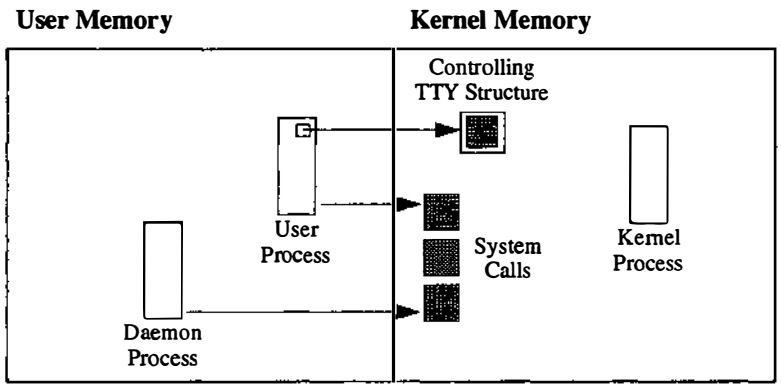

Figure 3.2 Types of processes.

```
$ ps -efk
USER   PID  PPID  C   STIME  TTY  TIME CMD
root   0    0     54  08:32:40 -   0:00 swapper
root   1    0     0   08:33:25 -   0:00 /etc/init
root   514  0     120 08:33:40 -   3:58 kproc
...    ...  ...   ... ...      ...  ...  ...
root   5240 1     0   08:35:09 hft/0 0:00 -ksh
...    ...  ...   ... ...      ...  ...  ...
```

Figure 3.3 The ps command and process types.

backend optimizes the code, if desired, to produce an object file. There are two outstanding benefits to this approach. First, the optimizing backend is designed to optimize code specifically for the RISC System/6000 or POWER/PC processors, taking advantage of their advanced architectural features. The optimizer rearranges code in such a way as to best fill the processor pipeline.

Second, the common intermediate language makes it possible for programs in one language to call routines in another language. For instance, a C program can call a Fortran routine to perform numerically intensive operations, or a Pascal program can call a C routine to interface with a device driver.

Another benefit of the XL compiler family is that they all share a common linkage editor.

3.4 The AIX Compilation Process

The AIX compilation process differs slightly from traditional compilation processes, as illustrated in Fig. 3.4. The traditional C compilation process in UNIX starts with the C preprocessor, a program called "cpp." The preprocessor handles all lines that start with the #directive token, such as #include and #define (see Chap. 1 for details on preprocessor directives that are pertinent to this book). The preprocessor creates a file with the same base name as the original source file, but with a ".i" extension. The .i file is in ASCII format.

Next, the compiler reads in the .i file and compiles the code into assembler source. The results are stored in a file with the same base name as the original source code file, but with a ".s" extension.

The assembler, a program named "as," converts the assembler source in the .s file into an object file. The results are stored in a file with the same base name as the original C source code file, but with a ".o" extension. This is where the compilation process stops if the -c option is given when the compiler is invoked. Object files containing useful routines are often archived into libraries.

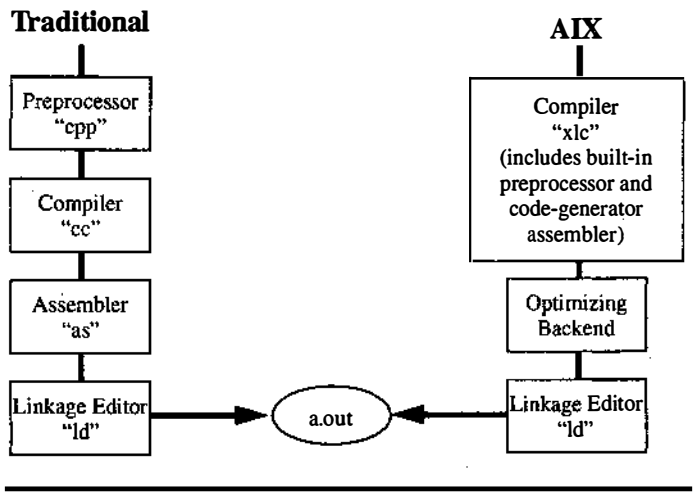

Figure 3.4 Traditional and AIX 3.2 program compilation.

Finally, to produce an executable file, the linkage editor is called. The linkage editor, a program named "ld," binds external objects to the program in order to satisfy unresolved references to functions and variables. For instance, many C programs call the printf() routine to send output to standard out. Since the code for printf() is not defined within the application, it must be brought in from somewhere else. In the case of printf(), the code is linked in from the standard C library, /lib/libc.a. The result of the linkage editor is an executable file named "a.out." The linkage editor can be told to name the file something other than a.out, for convenience.

The AIX compilation process accomplishes everything that the traditional UNIX compilation does, but there are a few differences. The AIX C compiler includes a built-in C preprocessor. It works with a superset of the standard cpp directives but includes a few AIX-specific pragma options, such as #pragma disjoint and #pragma isolated_call. See InfoExplorer for more information on these directives.

As mentioned earlier, the AIX C compiler consists of a front end, which produces an intermediate language, and an optimizing backend. The compiler backend includes a code generator, similar to the standard assembler, which produces an object file. While the compilation process does not use the traditional assembler "as," an assembler is included with AIX, allowing one to develop code using the RISC System/6000 and POWER/PC assembly language. However, it is difficult, using the assembler, to create code that is more efficient and faster than code optimized by the compiler backend. For example, the AIX optimizer treats all auto-type integers as register variables by assigning them to virtual registers. The -O option must be used when invoking the compiler to specify optimization.

3.5 The AIX 3.2 Linkage Editor

The AIX 3.2 linkage editor creates object files and executable files. It is named "ld" and is called for the last phase of the compilation process as long as the -c option is not specified when the compiler is invoked. The AIX 3.2 linkage editor performs dynamic binding and static binding, as described in the next section of this chapter. It also creates shared object code, which is described in Chap. 4.

Static binding

The "traditional" UNIX compilation process described above illustrated a linkage editor that performed static binding. This means that the code and data of all external symbols (a symbol is the name of a variable or a function) are brought together and placed in the executable file that the linkage editor produces. For instance, in the case of an application's call to the printf() routine, the code for printf() is copied from the standard C library into the application itself. Figure 3.5 illustrates static binding.

Dynamic binding

Dynamic binding was introduced to AIX in Version 3.1. It allows the linkage editor to delay resolving external symbol references until the application is exe-

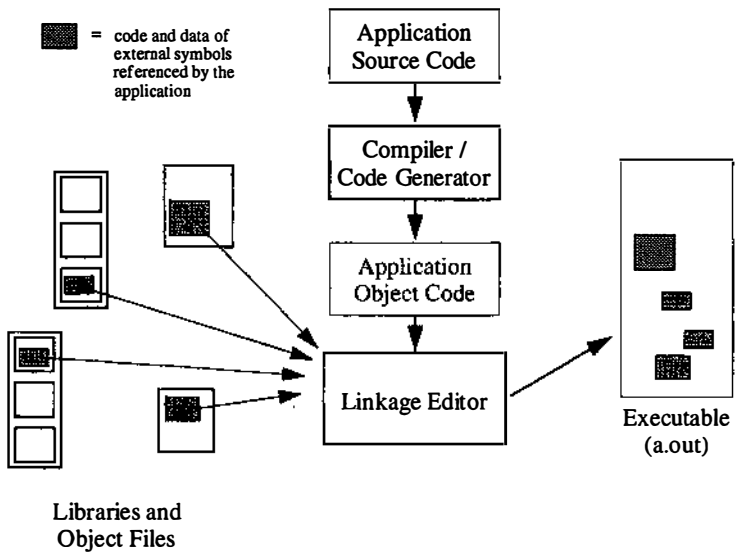

Figure 3.5 Static binding.

cuted. Instead of copying the code for the printf() routine into the application, the linkage editor places a small amount of "glue code" into the application. The glue code provides the name of the symbol (function or variable), as well as the file name of the library or external module where the symbol can be found. When the program is executed, the kernel's loader uses the glue code to locate and resolve the symbol. In the case of the printf() routine, the kernel's loader locates printf() in the standard C library (/lib/libc.a) and loads the code into the process image. Figure 3.6 illustrates dynamic binding.

Obviously, the biggest advantage of dynamic binding is that executable files are smaller, since code from frequently called routines is not copied to each executable file. Another benefit of dynamic binding is that since library routines are not linked to the application until run-time, library routines can be changed without needing to relink applications to them. The next time the application is executed it loads the new version of the library routine.

> **Author's Note:** I always point out that one of the possible disadvantages to dynamic binding is that the next time the application is executed it loads the new version of the library routine. When a new version of a library routine is introduced on the fly, it has been known to "break" an application or two.

A disadvantage of dynamic binding is that the glue code serves as a promise to the application that the specified files that define the external symbols will be available at run-time. If, for any reason, a file is not available, the promise is bro-

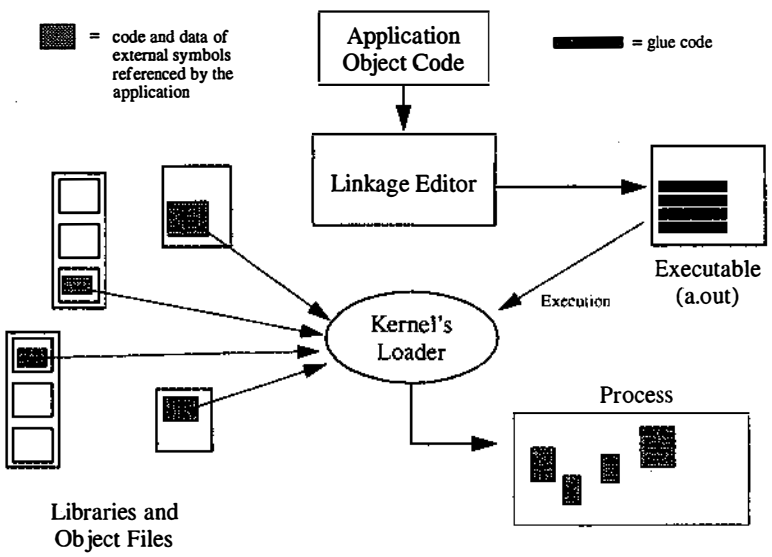

Figure 3.6 Dynamic binding.

ken and the loader fails when a user tries to execute the program. One must not remove, move, or change the name of any file that contains definitions of dynamically bound symbols. This also means that in order to distribute any application that is dynamically bound to libraries or external modules, the libraries and modules must also be distributed. Static binding creates a completely self-contained application. AIX 3.2 allows both dynamic and static binding.

Creating an object file for dynamic binding

Figure 3.7 illustrates how an object file is created so that it can be dynamically bound to an application. The source code file, colors.c, has three functions and a global variable with an initial value. The first step is to compile the source code file. It is necessary to use the -c flag when invoking the compiler to avoid calling the linkage editor. Since this source file does not have a main() function, the linkage editor would fail. The second step is to create an export list file. This is an ASCII file that lists the names of all symbols to be exported by the object file for dynamic binding and can be created with any text editor. The file named colors.exp in the example below is the export file. The final step is to call the linkage editor (ld) with the -bE:colors.exp option. This option specifies the export file name. The example designates colors as the final name for the object file.

> **Author's Note:** The ld command in Fig. 3.7 includes the -e red option and the -lc option. The -e option specifies the name of the function to use as an entry point for the object file. Any function name from the source code will do. It's provided because the linkage editor requires that an entry point be specified. The -lc option must be used so that the linkage editor can locate the printf() subroutine in the

colors.c

```
int myglob=7;
red() {
   ...
}

blue() {
   ...
}

green() {
   ...
}
```

colors.exp

```
# Export file for colors.o
myglob
red
blue
green
```

```
xlc -c colors.c
ld -o colors colors.o -bE:colors.exp -e red -lc
```

Figure 3.7 Creating an object file for dynamic binding.

standard C library. These two options must be explicitly given when using the ld command. They need not be given when using the xlc (or cc) command to invoke the compiler because the AIX 3.2 C compiler uses a configuration file, named /etc/xlc.cfg, to specify defaults to the linkage editor when it is called by the compiler. The configuration file instructs the linkage editor to include /lib/crt0.o, which makes the resulting output file executable by specifying the program's entry point, and to include the standard C library when resolving symbols. The ld command has no configuration file.

Creating an application for dynamic binding

Figure 3.8 illustrates how an application is created to dynamically bind with the colors object file from Fig. 3.7. First, an import list file is created. Like the export list file, it is an ASCII file. The import list file uses the #! syntax to specify the absolute or relative path name for the file to which the application will bind. Each #! path line is followed by the names of the symbols to be imported from the file. Multiple path names may be specified within a single import list file. The import list file in this example is myapp.imp. Note that it includes only the symbols required for myapp.c. The second step is to compile the application. The -bI:myapp.imp option instructs the linkage editor, when it is called by the compiler, to create the glue code necessary to dynamically bind the colors object file to the application.

Author's Note: I use this example in one of my classes. I once had an observant student who noticed that the size of the myapp executable was larger when dynamically bound to the colors object than when statically bound. The reason is that the glue code required for dynamic binding in this case ended up being more code than the bound routines themselves. This is often true with programs this small.

myapp.c

```
extern int myglob
main()
{
...
red();
myglob++;
green();
}
```

myapp.imp

```
# Import file for myapp
#!./colors
myglob
red
green
```

```
xlc myapp.c -o myapp -bI:myapp.imp
```

Figure 3.8 Creating an application for dynamic binding.

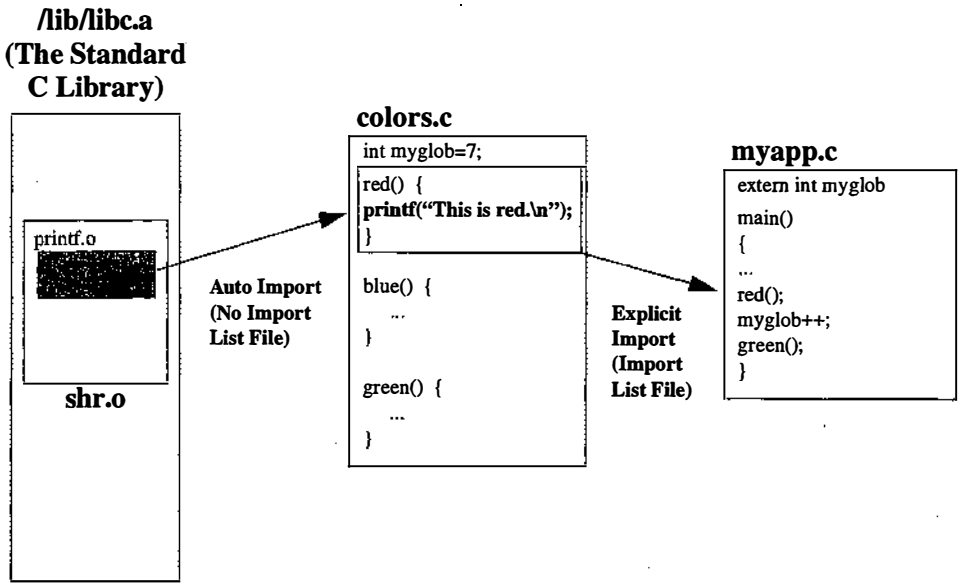

Figure 3.9 Dynamic binding and the standard C library.

To illustrate the relationship between myapp, colors, and the standard C library, Fig. 3.9 shows how each component is bound. Note that an object file that exports symbols, such as the colors object file, can also import symbols. One might notice that the colors object file did not use an import list file to dynamically bind with the standard C library. This is because AIX 3.2 performs automatic importing of symbols when linking to a shared object or shared library. The standard C library contains many shared objects, one of which includes the printf() subroutine. Shared objects and shared libraries are discussed in Chap. 4.

Load-time dynamic binding

The dynamic binding technique described in the previous section is called exec-time dynamic binding. This means that the resolution of external symbols takes place when a program is executed. AIX 3.2 supports another type of dynamic binding called load-time dynamic binding. This technique loads an object file into the process when the program running in the process calls the load() subroutine. Parameters to the load() subroutine include the name of the loadable object file and a flag option to control how the object file is loaded. The load() subroutine returns a pointer to the function that has been designated as the object file's entry point. The code of the object file is loaded into the process's data region (see Sec. 3.7). The unload() subroutine is called to unload the object file code from the process's data region when it is no longer needed.

See the manual pages or InfoExplorer for more information on the load() and unload() subroutines.

Load-time dynamic binding is useful when applications need to call a large utility that is required infrequently. By loading the large utility on-the-fly instead of upon execution of the program, the virtual memory requirements of the process are reduced. If the utility is not required by the user, it is never loaded. Figure 3.10 provides an example of load-time dynamic binding. Notice the definition of the entry point for the loadable module. Here, it is important to specify the desired entry point, as it links the two components.

> **Author's Note:** The load-time dynamic binding technique is similar to PC-DOS overlays.

3.6 The XCOFF File

The object file created by the code generator is in a format known as XCOFF (eXtended Common Object File Format), an IBM variation of AT&T's COFF (from System V Release 2). Figure 3.11 illustrates the components of the XCOFF file, which are also described by the header files listed. The structure of the XCOFF file is described in /usr/include/xcoff.h. XCOFF begins with a file header structure of type filehdr as defined in /usr/include/filehdr.h. The file

bigmod.c

```
/* Big routine, hardly ever used */
bigmod()
{
...
}
```

```
xlc -c bigmod.c
ld -o bigmod bigmod.o -e bigmod -lc
```

myapp.c

```
...
char *libpath="/lib:/usr/lib:.";
main()
{
int (*funcp)();
...
if((funcp=load("bigmod", libpath))==0)
{
perror("load");
exit(1);
}
...
(*funcp)();
...
if(unload(funcp))
{
perror("unload");
exit(2);
}
...
}
```

```
xlc -o myapp myapp.c
```

Figure 3.10 Load-time dynamic binding.

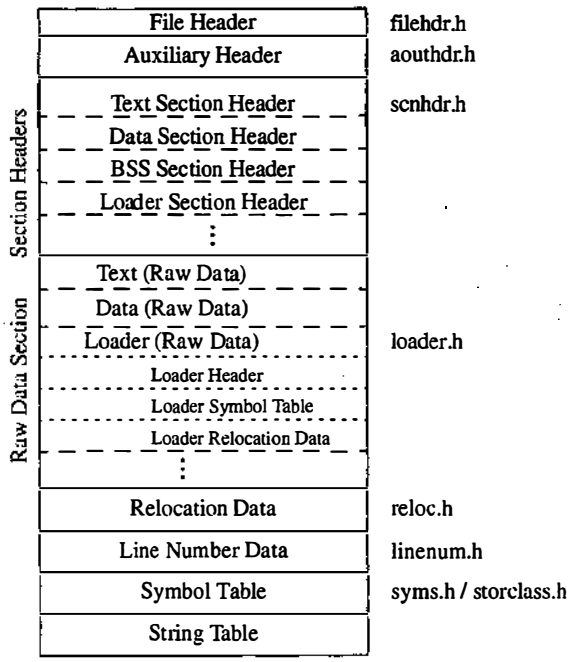

Figure 3.11 The XCOFF file image.

header includes fields that describe attributes of the XCOFF file. A field called f_magic indicates the target machine for the object file or executable.

The XCOFF file header is followed by an auxiliary header described by the aouthdr structure defined in the /usr/include/aouthdr.h header file. This structure also contains a magic number that indicates whether the file is executable, and if so, how it should be executed. It also holds the size values for the other sections of the XCOFF file. Another field in the auxiliary header, o_modtype, is an array of two characters that describes the module type. A module type of "RE" indicates a shared object (reentrant code). The use of shared objects is explained later in this chapter. The third part of the XCOFF file is the section headers. The section headers define attributes of each of the XCOFF sections, including their names, sizes, and starting locations. The structure scnhdr is defined in the header file /usr/include/scnhdr.h.

After the headers, the XCOFF file has a raw data area, which contains the main sections. The text section holds the machine instructions that represent the code of the program. The data section holds declarations of global variables which have initial values. Noninitialized global variables are tagged in the BSS section header within the section headers area, as illustrated in Fig. 3.11. BSS stands for block started by symbol and is actually an old mainframe assembler

term. It indicates that memory must be allocated at run-time to hold these variables.

The next section is the loader section. This component constitutes part of the AIX-specific information in the XCOFF image, as a loader section is not part of the standard COFF format. The loader section contains information used by the kernel loader in order to perform dynamic binding. This is where the "promises" made in the import list files described in Sec. 3.5 are stored.

Other sections of the XCOFF file include the debug section, which is present if the program was compiled with the -g option to the cc or xlc commands, the symbol table section, for locally defined symbols, and a string section, which holds symbol names that are longer than eight characters. The details of these sections are more appropriate for a book on compiler internals and are beyond the scope of this book.

3.7 The AIX 3.2 Process Image

The previous section of this chapter described the XCOFF image of a program. When a program is executed it runs as a process. This section describes the image of a process.

> **Author's Note:** To state that a program runs as a process is an oversimplification. It is more correct to say that a process provides an environment for the execution of a program. We discuss how processes are created and how they execute programs in Chap. 5.

The two major components of a process are the text and the data. The text is the machine instructions that make up the program code. The data are the user variables of the program and their values, and the system information about the process. AIX 3.2 allocates a memory segment for the text and a memory segment for the data of a process. The segments are maintained by the virtual memory manager (detailed in Chap. 4). Each segment is 256 megabytes of virtual memory.

> **Author's Note:** Actually, every process has 16 segments, which are described in Chap. 4. As an introduction to the process image, we discuss only the two most important segments here.

The text segment

The text segment holds the code of the executing program. Its contents are directly mapped from the raw text section of the XCOFF file. The kernel allocates only one text segment for the code of a given program, no matter how many processes are running the program concurrently. In other words, the text segment is shared by all processes executing the same program. The text segment is protected as read-only memory to prevent any process from changing

the code while it is being executed. In this way, AIX does not support self-modifying code.

> **Author's Note:** Actually, there is a way to implement self-modifying code. Load-time dynamic binding, described in Sec. 3.5, loads the code of a specified object file into the calling process's data segment. This is necessary since the text segment is read-only and cannot be changed during execution. It would also be inappropriate for the loadable object file to be loaded into a segment that might be shared by other processes, as could be the case with a text segment. A process has read-write authority for its own data segment. Therefore, code loaded into the data segment of a process via the load() subroutine can be altered. This is not the primary intent of load-time dynamic binding, but rather a coincidental characteristic.

Process data

A process has various classes of data that must be described before detailing the layout of the process's data segment. Each class is handled differently. The following list briefly describes some of the classes a process may have. It is not the intent of this section to describe the scope or other characteristics of all the data classes supported by the C language. The reader is encouraged to consult the C Language Reference Guide that is part of the InfoExplorer on-line documentation for more details.

Initialized global data. These are variables that are declared outside of the body of a function and have been given initial values by the program. The compiler places these variables together within the program's XCOFF file. The kernel's loader allocates memory for these variables within the process's data segment at run-time.

Noninitialized global data. These are variables that are declared outside of the body of a function but have not been given initial values. The compiler groups these variables together in the BSS section of the XCOFF file (see Sec. 3.6). Since these variables have no values at compile-time, the compiler does not allocate actual space in the XCOFF file for the variables. For instance, if a global array of 2000 emprec structures is declared by a program (where an emprec structure contains an employee record definition), the compiler does not actually allocate that much space in the XCOFF file. The loader, at run-time, does allocate enough memory in the process's data segment to hold the array. It also initializes the global data according to standard C language rules. See the C Language Reference Guide under InfoExplorer for more information on initialization rules.

Automatic local data. These are variables declared within the body of a function. They may be initialized or noninitialized. Their scope is the body of the function. In other words, when the function ends, they no longer exist. They are stored on a stack within the process's data segment. Stacks are described shortly.

Static local data. These are variables that are declared within the body of a function. Like automatic local data, their scope is the body of the function; however, they maintain their values after the function ends. If the function is called subsequent times, these variables are recalled with their last known values. For this reason, they are not stored on a stack. They are stored along with the initialized global data of the process.

Dynamically allocated global data. Programs often require more memory as they run to accommodate dynamic growth of data. Sometimes the additional memory is only needed temporarily. The malloc() subroutine provides programs with the ability to access more memory from their data segment. The new memory is allocated above the BSS and is called the heap. In other words, using malloc() allows a program to "throw more memory on the heap." The AIX 3.2 implementation of the malloc() subroutine is a little different from the way other UNIX-based systems implement it. This is described later in this chapter.

Figure 3.12 illustrates how the different classes of data are treated by the compiler and the kernel's loader.

How a stack works

Each function called within a program usually has local data. The local data of a function have a scope of the function's duration. Since functions can call other functions, local data of the calling function must be saved while the called func-

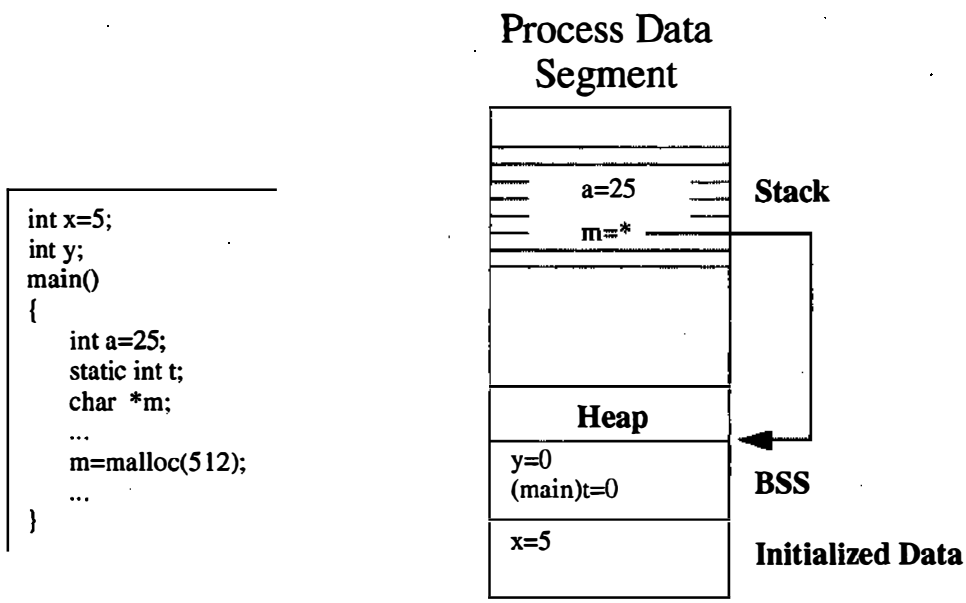

Figure 3.12 Data classifications.

tion is executing. When the nested called function completes, the caller's data must become available again. This is accomplished by using a stack. Each function has a stack frame. The size of each stack frame is determined by the compiler and corresponds to the needs of each function. When a function calls another function, a stack frame is created for the new function and is "pushed" onto the stack. AIX 3.2 stacks grow downward in memory, so a new stack frame is pushed below the current stack frame. The CPU maintains a stack frame pointer that points to the memory address of the current stack frame. When the nested function completes, its stack frame is "popped," or discarded. The stack frame pointer is changed to point back up to the previous stack frame. Figure 3.13 illustrates the code example and how the stack is managed.

In the example, the main() function has its own stack frame to hold the local variables "a" and "b". When main() calls foo() a stack frame is pushed on the stack for the foo() function. It holds the "x" and "y" local variables of foo(). When foo() calls bar() a stack frame is pushed for the bar() function. It holds the "a" and "x" local variables of bar(). Even though different functions have local variables of the same name, the system can tell them apart because each has its own stack frame in memory.

> **Author's Note:** I often explain stacks when teaching basic C programming since it helps illustrate the difference between passing parameters by value and passing parameters by reference when calling a function. Passing parameters by value copies the values from the caller function's stack frame to the called function's stack frame. Passing parameters by reference means passing the addresses of the original data in the calling function's stack frame to pointers defined in the called function's stack frame. This allows the called function to indirectly manipulate the values of the variables in the calling function's stack frame.

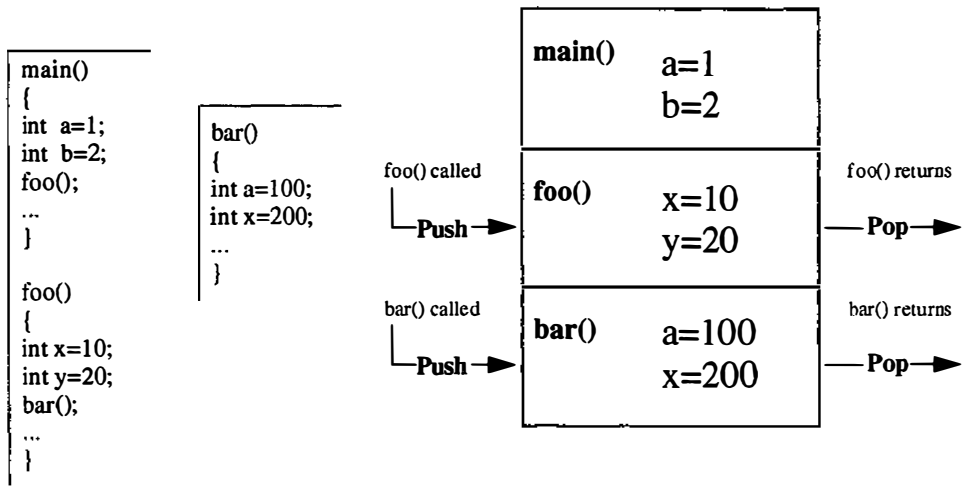

Figure 3.13 A stack example.

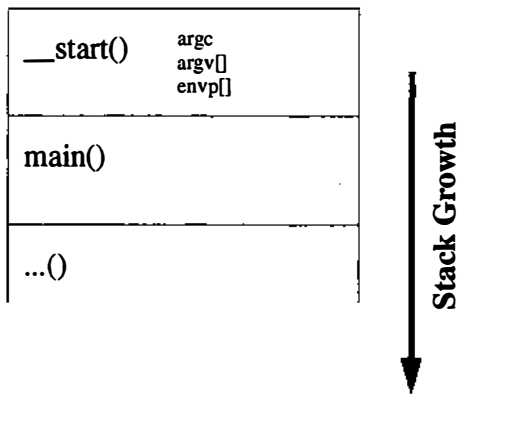

Figure 3.14 AIX 3.2 stack frame allocation.

By default, the AIX 3.2 linkage editor designates a program's entry point as a routine called _start(). The _start() routine is found in the /lib/crt0.o object file. The _start() routine calls main(), passing to it the parameters argc, argv[], and envp[]. Therefore, the first frame on the stack is allocated to _start(). The second frame is allocated to main(). Each stack frame has a pointer link back to stack frame of the function that called it. Figure 3.14 illustrates how AIX 3.2 builds a process's user stack.

The data segment

AIX 3.2 carves the 256-megabyte data segment of a process into two regions as shown in Fig. 3.15. The kernel region is 524,288 bytes (1024 * 512, or 1/2 megabyte). The remaining space in the segment is allocated to the user region. The point of separation between the two regions is called the "red zone." A

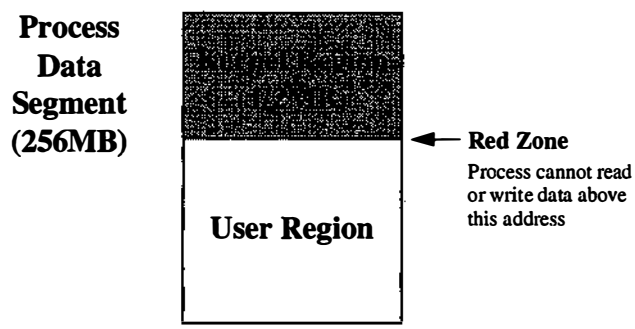

Figure 3.15 The process data segment.

process can read and write within its own user region, but it may not directly access the memory allocated to the kernel region (above the red zone).

When a program is executed, the kernel's loader places the program's initialized global data starting at the lowest address of the user region. This is also the lowest address of the data segment. The amount of memory required for initialized global data is known at load time and does not change during the lifetime of the process.

The loader allocates memory for the noninitialized global data directly above the initialized global data. It does so based on information found in the BSS section header of the program's XCOFF file. All arrays are expanded to their declared dimensions. The BSS data are initialized according to C language rules. For instance, all integers are initialized to zero. The loader sets a break value, called brk, at the top of the BSS. The brk value indicates the highest memory address assigned to global data.

As mentioned earlier, the malloc() subroutine, or one of its relatives [calloc() and realloc()], allocates additional global memory during program execution. This is done by raising the brk value. This means that the heap is allocated from memory directly above the BSS and grows upward into the data segment's user region, as shown in Fig. 3.16. System administrators and programmers can use the ulimit command to set the maximum amount of memory that can be allocated to global data, thus limiting the growth of the heap.

Author's Note: Actually, the malloc() subroutine calls the sbrk() system call, which resets the break point.

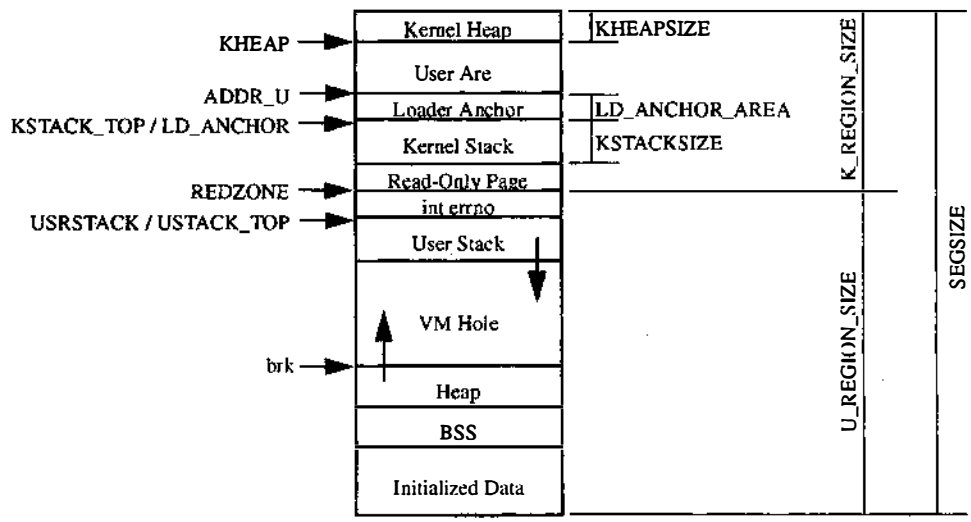

Figure 3.16 Data segment details.

At the top of the data segment's user region is an integer called errno. This variable, which is global and can be accessed by the process, is set to some value when a system call, running on behalf of the process, fails. The system call records the reason for the failure in the errno variable. The header file /usr/include/errno.h defines symbolic constants to match the errno values.

The process's user stack is directly below the errno variable. It holds the local data for user code functions. It grows downward into the data segment's user region as functions are nested and stack frames are pushed. It retreats upward as the nested functions complete and the stack frames are popped. System administrators and programmers can use the ulimit command to set the maximum amount of memory that can be allocated to the user stack.

The area between the top of the heap (the brk point) and the bottom of the user stack, for lack of a better name, is called the VM (virtual memory) hole. It is the remaining memory of the data segment's user region after accounting for the initialized global data, BSS, heap, errno, and user stack. This is a very large area and allows for a great deal of process data. It is, however, sometimes not enough memory for applications that require more than 256 megabytes of user data, such as some graphically based engineering and scientific programs. Chapter 4 describes how a program can request additional memory for data. IBM calls it the "huge data model" technique.

System calls and device driver routines running on behalf of a process sometimes need to allocate memory dynamically. They do so by calling the xmalloc() kernel service. Since AIX 3.2 has a preemptable kernel, which means that a system call running on behalf of a process is not guaranteed to complete before the process is preempted, each process must have its own kernel heap. This is found at the top of the data segment's kernel region. It has a size of about 398 kilobytes.

Below the per-process kernel heap is the user area. Each process has a user area, which holds information about the process. The user area is defined as a user structure in the /usr/include/sys/user.h header file and has a size of 18,576 bytes. It includes the process's credentials (user and group IDs), current directory, resource usage and limit information, and signal mask. It also includes the device name of the process's controlling terminal, as well as the name of the program executing within the process. The user area contains an array of pointers to files opened by the process. The array is called the file descriptor table. Details of the user area are provided in Chap. 5.

> **Author's Note:** The user area is sometimes called the U-Area or u-block. The size of a process's user area is 18,576 bytes in AIX 3.2. On most UNIX-based systems, the user area is pageable. In fact, a comment in the /usr/include/sys/user.h header file states "The u-block contains information about the process that need not be in memory when the process is swapped out. It is pinned when the process is swapped into memory, and unpinned out when the process is swapped out." AIX 3.2 pins the first two pages of the user area, even when the process is swapped out. The remaining portion of the user area contains the file descriptor table and is always pageable.

Below the user area, the process's data segment contains a loader section, used for dynamic binding. This section has a size of 256 bytes.

Below the loader section is the per-process kernel stack. It is used by system calls running on behalf of the process in the same way that the user stack is used. System calls can call other system calls or small routines called kernel services. Since each kernel service may have its own local data, a stack is required. It can grow to a size of approximately 96 kilobytes, which takes it one page (4 kilobytes) from the lowest memory address of the kernel region of the data segment.

The symbolic constant names displayed in Fig. 3.16 come from the /usr/include/sys/pseg.h header file. The name of this file comes from "private segment." The #defines take a while to sort through, but Fig. 3.16 should help.

The malloc(), realloc(), and free() subroutines

IBM changed the way the AIX kernel allocates and deallocates heap memory from AIX 3.1 to AIX 3.2. These are techniques used by the kernel when malloc(), realloc(), and free() are called. As described earlier, the malloc() subroutine requests that additional memory be allocated for data. The additional space is allocated from a chunk of contiguous virtual memory within the process's data segment. If a process calls malloc() multiple times, while each malloc() results in the allocation of contiguous virtual memory chunks, there is no guarantee that the chunks themselves will be contiguous to one another. It is also important to understand that each chunk requires additional memory to manage the allocated space. The management space is called the chunk's prefix. Programs must be careful not to clobber the prefix of a chunk.

> **Author's Note:** I use the term "chunk" to refer to the memory space returned by a call to malloc() or realloc(). The AIX documentation refers to it as a "block." I prefer not to use the term "block" since it is used many other places in AIX to refer to objects of a fixed size.

The realloc() subroutine is used by a process to change the size of an already allocated chunk of virtual memory in the heap. It often results in the original chunk's being moved to another virtual memory location to assure that the resized chunk remains contiguous.

When a process no longer needs memory that was dynamically allocated with malloc(), it can call the free() subroutine to deallocate the virtual memory. The free() subroutine places the freed chunk of contiguous virtual memory onto a free list per process. It is important to understand that the memory deallocated by the free() subroutine never goes back to the system's free virtual memory. It only becomes available to be allocated once again by another call to malloc() or realloc().

When malloc() or realloc() are called, the system looks first at the process's free list of memory chunks. The free list consists of previously allocated, then freed contiguous chunks of virtual memory. If a process has never called malloc(), the free list is empty. In that case, the sbrk() system call is used to raise

the break point of the heap. It does this by moving the _edata location. The amount of memory grabbed by sbrk() is determined by the size of the malloc() request, but rounding up does occur. Any memory grabbed by sbrk() that is beyond what was requested by malloc() is placed on the free list. Any memory deallocated by the free() subroutine is also placed on the free list.

The AIX 3.1 method

AIX 3.1 uses a set of 28 hash buckets to manage the heap. Each hash bucket points to a linked list of free virtual memory chunks of a given size. Table 3.1 lists the hash bucket indices and their corresponding memory chunk sizes. Each virtual memory chunk requires 8 bytes of management overhead.

When a malloc() call is made, the system queries the hash bucket that corresponds to the memory size requested. If the hash bucket points to an existing chunk, that chunk is allocated. If the hash bucket points to null (the bucket is empty), sbrk() is called to add blocks to the list. The sbrk() system call never grabs less than a 4096-byte page; therefore, if the requested memory size is smaller than a page, the request is honored and the remaining memory from the page is carved into chunks equal to the requested size and placed in the linked list maintained by the hash bucket. Figure 3.17 provides an example. When a chunk is freed, it is placed at the head of the linked list maintained by the hash bucket that corresponds to the freed chunk's size.

TABLE 3.1 AIX 3.1 malloc() Hash Buckets

Bucket	Block Size	Sizes Mapped	Pages Used
0	16	0 - 8	
1	32	9 - 24	
2	64	25 - 56	
3	128	57 - 120	
4	256	121 - 248	
5	512	249 - 504	
6	1K	505 - (1K-8)	
7	2K	(1K-7) - (2K-8)	
8	4K	(2K-7) - (4K-8)	2
9	8K	(4K-7) - (8K-8)	3
10	16K	(8K-7) - (16K-8)	5
11	32K	(16K-7) - (32K-8)	9
12	64K	(32K-7) - (64K-8)	17
13	128K	(64K-7) - (128K-8)	33
14	256K	(128K-7) - (256K-8)	65
15	512K	(256K-7) - (512K-8)	129
16	1M	(512K-7) - (1M-8)	257
17	2M	(1M-7) - (2M-8)	513

(*Continued*)

TABLE 3.1 AIX 3.1 malloc() Hash Buckets (*Continued*)

Bucket	Block Size	Sizes Mapped	Pages Used
18	4M	(2M-7) - (4M-8)	1K + 1
19	8M	(4M-7) - (8M-8)	2K + 1
20	16M	(8M-7) - (16M-8)	4K + 1
21	32M	(16M-7) - (32M-8)	8K + 1
22	64M	(32M-7) - (64M-8)	16K + 1
23	128M	(64M-7) - (128M-8)	32K + 1
24	256M	(128M-7) - (256M-8)	64K + 1
25	512M	(256M-7) - (512M-8)	128K + 1
26	1024M	(512M-7) - (1024M-8)	256K + 1
27	2048M	(1024M-7) - (2048M-8)	512K + 1

If the size of an existing chunk is changed via the realloc() subroutine, the system checks to see if the new size still falls within the range of sizes supported by the current chunk (see Table 3.1). If it does, then the system updates the length value in the chunk's prefix and returns the same chunk. If the new size is greater than the size of the current chunk, a new chunk is allocated from the appropriate hash bucket list and the data are moved from the current chunk to the new chunk. The smaller chunk is then placed on the free list.

The AIX 3.2 method

AIX 3.2 uses a binary tree to maintain free lists of virtual memory chunks within the heap. There is no limitation to the number of chunk sizes supported by the tree. This method also helps reduce virtual memory fragmentation.

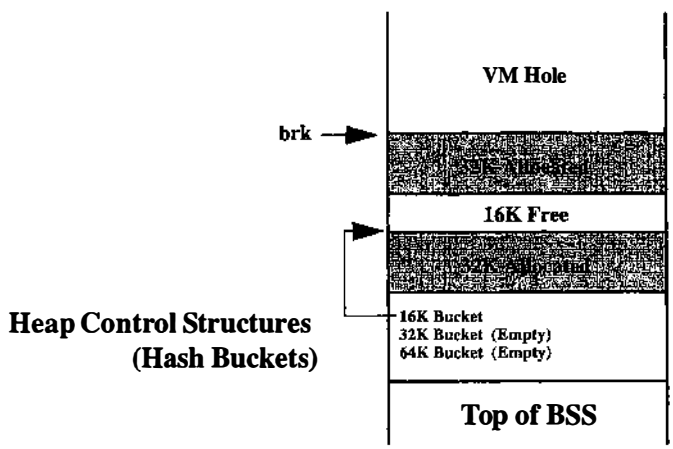

Figure 3.17 An AIX 3.1 malloc() example.

When a malloc() occurs, the system searches the tree for the first memory chunk large enough to accommodate the request. If the chunk found is larger than the requested size, the chunk is divided into two chunks. One chunk is the requested size and is returned to the program. The other chunk is the remainder (called the "runt") and is placed back on the tree at the appropriate location. If no chunk of the requested size is found on the tree, sbrk() is called to extend the heap. The requested chunk size is returned to the process and any remaining runt is placed on the tree at the appropriate location.

The most interesting part of the AIX 3.2 heap management occurs when a chunk is freed. Instead of simply placing the freed memory chunk back on the tree free list, the system checks to see if an adjacent virtual memory chunk exists on the tree. If it does, the two chunks are combined to form a larger chunk of contiguous virtual memory, which is placed back on the free list at the appropriate location in the tree.

When one compares the two methods, one sees that the AIX 3.1 hash bucket technique often returned memory chunks that were much larger than required by the process. The AIX 3.2 tree technique returns memory chunks that are often much closer in size to what the process requested. This means that it is possible for some AIX 3.1 applications to fail when executed in AIX 3.2. To provide compatibility with AIX 3.1 applications that rely on the hash bucket method of heap allocation, AIX 3.2 includes an environmental variable called MALLOCTYPE. For AIX 3.1 applications that experience problems in AIX 3.2, the following command should be issued at the shell prompt prior to executing the application:

```
export MALLOCTYPE = 3.1
```

There is another unusual characteristic of the AIX 3.2 malloc() subroutine that has to do with paging space allocation. It is discussed in Chap. 4.

3.8 The System Call Subsystem

The AIX 3.2 system call subsystem provides the API (application programming interface) to the kernel. As mentioned earlier, applications running as user processes cannot directly access kernel memory. They must use system calls to access kernel data and services. This section describes how system calls are used and what happens within the AIX 3.2 kernel when they are used.

System calls and library routines

System calls and library routines look alike but operate in a very different way. Library routines are commonly used functions, defined within object files that are archived into library files. These object files are linked to applications such that the code of the functions becomes part of the application at link time, if static binding is used, or at run-time, if dynamic binding is used. When an application calls a library routine function, the code executes in user mode and

the data are stored in the user stack. The kernel is not involved in executing a library routine.

System calls are commonly used functions that are part of the kernel. They perform system-level services on behalf of the calling process. When a process invokes a system call, a mode switch occurs. The mode switch changes the execution environment from user mode to system mode. The normally protected memory allocated to the kernel (/dev/kmem) becomes available to the application while the system call executes. A separate kernel stack is used to hold the data associated with system calls. Figure 3.18 illustrates the mode switch.

The symbol hash table

AIX 3.2 employs a unique way of accessing the code of a system call. When a process issues a system call, the name of the system call is used to search the kernel's symbol hash table. This table contains the names of all valid system calls. When a match is found in the symbol hash table, its corresponding index value is used to vector into an array of pointers to functions. The retrieved function address points to the system call code. The symbol hash table and array of pointers is shown in Fig. 3.19.

System call return

When a system call completes, it returns a value to the calling process. This is true whether the system call succeeds or fails. It is the responsibility of the program that issued the system call to check the return value. A programmer must rely on the system documentation for explanations of valid return values from

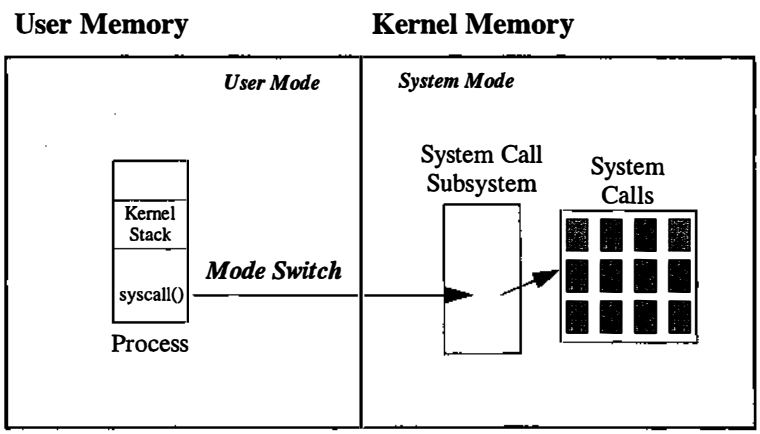

Figure 3.18 The mode switch.

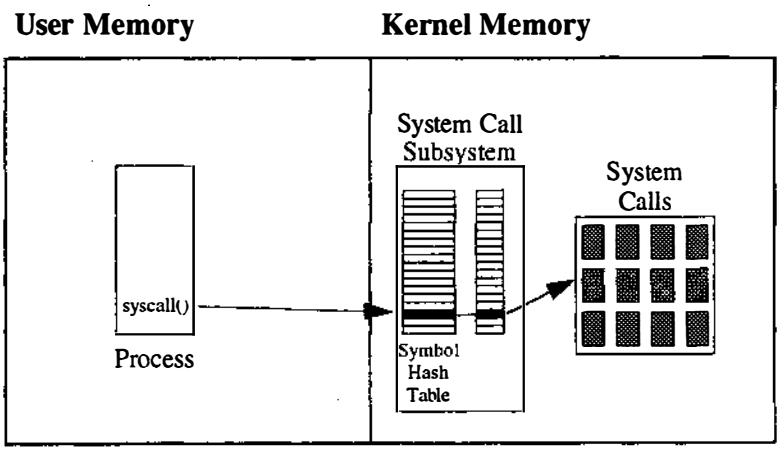

Figure 3.19 The symbol hash table.

each system call. The documentation for each system call describes the valid return values for success and failure.

When a system call fails, it returns a specific value that indicates failure to the calling process. It also assigns a value to the errno global variable described earlier in this chapter. The /usr/include/sys/errno.h header file defines symbolic names for the possible values for the errno variable. A program can query the errno value to determine the reason for a system call failure.

A system call example

Figure 3.20 provides an example of the open() system call. If the open() system call succeeds in opening the specified file, it returns the integer value of the

```
#include <fcntl.h>
...
main()
{
int fd;
...
if((fd=open("mydata".O_RDWR))==-1)
      {
      perror(argv[0]);
      exit(1);
      }
...
}
```

Figure 3.20 A system call example.

assigned file descriptor. The valid values for file descriptors are 0 through 1999. (The use of file descriptors is detailed in Chap. 7.) If the open() system call fails, it returns a -1 value. The program tests for the -1 value. If the open() system call fails, the program reports the error and exits. The open() system call fails if the file named "mydata" is not found in the current directory or the process does not have the authority to open the file for reading and writing.

Chapter

4

AIX 3.2 Memory Management

4.1 An Introduction to Virtual Memory

Virtual memory is a concept supported by most operating systems. Its primary purpose is to allow many programs of various sizes to run at the same time by giving the computer system the appearance of having more physical memory than it actually has. This is accomplished by supplementing the system's physical memory with a secondary storage medium such as disk space. Programs run in virtual memory, unaware of the fact that the virtual memory is a combination of the system's primary storage (physical memory) and secondary storage (such as disk space). A component of the operating system, the virtual memory manager (VMM) in the case of AIX 3.2, controls how the physical memory and secondary storage are used.

Another aspect of virtual memory is that every process has a virtual address range of byte zero to the highest addressable byte in the image of the process. Where each byte of a process resides in physical memory when it is accessed is inconsequential. The VMM translates each virtual address in a process's address space to its current physical address. Portions of a program's text or data can reside in any physical memory location at any time.

Swapping

Early UNIX systems used a virtual memory technique called swapping. When a program was executed an entire process image was created in physical memory. The process image was similar to the process image described in Chap. 3 but was much smaller, as to allow many processes to exist in physical memory simultaneously. Figure 4.1 illustrates many processes in memory at once. A process's image had to be in memory in order for the process to be dispatched. A process whose image was in memory was said to be "in core." It is common

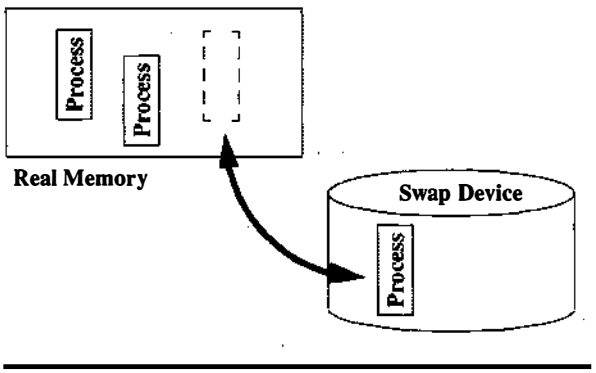

Figure 4.1 Swapping.

for processes to spend much of their time waiting on I/O requests to complete or waiting on some other event to occur. Processes that are waiting are said to be sleeping and cannot be dispatched. Older UNIX systems would move the images of sleeping processes from physical memory to secondary storage, which, in this case, was a raw disk partition called swap space. Such processes were said to be "swapped out." This would free up physical memory for other processes. When the event waited on by a sleeping process occurred, the operating system would wake the process, but before the process could be dispatched, it had to be swapped in from swap space to physical memory. Figure 4.1 illustrates swapping. Swapping had two major disadvantages. It was slow, since copying entire process images was involved, and it limited the size of a process.

Demand paging

Most UNIX-based systems today use a virtual memory technique called demand paging. It was introduced in BSD systems in the early eighties, although the concept of demand paging was found in mainframe operating systems prior to that. In demand paging virtual memory, a process's text and data are carved up into pages of a specified size. Most systems use a page size of 2, 4, or 8 kilobytes. When a program is executed, the VMM allocates physical memory to pages of the process's text and data as they are required. For instance, as a process runs, when a particular variable is referenced the VMM attempts to locate the data page that contains the variable in physical memory. If the page is found, the contents of the physical memory that holds the value of the variable are loaded into a CPU register for processing. If the page is not found a page fault occurs. The VMM locates the missing page in the secondary storage and pages it into memory. Once the page is in memory, the value of the variable can be loaded into a CPU register for processing. With demand paging, the secondary storage device is called "paging space."

The goal of demand paging is to keep the most popular pages in physical memory. When available physical memory runs short, the least popular pages

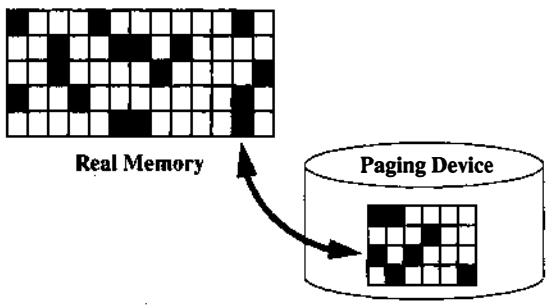

Figure 4.2 Demand paging.

(those that have not been referenced for a while) are paged out to paging space. Figure 4.2 illustrates demand paging. Most UNIX-based systems, including AIX 3.2, use a VMM technique that is a combination of swapping and demand paging. Most of the time AIX 3.2 uses demand paging to manage memory requirements. If the system becomes acutely short of free physical memory AIX 3.2 resorts to swapping entire processes in order to prevent a condition called "thrashing." Thrashing occurs when a system is spending more time moving pages in and out of physical memory than doing productive work. Thrashing is described in greater detail shortly.

> **Author's Note:** Some of the old swapping terminology persists today. For instance, paging space is still sometimes referred to as swap space. AIX 3.2 includes a command called "swapon" that is used to activate a paging space partition. Finally, the traditional UNIX scheduler process, which is responsible for arranging ready-to-run processes into run queues according to each process's priority, also had the responsibility of making sure that when a process was made ready-to-run (i.e., awakened from a sleep state when a specified event occurred) its image was swapped into physical memory. To this day, the scheduler still has the name "swapper," even though it swaps processes only when the system is thrashing. The swapper has process ID zero in AIX 3.2.

4.2 AIX 3.2 Virtual Memory

The RISC System/6000 supports real memory configurations from 16 megabytes to 2 gigabytes and beyond, based on the model. The AIX Version 3 kernel, working with the RISC System/6000 hardware, supports a virtual memory scheme of 4 petabytes.

> **Author's Note:** A petabyte is one million billion bytes, the level beyond a terabyte. Four petabytes is the same as 2^52. This is architecturally accomplished by using a 52-bit virtual address. While the idea of a system with 4 petabytes of virtual memory might have some programmers salivating with the thought of writing incredibly large programs, keep in mind that the 4 petabytes represents the total virtual memory size of the system. All processes must share this address space and

AIX implements a virtual memory scheme that provides 4 gigabytes per process. It must also be understood that utilized virtual memory must be backed by secondary storage, specifically paging space. The details of the per process usage of virtual memory are described shortly.

Virtual memory segments

AIX 3.2 uses a segmented memory scheme. Architecturally speaking, one can view the entire 4 petabytes of virtual memory as being carved into segments. Each virtual memory segment is 256 megabytes. Segments are allocated by the virtual memory manager as needed by processes and the operating system. The kernel includes a segment information table (SIT) to track allocated segments in a manner similar to how the process table tracks processes. Theoretically, the system can support 16 million segments (4 petabytes divided by 256 megabytes). Each segment is identified by a unique segment ID number, much as each process is identified by a unique process ID number. Segment ID numbers are often seen within kernel structures defined in header files as variables of type vmhandle_t. The header file /usr/include/sys/seg.h provides the typedef.

Each segment is carved into virtual memory pages. Each page is 4096 bytes. The RISC System/6000's real memory is divided up into 4K frames. It is at this point that virtual memory meets real memory in that the 4K virtual memory pages of segments are moved in and out of the 4K frames of real memory. Frames are the holders for the pages. The virtual memory manager uses a page frame table (PFT) to keep track of the state of real memory frames, and a series of external page tables (XPT) to keep track of the virtual memory pages. Figure 4.3 illustrates the components of virtual memory.

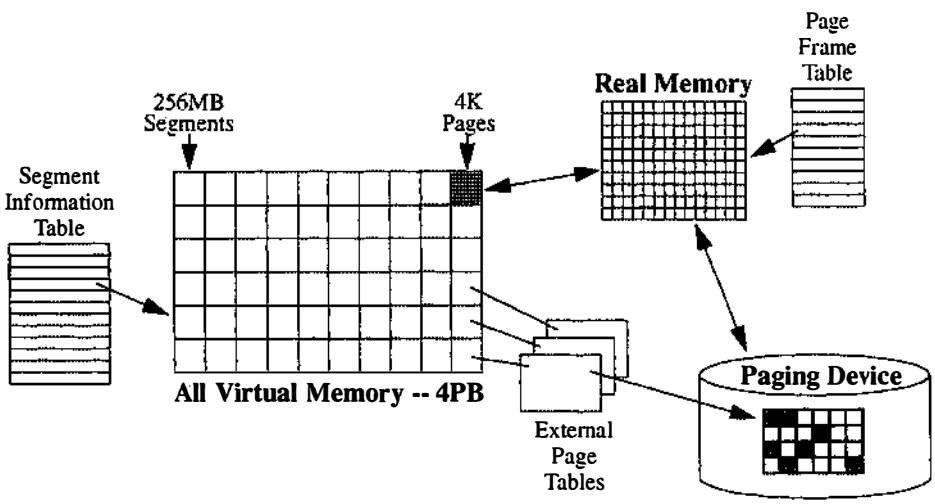

Figure 4.3 VMM components.

Segments and processes

The AIX virtual memory manager allocates 16 segments per process. Thus the virtual memory image of an AIX process is 4 gigabytes. The AIX compilers generate 32-bit effective addresses for each instruction and piece of data per program. These 32-bit effective addresses are used as pointers within the 4-gigabyte virtual memory image. Figure 4.4 illustrates the 16 segments that make up the virtual memory image of a process.

Segment 0—The kernel segment

The AIX kernel is always allocated to segment 0 by the Virtual Memory Manager. This segment is constant and remains part of every process's virtual memory image. This emphasizes the fact that the kernel is not a process itself, but rather a part of every process. Despite the fact that the kernel is part of every process, a process cannot directly access the kernel. Segment 0 is considered "off limits" to a process and can only be accessed via system calls.

Segment 1—The text segment

Segment 1 of each process contains the text of that process. The pages from the text section of the process image (from Chap. 3) are stored here. This indicates that, under normal conditions, the largest program (set of instructions) supported by AIX 3.2 is 256 megabytes. Recall that Chap. 3 described load-time

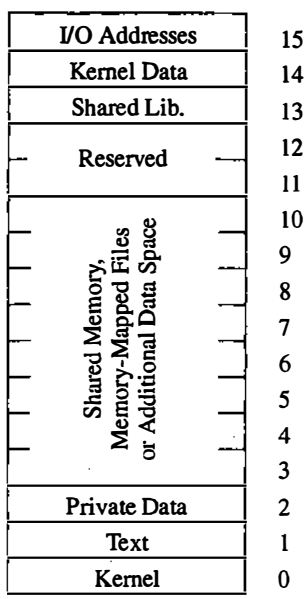

Figure 4.4 Process segments.

dynamic binding, which allows a program to load additional text into its data segment.

Text segments can be shared by multiple processes. For example, if three users on the same system are all running the vi editor, the instructions for the editor are loaded into memory only one time. Each user's vi process shares the same vi text pages. Text segment sharing is automatic in AIX 3.2.

The text segment of a process is read-only. The process cannot alter its content.

Segment 2—The data segment

The most interesting segment of a process is segment 2. This segment contains the initialized data, noninitialized data (BSS), heap, user area, user stack, and kernel stack, as described in Chap. 3.

The data segment's contents cannot be shared between different processes. Each process has its own private data segment. The data segment is often called the "per-process private segment."

Figure 4.5 illustrates the relationship of the process image, as described earlier, to the process's virtual memory segments. The kernel region of the data segment, which is 1/2 megabyte, can only be accessed by the kernel (for reasons that are explained shortly). The kernel stack is only used by system calls operating on behalf of the process. The initialized data, BSS, and heap sizes vary

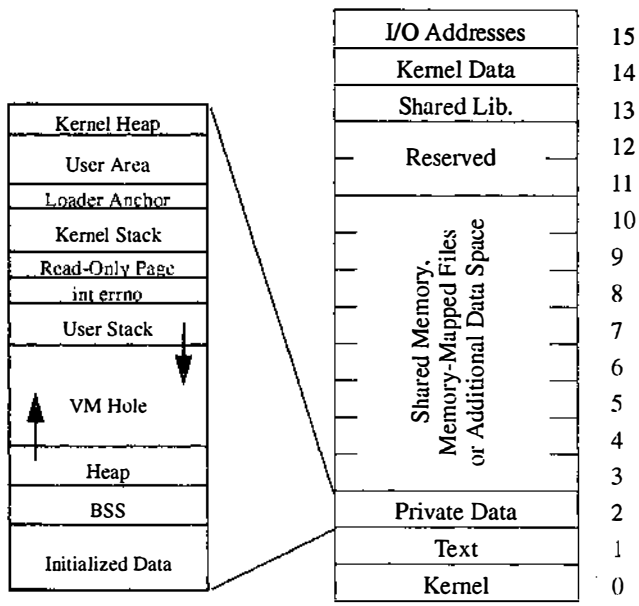

Figure 4.5 The per-process private data segment.

with each process. The fact that all of these sections are placed within the data segment helps one determine the size of the VM hole. The size of the VM hole is 256 megabytes minus 1/2 megabyte (the size of the kernel region) minus the initialized data, BSS, heap, and user stack.

As with the text segment, one might assume that 256 megabytes minus the small amount of space used for the kernel region would afford adequate space for user data. In most cases this is true. However, there are many applications, such as geophysical modeling, that require massive amounts of user data. Such programs can exhaust the available space in the data segment. AIX Version 3.2 provides the ability to allocate more segments for data storage. This concept, called the "huge data model," is detailed shortly.

A process has read and write capabilities on most of the pages of its own data segment. The only exceptions are the pages within the kernel region, which can only be accessed by the kernel.

Segments 3–10—Shared memory or memory mapped files

Every process has 8 of its 16 segments available for use as shared memory segments or for memory mapping files. Shared memory is a form of interprocess communication where more than one process accesses the same segment. Details of shared memory are discussed later in this chapter. Most UNIX-based systems, including AIX 3.2, support explicit file mapping, where the contents of an open file is mapped to data space within the process's image. In AIX 3.2, data blocks from an open file can be mapped into pages within segments 3 through 10. Memory mapped files are explained in Chap. 7.

Segments 11 and 12

These two segments are reserved by the operating system.

> **Author's Note:** Although IBM claims that segments 11 and 12 are reserved, I have found that they will work for shared memory IPC and explicitly memory mapped files. I advise that programmers avoid using these segments for shared memory or mapped files, however, since the future use of these two segments is unknown.

Segment 13—The shared text segment

AIX 3.2, like most other UNIX-based systems, supports the concept of shared libraries or, more specifically, shared text. When many programs use common routines from libraries or other external modules, the text (instructions) from these routines is loaded into a segment that is shared by all processes. Relocation code within each program provides a technique that allows many different processes, executing many different programs, to shared public versions of common routines such as printf(). Since AIX 3.2 uses a segment to store shared text, a limit of 256 megabytes of shared text space is imposed. Infor-

mation on how to create shared text modules and how the kernel implements the sharing is provided later in this chapter.

Segment 14—The kernel data segment

Like a program, the kernel consists of text (system calls and kernel services) and data (variables and tables). Most of the code for the kernel is found in segment 0. Most of the data of the kernel is found in segment 14. For instance, the process table starts at address 0XE0030000. Like segment 0, all processes share segment 14, although its contents can only be accessed by the kernel.

Segment 15—The I/O device address segment

This segment is truly a virtual segment, in that no physical memory is ever assigned to represent it. The virtual memory manager interprets any reference to an address within segment 15 as a reference to an I/O device. Such references are mainly made by device drivers. The virtual-to-real memory translation discussed shortly does not apply to this segment.

Process segments and the system

Figure 4.6 shows three processes and their segments. Two of the processes are executing the vi editor so they share a common text segment. All three processes share the kernel and kernel data segments, as well as the shared text segment. All three processes have their own data segments. None of the processes are implementing shared memory or explicitly memory mapped files.

AIX 3.2 allocates segments for other uses. The VMM has two segments allocated for its own use. They are the VMMDSEG (VMM data segment) and the

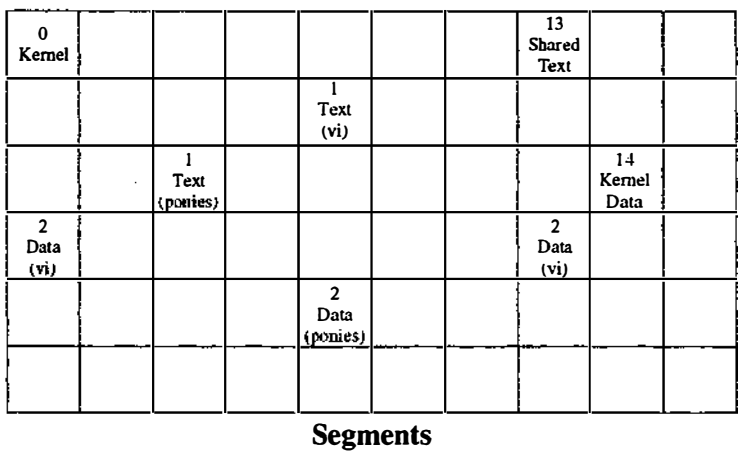

Figure 4.6 An example of process and system segments.

PTA (page table area). The use of these segments is described later in this chapter.

The VMM also allocates segments to opened ordinary files. The files may be local or remote (see Chap. 7). The VMM maps the file's pages into a segment when the file is opened by a process. The file segment is not one of the process's 16 segments. If a file is opened by multiple processes concurrently, the VMM only maps the file once. It is never mapped to multiple segments. The processes that opened the file access it via the same segment ID number. Figure 4.7 shows the two VMM segments as well as a file opened by one or more of the three processes.

The segment information table

The kernel maintains a table to keep track of segments. Each active segment is represented by a segment control block (SCB) structure. Each SCB is an entry in the kernel's segment information table. Since segments are allocated and deallocated as needed, the segment information table's size is dynamic.

> **Author's Note:** As mentioned in Chap. 1, IBM considers the VMM header files and codes as proprietary. The files and structures described throughout this chapter are not shipped with AIX 3.2; therefore, I've avoided mentioning specific field names.

4.3 The Huge Data Model

The private data segment of each process provides it with almost 256 megabytes of virtual memory for user data. Some applications, such as engineering and scientific programs that use large multidimensional arrays,

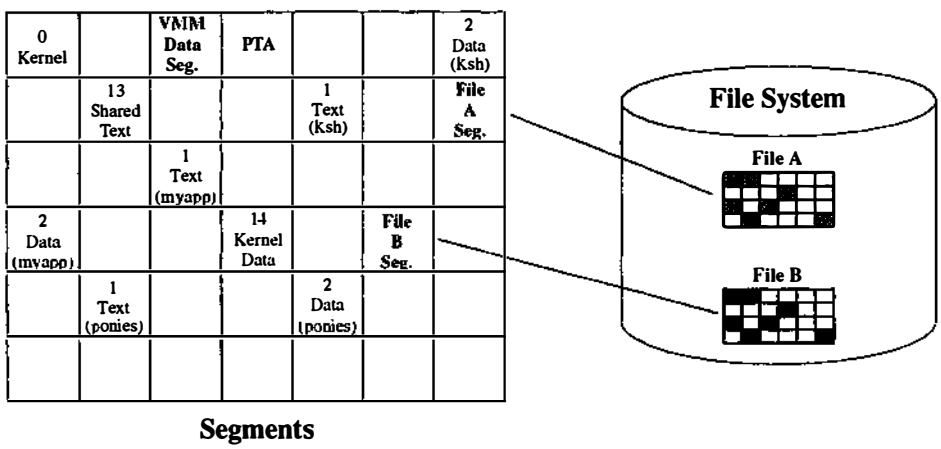

Figure 4.7 VMM and file segments.

require more data space. AIX 3.2 allows a process to claim the virtual memory of up to eight more of its own segments (segments 3 through 10) for additional data space. IBM calls this the huge data model.

The AIX 3.2 linkage editor includes an option, -bmaxdata, that allows a programmer to specify how many additional segments should be available to the process for data space. Figure 4.8 shows an example of creating an application that requests five additional segments for data space. Segments 3 through 7 are allocated. The additional data space cannot be used until it is allocated via the malloc() subroutine (see Chap. 3). Keep in mind that if the application intends to use all the additional data space it acquires with the -bmaxdata option, adequate paging space must exist to accommodate the virtual memory. Another point to remember is that the process in Fig. 4.8 can no longer use segments 3 through 7 for shared memory IPC or explicitly memory mapped files.

4.4 Virtual Memory Pages

Figure 4.9 illustrates how AIX 3.2 classifies page and storage types. Physical memory is carved into 4-kilobyte frames. These frames hold the 4-kilobyte pages of virtual memory. The contents of disk files, which can be programs or data files, are stored in 4-kilobyte disk data blocks (see Chap. 6). AIX 3.2 carves paging space into 4-kilobyte slots.

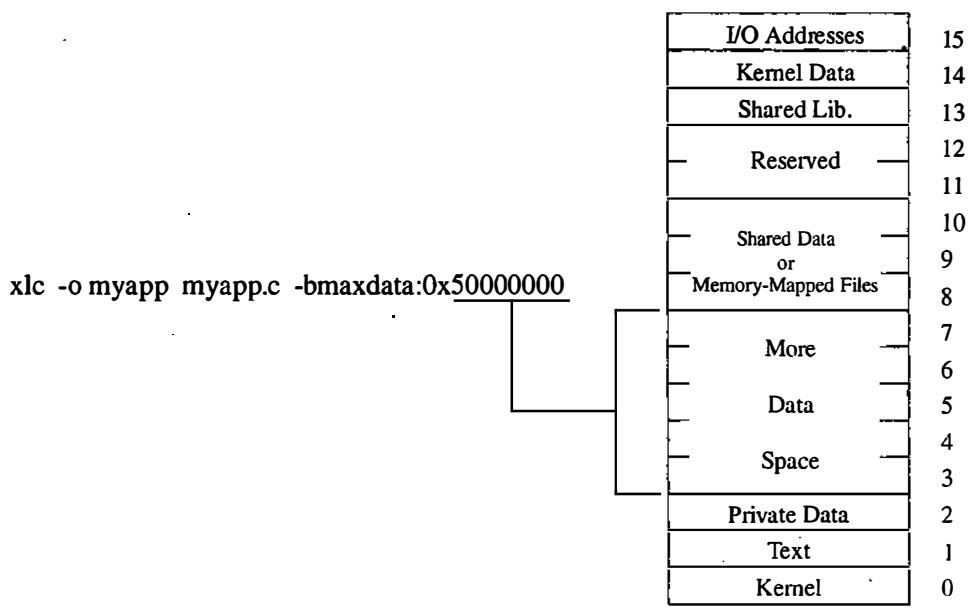

Figure 4.8 The huge data model.

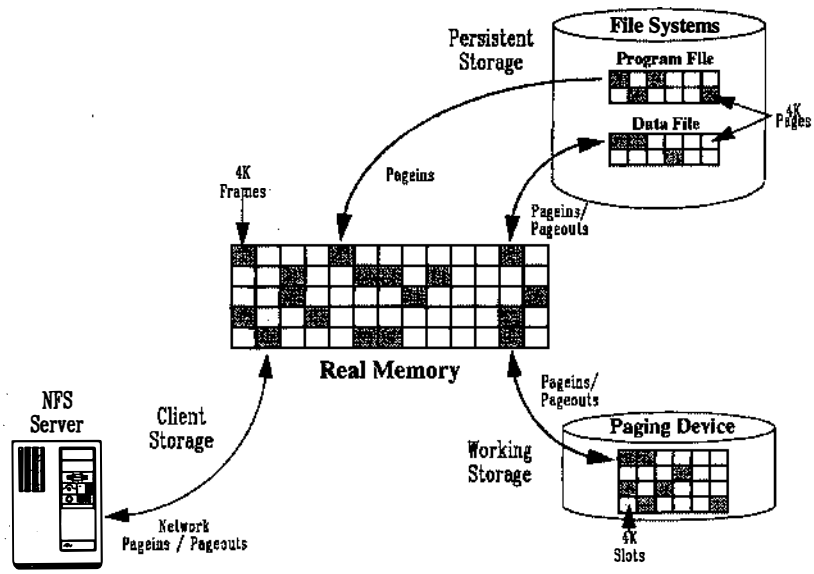

Figure 4.9 Page storage types.

Persistent storage pages

When a program is executed, required pages of the program's code (called text pages) are brought from disk into memory. The action of loading a page from disk to memory is called a pagein. When a process opens a data file for I/O, the requested pages of the data file are paged in. If a process performs a write operation on the opened file, the contents of the data file's page are modified. AIX 3.2 refers to a page of virtual memory whose contents have been changed as a "dirty" page. Eventually, the VMM will write the dirty page back out to the disk file, replacing the disk file copy of the page with the modified page. This operation is called a pageout. The text pages of a program are not allowed to be modified (recall that a process's text segment is read-only); therefore, text pages are never paged out. If the physical memory frame holding a process's text page is reassigned to another page, then the same text page is required again by the executing program, and the VMM performs another pagein operation from the same disk data block of the program. This is called a "repage," since the page had to be fetched more than once.

Pages that come from disk files, such as program pages and data file pages, are always paged back out or repaged back in from their original disk location. IBM calls this technique "persistent storage." Persistent storage pages never use paging space.

> **Author's Note:** The concept of persistent storage confused one of my students. He questioned the fact that the VMM would automatically page out a modified page of

a data file without the consent of the user. "What about when I edit a file with vi?" he asked. "Does the VMM page out pages of the file I'm editing before I've issued a save command? I thought vi gave me the option of quitting without saving, thus aborting the changes made to the pages of the file." What he did not understand about vi is that it uses a temporary file to store the file that is being edited. When the user issues the save command (:w), vi writes the temporary copy of the file over the original file pages in memory. The VMM pages out the modified pages of the original file at the appropriate time.

The data file illustrated in Fig. 4.9 represents a data file opened directly by a process. Pages of the data file are paged in when the process calls read(). The data file pages are modified when the process calls write(). The VMM pages the data file pages back out to disk when the file is closed or when the syncd daemon flushes all dirty pages to disk.

Writing persistent storage pages to disk files

The VMM writes dirty persistent storage pages to their respective disk files at various times. All dirty pages of a data file are paged out to disk when the data file is closed. If a process writes to a data file sequentially, the VMM pages out 16 kilobytes (four pages) every time it hits a 16-kilobyte boundary, as illustrated in Fig. 4.10. A dirty persistent storage page is paged out when the VMM requires the frame holding the dirty page for another page. All dirty persistent storage pages are paged out when a sync() operation occurs. AIX 3.2 has a syncd daemon which is started during system initialization by the /sbin/rc.boot shell script. It runs with a parameter of 60, which indicates that it calls the sync() subroutine once every 60 seconds. In addition, any user can call the sync command or the sync() subroutine at any time.

Working storage pages

In addition to text pages, processes have data pages. These are the pages that make up the process's data segment (see Chap. 3). Recall that this segment con-

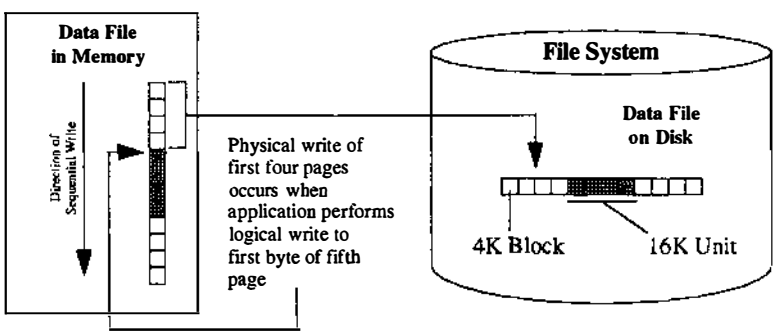

Figure 4.10 Writing sequential pages to disk files.

tains the process's initialized global data, BSS, heap, user stack, user area, kernel stack, and kernel heap. Process data are transient. They have no home on disk and are likely to change during the execution of the process. Since process data pages do not come from a disk file originally (they are constructed by the kernel's loader at run-time), they cannot be treated as persistent storage pages. They are called "working storage pages" and are paged in and out of paging space.

The pageable portion of the AIX 3.2 kernel is working storage. It pages in and out of the system's paging space. Figure 4.11 illustrates the 16 process segments and indicates which are persistent storage and which are working storage.

Client storage pages

The network file system (NFS) provides processes the ability to access files stored on remote systems. It does so by implementing a virtual file system (see Chap. 8) that makes a remote file look as though it is stored locally. Processes are not aware that the NFS files they open are stored remotely. When a process reads or writes to a remote file, NFS pages the file across the network. The VMM assigns a segment to a remote file in the same fashion as it assigns segments to local files, except the segment is called a "client segment."

Remote file pages can be text or data file pages. If a process opens a remote data file for I/O, pages of the data file are paged in and out of the original disk file across the network. NFS implements a file locking scheme to control concurrent access to the file by multiple NFS clients. If a process executes a program that is stored on an NFS server, NFS pages the program's text pages to

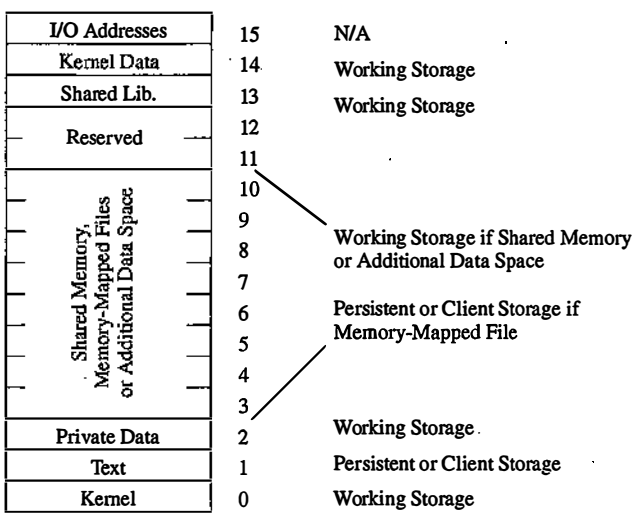

Figure 4.11 Process segment types.

the client. Since the text pages are read only, they need not be paged back to the server. If the VMM requires a frame that is holding a client text page that has not been referenced in a while, instead of discarding it, the VMM pages the client text page to local paging space where it can be retrieved if needed later. This reduces the possibility of a large number of network repages. Figure 4.12 illustrates client storage.

The page frame tables

The AIX 3.2 kernel maintains two tables to track physical memory frames. They are the page frame table hardware (PFTHW) and the page frame table software (PFTSW). Despite the name, the page frame table hardware is part of the kernel software. The PFTHW and PFTSW are twin tables with one entry for each frame of physical memory. The PFTHW contains fields that describe the hardware state of each physical memory frame. It is this table that is used by the VMM to determine how memory is currently allocated. Many of the PFTHW fields are updated directly by the system hardware. The PFTSW contains fields used by the VMM to perform memory aliasing, which allows a physical memory frame to contain a page that is mapped to different virtual addresses from multiple processes. Memory aliasing supports memory mapped files and the mmap() subroutine. The PFTSW also keeps statistics on each physical memory frame.

Author's Note: To simplify discussions in this chapter, the PFTHW and PFTSW tables will be referred to as the page frame table (PFT).

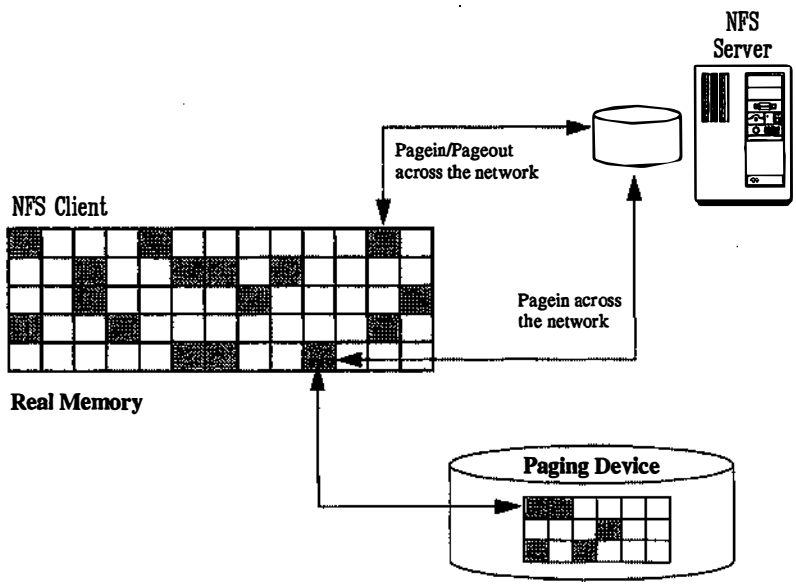

Figure 4.12 Client storage pages.

The size of the PFT is determined at system initialization and is static, since the amount of physical memory cannot change without a system restart. The PFT can be very large, up to 1 megabyte. Each entry in the PFT describes a frame of physical memory. Fields include the segment ID and page number for the page currently occupying the frame, a flag to indicate whether the page in the frame has been referenced recently, and a flag to indicate whether the page in the frame is dirty (has been modified since it was last written to disk). Figure 4.13 illustrates an excerpt from the PFT.

Page replacement

The key role of the VMM is to make sure that pages required by processes or the operating system are in physical memory. If they are not, the VMM must fetch them from secondary storage (paging space, in the case of working storage, or the file system, in the case of persistent storage). The access pattern of virtual memory pages by processes shows that pages are either accessed repeatedly or accessed infrequently. The goal of the VMM is to keep frequently accessed (popular) pages in physical memory. The VMM must also maintain an adequate number of available physical memory frames on a free list to accommodate activities such as the prefetch (read ahead) algorithm used during the sequential reading of a file. In order to accomplish these tasks, the VMM must occasionally select pages found in physical memory frames for replacement. It attempts to replace pages that have not been referenced in a while. The actual page replacement algorithms are discussed below.

The VMM maintains a threshold parameter called minfree that defines the minimum number of physical memory frames on the free list. When the number of free frames drops below this point, the VMM runs a page stealer routine, which uses the page replacement algorithm to free memory frames. It uses another parameter, maxfree, to indicate how many frames must be freed by the

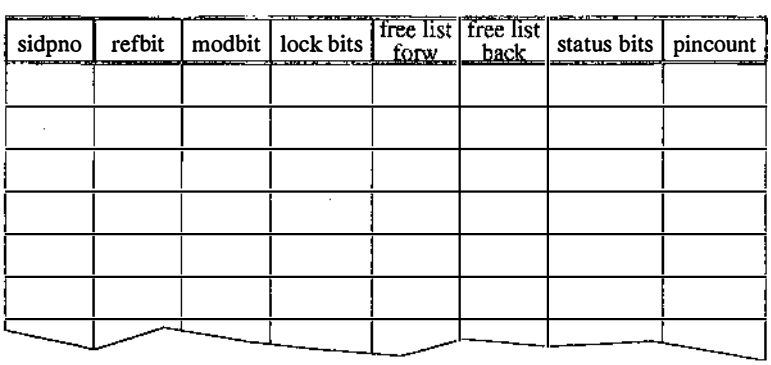

Figure 4.13 The page frame table.

page stealer. The default value for minfree is 120 frames. The default value for maxfree is 128 frames. Both minfree and maxfree can be tuned, as described later in this chapter.

Figure 4.14 uses an excerpt from the PFT to illustrate how the page stealer works. When the number of free frames of physical memory drops below the value of the minfree parameter, the page stealer begins scanning entries in the PFT looking for pages to steal. It determines whether a page should be stolen by examining the reference flag. If the reference flag is turned on, it indicates that the page has been recently referenced. In that case, the page stealer turns off the reference flag and moves on to the next frame. If the reference flag is still turned off the next time the page stealer examines that frame, the page is stolen.

> **Author's Note:** One of my students dubbed this the "Arnold Schwarzenegger algorithm." If the reference flag is turned on, the page stealer turns it off and says, "I'll be back." If the reference flag is still turned off when the page stealer comes back around to the frame, it's "Hasta La Vista, Baby!"

When the page stealer finds a PFT entry with the reference flag turned off, it selects the page in that frame for stealing. If the dirty flag is not turned on, the frame is placed on the free list. If the dirty flag is turned on, the page is scheduled for pageout and the frame is placed on the free list. The page is not actually paged out until another page needs the frame.

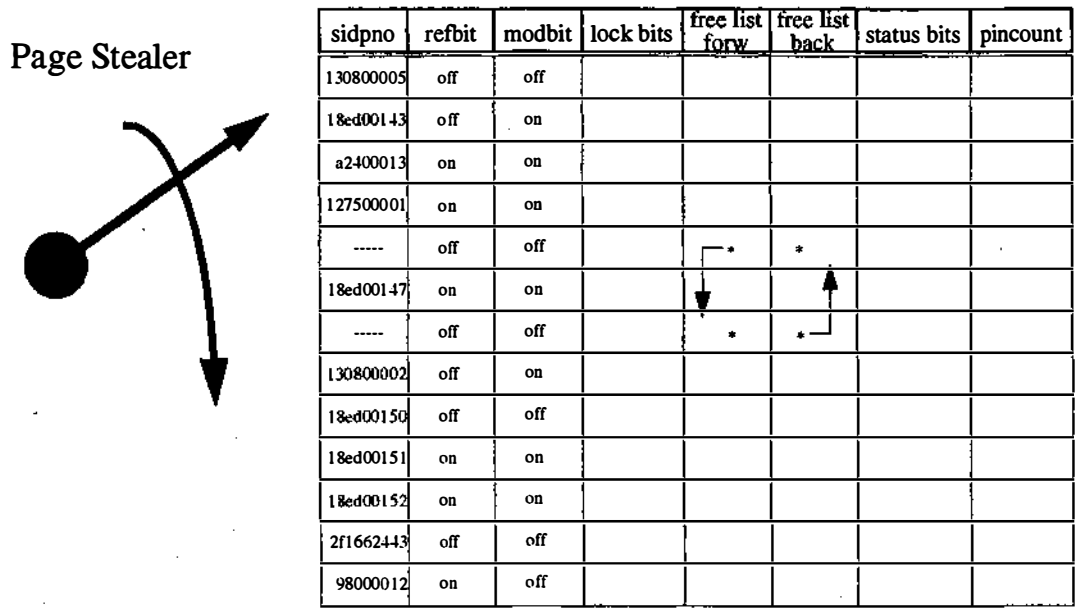

Figure 4.14 The page stealer and the page frame table.

The pages of frames that are placed on the free list are not lost until another page uses the frame. This allows a process to access a page that is on the free list. When this occurs, the VMM turns the reference flag back on and the page is saved. AIX 3.2 calls this a page reclaim.

The VMM includes a table called the page device table (PDT). Pages scheduled for pageout are logged in this table until they are actually paged out. The size of the PDT is 256 entries.

When the page stealer steals enough pages to bring the number of free frames to the maxfree parameter, it stops. The next time it runs it starts where it left off. When the page stealer makes a complete pass through the PFT it is called a cycle. Many performance measurement tools report statistics on pageins, pageouts, page steals, reclaims, and cycles.

Computational pages and data file pages

For performance reasons, AIX 3.2 differentiates between computational pages and data file pages. Computational pages are made up of processes' text and data. The VMM tracks and limits the total number of file pages that are in physical memory. Two parameters, minperm and maxperm, are used to set thresholds for data file pages. They are expressed as percentages of physical memory. When the percentage of data file pages in physical memory drops below the minperm value, the page stealer steals equally from data file pages and computational pages. When the percentage of data file pages in physical memory climbs above the maxperm value, the page stealer steals only data file pages. When the percentage of data file pages in physical memory is between the values of minperm and maxperm, the page stealer steals only data file pages unless the repage rate for file pages is higher than the repage rate for computational pages, in which case computational pages are also stolen. Figure 4.15 illustrates this relationship. The default value for minperm (in pages) is (number of memory frames - 1024) * .2. The default value for maxperm (in pages) is (number of memory frames - 1024) * .8.

The prefetch algorithm

The AIX 3.2 VMM performs read ahead when it detects that a file is being accessed sequentially. It saves time by prefetching consecutive pages from disk in anticipation of their need. When a process accesses two consecutive pages of a data file, the VMM schedules additional reads. The number of pages read by the VMM is determined by the threshold parameters minpgahead and maxpgahead. The minpgahead parameter specifies the number of pages to prefetch when the VMM first detects that a file is being accessed sequentially. The default value is two. The size of each subsequent prefetch is double the previous prefetch size; therefore, if minpgahead is two, the first prefetch reads two pages. The second prefetch reads four pages. The third prefetch reads eight pages, and so forth. The doubling of the prefetch size ends when the number

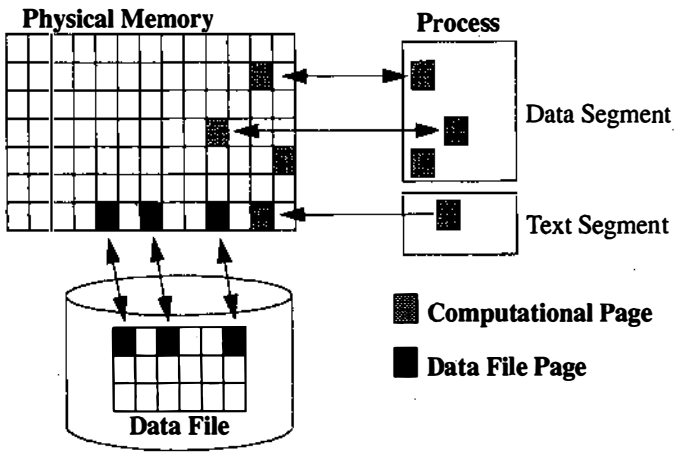

Figure 4.15 Computational and data file pages.

reaches the value of the maxpgahead parameter, which has a default value of eight. Each prefetch overlaps the process's access requests. Figure 4.16 illustrates the prefetch algorithm using the default values for minpgahead and maxpgahead.

In the example above, the first access by the process causes no assumptions to be made by the VMM. When the process accesses the first byte of the next page of the file, the VMM responds by fetching minpgahead number of pages.

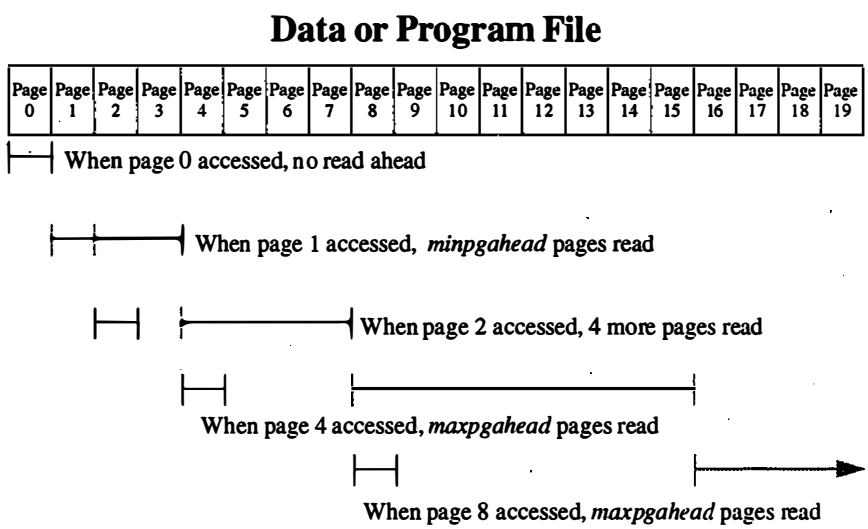

Figure 4.16 The prefetch (read ahead) algorithm.

```
# /usr/lpp/bos/samples/vmtune -P 50

minperm     maxperm     minpgahead     maxpgahead     minfree     maxfree     numperm
  1433        6554           2              8            56          64         1532

number of memory frames=8192      number of bad memory pages=0
maxperm=80% of real memory
minperm=17.5% of real memory

minperm     maxperm     minpgahead     maxpgahead     minfree     maxfree     numperm
  1433        4096           2              8            56          64         1532

number of memory frames=8192      number of bad memory pages=0
maxperm=50% of real memory
minperm=17.5% of real memory
```

Figure 4.17 The vmtune command.

When the process accesses the first byte of the first prefetched page, the VMM prefetches an additional four pages. When the process accesses the first byte of the second set of prefetched pages, the VMM prefetches an additional eight pages. Once the VMM has reached the value of maxpgahead it continues to prefetch that many pages. This entire algorithm stops if the process accesses any nonconsecutive page from the file.

VMM tunable parameters

The VMM parameters listed in this section are tunable. AIX 3.2 includes a command called /usr/lpp/bos/samples/vmtune that is used to examine or change the values of minfree, maxfree, minperm, maxperm, minpgahead, and maxpgahead. It is stored in the samples directory because it is release-specific and tightly coded to the VMM of each version of AIX. The samples directory also includes the source file vmtune.c and a README file for specific information.

When vmtune is executed with no arguments it displays the current parameter values. When vmtune is executed with arguments it displays two sets of values. The first set indicates what the parameter values were prior to making any changes. The second set indicates the parameter values after applying the desired changes. Figure 4.17 illustrates examples of the vmtune command.

The vmtune command uses the following options:

-f minfree (number of frames)

-F maxfree (number of frames)

-p minperm (percentage of physical memory)

-P maxperm (percentage of physical memory)

-r minpgahead (number of pages)

-R maxpgahead (number of pages)

Author's Note: The values for minperm and maxperm are expressed as a percentage of physical memory even though the formula described earlier in this chapter for determining the default values was expressed in pages.

Consult the AIX 3.2 Performance Tuning Guide (SC23-2365-03) for details on tuning the VMM parameters. This guide is also found within InfoExplorer.

VMM and system administration

The two main concerns of a system administrator when it comes to the VMM are making sure the system has enough physical memory to support the workload and making sure the system does not run too low on available paging space.

Without enough physical memory the system thrashes. That's when the system is spending more time shuffling pages from primary to secondary storage than performing useful work. There are two solutions to thrashing. One is to reduce the workload of the system by eliminating unnecessary processes or rescheduling jobs for nonpeak hours. The second solution is to add more physical memory.

AIX documentation states that one can identify a thrashing system by running the vmstat command and comparing the number of pageouts (po) to the number of pages freed by the page stealer (fr). If the number of pageouts is consistently greater than 17 percent (one-sixth) of the number of pages freed by the page stealer, the system is thrashing. AIX 3.2 has an algorithm to reduce the effects of thrashing. It is described in Chap. 5.

Author's Note: You usually don't need to run vmstat to figure out that the system is thrashing. The disk drives will begin to chatter and response time will slow dramatically!

Running low on available paging space slots can be an even bigger problem than thrashing. The system handles the situation in stages. When available paging space slots reaches the first low water mark the system prohibits the creation of new processes via the fork() system call. Processes attempting to fork() will retry five times at a specified frequency before returning a failure condition. (See Chap. 5 for a description of the fork() system call.) This way the system is able to stop new processes from being created since there is not enough paging space available to support their working storage requirements. Unfortunately, this action may not be enough to alleviate the problem. Existing processes may use malloc() to grab more virtual memory, thus more paging space.

When the system reaches another low water mark of available paging space slots, it sends a SIGDANGER signal to all processes. (See Chap. 10 for an explanation of signals.) The SIGDANGER signal is unique to AIX 3.2. Unfortunately, the default action for a process receiving a SIGDANGER signal is to ignore it. A process must register a signal handler to deal with the SIGDANGER signal, or the process can call psdanger() to query the status of available paging space. See the manual page for the psdanger() subroutine or the example program in the Performance Tuning Guide found in InfoExplorer. An appropriate action upon receipt of a SIGDANGER signal is to gracefully terminate the process, lest the operating system does it nongracefully, as described in the next paragraph.

When a final low water mark of available paging space slots is reached, the system has no choice but to terminate processes. It sends SIGKILL signals, which cannot be trapped or ignored, to those processes that are consuming the most virtual memory. These processes are immediately terminated.

> **Author's Note:** The system often terminates the X display server, which causes the termination of all window clients and their children. You've lost just about everything but at least you don't have a paging space shortage anymore! In traditional UNIX fashion, the only message the user receives when this happens is "Process killed." At least the cause is easy to identify. The lsps command displays the paging space devices and their utilization percentages. If the utilization is greater than 85 percent, you should consider adding more page space.

AIX 3.2 offers two options for increasing the paging space. Existing paging space logical volumes can be extended dynamically or new paging space logical volumes can be created and activated dynamically. If the system has multiple physical volumes (fixed disk drives) it is usually better to create a new paging space logical volume, as long as it is placed on a physical volume that does not already have a paging space logical volume or any other busy logical volume. The VMM uses a round-robin technique for allocating slots of paging space to virtual memory. If it is later determined that an additional paging space logical volume is not required, one can remove it. The process of removing a paging space logical volume is not difficult but it does require a system restart. The system management interface tool (SMIT) has an option for deactivating a paging space device upon the next system restart.

If the system has only one physical volume or the other physical volumes are already heavily utilized, it is best to extend the size of the current paging space. The drawback to this is that it is very difficult to reduce the size of the paging space logical volume if it is later determined that it need not be so large.

> **Author's Note:** IBM recommends that the total paging space size be 1.5 to 2 times the size of physical memory at smaller memory sizes. I usually allocate 2.5 times the size of physical memory for systems with 16, 32, or 64 megabytes of random access memory (RAM). In theory, as a system's physical memory increases the ratio of paging space should decrease since the system should do less paging. This is not always true, however. The most important factor in determining a system's paging

space requirements is the applications running on the system. Since AIX 3.2 uses persistent storage for data files, commercial applications that perform a lot of file I/O, such as data bases, do not require as much paging space as applications that process a lot of data, such as graphical engineering and scientific applications. The bottom line is that your system should always have at least as much paging space as it has physical memory.

The AIX 3.2 malloc() subroutine and paging space

AIX 3.2 uses a late allocation technique for paging space when processes call malloc(). This means that paging space slots are not allocated to the process's working storage until the process references pages within the memory region returned from malloc(). As each page of memory is accessed by the process, a paging space slot is allocated. This can lead to a potential problem. If many processes use malloc() to allocate large chunks of virtual memory, the system does not verify that there is enough paging space to accommodate the requests, since the allocation of paging space may never actually occur. If all of these processes then begin to reference large amounts of the space they acquired with malloc() the system might run out of paging space. This is where it is important to monitor paging space utilization with the psdanger() subroutine or by trapping the SIGDANGER signal.

Author's Note: This reminds me of the way the airlines overbook flights. They know, from historical data, that they can count on a certain percentage of no-shows. Therefore, they reserve more seats than actually exist. AIX 3.2 might promise more paging space than actually exists with the idea that processes don't always use everything they have asked for with the malloc() subroutine. This has sparked the debate regarding standards compliance described in Chap. 2. On most systems, malloc() fails if there is not enough paging space to honor the request. This may cause some programs that run correctly on other UNIX-based systems to fail when ported to AIX 3.2.

4.5 Address Translation

This section describes how the VMM translates addresses that find data and instructions in physical memory.

In order for a process to run, its instructions must be loaded into the RISC System/6000 or PowerPC processor pipeline. In order for a process to manipulate its data, the data must be loaded into CPU registers. A hierarchy of system resources is involved in the execution of a process, as shown in Fig. 4.18. CPU resources, the instruction pipeline, general-purpose registers, and floating-point registers are the scarcest resources. If a desired piece of data is located in a CPU register, access time is very fast; however, only a small amount of data can occupy the CPU registers at a time.

Traditionally, data and instructions are kept in memory. Load and store operations move the data and instructions to and from the memory and CPU. While

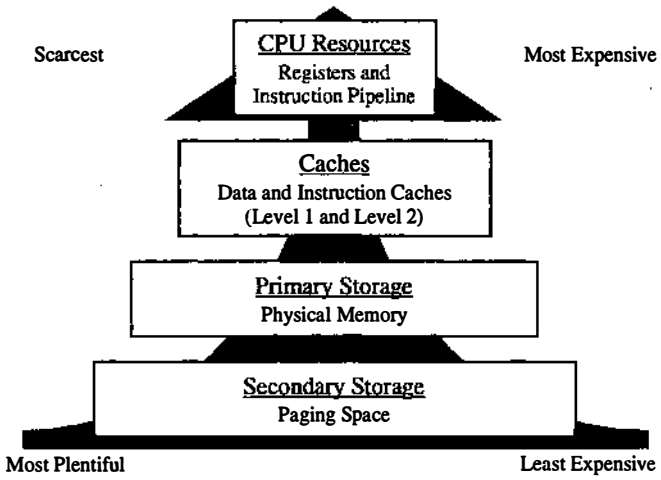

Figure 4.18 System resources.

there can be an abundance of memory (compared to CPU registers), locating a line of code or piece of data in memory is time-consuming. To help speed up the search, many computer systems, including the RISC System/6000 and PowerPC, implement multiple levels of memory caching. The caches, while more abundant than CPU registers but scarcer than memory, hold the most recently referenced data and code. If a desired piece of data or instruction is found in one of the caches, the access time is still very fast. If the desired data or instruction is not found in one of the caches, a cache miss occurs and the search continues throughout the entire physical memory.

If the desired piece of data or instruction is not found in physical memory, a page fault occurs. This means that the virtual memory page that contains the desired data or instruction has been paged out or was never paged in. The VMM must locate the missing page and fetch it into memory. This takes the greatest amount of time, but secondary storage via disk-based paging space is the most abundant and least expensive resource.

There are three different types of addresses associated with every program instruction or piece of data. When a program is compiled, the compiler assigns an effective address to every line of code and every piece of data in the program. An effective address in AIX 3.2 is 32 bits, which represents 4 gigabytes, the total size of a process's image. The VMM converts the 32-bit effective address into a 52-bit virtual address; 52 bits represent the 4-petabyte virtual memory size of AIX 3.2. The 52-bit address is then translated by the VMM into a 32-bit real address, which is used to locate the desired instruction or data in physical memory or secondary storage. Figure 4.19 illustrates the translation process.

In order to find a piece of data or a machine instruction, the VMM needs to know three things. First, the VMM must determine which segment holds the

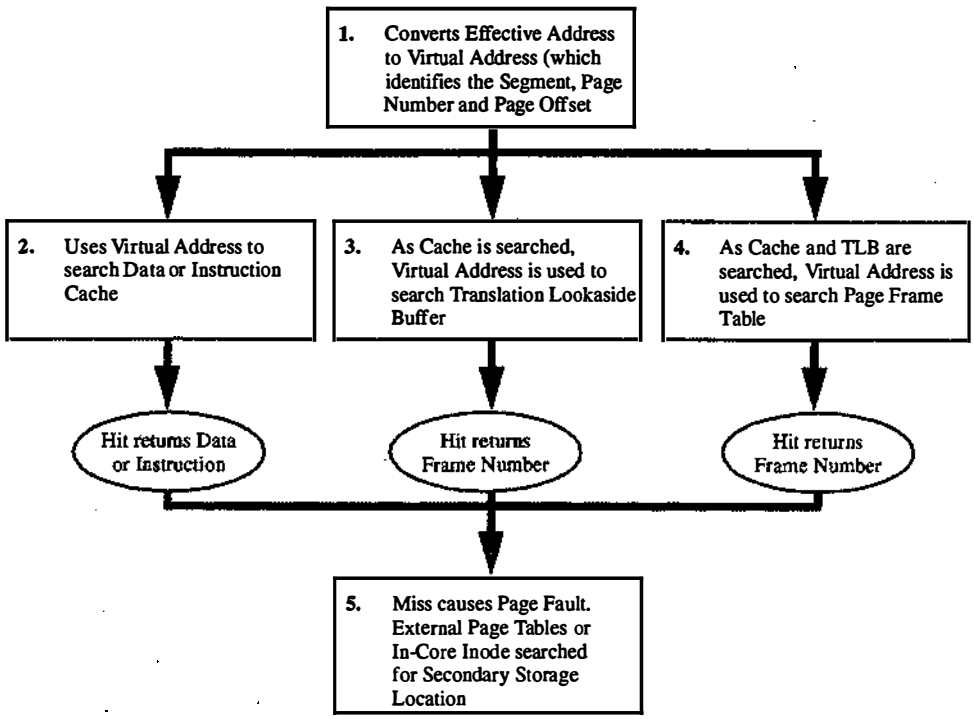

Figure 4.19 The address translation process.

desired object. Second, it must determine which page within the segment holds the desired object. Finally, the VMM must determine at which byte within the page the desired object begins. The last component is known as the page offset. Recall that there are 16,777,216 possible 256-megabyte segments in the 4 petabytes of virtual memory. There are 65,536 possible 4-kilobyte pages in a 256-megabyte segment. There are 4096 bytes in each page. The next section explains how the VMM locates the segment, page, and byte offset for a piece of data or instruction.

The effective address

As mentioned earlier, the compiler assigns a 32-bit effective address to every machine instruction (line of code) and every piece of data in the program. To illustrate how the VMM performs address translation, assume the following line of code from a program:

```
printf("The answer is %d.\n", a);
```

The program must pass the value of the variable "a" to the printf() subroutine. The system must locate the variable "a" so that its contents can be loaded into

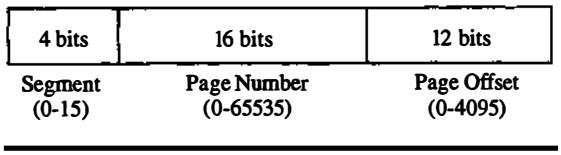

Figure 4.20 The 32-bit effective address.

a CPU register. The compiler generated a 32-bit effective address for "a" when the program was compiled. The VMM starts with that address, which is carved up as shown in Fig. 4.20.

The first 4 bits of the address are used to identify the segment that holds the variable. Four bits allows up to 16 segments to be referenced. This is a far cry from the possible 16,777,216 segments mentioned above, but recall that a process has 16 segments. All that is needed for now is the segment number within the process. The 16 bits from the middle of the effective address indicate the page number within the segment. The 16 bits can reference up to 65,536 pages, or the number of pages in a segment. The last 12 bits indicate the byte offset of the object (in this case, the storage location of the variable "a") within the page.

The first step involves transforming the 4-bit segment number within the process into a 24-bit segment identifier for all of virtual memory.

Segment registers

The RISC System/6000 and PowerPC include a set of registers called the segment registers. There are 16 segment registers. They hold the segment ID numbers for the 16 segments of the current process. During address translation, the VMM uses the first 4 bits of the effective address to index into the segment registers. Each segment register holds a 32-bit value, although only 24 bits are returned for address resolution. The 24 bits represent the unique segment ID number (24 bits can reference up to 16,777,216 segments). Figure 4.21 illustrates how the segment registers are indexed for the variable "a." The first 4 bits of the effective address for "a" would be 0010 since the variable is data and would be found in a process's data segment (segment 2).

The 24-bit segment ID is attached to the remaining 28 bits of the effective address to form the 52-bit virtual address. The 52-bit virtual address is capable of referencing any byte of the 4-petabyte virtual memory range. Now the search for the variable "a" begins. Figure 4.22 shows how the search takes place. It is not a sequential search, but rather a parallel search. The 52-bit virtual address is used many ways by the VMM during the search, as illustrated in the following sections.

Data/instruction caches

As mentioned earlier, many computer systems' CPUs include data and instruction caches that serve as a form of fast memory. They hold copies of the most

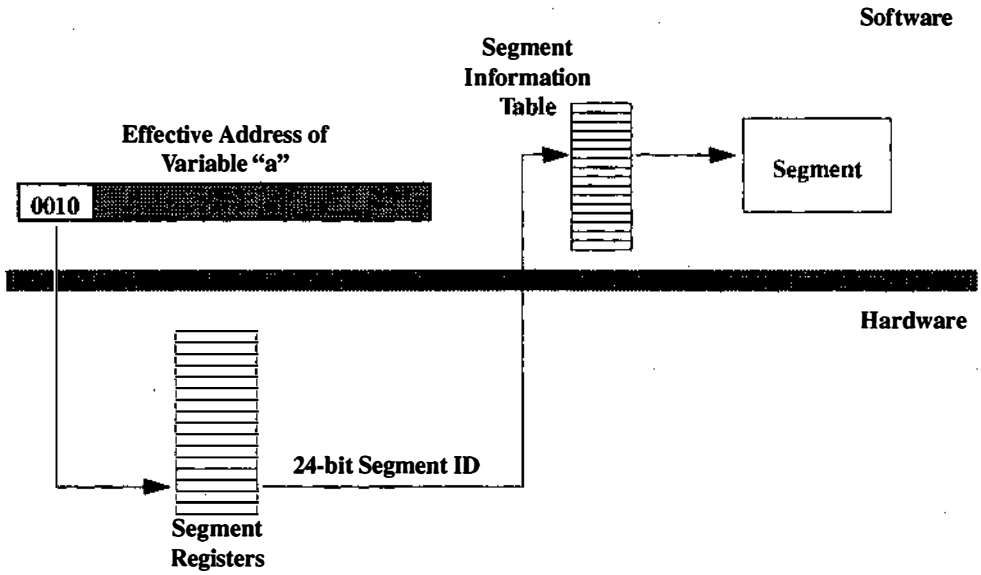

Figure 4.21 The segment registers.

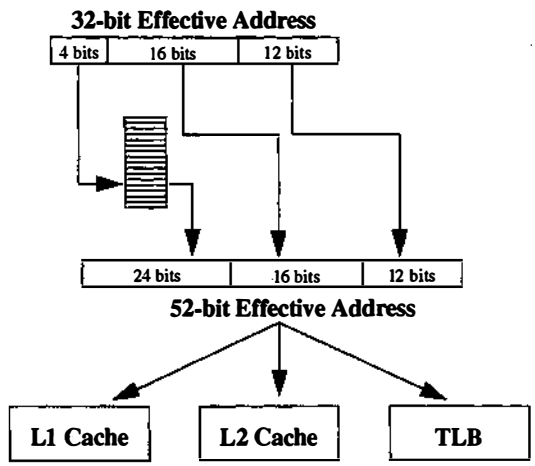

Figure 4.22 Cache and memory searching.

recently accessed data and instructions from memory. The RISC System/6000 and PowerPC implement data and instruction caches for such a purpose. The sizes of these caches are model-specific. Consult the hardware specifications for your particular system for details. This example assumes that the data cache size is 128 kilobytes.

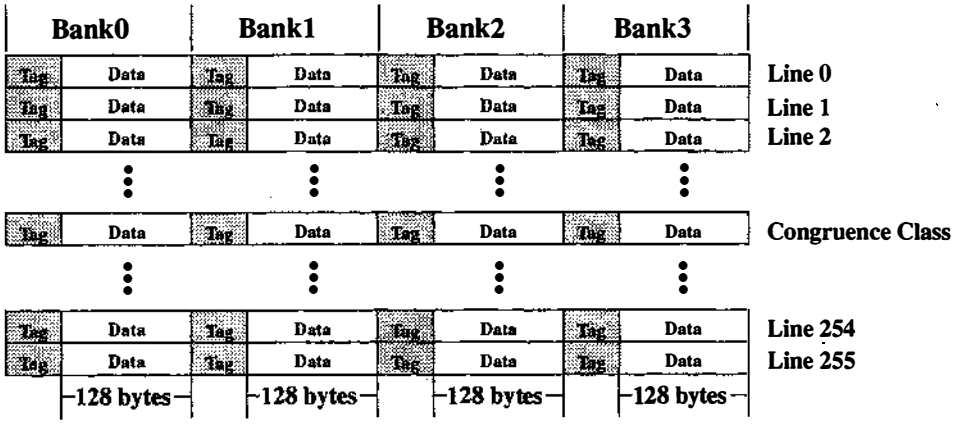

Figure 4.23 The data cache.

Each cache line is 128 bytes long. The data cache is four-way set associative, which means that there are four banks of 128-byte lines. For a system with 128 kilobytes of data cache, such as the example here, there are 256 cache lines in each bank. The combination of the four corresponding cache lines from the four banks is called a congruence class, so the cache in this example would have 256 congruence classes. Figure 4.23 illustrates the data cache configuration for the example.

For cache searching, the 52-bit virtual address derived from the 32-bit effective address is carved up as shown in Fig. 4.24. The rightmost 7 bits represent the cache line offset since 7 bits can represent up to 128 bytes. The next rightmost 8 bits represent the line number in the cache, since 8 bits can represent up to 256 congruence classes. The leftmost 37 bits represent a tag, which is used to uniquely identify the correct cache line when a congruence class is searched. Each 128-byte cache line includes a 37-bit tag prefix. This way, the cache search is done by vectoring to the correct congruence class, then comparing the four tags with the virtual address tag. If a match is found, there is a cache hit and the line offset from the virtual address is used to load the CPU register. If no tag match is found, a cache miss occurs. While this very fast search is taking place, other searches are progressing in parallel, so a cache miss simply ends the cache search.

Level 2 caches

Some RISC System/6000s and PowerPC models include an L2 cache. L2 caches are larger than the primary data cache described above but take longer to search. The L2 caches are direct mapped, instead of four-way set associative. There are 8192 lines of 128 bytes each. For L2 caches, the virtual address is carved up as shown in Fig. 4.25. The rightmost 7 bits still reference the line off-

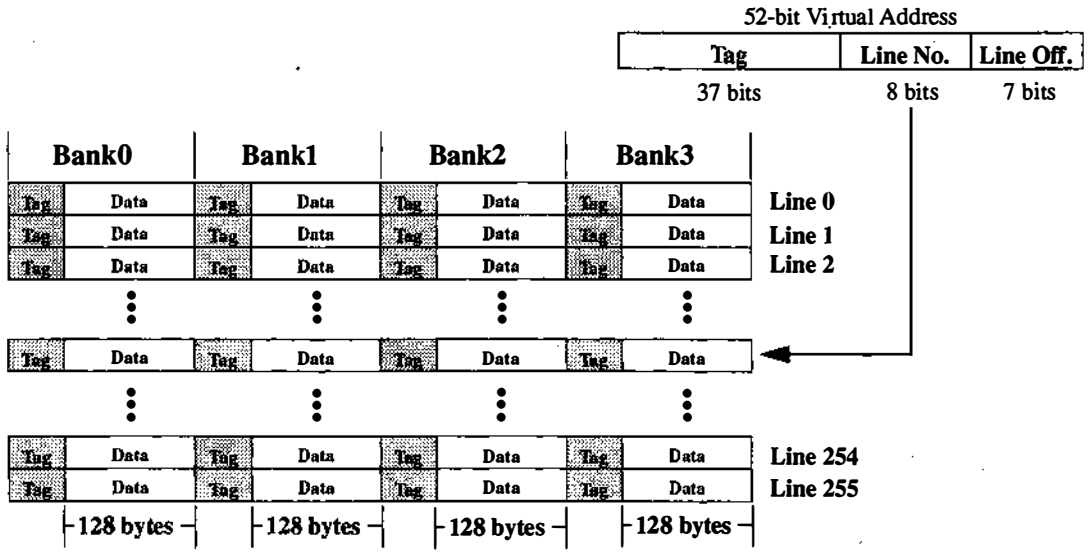

Figure 4.24 Searching the data cache.

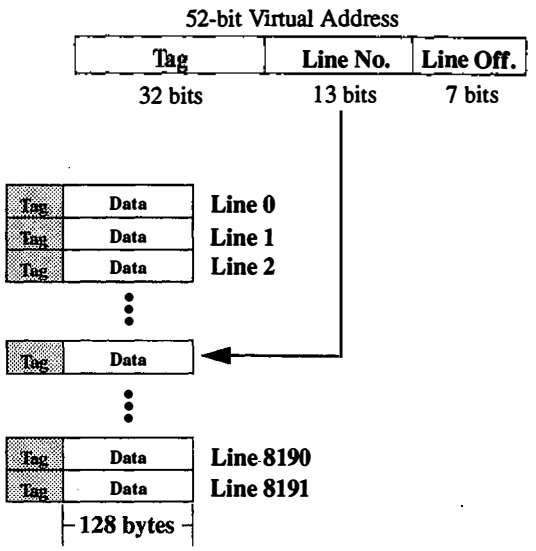

Figure 4.25 Searching the L2 cache.

set, but the next rightmost 13 bits reference the line number, since 13 bits can reference up to 8192 cache lines. The remaining 32 bits serve as the tag.

The translation lookaside buffer

The translation lookaside buffer (TLB) is another hardware cache that keeps track of the most recently referenced pages and their current real memory locations. It can hold up to 512 references. It is two-way set associative and has 256 congruence classes. The 52-bit virtual address is carved up as shown in Fig. 4.26 for searching the TLB. The leftmost 32 bits represent the tag. The next 8 bits indicate the TLB line number. If a tag match is found, the TLB returns the 20-bit physical memory frame. The 20-bit frame number is reattached to the rightmost 12 bits from the effective address (the page offset bits) to form the 32-bit real address. If a tag match is not found, the desired page may still be in physical memory, but now the search takes place in the software.

The hash anchor table and page frame tables

While the system is searching the caches and TLB, the VMM is also searching for the desired page by looking through the page frame tables. Since this is done in software, it is much slower than any of the examples presented so far. As mentioned earlier, the PFT contains one entry for every physical memory frame. Its size, which can be quite large, is configured at system initialization and never changes.

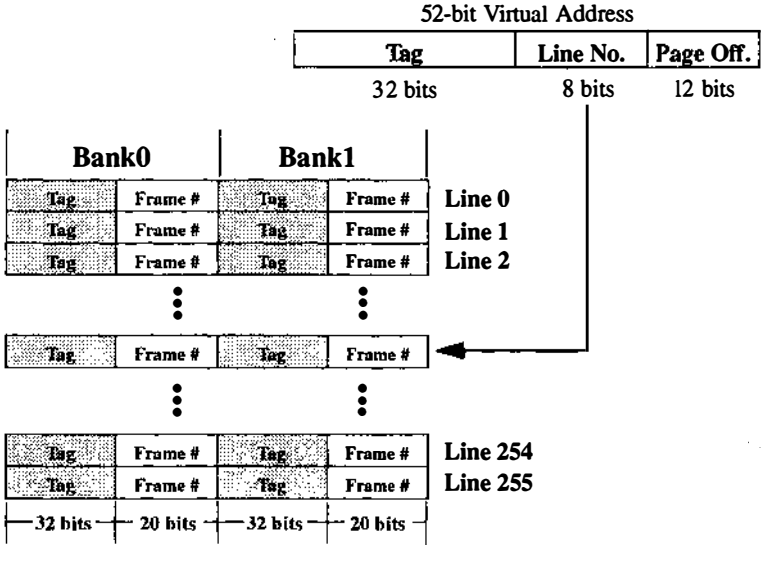

Figure 4.26 Searching the TLB.

To speed the PFT search, the kernel maintains a hash anchor table (HAT). Hashing is performed on the leftmost 40 bits of the 52-bit virtual address (the segment ID and page number). A value from the HAT indexes to the head of a linked list found in the PFT. The PFT uses linked lists to chain all pages that share common segment IDs. Only the linked list must be searched, which reduces the search time. The search is performed using the segment ID number and the first three bits of the page number. If a match is found, the index value of the match in the PFT is used as a 20-bit frame ID. The 20-bit frame ID is attached to the 12-bit page offset from the original effect address to form the real address. The entire 128-byte memory line that corresponds to the real address is loaded into the data cache, possibly replacing another line. The tag in the data cache line is updated to reflect the new contents of the cache line. The value at the desired address is loaded into a CPU register. Figure 4.27 illustrates the PFT search. If the PFT search does not yield a match, a page fault occurs and the VMM must look to paging space if the desired page is working storage (as with the example of variable "a"), or a file system if the desired page is persistent storage.

Page faults

The VMM uses a series of tables called external page tables, or XPTs, to keep track of working storage pages on paging space. AIX 3.2 allocates one XPT for each working storage segment. It is used by the VMM to locate working storage pages in paging space.

An interesting characteristic of the AIX 3.2 XPTs is that they, themselves, are pageable. This is to allow them to grow to a size adequate to handle the

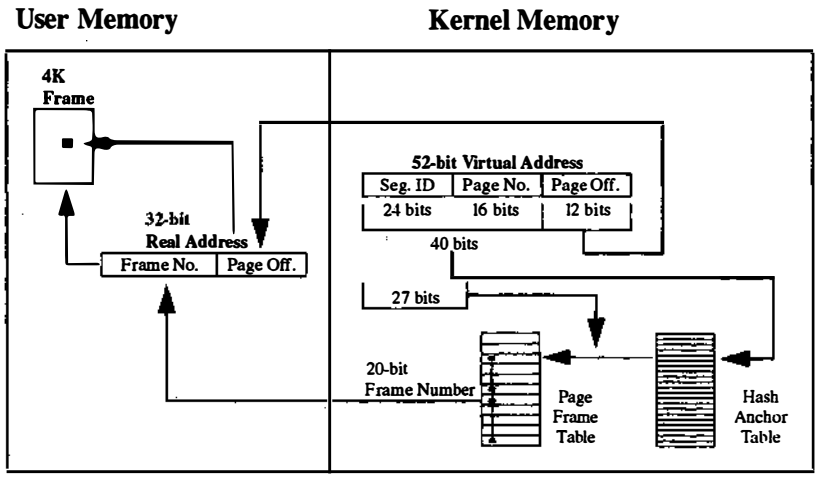

Figure 4.27 Searching the page frame table.

large virtual address space of AIX 3.2. Since the XPTs are pageable it's possible that the VMM can experience a page fault while resolving a process's page fault. When this happens, the VMM must first locate and page in the missing page of the XPT. It can then finish locating the missing page to resolve the original page fault. This double page fault resolution is called "back tracking." The frequency of back tracking is reported by the vmstat command. See the manual page for vmstat for details.

Figure 4.28 illustrates how the XPTs are configured. Each XPT has a pinned anchor structure, called an xptroot, that allows the rest of the XPT to be paged out. The size of the xptroot structure is 1024 bytes. Each one holds up to 256 pointers to XPT data blocks. Each XPT data block, defined as an xptdblk structure, holds up to 256 XPT entries. These entries identify the locations of working storage pages using 4 bits to identify the paging space logical volume (there is a system limit of 16 paging space logical volumes) and 24 bits to identify the block number.

Persistent storage segments do not use XPTs. Since the data block addresses of a local disk file are maintained in a structure called an inode (see Chap. 6) and this structure is cached in the kernel when a file is open, the VMM uses the inode information to locate missing pages of the file. Figure 4.29 illustrates how this works. See the section called "Linking an In-Core Inode to a File Segment" in Chap. 7 for more information.

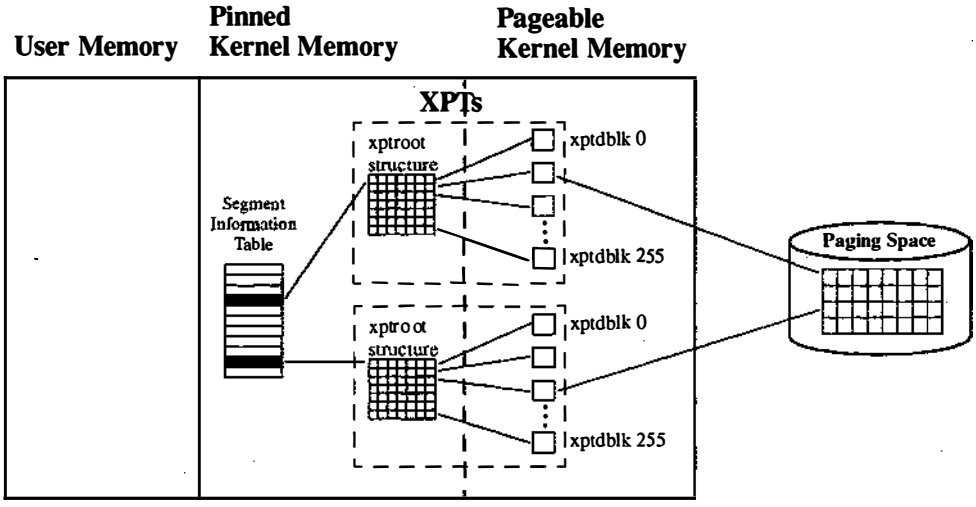

Figure 4.28 External page tables.

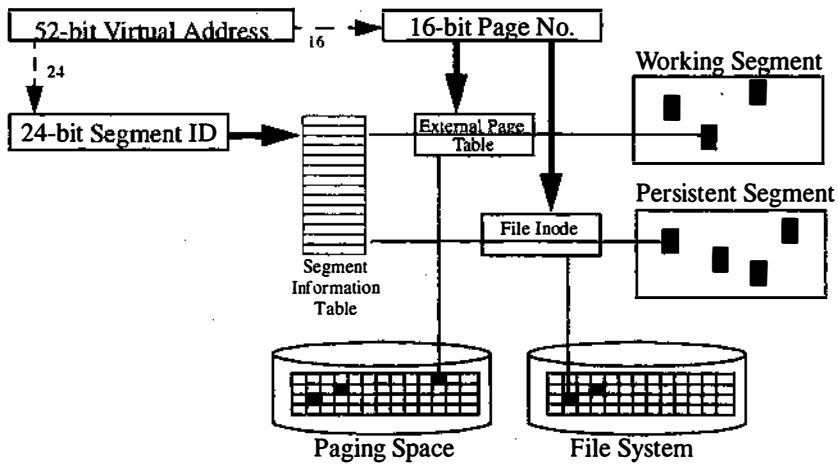

Figure 4.29 Handling a page fault.

4.6 The VMM Segments

As mentioned earlier in this chapter, the VMM has two segments of its own, the VMMDSEG (VMM data segment) and the PTA (page table area). The VMMDSEG contains most of the tables discussed in this chapter, as well as miscellaneous VMM variables. The VMMDSEG includes the page frame tables, the hash anchor table, the page device table, and the segment information table.

The page table area holds the external page tables. It includes a set of area page map (apm) structures at the beginning of the segment. Each apm structure defines the status of a page within the PTA. Figure 4.30 illustrates the layout of the PTA segment.

4.7 The Shared Text Segment and Shared Libraries

Segment 13 of every process is the system's shared text segment. It contains the code of subroutines linked from a shared object file or shared library. This allows many processes to access the same memory location for a commonly used subroutine instead of each process having its own private copy of the subroutine. The section explains why shared libraries are important and describes how to create them.

> **Author's Note:** The term "shared library" is misleading in AIX 3.2 because it's the objects in the libraries that are either sharable or nonsharable. A library might contain a mix of both shared and nonshared objects.

Why shared text?

Before the days of shared text, all code found in UNIX process at program runtime was private. This meant that if a dozen processes, all executing different

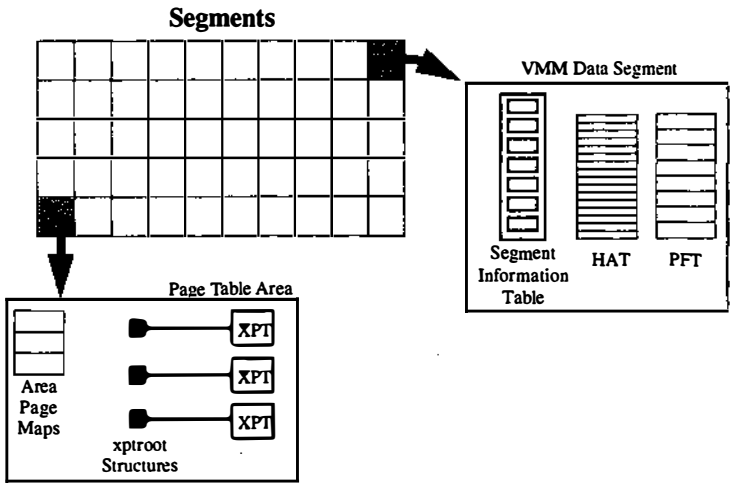

Figure 4.30 The VMM segments.

programs, included references to the printf() subroutine found in the standard C library, each process had its own private copy of the text for the printf(). One can easily see that memory is being wasted, especially considering that the text never changes.

The concept of shared text means that the code for commonly referenced subroutines is placed in a global memory location that can be accessed by any process. That is exactly what segment 13 of every AIX 3.2 process is, a common location for shared text. Figure 4.31 contrasts nonshared text with shared text in AIX 3.2. Since every subroutine includes data, which cannot be shared among different processes, they are loaded into the process–private data segment.

In addition to reducing the memory requirements of processes, shared text can also reduce paging activity because frequently referenced pages of shared subroutines tend to stay in physical memory more often. There is one drawback, however, to shared text. Its location in segment 13 causes an executing process to have a poorer locality of reference, which means that the virtual memory pages touched by the process are further spread out than they would be if the shared text were private. The AIX Performance Tuning Guide suggests trying application with shared and nonshared versions of object files or libraries to determine if the use of shared text degrades process performance.

Creating a shared object

Figure 4.32 provides an example of how to create a shared object. The source code file red.c includes a subroutine called red(). An export file is created for the object so that dynamic binding can be implemented. See Chap. 3 for details on dynamic binding and export files. The source code file is compiled into the

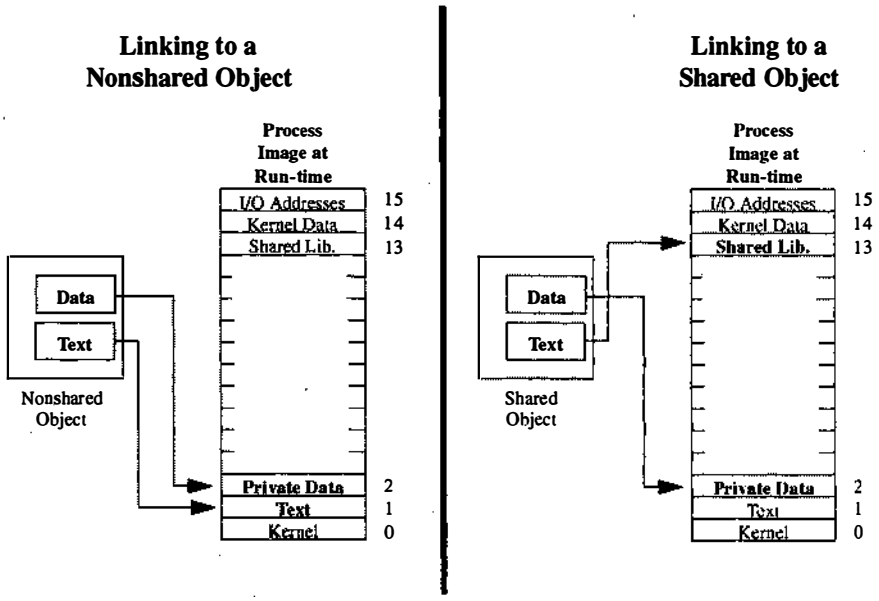

Figure 4.31 Nonshared vs. shared text.

object file red.o. The ld command is then used with the -bE:red.exp option to identify the export file. The most important option is -bM:SRE, which designates the object file as shared. The option means "Mode: Shared Reusable" (or reentrant). This is all that is required to make the object shared. See Chap. 3 for a description of the other options given with the ld command.

An object file that is marked for shared text has the value "RE" stored in the o_modtype field of the auxiliary header portion of its XCOFF file. See Chap. 3 for an explanation of the XCOFF file. A simple C program can be written to test the value of this field to determine if an object file has been marked for shared text.

red.c

```
red()
{
  ...
}
```

red.exp

```
# Export list for red.o
red
```

```
xlc -c red.c
ld -o red.o red.o -bE:red.exp -bM:SRE -e red -lc
```

Figure 4.32 Creating a shared object.

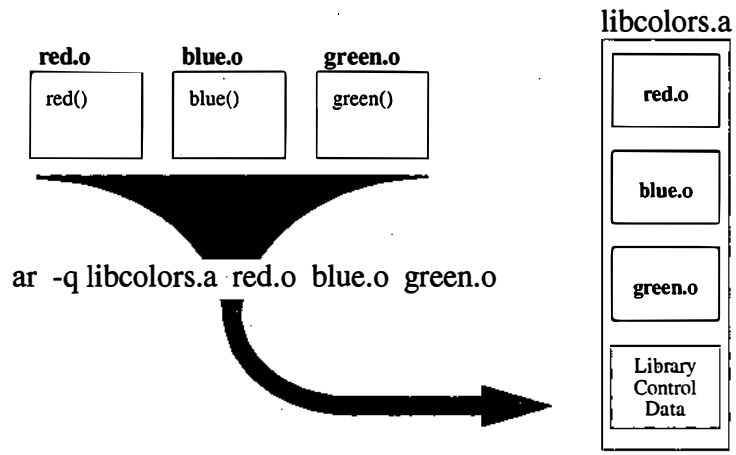

Figure 4.33 Creating a shared library.

Creating a shared library

Figure 4.33 illustrates how a library is created. The ar command is used to archive three object files, all of which have been marked for shared text, into a library called libcolors.a. The ar command includes other options for adding new object files to a library, replacing existing object files in a library, and extracting object files from a library. See the manual page for the ar command for more information.

Linking an application to a shared library object

Figure 4.34 illustrates how an application links to a shared library. The -l option, which is passed by the compiler to the linkage editor, specifies the library name. The -L option indicates that the current directory should be searched for the library. The most important point of the example in Fig. 4.34 is that AIX 3.2 will perform dynamic binding of the red() function to the application when it is executed, even though the application was compiled without an import list file (see Chap. 3). This is because "auto import" is the default when linking to any shared object.

Replacing a shared library object

As mentioned in Chap. 3, one of the advantages of dynamic binding is that if a library object is updated, the applications that are linked to the library need not be relinked. The applications inherit the object revision the next time they are executed. It is important to point out, however, that AIX 3.2 does not allow one to update a shared object within a library once any program that is linked

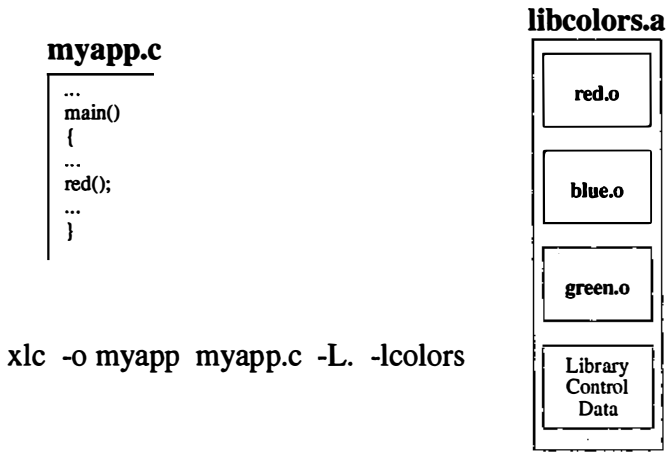

Figure 4.34 Linking to a shared library.

to the shared object has been executed since the previous system initialization, even if the program has terminated. In other words, once a program that uses a shared object from a library has been executed, the system must be rebooted in order to update the shared object in the library, even if the program is no longer running.

> **Author's Note:** Actually, the system need not be rebooted, but a special command must be run before the library object can be replaced.

The reason for this is simple. Once shared text is loaded into segment 13 it stays there, even if all programs that have used it are no longer executing. AIX 3.2 assumes that the shared text will be needed again later, and it is easier to leave it in place. Because of this, the system prevents the library object from being updated.

AIX 3.2 includes a command called "slibclean," which cleans out all shared text subroutines that have a current reference count of zero from segment 13. The command must be run prior to updating any shared library. See the manual page for slibclean for more information.

Chapter

5

The Process Management Subsystem

Process management is one of the most important subsystems of the AIX 3.2 kernel. It is responsible for creating processes and cleaning up after them when they terminate. It also schedules processes by arranging them into run queues based on their priorities and dispatches the most favored processes at regular intervals. Finally, the process management subsystem provides timer mechanisms that are used by processes to trigger timed events. This chapter describes all of these characteristics of the AIX 3.2 process management subsystem.

5.1 Processes

The process is the most important feature of AIX 3.2. It is the scheduled entity of work, providing the environment and resources required to execute a program. Most of the commands issued at the shell's prompt result in the creation of a process to execute the command. In fact, the primary purpose of the shell, as a command interpreter, is to create processes for the execution of commands. When a user invokes a command that is the name of an executable file, the shell, which is itself a process, creates a child process, then loads the program specified as the command into the process image for execution. Figure 5.1 illustrates how the shell executes commands as child processes.

> **Author's Note:** Not all commands issued at the shell prompt result in the creation of a child process. Some commands are recognized by the shell as subroutines within the code of the shell's program. Such commands are called "shell built-ins." They run within the shell's process. Examples of shell built-ins include the cd command and the Korn shell's test command.

The child process created by the shell initially executes another copy of the shell. Soon after its creation, it loads and executes the desired program.

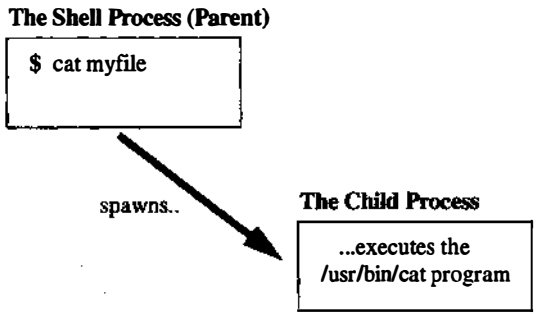

Figure 5.1 The shell and processes.

This is called "spawning a process." The details of spawning are discussed shortly.

With the exception of two special processes, all user processes are created by other user processes. The creating process is known as the parent process and the created process is known as the child process. See Chap. 3 for details on different types of processes.

Each process has its own environment. A child process inherits much of its environment and many of its attributes from its parent. The child can decide whether to keep the environment or modify it. Modifications made to the child's environment are not passed back to the parent process but would be inherited by any processes created by the child. Figure 5.2 illustrates how processes

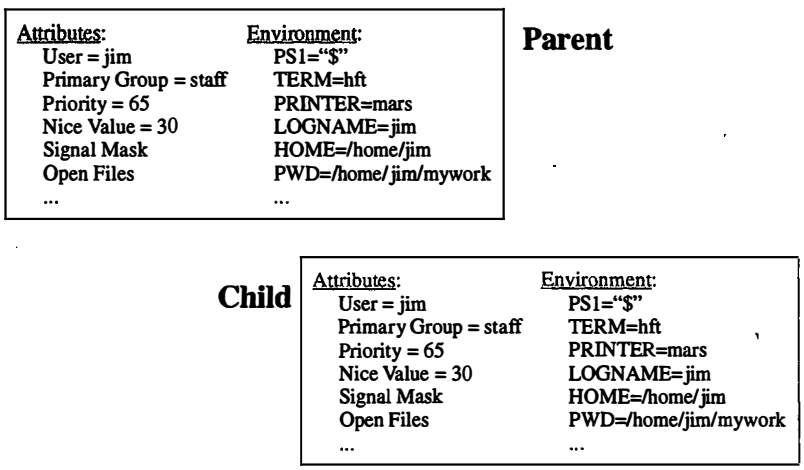

Figure 5.2 Process inheritance.

inherit environments and attributes. When a process terminates its environment also terminates.

Process attributes

In addition to text and user data, processes maintain information used by the operating system. Chapter 3 mentioned some of the attributes of a process. Here, we provide more details.

Process identification number. Probably the most well-known attribute of a process is its process ID number, or PID. Each process is guaranteed to have a unique PID within the system. The PID is used by many shell-level commands, such as kill. It is also used as a parameter to many system calls and subroutines, such as signal(). The /usr/include/sys/types.h header file defines a data type called pid_t, which is an integer. AIX 3.2 has a special way of assigning PIDs, which is described shortly.

Process group ID. Each process belongs to a process group. Processes can change process groups or create new process groups. The main purpose of a process group is to organize related processes such that an entire group can be controlled as a job. The C shell and the Korn shell use process groups to implement job control. Signals are sent to all processes within a process group via the killpg() subroutine (see the manual page for killpg() for more details). Each process group has a process identified as the group leader. The process group ID (PGID) is equal to the PID of the group leader process.

Session ID. Process groups can be collected into sessions. A session is one or more process groups. Each session has a leader. The session ID is equal to the PID of the session leader.

Credentials. Every process has a set of credentials that identify the user associated with the process. The credentials are actually a set of user and group IDs which are described shortly. The credentials are used by the system to determine the authority a process has over files and other resources.

Priority. Every process has a priority value used by the dispatcher to determine the order of time sharing of the CPU. The priority value for an AIX 3.2 process has a range of 0 through 127, where the lower the value, the more favored the process. For instance, a process with a priority of 62 is more favored than a process with a priority of 68. For most processes, the priority value changes frequently, as described later in this chapter.

Nice value. The nice value is a factor used in calculating the priority of a process. Users have some control over the nice values of their processes. See the

discussion of process scheduling later in this chapter for more details on process priorities and nice values.

Signal handling information. Signals are notifiers sent to a process by the kernel or by another process. A process has some control over how incoming signals are treated. See Chap. 10 for details on signals.

Accounting Data. As a process runs, it gathers statistics about itself, such as the amount of accumulated CPU time. These statistics are available to the process or its parent.

Opened files. Each process maintains a table of pointers to files the process has opened. The table is called the file descriptor table.

Operational environments

A RISC processor is always executing code from one of two different environments. It is either executing code from a process (or a system call running on behalf of a process), or it is executing the code of an interrupt handler. When the processor is executing process code or the code of a system call, it is running in the process environment. When an event occurs that interrupts the process environment, the system must handle the event by running an interrupt handler. The processor is then said to be running in the interrupt handler environment. The fundamental differences between the two environments are described in this section.

During process environment, the system operates in one of two modes: user mode, when executing the user code of a process, and system mode, when executing the kernel code of a system call. Recall from Chap. 3 that the system performs a mode switch when a system call is encountered during the execution of a process. The ID numbers of a process's 16 segments occupy the segment registers in hardware when a process is running. Processes may use the floating-point registers. The process environment is subject to page faults.

Interrupt handlers are always more favored than any process. They are found in the device driver which corresponds to the device whose interrupts they service. In addition, there are many interrupt handlers that react to interrupts generated by the system hardware, such as timer interrupt handlers.

Each interrupt handler runs with a priority value. While an interrupt handler is running, it blocks all interrupts that are less favored than itself. For this reason, interrupt handlers must complete as quickly as possible. Therefore, interrupt handlers may not experience page faults. All code and data touched by an interrupt handler must be pinned in memory. Figure 5.3 illustrates the process environment and the interrupt handler environment.

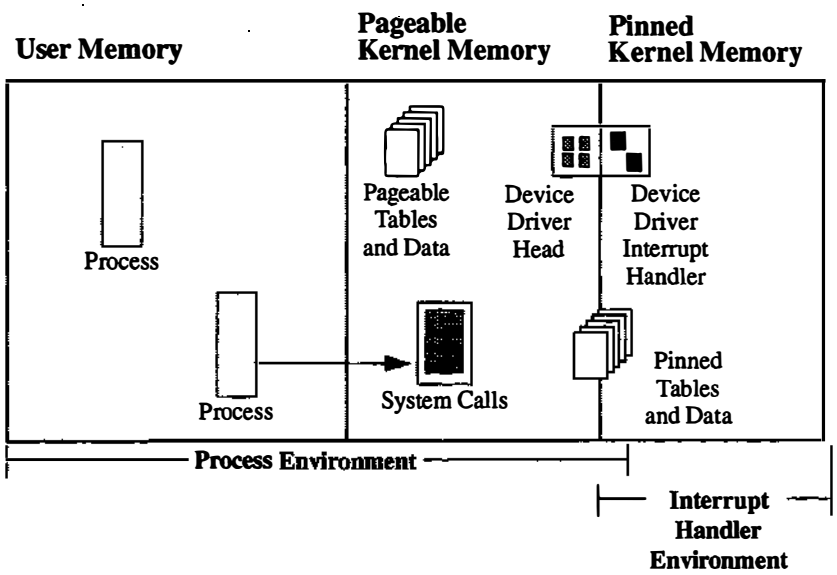

Figure 5.3 Operational environments.

5.2 Process Management Data Structures

This section describes kernel tables and other data structures used to manage processes.

The process table

The kernel's process table is at the heart of the process management subsystem. It is an array of proc structures as defined in the /usr/include/sys/proc.h header file. The size of the table is defined as NPROC, which is 131,072 entries. The process table is stored in the kernel's data segment (segment 14 of the 16 process segments).

With most UNIX-based operating systems, the NPROC value is tunable. It is not tunable in AIX 3.2. The /usr/include/sys/proc.h header file defines NPROC as (1 <<PROCSHIFT), where PROCSHIFT is 17. This is binary 100000000000000000, which is 131,072 decimal. The size of the process table is obviously larger than would ever be required. Traditionally, the process table must be pinned in memory since it is often accessed by interrupt handlers. Pinning a table as large as the AIX 3.2 process table would be unreasonable; therefore, AIX 3.2 only pins pages of the process table that have active entries.

Interesting fields in the proc structure include:

p_stat. A character field that indicates the state of the process table slot. Values are defined in the /usr/include/sys/proc.h header file and are discussed in Sec. 5.3.

p_wtype. A character field that indicates the event for which a process is waiting. Values are defined in the /usr/include/sys/proc.h header file.

p_flag. An unsigned long integer that indicates the current flag settings for the process. Values are defined in the /usr/include/sys/proc.h header file and are discussed throughout this chapter. The flag values are ORed to allow multiple flags to be set simultaneously.

p_child. A pointer to another proc structure (entry within the process table) for the head of the list of child processes. The use of this field is described shortly.

p_siblings. A pointer to another proc structure that forms a linked list of processes which, along with this process, share a common parent process.

p_pri. A character field that holds the process's current priority value. It is used by the dispatcher when determining which process to run.

p_uid. A uid_t data type that holds the real user ID of the process.

p_pid. A pid_t data type that holds the process's ID number.

p_ppid. A pid_t data type that holds the process's parent PID number.

p_pgrp. A pid_t data type that holds the process group ID number for this process.

There are many other fields in the process table which are described throughout this chapter. They are grouped according to their use. For instance, there are fields used by the process scheduler and dispatcher, fields used for signals, and fields used for identification.

Author's Note: The proc structure and the /usr/include/sys/proc.h header file in general provide a great deal of information about any UNIX-based operating system. It is one of the first files I examine when I encounter a new system.

The user area

Each process has, defined within its data segment, a user area, or U area. It was first described in Chap. 3 as part of the image of a process. A process's user area contains information about the process that is not needed when the process is not executing. Unlike the process table, process user areas are not accessed by interrupt handlers. The user area is probably the second most important component of process management, after the process table.

The user area is defined as a user structure in the /usr/include/sys/user.h header file. Interesting fields include:

u_procp. A pointer to the proc structure (entry in the process table) that represents this process.

u_signal[]. An array of pointers to functions that is used to vector to signal handling routines within the process's text segment. See Chap. 10 for details on how this array is used.

u_sigmask[]. An array of sigset_t data types that indicates which signals to block when executing a signal handler. See Chap. 10 for details on signal blocking.

u_segst[]. An array of segstate structure which defines the current state of the process's 16 segments. The array has 16 elements.

u_exh. A union that contains either the XCOFF header, if the process is executing a binary program, or the name of the shell interpreter, if the process is executing a shell script. For binaries, the union holds an xcoffhdr structure, as defined in the /usr/include/xcoff.h header file. The xcoffhdr structure holds the program's filehdr structure and aouthdr structure. See Chap. 3 for information on these two structures. For shell scripts the union holds an array of 32 characters called u_exshell[]. The array contains the string "#!" plus the path name of the shell, such as "#!/usr/bin/ksh." This notation is often used in shell scripts to identify which program to use when executing the script.

u_comm[]. An array of 33 characters used to hold the basename of the executable running in this process.

u_cred. A pointer to a ucred structure. The ucred structure holds the credentials of the process. Its definition and use are described shortly.

u_timer. A structure used to hold timer information. Timers are discussed later in this chapter.

u_ru. A rusage structure that holds resource usage information for this process. The rusage structure, defined in /usr/include/sys/resource.h, includes fields that hold run-time statistics that are available to the process's parent. For instance, the rusage structure holds the process's accumulated user mode and system mode times, which can be viewed by a user by issuing the time or timex commands. See the manual pages for time and timex for more details.

u_cru. Another rusage structure; however, this one holds accumulated resource usage statistics for this process and all of its children.

u_ttyd. A dev_t data type (device major and minor numbers) that indicates the controlling terminal for this process. The dev_t data type is described in Chap. 9.

u_cdir. A pointer to a vnode for the process's current directory. See Chaps. 7 and 8 for details on vnodes. This is how a process knows the starting point for relative path name resolution.

u_ufd[]. As described earlier, this array is the process's file descriptor table. It has a size of OPENMAX (2000 elements), where each active element points to an entry in the kernel's file table. A file descriptor table slot is allocated each time a process opens any type of file. See Chap. 7 for a complete discussion on the use of the per-process file descriptor table.

The u_procp pointer in a process's user area points from the process image to its entry in the kernel's process table, as illustrated in Fig. 5.4. The process table entry also has a link to the process's user area, although it's not a simple pointer. The p_adspace field, which is a vmhandle_t data type in the proc structure, holds the segment ID number of the process's data segment. The segment ID number is accessed through the kernel's segment information table to locate the process's user area, which is always at the same offset within the data segment.

A process's credentials

Each process's user area includes a pointer called u_cred which points to a ucred structure within the kernel memory. The ucred structure is defined in the /usr/include/sys/cred.h header file. It includes fields for user IDs and group IDs. The user ID fields are all uid_t data types and include:

cr_uid. The process's effective user ID

cr_ruid. The process's real user ID

cr_suid. The process's saved user ID

cr_luid. The process's login user ID

The group ID fields are all gid_t data types and include:

cr_gid. The process's effective group ID

cr_rgid. The process's real group ID

cr_sgid. The process's saved group ID

cr_groups[]. An array of size NGROUPS_MAX (defined as 32 in the /usr/include/sys/limits.h header file) which holds the group IDs for the concurrent group set.

AIX 3.2 allows a user to belong to up to 32 groups at a time. Group access to files and other objects is granted if the group associated with the file or other object is present in the process's concurrent group set. One group is designated as the process's primary group. By default, this is the group designated in the /etc/passwd file for the user account. A user can change their primary group for a login session by using the newgrp command. The primary group is associated with any file created by the process. Figure 5.5 illustrates AIX 3.2 group sets.

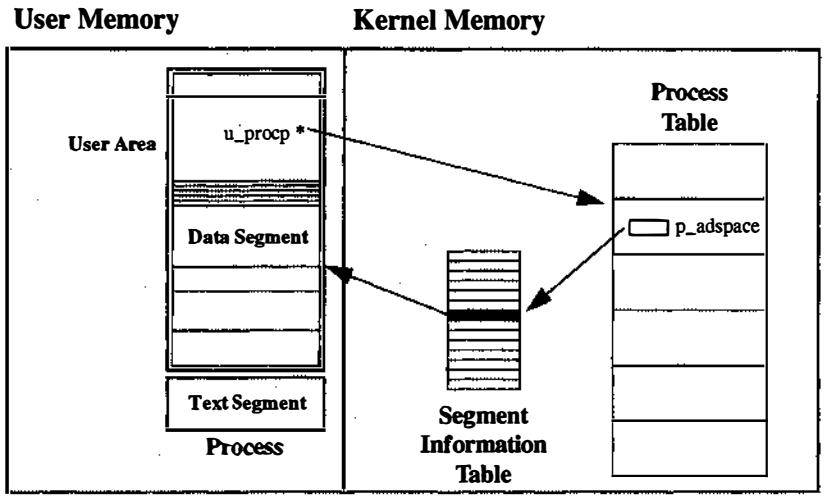

Figure 5.4 The user area and the process table.

As seen in the ucred structure, four different user IDs are maintained per process. The login user ID (luid) is the ID number of the account used to log in to the system. It never changes as long as the user is logged in. The real user ID (ruid) is the ID number of the account for the user of the system. It changes whenever a user successfully issues the su (switch user) command. It can also be changed by a process running with root authority (user ID = 0) issuing the setuid() subroutine.

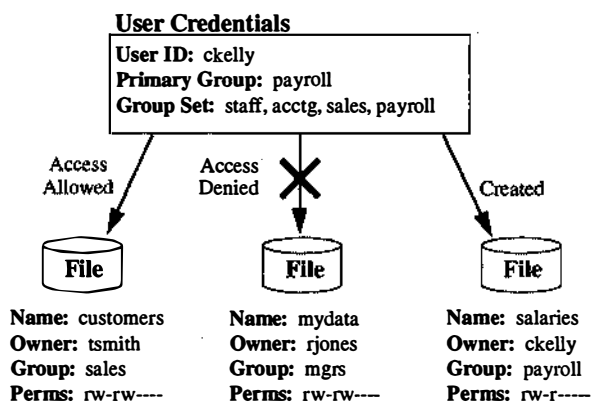

Figure 5.5 Group sets.

The effective user ID (euid) is the same as the real user ID unless the process has executed a program that has set-user-ID-bit turned on. The set-user-ID-bit is activated for an executable file by ORing the value 04000 with the other octal notation of the chmod command or chmod() subroutine. When a user executes a program that has the set-user-ID-bit turned on, the effective user ID of the process is set to the user ID value of the owner of the program. This feature allows system administrators to grant root or other special privileges to a non-privileged user when they execute a specific program. It also allows programmers to grant users access to data files only when the users execute specific programs.

Figure 5.6 illustrates an example of a set-user-ID-bit program. The program, called ponies, handicaps horse races. Its owner, dkelly, allows any user to execute the program. During execution, the ponies program opens and performs I/O operations on a file called horses.ascii, which is an ASCII data file. dkelly does not want any user to be able to directly access the horses.ascii file with an editor or other tool, so he sets the permissions such that only dkelly can perform read and write operations on the ASCII file. In order to allow other users to access the horses.ascii file when executing the ponies program, the set-user-ID-bit is activated for the ponies program. A set-group-ID-bit works in a similar fashion to the set-user-ID-bit. When it is activated for a program file, any process that executes the program has its effective group ID set to the group ID associated with the program.

The set-group-ID-bit has a special meaning for data files. It is used to implement mandatory file and record locking as described in Chap. 7.

Programs that have the set-user-ID-bit activated sometimes call the setuid() subroutine to change the effective user ID value back to the value of the real user ID. This is usually done after some privileged access has taken place and the special effective user ID is no longer needed. If the program needs to use the special set-user-ID-bit value, it can get it from the saved user ID field. Figure 5.7 shows how the four user ID fields evolve during the execution of a program.

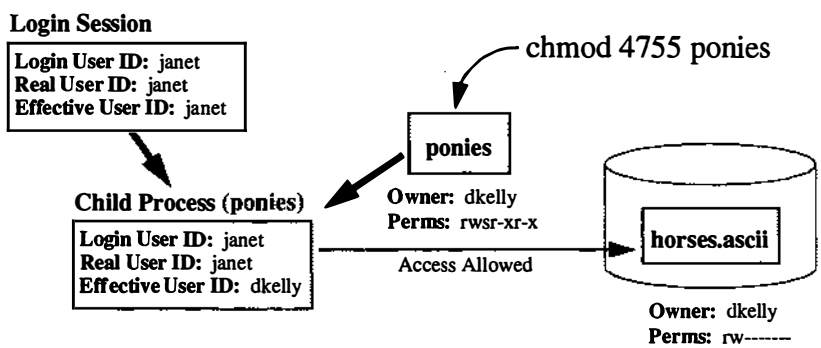

Figure 5.6 A set-UID-bit program.

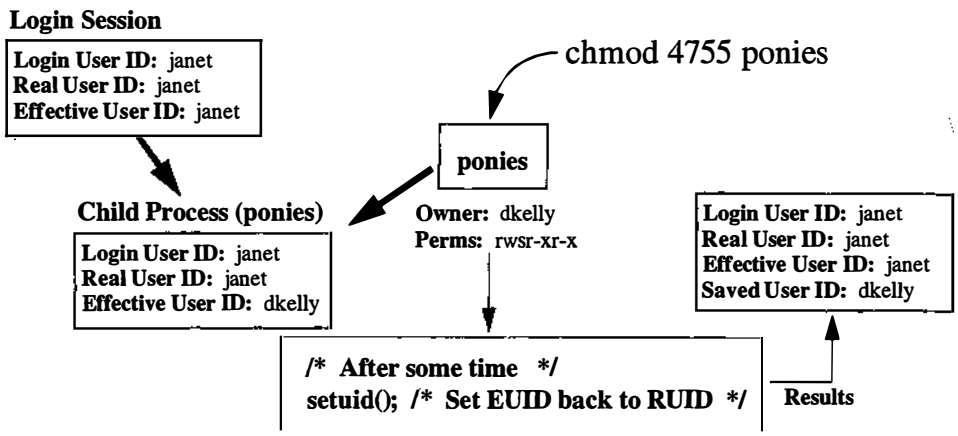

Figure 5.7 User IDs and the setuid() subroutine.

Author's Note: Most UNIX-based systems store the user IDs and group IDs directly inside of the user area. By placing these values in a separate ucred structure, AIX 3.2 accomplishes two things. It allows multiple processes to point to the same ucred structure (note that the ucred structure has a field called cr_ref, which serves as a reference count) instead of having the same information repeated in each process. This saves space. It also allows the credential information to be stored in kernel memory, which is much more secure than user memory. The credential structure can then be used by various security components of the operating system.

How AIX 3.2 generates process ID numbers

Most UNIX-based systems assign process ID numbers sequentially as new processes are created. When the last available process ID number is assigned, the sequence starts over with the lowest available nonassigned value. AIX 3.2 does not use this technique. Instead, the process ID number is broken down bitwise into two parts. As an integer, the process ID number consists of 32 bits, but only the rightmost 26 bits of the 32-bit word are used. From that 26 bits, the leftmost 18 bits are used as an index into the process table. The remaining (rightmost) 8 bits serve as a generation number which is incremented each time a process table slot is assigned. In this way, a process's PID can be used to determine its slot in the process table. Figure 5.8 illustrates the configuration of an AIX 3.2 PID.

The /usr/include/sys/proc.h header file defines macros for converting a PID into a process table slot number. The conversion, shown in Fig. 5.9, is based on bitwise ANDing the PID with the value 0x7ffff00, then shifting the results to the right by 8 bits.

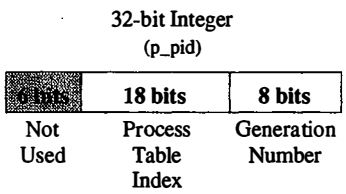

Figure 5.8 AIX 3.2 process ID numbers.

proc.h

```
#define GENSHIFT 8
...
# define PROCPTR(pid)                                    \
      ((pid) != 1)  ?                                    \
      &proc[((pid) & 0x7ffff00) >> GENSHIFT] :   \
      &proc[1])
```

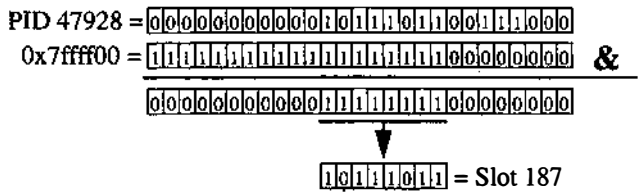

Figure 5.9 Converting a PID into a process table slot number.

Author's Note: Another method I have used for determining a process's slot number in the process table is to convert the decimal-based PID to hexadecimal, drop the right two digits, then convert the number back to decimal. For instance, a PID value of 47928 converts to 0xbb38. Dropping the right two digits leaves 0xbb which, when converted back to decimal, is 187. Therefore, a process with a PID of 47928 would always occupy process table slot 187.

Author's Note: There is a concern about the way AIX 3.2 assigns PIDs. Since their value is determined by the index of a process in the process table, they are not assigned sequentially. This means that it is possible for a child process to have a lower PID value than its parent. I have encountered shell scripts over the years that count on a child's PID being higher than that of its parent. While this is not a good programming assumption to make, it does happen. Such shell scripts, or any kind of program for that matter that depends on sequential PID assignment, will find difficulties in AIX 3.2.

5.3 User Process Creation

This section details the life cycle of a process, from its creation to its termination. It also describes family relations between processes and how the kernel keeps track of those relationships. The fork(), exit(), wait(), and exec family of subroutines are also presented.

The fork() system call

As mentioned earlier, almost all user processes exist because a parent process created them. The exceptions are PID 0, the system scheduler (whose name is "swapper") and PID 1, the "init" process. These two processes are hand-crafted at system start. Actually, all other user processes can trace their ancestry back to init.

A new user process is created when an existing user process issues the fork() system call. The definition of the fork() system call is that it makes a new process (child) that is an exact clone of the calling process (parent). This is not completely accurate, however. The child process does inherit most of its attributes and properties from its parent, but there are obvious differences. For instance, the child process has its own unique PID. It also has a different PPID (parent's process ID number) than its parent. Finally, resource statistics for the child process are initialized to zero to give the child a fresh start. Other attributes, many of which have been discussed already, are inherited by the child. They include the priority and nice values, the credentials, the process group membership, and the parent's opened files.

> **Author's Note:** The fact that the child process inherits the parent's opened files by receiving a copy of the parent's file descriptor table is not a trivial point, as is described in Chap. 7.

The fork() system call works with the virtual memory manager (VMM) to establish segments for the new process. A child process shares the same text segment as its parent. In fact, the child process inherits the same program counter address (which is stored in the instruction address register, or IAR). This means that the child process comes to life running the fork() system call. The VMM allocates a new data segment for the child process; however, pages from the parent's data segment are only copied as either the parent or child modifies them. This is a VMM technique called "copy on write" which helps reduce the overhead of the fork() system call. The child process receives a new user area structure, but many of the fields from the parent's user area are copied into the child's user area.

The first thing the fork() system call does is verify that the calling process's user ID has not reached the maxuproc value. The maxuproc parameter, which is tunable via the system management interface tool (SMIT), specifies the largest number of processes allowed per nonroot user ID. By default, the value of maxuproc is 40. It prevents users from accidentally or intentionally creating recursive programs that call fork().

The fork() system call also verifies that the VMM has an adequate amount of free paging space slots. If this is not the case, fork() retries after waiting a specified amount of time. If the fifth retry fails, the fork() system call fails and returns a value of -1 to the calling process. The wait time between retries is tunable via the /usr/lpp/bos/samples/schedtune command. See the manual page for schedtune for more details.

The fork() system call assigns a slot from the process table. A page fault may occur if the assigned slot is from a page of the process table that has no other active entries and is therefore not pinned. The proc structure includes a field called p_stat, which is set to the value of SNONE when the slot is not in use. It is changed to the value of SIDL while fork() is creating the child process.

> **Author's Note:** The SIDL value for p_stat means that the process is "under construction," since the process table slot is no longer available yet the process has not been realized in memory. The p_stat field holds this value for a very short time and never returns to this state.

The fork() system call generates a unique PID based on the process table slot index. It places a lock on the process table while the slot is allocated and the PID is generated. This prevents any other process or interrupt handler from interfering with the fork() system call while it is executing a critical section of code.

Kernel services called by fork() handle the creation of the new segments, as well as the copying of values from the parent to the child. Pointers are also updated to include the child process in the appropriate linked lists, as described shortly. Once the child's VMM segments are in place, fork() sets the value of the p_flag field in the child's process table slot to the value of SLOAD, which means that the process is "in core."

One of the last things the fork() system call does, running on behalf of the parent process, is to link the child to a run queue and set the p_stat field to a value of SRUN. This makes the child process "ready to run." The fork() system call then returns the PID of the child back to the parent. When the child is selected to run, it starts in the middle of the fork() system call! This is to allow the child process to finish its side of the fork(). The child's version of the fork() system call returns a zero to the child process. Figure 5.10 illustrates the fork() system call. The fork() system call returns a -1 to the calling process if it fails to create a child. Two conditions that may cause fork() to fail are insufficient paging space and the maxuproc value reached for the caller's user ID.

The process family tree

The process table includes linked lists to keep families of process together. First, every proc structure includes a pointer called p_child. It points to another proc structure that represents a child of the process. Since a process can create many children, this pointer points to a child that is designated as the head

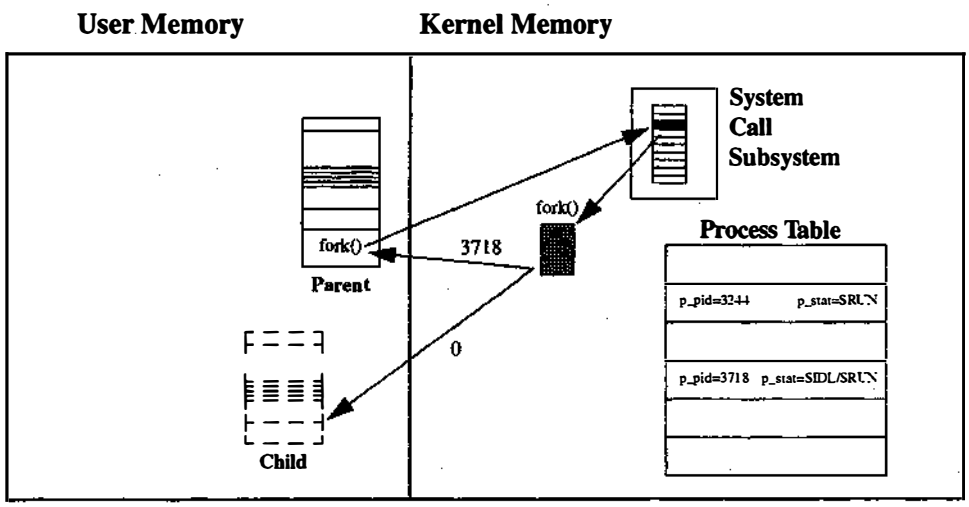

Figure 5.10 The fork () system call.

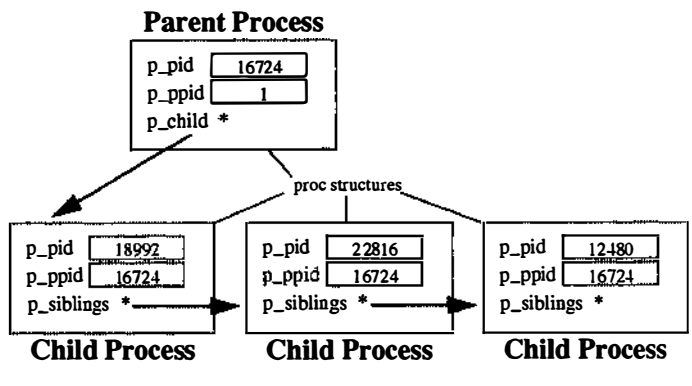

Figure 5.11 A process family tree.

of the child list. The child processes of any parent keep track of one another through a null terminated link list associated with the p_siblings pointer. Figure 5.11 displays the relationship between a parent process and its linked list of children.

> **Author's Note:** This is a strange family. The parent is only able to remember one child, so it tells that child to remember one of its siblings, which, in turn, must remember one of its siblings, and so forth. Sounds like the theme for a television talk show.

Whenever a new process is created or an existing process terminates, the p_siblings list is reconstructed. If the child process that is the head of the sib-

lings list terminates, the parent's p_child pointer must be relinked to the new head of the siblings list.

Process groups

Process groups were introduced earlier in this chapter. Each proc structure has two fields that manage a process group. The p_pgrp is a pid_t data type that holds the process group ID number, which is also the PID of the process group leader. A pointer called p_pgrpl is used to create a circularly linked list of all processes within the process group. Figure 5.12 illustrates how the process table maintains process groups.

The process life cycle

Figure 5.13 illustrates the life cycle of a user process. The labeled states correspond to values of the p_stat field in a process table slot (proc structure). A slot that is not active has a p_stat value of SNONE. During the creation of a new process, the p_stat field is set to SIDL, or "intermediate state of process creation." This means that the slot is spoken for, even though the process is not ready to run yet. A new process passes through this stage only once and for a very short period of time.

The fork() system call makes the new process ready to run by changing the value of p_stat to SRUN. The process is now said to be on a run queue and fair game for selection by the kernel's dispatcher routine. An SRUN process is not

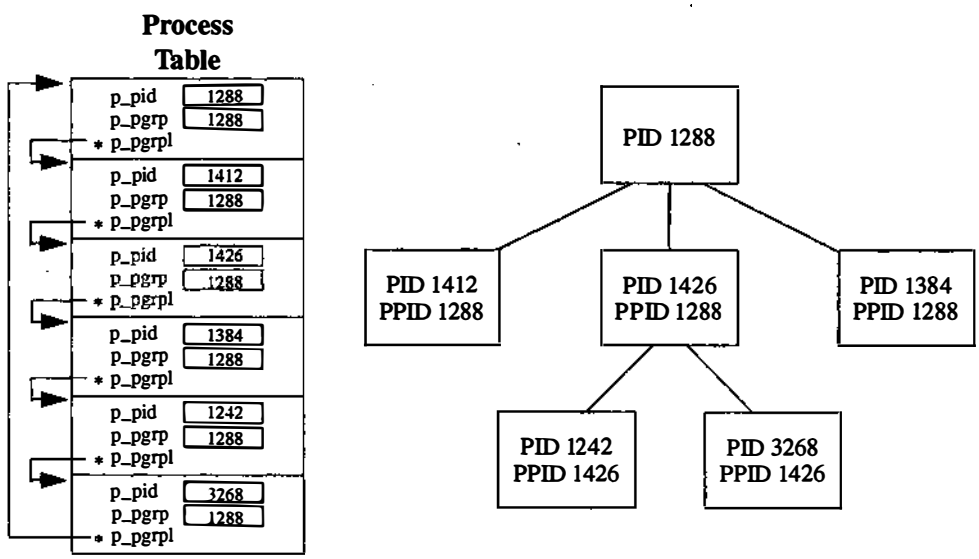

Figure 5.12 Process groups.

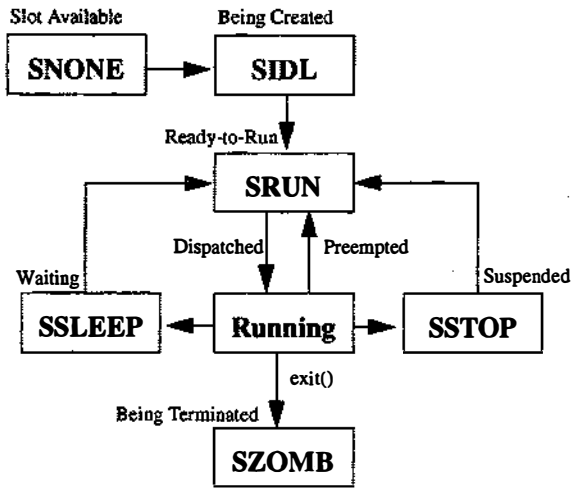

Figure 5.13 The process life cycle.

waiting on any event or condition. Since AIX 3.2 does not support multiprocessor hardware, there will only ever be one process selected to run at any given time. The p_stat value for the current running process is still SRUN. AIX 3.2 does not have a name for the running state.

There are five possible things that can happen to a running process: it may be preempted by the operating system, making way for another process to run; it may be interrupted by an event that causes an interrupt handler to run; it may voluntarily put itself to sleep while waiting on an event or condition; it may be stopped (not terminated, simply suspended) by the operating system; or it might call the exit() subroutine and terminate. Each of these possibilities is explored now.

Preemption occurs for the current running process when the kernel's dispatcher has determined that a more favored process is ready to run. The current running process could still use the CPU, but it will have to wait until it is once again the most favored ready-to-run process. The preempted process is placed back on the run queue.

If a system call running on behalf of a process must wait on an event, such as disk I/O, page fault resolution (which usually involves disk I/O), a timer, or any other type of event, the system call will cause the process to voluntarily give up the CPU by calling the esleep() kernel service. This changes the p_stat field to the value of SSLEEP. Sleeping processes are not considered by the dispatcher when selecting a process to run, since they are not in a ready-to-run state. Most interactive processes spend the majority of their time sleeping. When the event that the sleeping process is waiting on occurs, the interrupt handler associated with the event wakes the sleeping process by setting p_stat to the value of SRUN, making the process ready to run. Note that when a sleep-

ing process wakes, it does not automatically go back to running but rather must contend for the CPU with all other ready-to-run processes. The scheduling algorithm used by AIX 3.2, however, fairly guarantees that a freshly awakened process is dispatched quickly to respond to the event.

Interrupts are driven by events associated with devices. For instance, an interrupt occurs when a disk drive controller notifies the system that data requested by some process have been fetched and are ready for transfer to the process. Since the process that requested the disk data is usually sleeping while waiting for the data, interrupts are seldom associated with the current running process. An interesting characteristic of AIX 3.2 is that whenever an interrupt wakes a sleeping process and places it back on a run queue, the interrupt handler should call the dispatcher so that it sees the ready-to-run process. More on this shortly.

A process with a p_stat value of SSTOP is suspended. This occurs when the kernel or another process sends a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal to the current running process. There are various ways of suspending a process. For instance, the <CTRL>-<z> key combination instructs the C shell or the Korn shell to suspend the current running foreground process. This is part of those shells' job control feature. Processes are also stopped by break points set within a debug session. Finally, the stty command has options, such as tostop, which instruct a terminal session to suspend any background process that attempts to send output or request input for the controlling terminal. The process leaves the SSTOP state when it receives a SIGCONT signal. Note that the process returns to the ready-to-run state.

The final thing that can happen to the current running process is for the process to exit. This causes the p_stat field to change to the value of SZOMB. The process is now said to be in zombie state. All processes become zombies while they exit. A process should only be in this state for a short period of time, while its resources are deallocated. The SZOMB state is the opposite of the SIDL state. Zombies are discussed in more detail later in this section.

The exec family of system calls

As previously mentioned, the fork() system call creates a new process that is almost a clone of the calling process. In fact, after the fork() call, both the parent and child processes are executing the same text. This, by itself, may not seem very useful. Usually, it makes more sense for the parent and child processes to perform their own specific tasks. This is accomplished by having the child process execute another program within its process image. The exec family of system calls makes this possible.

There is no exec() system call, but rather a set of system calls based on the name. Figure 5.14 lists the various calls used to execute programs along with the characteristics of each. Consult the manual page for the execl() subroutine (all of the variations are described on the same manual page), or InfoExplorer for more information on implementation specifics.

```
execl(char *path, char arg0, ..., (char *) 0)

execle(char *path, char *arg0, ..., (char *) 0, char *envp[])

execlp(char *filename, char *arg0, ..., (char *) 0)

execv(char *path, char *argv[])

execve(char *path, char *argv[], char *envp[])

execvp(char *filename, char *argv[])
```

Figure 5.14 The exec family of system calls.

> **Author's Note:** For simplicity, I will refer to the exec family of calls as a generic exec() call throughout this chapter. From an internals perspective, the kernel activities for each of the exec calls are almost identical.

When a process calls exec() the kernel's loader routine locates the specified executable file, performs all the necessary tasks to make the program runable, and loads and executes the program's text within the image of the calling process. To make the program runable, the loader must resolve all references to external symbols as described in the loader section of the program's XCOFF file. This is dynamic binding. The virtual memory manager allocates a new text segment for the program code if no other process is executing the program. If one or more processes are already executing the program, the virtual memory manager allows the process calling exec() to share the existing text segment. The loader also creates, within the process's data segment, the areas for initialized global data, noninitialized global data (the BSS), and the user stack. The process's user area survives the exec() call, but some of the field values within the user area change to reflect information specific to the newly executed program. One important point to emphasize is that the process's file descriptor table, which references files already opened by the process, survives the exec() call. The only exceptions are any files that were previously opened by the process with the CLOSE_ON_EXEC flag specified when the files were opened. The fact that the process's file descriptor table survives an exec() call is crucial to pipes, sockets, and other forms of interprocess communications, as described later in this book.

The exec() call returns a -1 value upon failure. Reasons for failure include the nonexistence of the specified program, or lack of privilege needed to execute the program. There is no return from exec() if the call succeeds. This is because the text is replaced by the text of the newly executed program. This is illustrated in Fig. 5.15. The program example assumes that if it reaches the line of code following the execl() call, the execl() call must have failed.

```
...
main()
{
   ...
   if(fork()==0)     /* We must be the child */
   {
      execl("./myprog","myprog",0);
      perror("Exec failed");
      exit(1);
   }
   else   /* We must be the parent */
   wait();
   ...
}
```

Figure 5.15 An exec example.

Process termination

There are various events that cause a process to terminate. A process can choose to terminate by calling exit() or abort(). A process will also terminate when it reaches the closing brace (}) of the main() function. (Actually, the linkage editor supplies a call to exit() if it is not included by the programmer.) Finally, most signals received by a process will cause termination of the process. See Chap. 10 for details on signals. In all of these cases, the events in the kernel that handle process termination are the same.

The exit() subroutine not only terminates the calling process but also allows the programmer to specify an exit value, which is an integer parameter to the exit() call. The exit value is passed back to the calling process's parent. The exit values used by the programmer have no special meaning to the kernel. Their meanings are significant only between the child and the parent. It is customary in UNIX, however, to use an exit value of zero to indicate successful program completion, and a nonzero value to indicate an abnormal, or failed, program completion.

> **Author's Note:** The exit value parameter to the exit() subroutine is used in the same way as the argument to the exit shell command, frequently used in shell scripts. The $? special variable of the Bourne and Korn shells, or the $status special variable of the C shell, reports the exit value of the previously terminated child process, whether the child used the exit command (as in the case of a shell script) or the exit() subroutine (as in the case of a binary executable).

There is a great deal of kernel activity involved in the termination of a process. First, the kernel changes the p_stat field of the terminating process's process table entry to the value SZOMB. This means that the process has become a zombie. This is a normal occurrence in the death of a process, but a process should only remain in the zombie state for a short period of time. More details on zombies are given shortly.

Next, the kernel places the exit value of the terminating process in the _p_xstat field of that process's slot in the process table. The _p_xstat field in the process's proc structure holds the exit status. Another field, _p_ru, which is a rusage structure, holds the resource usage information for the terminating process. This information is copied from the terminating process's user area and becomes available to the parent process. Note that the _p_xstat and the _p_ru fields, along with some padded space, are part of a structure called _p_szomb, which, in turn, is part of a union called _p_ovly. The comments in the proc structure indicate that the _p_szomb structure is only used when the p_stat field has a value of SZOMB. The other structure of the union, _p_signal, is used to handle signals received by the process. A terminating process may not receive signals; therefore, the two union members constitute a mutually exclusive condition. This saves space in the process table, as illustrated in Fig. 5.16.

As a process terminates, all files held open by the process are closed. This includes sockets, as well as ordinary files and directories. The reference counts for all file table entries associated with the open files are decremented as the files are closed. Any file locks held by the process are released. See Chap. 7 for details on the kernel's file table and file locks. The virtual memory manager deallocates the virtual memory segments that make up the process. At this point, the process ceases to exist. All that remains is the exit status information in the zombie slot of the process table.

When a process terminates, the kernel sends a SIGCHLD signal (signal number 20) to the parent. This notifies the parent process of the death of the child. Keep in mind that the child process is still linked to the sibling list. In other words, although the child process has terminated and has become a zombie, it's still a family member, as illustrated in Fig. 5.17.

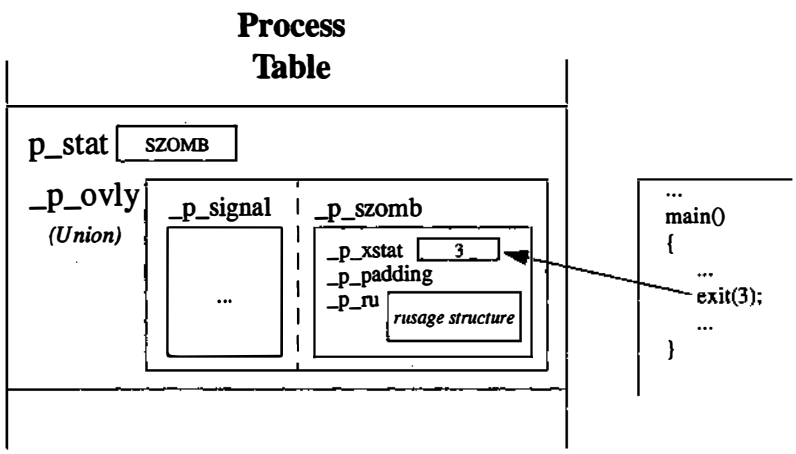

Figure 5.16 Exit status and the process table.

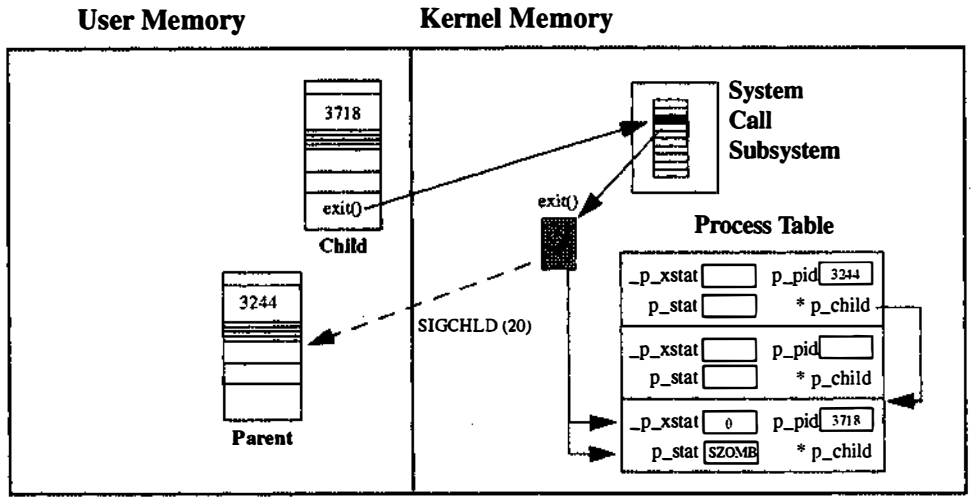

Figure 5.17 Process termination.

Cleaning up zombies—The wait() system call

When a parent process creates a child process via the fork() system call, the parent normally issues a wait() system call, which suspends the parent until the child terminates and the parent receives the SIGCHLD signal. The wait() system call then fetches the exit status value from the child's process table slot and clears the zombie by changing the p_stat field to SNONE (slot available). If the parent issues the wait() system call immediately after returning from fork(), the parent is serialized to run again only after the child has terminated. A parent need not call wait() immediately after returning from fork() but can, instead, continue to execute other instructions. This means that the parent and child processes are running in parallel and contend with each other for CPU scheduling. The two scenarios just given describe how the shell performs foreground and background processing. Figure 5.18 shows pseudo code for a command interpreter. If a command is entered with an ampersand (&) symbol at the end of the command string, the interpreter does not issue the wait() call after spawning the new process but continues with the next iteration of the loop and prompts the user for another command. If there is no ampersand symbol, the interpreter spawns the child process, then waits until the child terminates to perform the next iteration of the loop. While this example is an oversimplified version of a shell, and there is a problem that occurs when the user issues a number of background commands, it demonstrates how the shell creates children to perform the specified commands.

The most common technique used by applications that don't want to block until their child process terminates is to register a signal handler for the

```
endless loop
{
    prompt user for a command
    if command is a shell built-in
    {
        branch to built-in routine
        continue next iteration of the loop
    }
    if last character of command string is '&'
    {
        create a child process via fork()
        child: execute the specified command
        parent: continue next iteration of the loop
    }
    else
    {
        create a child process via fork()
        child: execute the specified command
        parent: wait() for child to terminate
    }
}
```

Figure 5.18 An example of a command interpreter.

SIGCHLD signal. Signals and signal handlers are detailed in Chap. 10, but a simple signal handler must be described here in order to demonstrate how a process can continue to execute instructions while its children are also active. The signal() subroutine registers a signal handler for a process by allowing the program to specify the signal to watch for and the action to take when the signal is received. The program example in Fig. 5.19 registers a signal handler, which is a function called deal_with_it(), and instructs the program to call the handler when a SIGCHLD signal is received. When the child process terminates and the parent receives the SIGCHLD signal, the program suspends the activities of the some_very_long_routine() function and branches to deal_with_it(). The deal_with_it() handler issues a wait() call to fetch the exit status of the child and to clean up the zombie. Once this is completed, the program returns to the some_very_long_routine() function and continues where it left off. This program only creates one child so the signal handler need only be called once. If a program creates many children, and some of the children terminate at almost the same time, the SIGCHLD signal handler may miss some of the signals. This is because the parent can only process one signal at a time and signals of the same type do not stack or queue. In other words, if the parent receives two or more SIGCHLD signals nearly simultaneously, it will only see one. The reasons for this are explained in Chap. 10, but the following text describes one method for overcoming this problem.

AIX 3.2 supports three versions of the wait() system call. The traditional wait() system call takes, as a parameter, the address of an integer where the child's exit status value is to be stored, and returns the process ID number of

```
...
main()
{
        ...
        signal(SIGCHLD, deal_with_it);
        ...
        if(fork()==0)     /* Child  */
        {
                execl("./myprog","myprog",0);
                perror("Could not execute");
                exit(1);
        }
        else
                some_very_log_routine();
        ...
}

deal_with_it()
{
        wait();
}
```

Figure 5.19 Clearing zombies with a signal handler.

the deceased child. The waitpid() system call has three parameters; the address of an integer where the child's exit status value is to be stored, a pid_t data type (integer) that specifies the process ID number of the child for which the calling process is waiting, and a flag value. If the second parameter (the child's process ID) is zero, the waitpid() system call fetches the exit status of the first zombie process in the linked list of children. This technique is the same as the traditional wait() system call. The flag parameter can be set to the value W_NOHANG, which causes the waitpid() call to return a value of zero when there are no zombies on the linked list of children. Figure 5.20 shows how the waitpid() system call is used with the W_NOHANG flag to clean up multiple zombies that have occurred nearly simultaneously. When the parent receives a SIGCHLD signal, it branches to the zclean() handler, which loops until all zombies within the parent's linked list of children have been cleaned up. When there are no more zombies, the waitpid() call returns zero without hanging, which terminates the loop and returns from the handler.

```
#include <sys/wait.h>
...
zclean()
{
    signal(SIGCHLD,zclean);
    int rv=1;
    while(rv=waitpid(0,0,WNOHANG));
}
```

Figure 5.20 Another signal handler.

Author's Note: Notice that the signal handler reregisters itself within the zclean() routine. The reason for this is explained in Chap. 10.

The third version of waiting supported by AIX 3.2 is the wait3() system call. This call fetches the resource usage information of the deceased child process and its descendants, as well as the exit status of the child. See the manual page for wait(), waitpid(), and wait3() for more information.

There is a way for a parent process to clean up a zombie without using one of the wait() routines. A signal handler can be registered with the SIGCHLD signal specified and the action of SIG_IGN (ignore the signal). This causes the p_stat field of the zombie's process table slot to change from SZOMB to SNONE. The exit status information is ignored. Figure 5.21 shows the command interpreter example from earlier with the addition of the signal handler to ignore and clean up zombies.

If a process never waits for a child or does not register a signal handler to ignore the SIGCHLD signal, the child will remain in a zombie state until the parent terminates. Long-lived zombies show up as <defunct> processes in the output of the ps command. The kill command does not remove a zombie because the kill command works by sending a signal to the specified process, and the receiving process must run in order to act upon the signal. Zombies can't run because they no longer exist; thus the signal is never delivered. This is why AIX 3.2 uses a union to hold either signal information or the process's exit status. Once a process terminates and becomes a zombie it has no need for signal handling fields.

```
#include <sys/signal.h>
...
signal(SIGCHLD,SIG_IGN);
endless loop
{
    prompt user for a command
    if command is a shell built-in
    {
        branch to built-in routine
        continue next iteration of the loop
    }
    if last character of command string is '&'
    {
        create a child process via fork()
        child: execute the specified command
        parent: continue next iteration of the loop
    }
    else
    {
        create a child process via fork()
        child: execute the specified command
        parent: wait() for child to terminate
    }
}
```

Figure 5.21 The command interpreter revisited.

Author's Note: Students often ask what penalty is suffered if a system has a lot of zombies that aren't quickly cleaned up. Aside from each zombie's taking up a slot in the process table (256 bytes per slot), since they no longer exist, they take up no other system resources; however, each zombie counts toward the total number of processes that a user has at any given time. This might cause a user to hit the maxuproc limit, discussed earlier in this chapter.

5.4 An Introduction to Process Scheduling

At the heart of the AIX 3.2 process management subsystem is process scheduling. It consists of a kernel subroutine called dispatch(), known as the dispatcher, a process called swapper, which is the scheduler process, a set of linked lists of proc structures within the process table, known as run queues, and a set of internal variables and algorithms. The goal of the process scheduling subsystem is to make sure the most favored ready-to-run process is running at all times.

Process priorities and run queues

As mentioned earlier, each process has a priority value, stored in the p_pri field of the proc structure. The priority value is a number between 0 and 127 and, for most processes, changes frequently. The lower the priority value of a process, the more favored it is. In other words, a process with a priority value of 62 is more favored than a process with a priority value of 65. When a process is in a ready-to-run state (i.e., not waiting on any event, such as I/O completion), the process is placed on a run queue. AIX 3.2 has 128 run queues, numbered 0 through 127. A ready-to-run process is placed on the run queue that corresponds to the process's priority value. For example, a ready-to-run process with a priority value of 65 is placed on run queue 65. Since most process priorities change frequently as they run, their run queue associations also change. The run queues are actually doubly linked lists created by the p_next and p_prior pointers in the proc structure, as illustrated in Fig. 5.22. Processes that have a common priority value are on the same linked list, or run queue.

Figure 5.23 illustrates the AIX 3.2 run queues. Ordinary, nonroot user processes always have priority values between 60 and 126. (Priority value 127 is reserved for a special kernel process that is described shortly.) Only processes granted special consideration by root authority can have priority values between 1 and 39. (Priority value 0 is reserved for the page stealer, which was described in Chap. 4.) Priority values between 40 and 59 are reserved for processes running with root authority and a negative nice value. The nice command and nice values are discussed shortly.

The process with a priority value of 127 is a special kernel process called "wait." In AIX 3.2 it always has a process ID number of 514 and is always ready to run; therefore, it only runs when there is no other process ready to run. The "wait" process represents processor idle time, as displayed by performance monitoring tools such as sar and iostat.

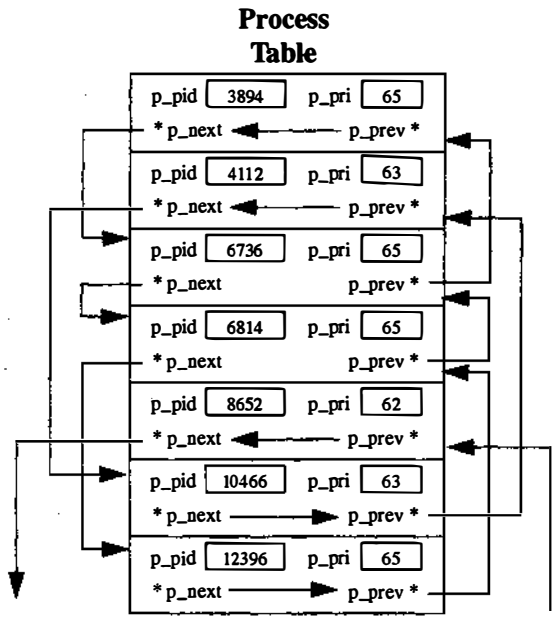

Figure 5.22 Process priorities and run queues.

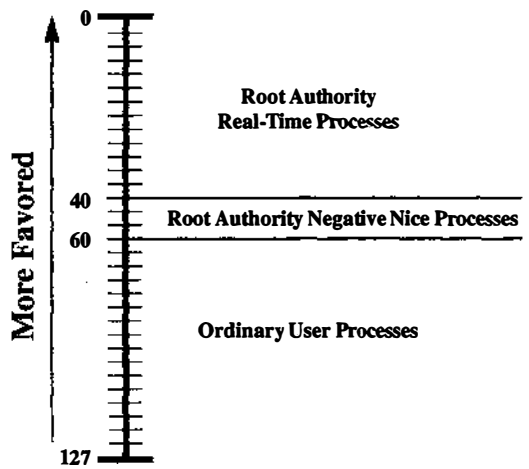

Figure 5.23 Run queue levels.

Author's Note: I once had a student ask if his system was CPU-bound because he noticed, when he did a ps -elk command, that process ID 514 was using 98 percent of the CPU time. In fairness to the student, since the ps -k command in AIX 3.2 only lists the names of kernel processes as "kproc," it's often difficult to determine their purpose.

The dispatcher

The dispatcher is a kernel routine called dispatch(). As a kernel routine, it can only be called by system calls or interrupt handlers within device drivers. It cannot be called directly by an application. The dispatcher scans the run queues, selects the most favored ready-to-run process, and dispatches it to run. Specifically, it resumes running the current process if that process is still the most favored, or it calls the swtch() kernel routine to perform a context switch.

Author's Note: The swtch() routine is spelled without an "i" because "switch" is a reserved word in the C programming language.

To quickly select the most favored process, the dispatcher scans a 128-bit mask. The mask has one bit for each run queue. If there are no ready-to-run processes linked to a particular run queue, the corresponding bit in the mask is 0. When a run queue has proc structures linked to it, the corresponding bit in the mask is set to 1. A kernel routine called requeue(), which is called whenever a process is placed on a run queue, is responsible for setting and unsetting bits in the mask. Once the dispatcher has determined the most favored run queue with a ready-to-run process, it consults an array called proc_run[], which has 128 elements, each containing a pointer to a proc structure. It uses the bit number from the mask as an index into the proc_run[] array. The pointer in the array element points to the proc structure that represents the process that is at the head of that particular run queue. That process is then dispatched. Figure 5.24 illustrates how the dispatcher selects a process to run.

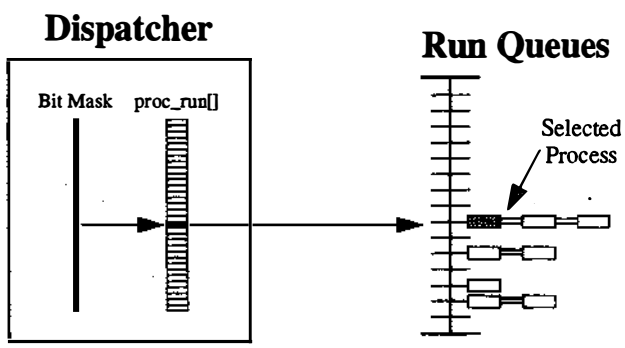

Figure 5.24 The dispatcher.

As mentioned earlier, the dispatcher is called by interrupt handlers and system calls. This is usually done whenever a process is placed on a run queue, such as when a process is awakened from a sleep state, or when a process is moved from one queue to another. The dispatcher is called by setting a global kernel flag called "runrun." Systems programmers who write device drivers or system calls must use the runrun flag to call the dispatcher whenever a process is placed on a run queue.

Process scheduling fields

Each proc structure (entry in the process table) has three fields that are used to maintain a process's priority. The p_pri field holds the priority itself. A field called p_cpu holds the count of recently accumulated clock ticks of CPU time. A clock tick in AIX 3.2 is 1/100 second. The concept of "recently accumulated" means the value of this field decays periodically, as is demonstrated shortly. The third field is the p_nice. It holds the process's "nice" value. Figure 5.25 shows the fields used for process scheduling.

The nice value

Every process has a nice value. By default, the nice value is 20. If a user is about to run a program whose response time is not critical, and the user does not want to wrest CPU time away from other processes running at the same time, the user can "nice" the program. This is done by adding the nice command to the beginning of the command string, as shown in Fig. 5.26. If used alone, the nice command adds a value of 10 to the default nice value of 20, making the nice value 30. The higher the nice value, the "nicer" the process is. If a user issues a command preceded by "nice -15," 15 is added to the default nice value of 20, making the nice value 35. The highest value that can be given with the nice command is "nice -20," for a nice value of 40. The superuser (root) can use negative values for nice, such as "nice --15," which subtracts 15 from the default nice value of 20, making the nice value 5.

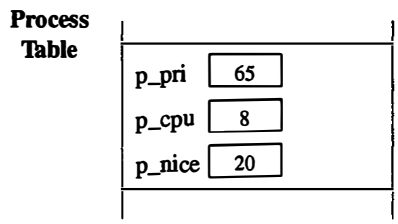

Figure 5.25 Process scheduling fields.

```
$ bigprog
      nice=20 (default)

$ nice bigprog
      nice=30

$ nice -15 bigprog
      nice=35

# nice --15 bigprog
      nice=5
```

Figure 5.26 The nice command.

> **Author's Note:** I have sometimes heard people mistakenly say that the nice command allows a user to set the priority of his or her process. This is not true. The nice command only sets the nice value, which is one factor in calculating a process's priority. The nice command is like allowing someone else, with only one or two pages to reproduce at a copy machine, to go ahead of you when you are about to reproduce a large volume of pages.

The nice command is only used to set the nice value of a process at the time of execution. It cannot change the nice value of an already running process. AIX 3.2, however, supports the renice command for changing the nice value of an already executing process. See the manual pages for nice and renice for more information. The Korn shell automatically sets the nice value of background processes at 24.

5.5 A Process Scheduling Example

This section describes, step by step, how processes are scheduled and dispatched. It uses three ready-to-run processes (the "wait" process at run queue 127 is ignored for this example). The processes and their scheduling attributes are shown in Fig. 5.27. The dispatcher selects process B to run, since it has the most favored p_pri value. Process B executes for one clock tick. The system is interrupted after each clock tick by the system clock interrupt. In addition to updating all system timers, the clock tick interrupt handler charges the current process for the clock tick of CPU time it just used. This is done by incrementing the value of p_cpu for process B. The clock tick interrupt handler then recalculates the priority value of the current running process by using the following algorithm:

$$p\_pri = PUSER + p\_nice + (p\_cpu >> 1)$$

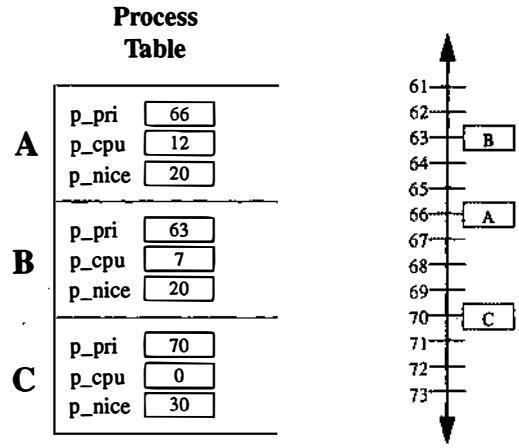

Figure 5.27 A process scheduling example.

The algorithm starts with a symbolic constant called PUSER, which is set to a value of 40 in the /usr/include/sys/pri.h header file. PUSER keeps non-real-time processes from ever having a priority value more favored than 40. The nice value, discussed in the previous section of this chapter, is added to the PUSER constant. (Actually, the PUSER constant is added to the nice value to establish the value of p_nice, but they are shown as two separate values in the algorithm above to simplify the example.) The next portion of the algorithm, (p_cpu >> 1), right-shifts the bits of the current CPU clock tick count by 1 bit. This operation is the same as "divide by 2." Therefore, the algorithm is simply dividing the current CPU clock tick count by 2 and adding the result to PUSER and p_nice. In the case of process B from the example, the algorithm would look like this:

$$p\_pri = 40 + 20 + (8/2)$$

The priority calculates to 64. This means that process B must be requeued from run queue 63 to run queue 64, as shown in Fig. 5.28.

The clock tick interrupt handler always calls the dispatcher when it completes. This means that the dispatcher is called at least 100 times per second. In the example, the dispatcher selects process B as the most favored process. The swtch() routine is not called since process B is already running. Process B simply resumes execution. After another clock tick, process B's p_cpu is incremented to 9. The algorithm still results in a priority value of 64 because the division of the odd number is handled as an integer with truncation of the decimal value:

$$p\_pri = 40 + 20 + (9/2)$$

$$p\_pri = 64$$

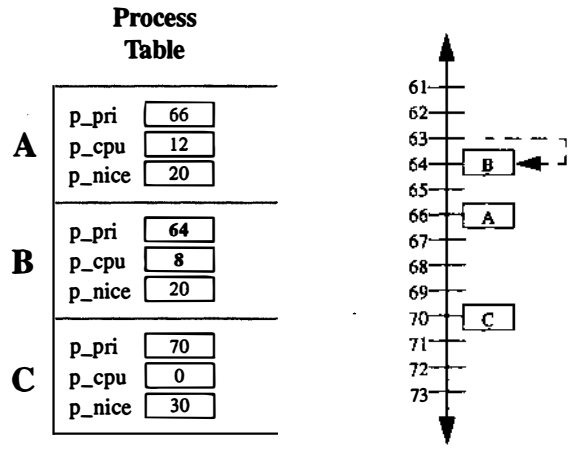

Figure 5.28 Process requeuing.

As each clock tick occurs, process B's priority continues to become less favored. Eventually, when process B's p_cpu value reaches 12, the priority calculation algorithm results in a priority value of 66, which is the same as process A's priority value. Process B is placed at the head of the run queue linked list. The dispatcher selects process B to continue running, as illustrated in Fig. 5.29.

Author's Note: The fact that AIX 3.2 places a proc structure at the head of the linked list of a run queue, instead of at the tail end, is interesting. By doing this, the kernel delays the context switch that would occur between process A and process B for another two clock ticks.

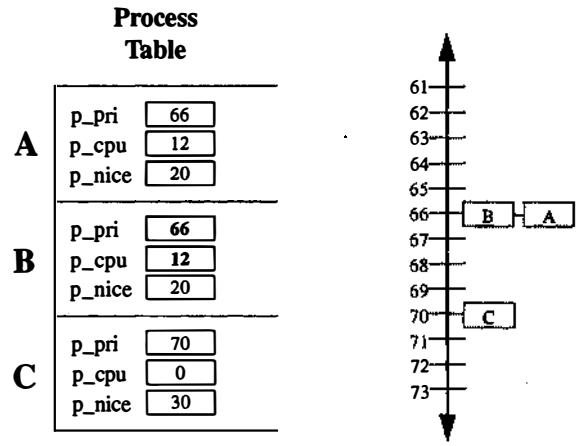

Figure5.29 The process scheduling example continues.

After two more clock ticks, process B's priority becomes 67. It is no longer the most favored process and, for the first time in the example, the dispatcher calls the swtch() routine. The swtch() routine saves the context of process B, then restores the context of process A into the machine state and resumes process A. The details of a context switch are provided in the next section.

The algorithm demonstrated here might lead one to believe that, since running processes become less favored as they run, all ready-to-run processes end up, sooner or later, with the same priority value, and that they simply take turns running in a round robin fashion. This is somewhat true; however, one must remember that processes frequently go to sleep, thus removing themselves from the run queues. Conversely, when a sleeping process awakens, it is placed on the run queue that corresponds to its priority value, so newly made ready-to-run processes can join the work load at any priority level. There is one other aspect of process scheduling that has yet to be described. That is the role of the scheduler process.

The scheduler process, whose process ID number is zero and whose name appears as "swapper" in the output from the ps -e command, spends most of the time sleeping. It wakes once every 100 clock ticks (once each second), performs a series of process scheduling tasks, then goes back to sleep. One of the tasks of the scheduler process is to cut the p_cpu value of all processes in half. As the scheduler runs through the process table, it performs the following algorithm:

```
p->p_cpu = p->p_cpu >>1
```

The variable "p" is a pointer to a proc structure and is used to indicate the current process table entry as the scheduler loops through each slot in the process table. Once again, the right-shift 1-bit operation is the same as "divide by 2." The result is that once each second, all processes have their accumulated CPU time cut in half. This is the decay factor mentioned earlier.

Another operation performed by the scheduler is to recalculate all processes's p_pri values using the same algorithm used by the clock tick interrupt handler:

```
p->p_pri = PUSER + p->p_nice + (p->p_cpu >>1)
```

All processes are then requeued as necessary and the dispatcher is called to select the most favored ready-to-run process for execution. The bottom line of the scheduler algorithm is that running processes become less favored as they use CPU time, since their p_cpu values increase. On the other hand, sleeping processes become more favored since their p_cpu values decay rapidly. This means that the AIX 3.2 process scheduling policies favor I/O bound processes over CPU bound processes.

Time slices

With most UNIX-based operating systems, a process is dispatched with a time slice, which is a guaranteed minimum amount of time that the process may run

before the dispatcher is called to decide whether that process is still the most favored process. In AIX 3.2, the dispatcher can be called at the conclusion of any interrupt handler. This means that the dispatcher can be called with the same frequency as interrupts, which can occur between any two RISC instructions. The likelihood of an interrupt's occurring after each RISC instruction is extremely remote, but when one defines "time slice" as the guaranteed minimum amount of time a process may run before it is evaluated by the dispatcher, in theory that time is one RISC instruction. For practical purposes, AIX 3.2 defines a time slice as a clock tick, or 1/100th second.

> **Author's Note:** You will see shortly that AIX 3.2 includes a tunable parameter for the time slice. This parameter actually controls how often the dispatcher is called by the clock tick interrupt handler. The shortest allowed time slice is 1/100 second.

5.6 The Context Switch

A context switch is the act of preempting the current running process, saving its machine state, restoring the machine state of another process, and resuming the execution of the newly dispatched process. The machine state of a process includes the information that populates the various hardware registers at the time of preemption. This information is saved within the preempted process's user area. In a larger sense, the "context" of a process includes the machine state as well as other information in the kernel pertaining to the process.

The user area of each process (defined as a user structure in the /usr/include/sys/user.h header file) includes an embedded machine state save area, which holds the machine state of a process when it is not running. The structure, called u_save, is an mstsave structure as defined in the /usr/include/sys/mstsave.h header file.

Three of the most interesting fields in the mstsave structure are:

gpr[NGPRS]. An array of unsigned long integers that holds the contents of the computer's general-purpose registers at the time of preemption. The size of the array is NGPRS, which is defined in the /usr/include/sys/m_param.h header file as 32. Recall that header files whose names begin with "m_" are machine-specific, as would be the case when dealing with the number of general-purpose registers.

fpr[NFPRS]. An array of double precision floating point numbers that holds the contents of the floating point registers at the time of preemption. The size of the array is NFPRS, which is also defined as 32 in the /usr/include/sys/m_param.h header file. This is the number of floating point registers in the RISC System/6000 and the PowerPC.

as. An adspace_t data type, which is actually an array of 16 vmhandle_t data types. It holds the segment ID numbers of the 16 segments of the process, per the segment registers in the hardware. The adspace_t data type is defined in the /usr/include/sys/m_types.h header file. It includes an array,

called srval[], with 16 elements, each holding a segment ID number as a vmhandle_t data type.

When a process is running, its user area is mapped into the kernel memory. A kernel variable, called u, holds the current process's user area. It is accessed via &u (the address of u). A context switch involves mapping the incoming process's user area to the address of u. This presents an interesting problem for the operating system. The u_save structure, which holds the segment ID numbers of the incoming process, is itself in the process's data segment. So how does the operating system know the segment ID number of the incoming process's data segment? Recall that each entry in the process table includes a field called p_adspace, which holds the segment ID number of that process's data segment. Since the process table slots are always pinned in memory, and every process's user area is always at the same location within the data segment, the swtch() kernel routine has all the information it needs to locate and map the incoming process's user area to the address of u.

> **Author's Note:** Recall from the discussion of the process's data segment in Sec. 3.7 that a process may not directly access its own user area or any other portion of the kernel section of the data segment. That is because the user area is mapped into kernel memory, which is only available to a process's application code via system calls.

Figure 5.30 illustrates how the mstsave structure (u_save) is used during a context switch. Most of the kernel code involved in performing context switches is written in assembler, since it deals with the system hardware.

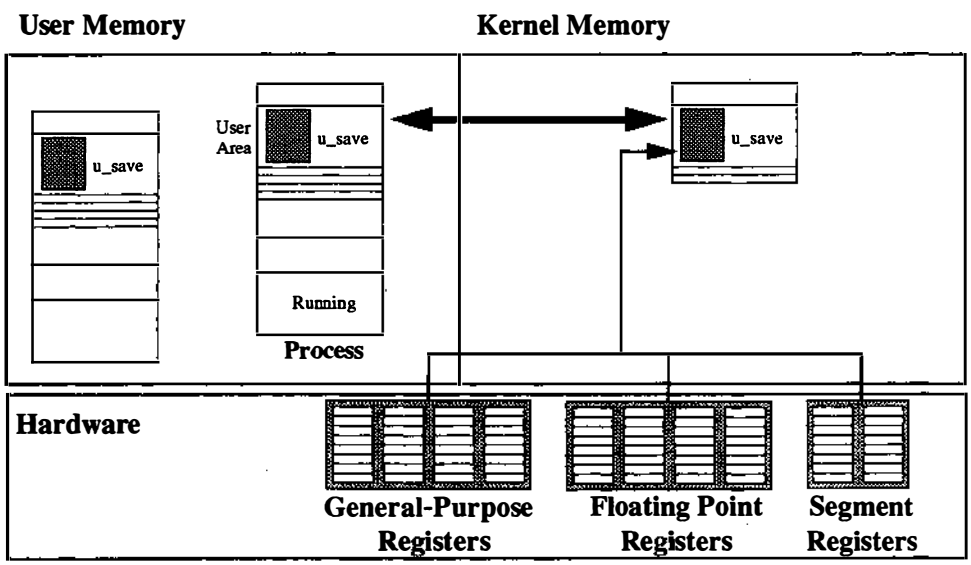

Figure 5.30 The mstsave structure.

5.7 Interrupts

The process management subsystem includes mechanisms for handling interrupts. These interrupts occur when physical devices, such as network controllers or the hardware clock, notify the operating system of some condition or situation. For instance, the SCSI (small computer system interface) controller interrupts the operating system when an I/O request has completed. The interrupts generated by physical devices (other than the CPU) are bus level interrupts. In addition to bus level interrupts, AIX 3.2 supports a series of software and CPU level interrupts.

As mentioned earlier, AIX 3.2 is always executing in one of two different environments. It is said to be in the process environment when executing the code of a process or a system call made by a process, and in interrupt handler environment when executing code that services a hardware or software interrupt. This section describes what happens when the operating system is in the interrupt handling environment.

Interrupt classes and priorities

AIX 3.2 supports 16 classes of bus level (device) interrupts and 34 classes of software and CPU interrupts. They are defined in the /usr/include/sys/m_intr.h header file. In addition to interrupt classes, AIX 3.2 provides 12 priority levels for interrupts. These priority levels are similar to the 128 priority levels for processes; however, interrupt handlers are not time sliced. Figure 5.31 shows how interrupt handler priorities compare to process priorities.

To understand the relationship of interrupt classes and priorities to processes, one must understand what happens when an interrupt occurs. Figure 5.32

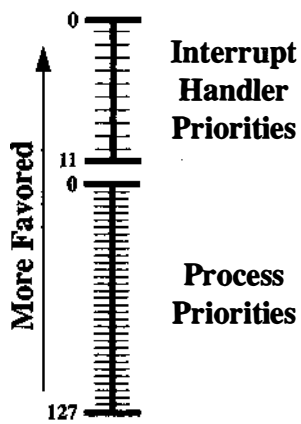

Figure 5.31 Interrupt handler priorities.

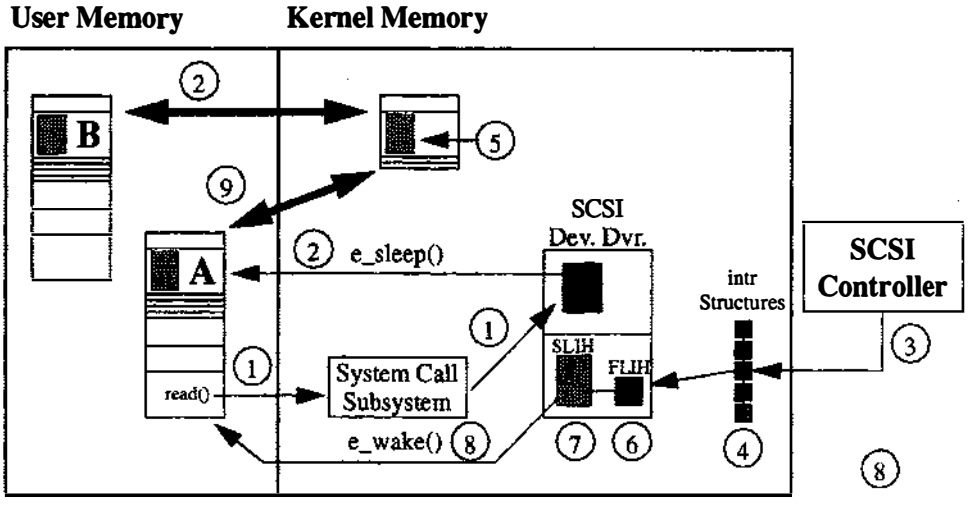

Figure 5.32 An interrupt handler example.

illustrates the components involved in handling an interrupt. It uses a SCSI controller as an example of a device that generates an interrupt. The events of the example are numbered.

1. Process A issues a read() system call on a file that it has opened. The file resides on fixed disk "hdisk0." The portion of the file requested by process A is not in memory and must be fetched from disk. This is done by the logical volume manager (described in Chap. 9), which issues a physical I/O request to the SCSI device driver.
2. The SCSI device driver calls the e_sleep() kernel service, causing process A to sleep while awaiting the completion of the I/O request. This allows the dispatcher to run process B.
3. When the SCSI controller is ready to transfer the requested file pages, it interrupts the CPU. This causes process B to stop executing while the CPU and the operating system handle the interrupt.
4. The operating system includes a linked list of intr structures, defined in the /usr/include/sys/intr.h header file. Each device has an intr structure in this linked list. The kernel contains interrupt processing logic that locates the correct intr structure. The intr structure contains a pointer to a function called an interrupt handler. The interrupt handler is inside of the SCSI device driver.
5. A context switch occurs, saving the machine state of process B so that the interrupt handler can execute. The context switch between a process and an interrupt handler is actually faster than a context switch between processes. This is because interrupt handlers only use segments 0 (the kernel segment) and 14 (the kernel data segment), which are already part of the process con-

text. The contents of the segment registers need not be changed to accommodate an interrupt handler. Another characteristic of interrupt handlers is that they do not use the floating point registers, so the floating point registers allocated to the current process need not be saved during a context switch to an interrupt handler. The context switch between a process and an interrupt handler is said to be an abbreviated context switch.

6. Initially, the SCSI interrupt handler interrupts the CPU at a high priority level to get the system's attention. This is called the first-level interrupt handler, or FLIH. As the FLIH executes, it blocks all interrupts that occur at less favored priorities. The higher (more favored) the priority of an interrupt handler, the shorter its execution time must be, since it blocks all other interrupts at its priority level and below. The FLIH performs whatever tasks must be done at this high level of priority.

7. If the interrupt handler has tasks to perform that are less important than other interrupts that may occur, the first-level interrupt handler calls a second-level interrupt handler (SLIH), which runs at a less favored priority. This is called "off-level scheduling" and is done via the i_sched() kernel service. Each FLIH priority level has a corresponding SLIH priority, as shown in Fig. 5.33. Figure 5.33 also shows how interrupt handler path length (time of execution) should be matched to the interrupt handler priorities. There is nothing within AIX 3.2 that enforces any of these concepts. Rather, it is up to the system program to adhere to these guidelines when writing device driver interrupt handlers.

8. The SLIH in this example uses the e_wake() kernel service to wake the sleeping process A. This causes process A to be placed back on the run queue. Upon completion of the SLIH, the dispatcher is called, since a process has been made ready-to-run.

9. The dispatcher performs a context switch from process B back to process A.

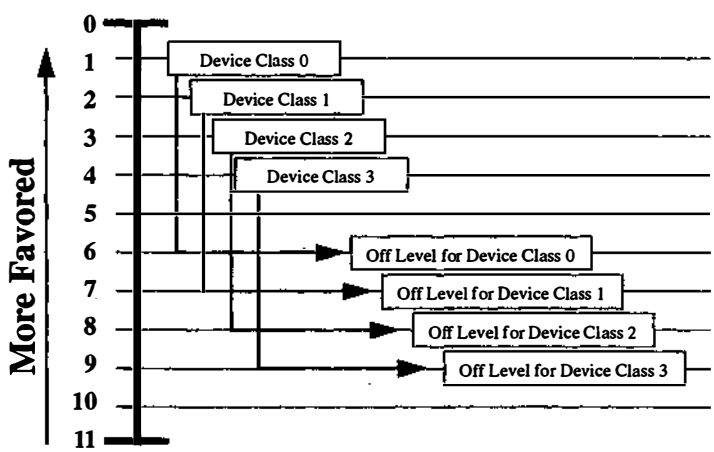

Figure 5.33 Off-level interrupt handling.

Interrupt-to-interrupt context switches

As mentioned earlier, AIX 3.2 supports 12 priorities of interrupts. As an interrupt handler runs, it blocks all other interrupts at its priority level, as well as interrupts of a less favored priority. An interrupt with a more favored priority than an executing interrupt handler will preempt the executing interrupt handler. This means that a context switch takes place between the interrupt handler of lesser priority and the interrupt handler of greater priority.

When a context switch takes place between interrupt handlers, the preempted handler's context is saved in an mstsave structure, as defined in the /usr/include/sys/mstsave.h header file. The mstsave structure that saves an interrupt handler's context is just like the u_save structure of a process. The kernel has a chain of 12 mstsave structures, one for each interrupt priority level. The first field in each mstsave structure is a pointer called prev that points back to the previous mstsave structure in the chain. As interrupts are nested, mstsave structures are allocated off of the chain in a fashion similar to a stack (see Chap. 3 for a discussion of how stacks work). A pointer called csa points to the mstsave structure associated with the current interrupt handler. The prev field of the first interrupt handler's mstsave structure points to the u_save structure of the current process.

Figure 5.34*a–f* illustrate how nested interrupts and their handlers use the chain of mstsave structures to save their context. At Time 1 (Fig. 5.34*a*), the current process is executing when an interrupt occurs. The priority of the interrupt, for this example, is 4. The context of the process is saved in the u_save structure and the csa pointer changes to point to the first mstsave structure in the chain. At Time 2 (Fig. 5.34*b*), as the first interrupt handler is executing, another interrupt occurs. This interrupt has a priority of 3, so the first interrupt handler is preempted. The context of the first interrupt handler is saved in its mstsave structure while the csa pointer is changed to point to the next mstsave structure in the chain. The second interrupt handler now executes. At Time 3 (Fig. 5.34*c*), a third interrupt occurs with a priority of 2. This interrupt preempts the second interrupt handler. The context of the second interrupt handler is saved in its mstsave structure and the csa pointer is incremented to point to the next mstsave structure in the chain. Now, the third interrupt handler executes. At Time 4 (Fig. 5.34*d*), the third interrupt handler has completed. The csa pointer is set to point to the mstsave structure of the second interrupt handler. This is done by using the prev pointer to walk back up the chain. The second interrupt handler can now resume its execution. At Time 5 (Fig. 5.34*e*), the second interrupt handler has completed and the context of the first interrupt handler is restored via the csa pointer. The first interrupt handler can now resume. Time 6 (Fig. 5.34*f*) shows the completion of the first interrupt handler. The csa pointer is set to point to the mstsave structure that is u_save (within the process's user area). This allows the process to continue execution. Keep in mind that the dispatcher may be called before allowing the process to resume execution.

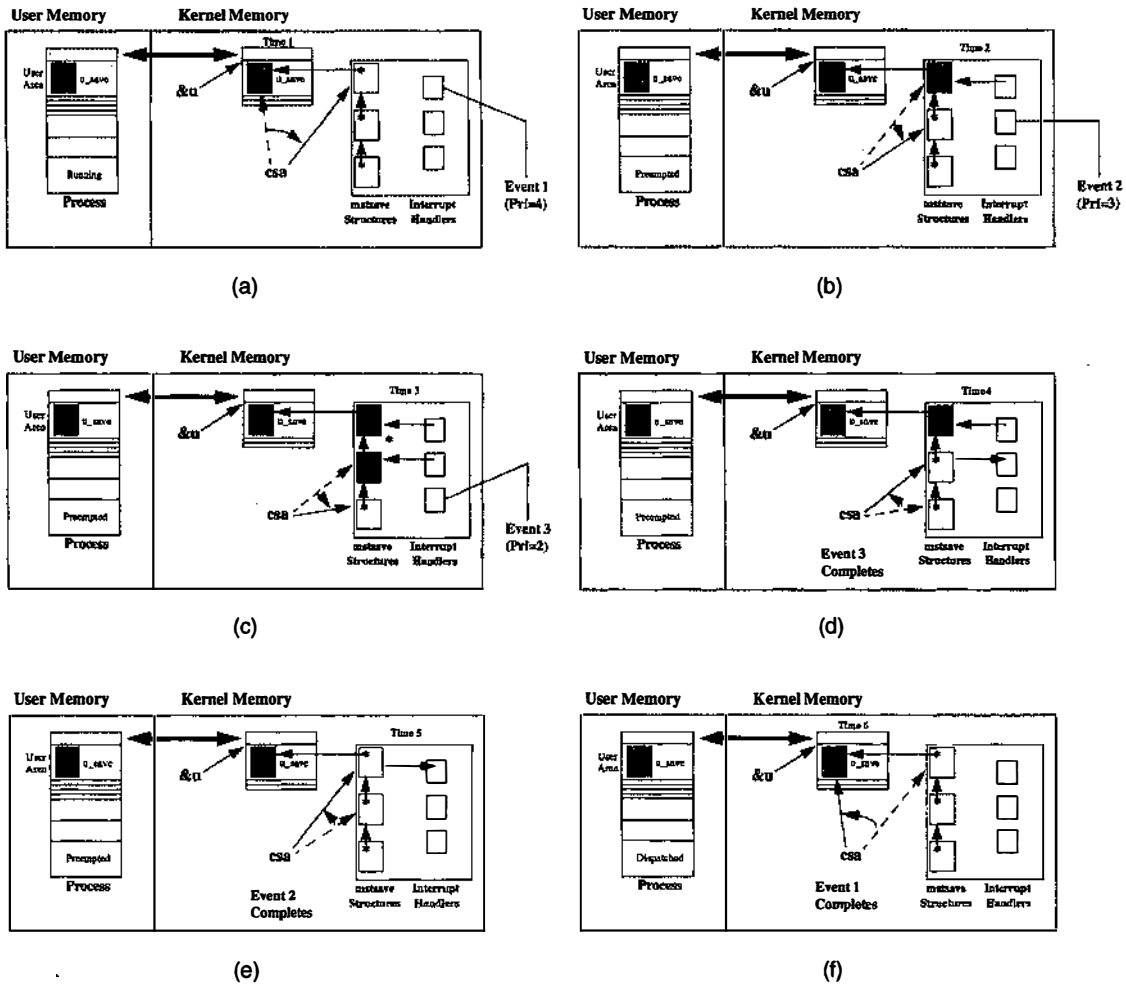

Figure 5.34*a–f* Interrupt-to-interrupt context switches.

Disabling interrupts

Kernel-level routines, such as system calls, running in the process environment, and interrupt handlers, running in the interrupt handler environment, often share common kernel data structures. To prevent contention problems, kernel-level routines can disable interrupts when updating these shared data structures. For instance, a device driver might link I/O requests to a queue when running in the process environment. An interrupt handler associated with the same device driver might also update the queue as requests are serviced. Since interrupts always take priority over processes, the system calls and device driver routines running on behalf of processes must have a way of preventing interrupt handlers from running while the system calls and device

driver routines update the shared data structures. In the same fashion, less favored interrupt handlers must be able to prevent more favored interrupts from occurring while they update kernel data.

Serializing access to shared kernel data structures is accomplished in AIX 3.2 via the i_disable() and i_enable() kernel services. A parameter to the i_disable() kernel service sets the current interrupt priority to a level that prevents interrupts of the same level or lower (less favored) priority from occurring. The i_enable() kernel service resets the interrupt priority level. The code between the i_disable() call and the i_enable() call is known as the "critical section" of code. A symbolic constant called INTMAX is used to disable all interrupts. A symbolic constant called INTBASE is used to enable all interrupts. Figure 5.35 illustrates how interrupts are disabled by a system call. See the manual pages for i_disable() and i_enable(), as well as the InfoExplorer documentation on disabling interrupts for more information.

AIX 3.2 system calls, kernel services, and interrupt handlers are written to implement the shortest path lengths possible for critical sections of code. In other words, interrupts are disabled only when absolutely necessary to assure data integrity, and for the shortest amount of time possible. Systems programmers who write device drivers must understand the use of i_disable() and i_enable() and appreciate the ramifications of their utilization. The IBM manual called "Kernel Extensions and Device Support Programming Concepts" within InfoExplorer or available in hard copy form provides guidelines for disabling interrupts.

```
/*  Code from some kernel extension  */
...
/*  Start critical section of code  */
i_disable(INTMAX); /*  Disable all interrupts  */
...   /*  Critical section here  */
i_enable(INTMAX); /*  Enable all interrupts  */
...
```

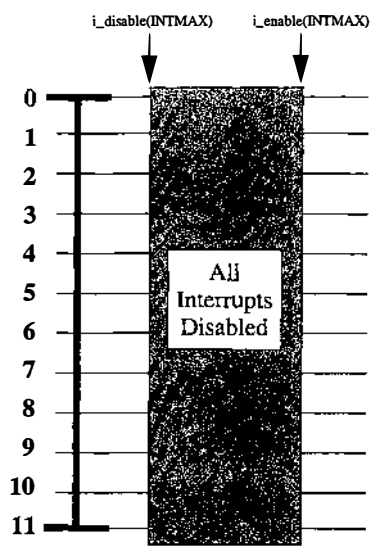

Figure 5.35 Disabling interrupts.

Locking kernel data structures

Access to kernel data structures by processes can be serialized by locks. The lockl() and unlockl() kernel services allow kernel routines, such as system calls running on behalf of processes, to lock and unlock kernel data structures. These kernel services provide advisory locks, which means that the kernel does not enforce these locks. Rather, kernel services and system calls must request a lock for a kernel resource. The lockl() kernel service includes a flag parameter to indicate the desired action to take if the lock is not available. The flag allows the process to sleep until the lock becomes available, or immediately return with a failed condition.

To avoid deadlock conditions, kernel locks are obtained in a hierarchical fashion that progresses from the coarsest lock level to the finest lock level. Another feature of kernel locks is the artificial raising of the priority of a process that owns a kernel lock to the level of a process that requests the same lock. This concept is illustrated in Fig. 5.36. Process A, which has a priority value of 68, issues a system call that acquires a lock on the process table. Soon afterward, process B, which has a priority value of 60 (which is more favored than process A) and has been sleeping while waiting for some event to occur, has awakened, since the event has occurred. The dispatcher causes process A to be preempted (even while holding a kernel lock) and allows process B to execute. Process B then issues a system call that attempts to lock the process table. The lock request fails since the lock is still owned by process A. The system artificially raises process A's priority to 60 so that process A can run. As soon as process A releases the lock, its priority is set back to 68. This allows process B to run and acquire the process table lock.

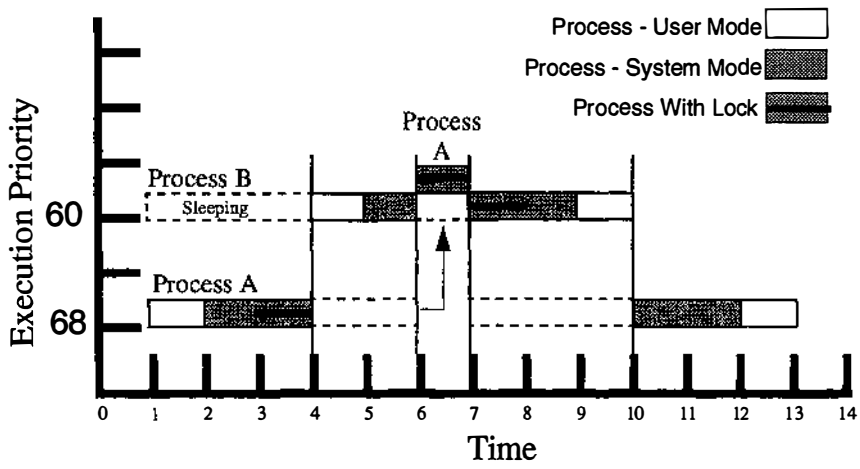

Figure 5.36 Kernel locks and process priorities.

5.8 The Scheduler and the Suspension Queue

AIX 3.2 adds a new scheduling feature that helps reduce the effects of thrashing due to overcommitted memory. (See Chap. 4 for an explanation of thrashing.) It is a memory load control algorithm which is executed by the scheduler process. Recall that the scheduler process (PID 0, a.k.a. "swapper") runs once each second. In addition to its traditional job of calculating process priorities and requeuing ready-to-run processes, it checks to see if the system is thrashing. If it detects a thrashing condition, it suspends certain processes and prevents new processes from entering the system. The pages associated with a suspended process age very quickly and are removed from physical memory by the page stealer. This reduces the load on physical memory and allows the system to recover from the thrashing condition. As soon as the system is no longer thrashing, the suspended processes are allowed to continue execution. This section provides details on the memory load control algorithm.

The memory load control algorithm

Figure 5.37 illustrates the logic flow of the memory load control algorithm. The flowchart begins once each second when the scheduler wakes.

1. The scheduler determines if the system is thrashing. This is accomplished by examining two fields already maintained by the virtual memory manager (VMM): the number of page steals within the last second and the number of pageouts resulting from page steals within the last second. By default, when the scheduler detects that the number of page steals requiring pageouts is greater than one-sixth of all pages stolen within the last second, the system is thrashing. The formula is

$$(\text{pageouts}/\text{pages stolen}) > (1/6)$$

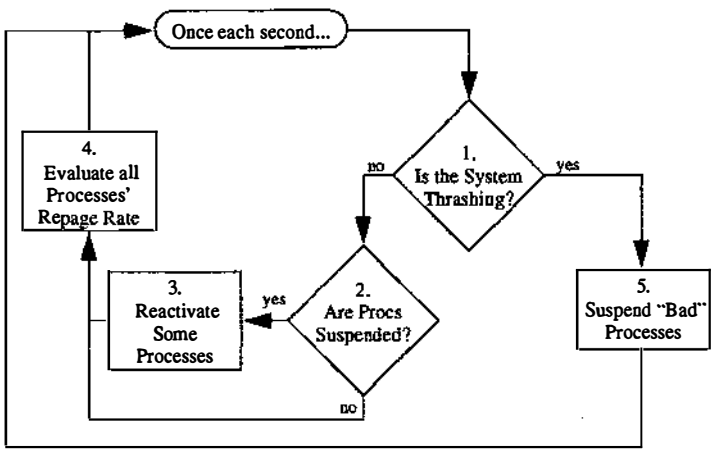

Figure 5.37 The memory load control algorithm.

2. If the system is not thrashing and has not been thrashing for a designated period of time (1 second, by default), and there are suspended processes, the scheduler allows a certain number of processes to leave the suspension queue and resume execution. Suspended processes are selected for reactivation first by their priority value and second according to which processes have been suspended the longest. In other words, when two or more suspended processes have the same priority value, the scheduler reactivates them in a FIFO fashion, according to the suspension queue.

> **Author's Note:** The 1-second delay between the time the system stops thrashing and the reactivation of the suspended processes allows the system a safety period to recover from the thrashing state. This delay time is tunable. One second is the default value for this parameter.

The number of processes reactivated by the scheduler is based on an internal algorithm. It is designed to reactivate enough suspended processes to equal about a 20 percent increase in active processes per second. This scheme allows the workload to build slowly after recovering from a thrashing condition.

3. A reactivated process is given a grace period (2 seconds, by default) during which it is not qualified for suspension. The reason for this is explained shortly.

4. Another operation performed by the scheduler when the system is not thrashing is to determine which processes should be suspended if the system starts thrashing. This is done by examining the repage rate of each eligible process. Recall from Chap. 4 that a repage is a pagein of a page that had previously been in physical memory but was then paged out. The VMM tracks the repage rate for every process. By default, if more than 25 percent (one-fourth) of all pageins for a process within the last second were repages, the scheduler marks the process as "bad," or qualified for suspension. Processes that are immune to suspension include sleeping processes, fixed-priority processes with priority values less than 60 (see the next section of this chapter for a discussion of fixed-priority processes), processes that have just been reactivated and are still within their grace periods, processes that have pinned memory, and kernel processes. Keep in mind that, at this point, the scheduler is only selecting processes that *should* be suspended when action is taken to correct a thrashing condition. No processes are actually being suspended.

5. When the scheduler determines that the system is thrashing (based on the formula presented in step 1), it takes action by suspending all processes marked as "bad" (in step 4). It also prevents new processes from executing by having them placed on the suspension queue as well. This causes the pages associated with these processes to quickly age. The pages are stolen from physical memory, allowing the system to recover from the thrashing condition.

The system maintains another tunable parameter which specifies the minimum number of processes to leave active. By default, the value of this para-

meter is two. It can be adjusted to control the aggressiveness of the memory load control algorithm at various physical memory sizes and workloads.

> **Author's Note:** The term "bad" process is interesting because, in most cases, processes that do a lot of repaging can cause a system to thrash. The high repage rate may be the result of poor locality of reference. It cannot always be assumed, however, that a high repage rate occurs only within bad processes. Another process might pin a large number of pages to physical memory. Recall that processes that have pinned memory are not eligible for suspension and are therefore never "bad" processes. They can be the direct cause of overcommitted memory, however, thus causing other processes to have higher repage rates than they would normally have. The bottom line is that sometimes a "bad" process isn't really bad. It's just a victim of its environment!

Suspension flags and fields

The kernel maintains the following fields in the process table (proc structure as defined in /usr/include/sys/proc.h) to aid the memory load control algorithm:

p_repage. An integer that holds the repage count for the process.

p_sched_next. A pointer to the next proc structure on the suspension queue.

p_sched_back. A pointer to the previous proc structure on the suspension queue. The suspension queue is a doubly linked list within the process table.

p_sched_flags. A character field that holds the current suspension status flag. The possible values, as defined in the /usr/include/sys/proc.h header file, include:

SSWAPNONE. Defined as 0x00, it indicates a normal (not "bad") process.

SGETOUT. Defined as 0x01, it indicates a process that the scheduler has marked as "bad." Its repage rate is higher than the defined system parameter (default 25 percent of all pageins). The process is not suspended but will be if the scheduler detects a thrashing condition.

SSWAPPED. Defined as 0x02, it indicates that the scheduler has suspended the process. The process is said to be on the suspension queue.

SJUSTBACKIN. Defined as 0x03, it indicates that the process has just been reactivated and is still operating within its grace period. As mentioned earlier, the process is immune to further suspension as long as it is in this grace period, which has a default time value of 2 seconds. The grace period is necessary since, while a process is suspended, most, if not all, of its pages are aged and paged out. A high repage rate is only natural in order to get the process running again.

Tuning the memory load control algorithm—The schedtune utility

AIX 3.2 includes a utility, /usr/lpp/bos/samples/schedtune, which provides the ability to adjust some of the parameters discussed in this section. For instance, the parameter that defines a thrashing condition as more than one-sixth of all page steals requiring pageouts can be tuned. When issued with no options, the schedtune command displays the current settings, as shown in Fig. 5.38.

The SYS value, altered via the -h option, is used to determine when a system is thrashing. The default value of six provides the 1/6 operand in the equation given earlier in this chapter. Using the -h option, the formula can be displayed as

(pageouts/pages stolen)>(1/h)

When the equation above is true, the scheduler assumes that the system is thrashing and takes appropriate action.

The PROC value, altered via the -p option, is used by the scheduler to determine which processes are candidates for suspension. The default value of four provides the one-fourth (25 percent) repage rate that identifies "bad" processes. Using the -p option, the formula is expressed as

(number of repages in last second/number of page faults in last second)>(1/p)

When the equation above is true, the scheduler sets the p_sched_flags field of the process to SGETOUT.

The MULTI value, altered via the -m option, indicates the minimum number of processes to leave active when suspending processes. This value can be increased for a system with a large amount of physical memory. See the AIX 3.2 Performance Tuning Guide (SC23-2365-03) for more details on tuning this parameter.

The WAIT value, altered via the -w option, indicates the number of seconds that the scheduler waits before reactivating processes once the thrashing condition has passed. Setting this number higher than 1 second may cause poor response time and excessive CPU idle time after a thrashing condition.

```
# /usr/lpp/bos/samples/schedtune

                THRASH                  SUSP                    FORK    SCHED
-h              -p          -m          -w          -e          -f      -t
SYS             PROC        MULTI       WAIT        GRACE       TICKS   TIME-SLICE
6               4           2           1           2           10      0
```

Figure 5.38 The schedtune command.

The GRACE value, altered via the -e option, indicates the grace period of unlimited repages given to a suspended process when it is reactivated. The default value is 2 seconds.

The TICKS value, altered via the -f option, indicates the time interval, in clock ticks (1/100 second), between retries of the fork() system call to create a new process when the system is low on paging space. Recall from Chap. 4 that the VMM prevents new processes from coming to life when the number of available paging space slots is below a low water mark. A fork() system call will retry at the specified interval for five iterations before failing.

The TIME_SLICE value, altered via the -t option, allows the system administrator to change the frequency with which the dispatcher is called. By default (0), the dispatcher is called after each clock tick interrupt, as described earlier in this chapter. For each value of 1 added to this parameter, the time slice increases by 10 milliseconds (one clock tick).

> **Author's Note:** So you can tune the time slice of AIX 3.2! Well, sort of. The -t option of the schedtune command adjusts the parameter that controls the minimum frequency with which the dispatcher is called, or, in other words, the maximum time a process is allowed to run before its priority is reevaluated. Keep in mind that the dispatcher is also called whenever any interrupt awakens a sleeping process. Since the true definition of time slice is the guaranteed minimum amount of time that a process will run before the dispatcher is called, the -t option of the schedtune command does not actually change the time slice. Adjusting this parameter might improve the performance of a system that runs very long, CPU-intensive applications.

One final note on the memory load control algorithm: It is important to remember that the goal of suspending processes when the system is thrashing is to allow the system to smooth out and recover from a thrashing state. Without process suspension, a small thrashing condition can grow steadily larger until the system cannot recover. The memory load control algorithm is not a permanent solution to the problem of overcommitted memory. That condition can only be solved by adding more physical memory to the system or reducing the memory requirements.

5.9 AIX 3.2 Real-Time Processes

Real-time programs, as described in Chap. 2, guarantee a maximum response time (context latency) to specific events. Many of the features of the AIX 3.2 kernel were designed to support a "near-real-time" environment. For instance, the preemptable kernel, as described in Chap. 2, allows a real-time process to preempt an ordinary process that is in the middle of executing a system call. The improved method by which the dispatcher locates the most favored ready-to-run process, discussed earlier in this chapter, also contributes to the real-time capabilities of AIX 3.2. But one feature stands out above all others when

it comes to implementing real time in AIX 3.2: the ability to fix a process's priority to a specific value.

Fixed-priority processes and the setpri() system call

AIX 3.2 provides the setpri() system call to allow a program to fix its priority. Recall that AIX 3.2 includes 128 priority values and run queues (0–127), as illustrated in Fig. 5.39. The PUSER value used in calculating a process's priority value is a constant 40. The only way for a process to have a priority more favored than PUSER (less than 40) is via the setpri() system call. The setpri() system call takes two parameters: a pid_t data type (or integer) that indicates the ID number of the process whose priority is to be fixed, and an integer value that indicates the desired priority level. If the process ID number parameter is zero, the setpri() system call fixes the priority of the calling process. Figure 5.40 provides a code example of how the setpri() system call is used to fix a real-time process to a highly favored run queue.

In the example, the process calls setpri() to fix its own priority to 38. This makes it more favored than any ordinary user process. Note that after setting its priority, the process sleeps, waiting for the event for which it must quickly respond. Well-behaved real-time processes spend most of their time sleeping at very favored priority levels. This prevents them from interfering with other processes.

Fixed-priority processes behave differently from ordinary processes. They are immune to the process scheduling algorithms discussed in Sec. 5.5. In fact, the equations presented in that section were not complete. They are wrapped

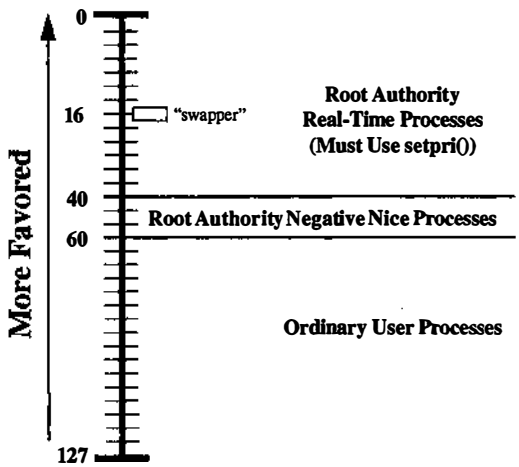

Figure 5.39 Fixed-priority processes.

```
...
main()
{
    ...
    setpri(0,38);
    ... /* sleeps, waiting for real-time event */
    ...
}
```

Figure 5.40 The setpri() system call.

within a test equation that verifies that the process being evaluated is not a fixed-priority process. The p_flag field of the proc structure (process table entry) includes a definition of SFIXPRI (0x00000100) that indicates a process that has called setpri(). The actual calculation performed by the clock tick interrupt handler and scheduler, when determining the priority of a process, is

```
if(p->p_flag !& SFIXPRI)

p->p_pri = p->p_nice+(p->p_cpu>>1)
```

Recall that p is a pointer to the proc structure being evaluated, and that p_nice has already had the PUSER constant of 40 added to the process's nice value (0–40). Note that this algorithm only takes place if the process's p_flag value does not include the value of SFIXPRI.

> **Author's Note:** The bitwise AND (&) operator is used to test for the existence of a value within another value. In this case, it is used to test for the existence of the value 0x00000100 in the process's p_flag value. The ! negates the logics of the test. Therefore, the test is TRUE if the SFIXPRI flag is not set.

The setpri() system call fixes a process to a designated priority. Subsequent calls to setpri() can be used to change the priority of the process, but there is no mechanism by which a process can revert back to normal scheduling characteristics.

> **Author's Note:** I tell my students that a process is like a cat: once you fix it you can't unfix it!

The setpri() system call can be dangerous if not used properly. Note in Fig. 5.39 that the scheduler process, known as "swapper" (PID 0), is fixed at priority 16. Many other important system processes are fixed at various priority levels. The /usr/include/sys/pri.h header file includes definitions of some of these processes and their priority levels. It also includes a warning that any process subject to scheduling must have a priority level less favored than the scheduler. Figure

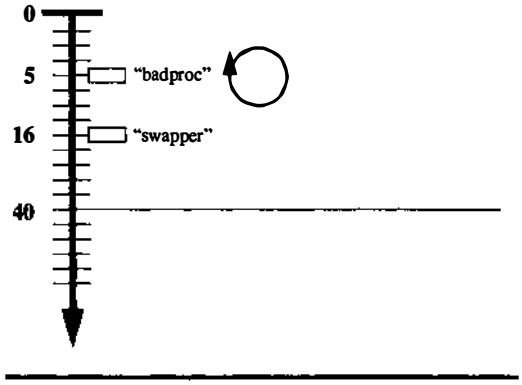

Figure 5.41 The wrong way to use setpri().

5.41 shows how a program can abuse the setpri() system call by setting its own priority to five, then entering an endless loop. The effect appears to lock up the system. Actually, the system is still running, but only the process that has set its priority to level 5. Even the scheduler cannot run. For this reason, only a process with root authority can successfully call setpri().

Chapter

6

The Journaled File System

6.1 An Overview of File Systems

The term "file system" has two distinct meanings for UNIX-based systems. The "global file system" refers to the file tree as viewed by the user. It includes the entire hierarchical arrangement of directories and files, from a logical perspective, regardless of the physical components that comprise the tree. In reality, the global file system is made up of one or more physical file systems, which reside on separate disk partitions or other storage media. These physical file systems are connected to form the global file system.

The global file system

Figure 6.1 illustrates the AIX 3.2 file tree and a simplified representation of the file systems on disk.

> **Author's Note:** The details of the files found in each directory are appropriate for a system administration discussion and are not provided here. While Fig. 6.1 shows each file system as a contiguous disk partition, the AIX logical volume manager allows file systems to be fragmented and spread across one or more physical disk drives, as described in Chap. 2.

Each file system has its own root directory, which is mounted onto a stub directory in the file system above. The stub directory is called the mount point. Each mount point directory is shown as boxed in Fig. 6.1.

AIX local disk file systems must reside within disk partitions called "logical volumes." Each logical volume is considered a device and thus includes a device file abstraction in the /dev directory. Table 6.1 lists the AIX 3.2 file systems, describes their general use, and indicates the device names for their logical volumes.

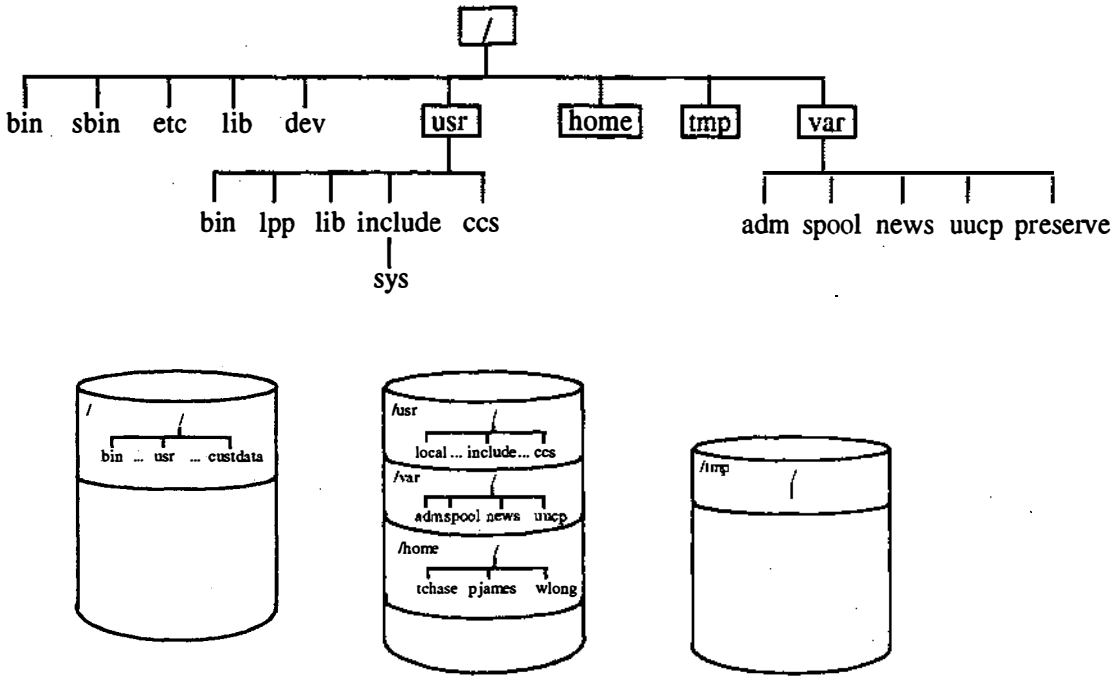

Figure 6.1 The AIX 3.2 file tree and disk file systems.

TABLE 6.1 AIX 3.2 File Systems

FS Name	Device Name	Description
/	/dev/hd4	Holds configuration and boot files specific to the system
/home	/dev/hd1	Holds users' HOME directory trees
/usr	/dev/hd2	Holds Licensed Program Products (LPPs)
/tmp	/dev/hd3	Holds temporary files
/var	/dev/hd9var	Holds transient system files such as mail and news

The physical file system

AIX implements local disk file systems via the journaled file system, or JFS. The JFS is similar to many other UNIX file systems, providing an interface that is compatible with those file systems. The main difference is that the JFS logs operations, such as the creation and removal of files and directories, that change file system control structures. The logging provides a means of reconstructing critical pointers in the event of a system crash. The result is a much more robust file system.

The virtual file system

The JFS is one type of virtual file system. AIX supports three virtual file system types: JFS for local disk files, network file system (NFS) for remote file systems, and CD-ROM file systems. The virtual file system concept, which is detailed in Chap. 8, allows different types of file systems to be accessed from applications without the applications needing to know the details of operation for each file system type.

Other virtual file system types can be added to the AIX kernel. For instance, support for the Andrew file system (AFS), developed at Carnegie-Mellon University, which provides remote file access in quite a different fashion from NFS, can be added to AIX as a kernel extension.

> **Author's Note:** IBM offers the distributed file system (DFS) as part of the distributed computing environment (DCE) from the Open Software Foundation (OSF). DFS has its roots in AFS. Therefore, it can be said that IBM offers AFS for AIX as the DFS of DCE from the OSF!

6.2 File System Components

At the surface, file systems consist of three logical components: files, inodes, and directories.

Files

In AIX, as with other UNIX-based systems, ordinary files have no structure, as far as the operating system is concerned. An ordinary file, which might contain data or executable code, is nothing more than a string of bytes. There is no such thing as a record size or format. Any format expected by an application must be superimposed by the application on the raw data of the file.

> **Author's Note:** As seen in Chap. 4, there is one instance when a kernel component expects an ordinary file to have a format. The kernel loader expects object files and compiled executables to be in XCOFF format. This is necessary in order to properly allocate memory for text, initialized data, and noninitialized data, as well as resolve references to symbols found in dynamically bound objects.

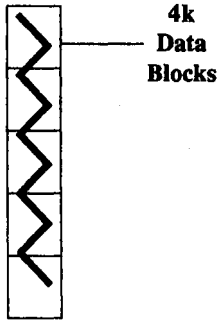

Figure 6.2 AIX 3.2 files.

The ordinary file is one type of AIX file. Other types of files include directories, symbolic links, block device special files, character device special files, named pipes (FIFOs), and sockets. Details of the structures and uses of each of these file types are provided in appropriate chapters of this book. Directories are described shortly.

The raw data of an ordinary file are stored in data blocks, as seen in Fig. 6.2. For local disk files (JFS), the data block size is 4096 bytes (4 kb). This means that the smallest disk allocation size for an ordinary file that is not empty is 4 kb. Also, the last data block of a file is always 4 kb, regardless of how much of the block is actually used. While this tends to waste disk space, especially in file systems that hold many small files (the JFS in AIX 3.2 does not support data block fragmentation), the benefit comes from the fact that file I/O is performed in 4-kb chunks via pageins and pageouts (see Chap. 4).

Inodes

Since the data blocks of an ordinary file contain nothing but raw data, the attributes of a file, such as the UID of the owner of the file, the GID of the group associated with the file, and the permission bits and file type, must be kept elsewhere. These attributes are stored in the file's inode (short for information node). Each file has an inode. Figure 6.3 illustrates the JFS inode.

The AIX header file /usr/include/jfs/ino.h contains the definition of the disk inode as a struct dinode. This file is one of the more difficult files to read for two reasons. The comments are found above the member definitions, instead of to the right of each definition, and, since there are many different types of files, the latter portion of the dinode structure is a complex set of nested unions and structures. An experienced C programmer should have little trouble untangling this convoluted mess, but the novice may need to spend more time on it.

Important members of the dinode structure include:

di_mode. A mode_t data type that includes the file type and permission bits

di_nlink. The number of hard links for the inode (described shortly)

di_uid and di_gid. The user ID of the owner and the group ID of the associated group

di_size. An off_t data type that holds the logical file size in bytes

di_nblocks. The number of data blocks actually used by the file

di_atime. The timestamp of the file's last access

di_mtime. The timestamp of the file's last modification

di_ctime. The timestamp of the inode's last modification

di_acl. A pointer to the file's access control list, if any (described shortly)

> **Author's Note:** The "ctime" timestamp is wrongly commented in some UNIX-based system header files as holding the file's creation time. Rather, it holds the date and time of the last change made to the inode itself, such as changed permissions via the chmod command.

The remainder of the dinode structure is specific to the file type. For ordinary files, this portion of the inode provides an array of logical block numbers for the data blocks of the file. A detailed explanation of how the data block addresses are maintained is found in Sec. 6.5.

The size of an AIX 3.2 inode is 128 bytes. Each file system has its own set of inodes. A file system's inodes are maintained in an array allocated to a set of data blocks within the file system. This is known as an inode table. Each inode has a number which corresponds to the inode's index within the table.

Directories

It's important to note that nowhere in the discussion of the file or the inode has the file name appeared. That's because file names are not found within the files (or at least as far as the operating system is aware), nor are they found within

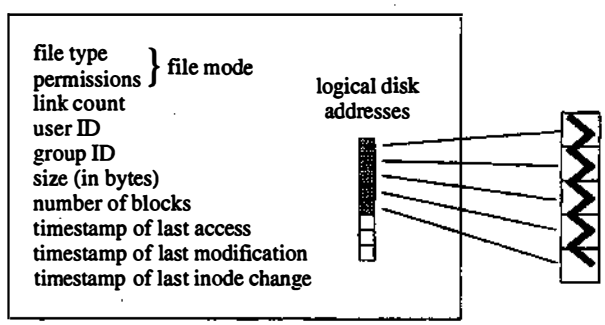

Figure 6.3 The JFS disk inode.

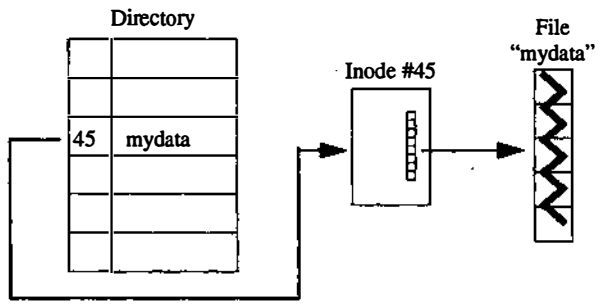

Figure 6.4 Directories and inodes.

the inode. Actually, file names exist only to provide the user with a symbolic reference to a file. The system performs all file operations based on inode numbers. File names are found only within directories.

A directory is a type of file that does have a structure recognized by the operating system. Each directory is made up of slots that hold file names and their corresponding inode numbers. In other words, a directory serves as a lookup table for converting file names to inode numbers. Figure 6.4 provides a simplified view of the relationship of a directory to an inode to a file.

Figure 6.5 shows how a complete pathname to a file is represented through directories and inodes. Since a directory is a type of file, it will have an inode which is referenced in that directory's "parent" directory. Thus a linked list, of sorts, is formed from the root directory to the file. Note that each directory contains a file named "." which represents that directory. The "." (dot) name is useful when issuing commands such as cp /etc/motd ./mymotd. Each directory also contains a file named ".." (dot-dot) which represents the directory's parent directory. This is used in commands such as cd .. to change the current directory to the parent directory. These file names are shown in Fig. 6.5.

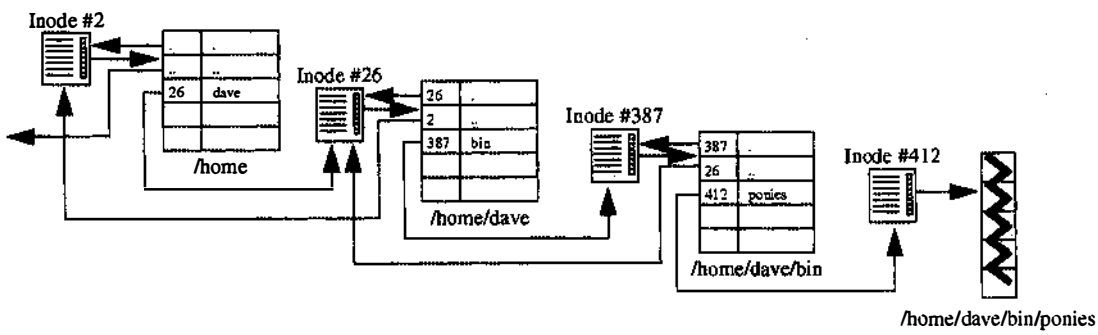

Figure 6.5 A pathname example.

The old style directory, as was used in AIX Version 2 for the RT, had a simple directory structure that consisted of two fields, a short integer for the inode number and a character array of 14 characters for the file name. The file name limit, therefore, was 14 characters. AIX 3.2 employs a directory structure similar to that of the BSD version of UNIX, which provides for variable-length file names.

The AIX 3.2 directory structure is defined as a struct direct in the file /usr/include/jfs/dir.h. The structure has four fields:

d_ino. An unsigned long for the inode number

d_reclen. An unsigned short for the length of the entire directory entry

d_namlen. An unsigned short for the length of the file name

d_name[_D_NAME_MAX + 1]. A character array for the file name

_D_NAME_MAX is defined in the dir.h file as 255 and is the size of the longest allowed file name. The MAXNAMELEN symbolic constant is also defined within the same file for BSD compatibility. Since AIX file names are terminated with a null character (\0), the d_name[] array is 256 characters. The d_namlen field contains the length of the file name. The file name is always padded up to the next 4-byte boundary. Figure 6.6 illustrates an example of a directory.

AIX allocates directory space in 512-byte blocks (see DIRBLKSIZ in dir.h). A directory consists of one or more of these blocks. Directory entries claim all of the bytes in a block. This is accomplished by having the last directory entry claim all of the remaining free bytes in the block within that entry's d_reclen field. When an entry is deleted from a directory, the free bytes from the deleted entry are claimed by the previous entry's d_reclen field. Figure 6.7 represents a before and after example. The dir.h header file includes a macro called DIRSIZ that indicates the minimum record length required to hold a directory entry. The dir.h header file also provides information on many library routines for manipulating directories, including opendir(), readdir(), closedir(), and rewinddir().

Links

AIX 3.2 supports hard links and symbolic links. A hard link exists between a directory entry and an inode within the same file system. The ln command

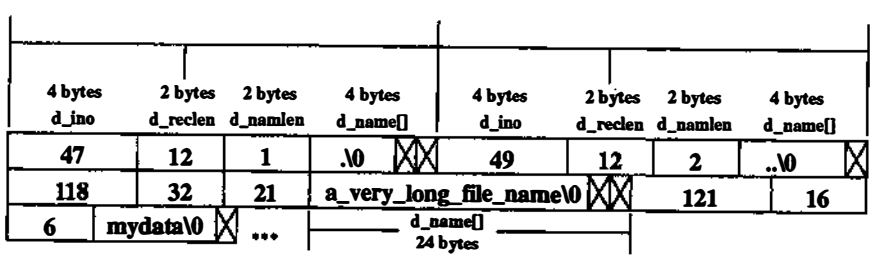

Figure 6.6 A directory example.

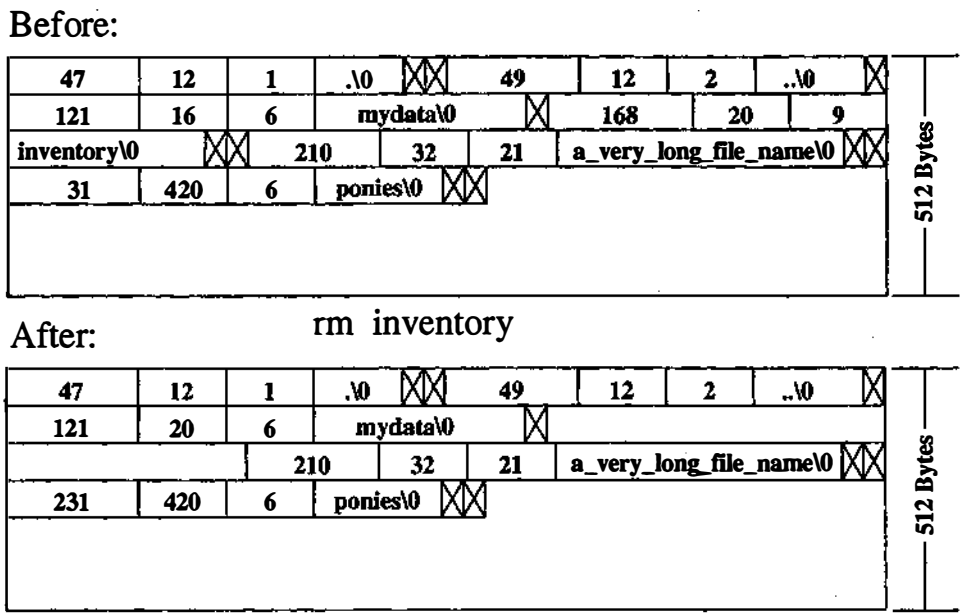

Figure 6.7 Directory space reallocation.

allows users to create additional links for an inode by establishing additional directory entries which reference that inode. For example, two users can create links to a data file or executable file such that the file appears to exist in each of their home directories. Figure 6.8 illustrates such an example. Hard links also allow a single file to have multiple names within the same directory. For instance, AIX 3.2 links /usr/bin/sh to /usr/bin/ksh to treat the Korn shell as the default shell.

The disk inode structure (dinode) includes a field named di_nlink which indicates the number of hard links to the file represented by the inode. In the example in Fig. 6.8, the inode has a link count of three. When a user removes a file via the rm command, the directory entry for that file is deleted and the di_nlink value is decremented for that inode. When an inode's di_nlink count reaches zero, the file is considered completely removed. The inode is then placed back on the free list of inodes and the file's data blocks are released back to the free list of data blocks.

The di_nlink count shows up in the output from ls -l, as shown in Fig. 6.9. File inode numbers can be determined with the ls -i command. The ncheck command converts inode numbers to file path names. The find command also includes a switch for determining file names for inode numbers. Hard links can only exist between directory entries and inodes within the same file system. Symbolic links were added to UNIX to provide links between directory entries

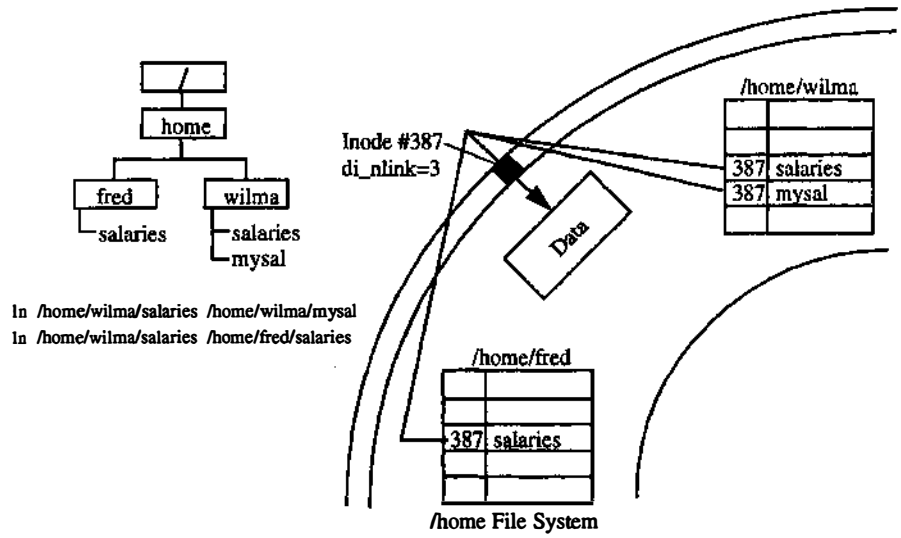

Figure 6.8 A hard link example.

```
# pwd
/home/wilma
# ls -l
...          ...  ...    ...      ...    ...          ...
-rw-r-----   3    wilma  payroll  38755  June 12 08:31  mysal
-rw-r-----   3    wilma  payroll  38755  June 12 08:31  salaries
...          ...  ...    ...      ...    ...          ...
# ls -i
...   ...
387   mysal
387   salaries
# ncheck -i 387 /home
/dev/hd2:
      /home/fred/salaries
      /home/wilma/mysal
      /home/wilma/salaries
# find /home -inum 387 -print
/home/fred/salaries
/home/wilma/mysal
/home/wilma/salaries
```

Figure 6.9 File names and inode numbers.

and inodes that are in different file systems. The ln -s command is used to create a symbolic link.

Figure 6.10 illustrates how a symbolic link is created between two file systems. The symbolic link (/home/carolyn/bin/ponies) is a special type of file. Its inode contains the full path name of the target file (/usr/local/bin/ponies). The dinode struct defined in /usr/include/jfs/ino.h includes a character array called

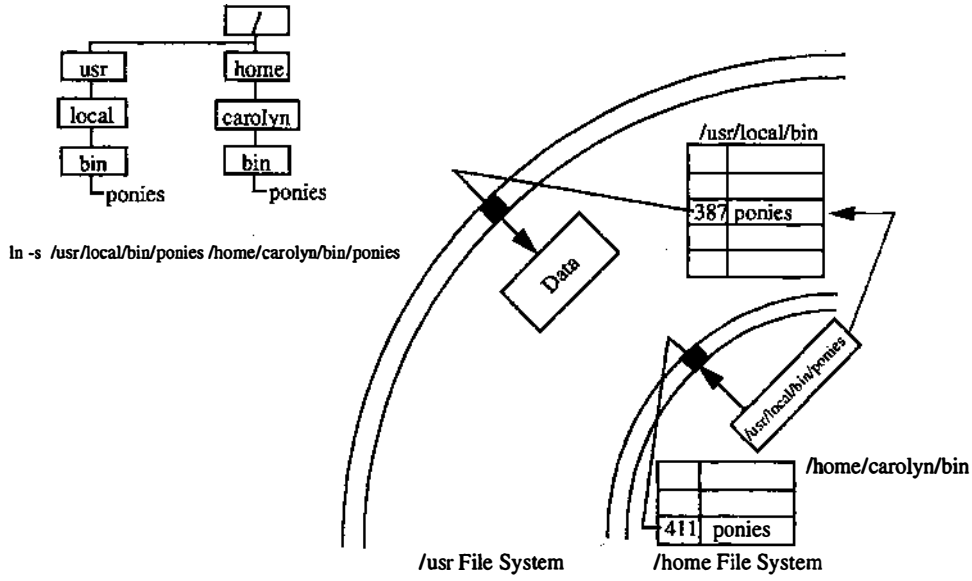

Figure 6.10 A symbolic link example.

_s_private[D_PRIVATE] which holds the path name of the target file as long as the path name is less than D_PRIVATE, which is defined as 48 characters. If the target file's path name is 48 characters or more (don't forget the null termination character), the path name is stored in a data block pointed to by the inode. This is all defined in the di_sym union.

The ls -l /home/carolyn/bin command shows the following (Fig. 6.11): The "l" character for the file type indicates the symbolic link. Notice the permissions for the symbolic link file. They are "wide open" since the target file's inode holds the actual permissions for the target file. Symbolic links do not increment the target file's link count.

Symbolic links, while necessary, include a number of potential problems. First, if the target file of a symbolic link is remove, moved, or renamed, the link is broken. For instance, if the file named /usr/local/bin/ponies is renamed to /usr/local/bin/horses, trying to execute /home/carolyn/bin/ponies would result in the error message "Cannot open /home/carolyn/bin/ponies." The error mes-

```
$ ls -l /home/carolyn/bin
...          ...  ...      ...    ...  ...      ...
lrwxrwxrwx   1    carolyn  staff  0    July 18 09:43  ponies->/usr/local/bin/ponies
...          ...  ...      ...    ...  ...      ...
```

Figure 6.11 Symbolic links and the ls command.

sage is misleading because it does not indicate the fact that the system cannot traverse the link. Care must also be taken when executing any type of file manipulation command on a symbolic link file. The results depend on whether or not the command knows how to deal with symbolic links.

Symbolic links offer one big advantage over hard links, aside from allowing links across file systems. Symbolic links can be established for entire directories. For instance, the /bin directory in AIX 3.2 is actually a symbolic link to the /usr/bin directory. This way, not every file in the /bin directory has to be established as a link.

> **Author's Note:** Some UNIX-based systems allow hard links of entire directories as long as the directories are within the same file system. AIX 3.2 does not allow this option for hard links.

> **Author's Note:** AIX 3.2 introduced the symbolic links of the /bin directory to /usr/bin and the /lib directory to /usr/lib to help maintain backward compatibility with AIX 3.1. Most of the /bin and /lib directory contents were moved to /usr/bin and /usr/lib, respectively, in AIX 3.2 to reduce the size of the root file system for support of diskless workstations.

Finally, if a user moves from the parent directory of a symbolically linked directory, down through the symbolically linked directory, then references the ".." directory name or issues the pwd command, the results can be surprising. For instance, consider the example in Fig. 6.12. If a user starts at /home/carolyn and moves to /home/carolyn/lib (which is really /usr/local/lib), then issues the pwd command, the output is "/usr/local/lib" if the user is using the Bourne shell or

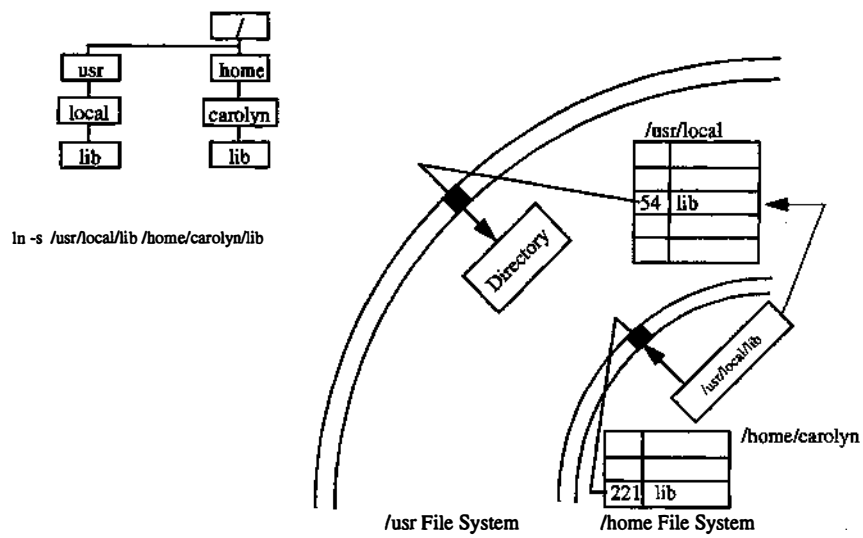

Figure 6.12 Symbolic links and directories.

the C shell. If the user is using the Korn shell, however, the output is "/home/carolyn/lib"! Similar results occur when the user issues a "cd .." command.

Access control lists

AIX 3.2 implements access control lists (ACLs) as a way of providing extended permission capability. The owner of a file can issue the commands acledit, aclget, and aclput to look at or modify the extended permissions of that file. (See the manual pages or InfoExplorer for more details on these commands.) Since the access control lists allow permission sets to be created for any user or group, additional space is required within the file's control data structure to hold this information. To avoid increasing the size of the disk inode, since ACLs might only be applied to a small number of files on most systems, IBM created a structure called an extended inode. The extended inode is an additional amount of disk space allocated only for files that require it. The dinode structure defined in /usr/include/jfs/ino.h has a field called di_acl which points to the data block holding the file's extended inode. If the value of di_acl is null, the file has no extended inode. The header file /usr/include/sys/acl.h defines the structures used to maintain ACLs. The header file lacks comments, but the structures are easy to understand once one understands the application of ACLs (see the manual page for acledit).

6.3 The Journaled File System (JFS)

The JFS represents one of IBM's finest contributions to the open systems marketplace, although it is not without some controversy. The JFS applies a data base logging approach to file system control structures. In this way, if a system crashes while these control structures are being updated, a log redo utility allows the file system to be returned to a known state. It is important to understand that the JFS does not log changes made to user data. Therefore, a system crash might still result in the loss of user data. The JFS attempts to assure that the file system maintains its integrity through a crash.

Prior to the JFS, a system administrator relied on tools such as fsck or fsdb to fix a corrupted file system. The fsck utility looks for inodes that have nonzero link counts yet are not claimed by any directory entry, or data blocks which are not on the free list yet are not claimed by any inode. When fsck finds such "orphans," it has little choice but to place the inodes (files) or data blocks (assigned to a file by fsck) into the lost+found directory located in the root directory of the file system. Finding the owners of the contents of the lost+found directory after running fsck relies upon the UNIX savvy of the system administrator and is usually not an easy task.

Incidentally, AIX 3.2 includes the fsck utility which still can be used to check the integrity of a file system. In fact, if the JFS log redo fails for any reason, fsck may be the only way to fix a corrupted file system, short of restoring from a backup. AIX 3.2 also offers the fsdb (file system debugger), which allows a user to examine and change the data found in file system control structures.

Each journaled file system within a volume group (see Sec. 2.3 for information on volume groups) usually shares a common JFS log. The JFS log for the root volume group is in the /dev/hd8 logical volume. It is a 4-megabyte circular log which is updated by the operating system at regular intervals. The details of the JFS logging are described in Sec. 6.6.

One unfortunate aspect of the JFS is that it is not supported for floppy diskette file systems.

6.4 The JFS Architecture

As previously mentioned, the JFS design is similar to other UNIX-type file systems. It includes a boot block, a super block, inodes, indirect blocks, and data blocks. Anyone familiar with other types of UNIX file systems might be tempted to skip this section of the book. However, the JFS supplies a few interesting twists to the traditional UNIX file system paradigm. This section explores those nuances. It also serves as an introduction to the fundamental concepts of file systems.

Before describing the design on the JFS it's necessary to say a few things about how AIX 3.2 manages disk space. As explained in Chap. 2, the LVM carves all physical disk space up into contiguous physical partitions. The default size of these partitions is 4 megabytes on most RISC System/6000 models. Some smaller RISC System/6000 models use a 2-megabyte default physical partition size. In any case, the size of a volume group's physical partitioning is constant throughout the volume group and can be set by the system administrator when the volume group is created. IBM recommends 4 megabytes as the optimal size.

When a journaled file system is created, the system allocates it to one or more physical partitions. This means that the smallest file system that can be created in a volume group is equal to the size of the physical partitioning of that volume group. Figure 6.13 shows a volume group with 4-megabyte physical partitions. It also shows a couple of 4-megabyte file systems and a 12-megabyte file system. Another nice feature of the JFS (with help from the LVM) is the ability to easily extend a file system. When a file system becomes full, the system administrator

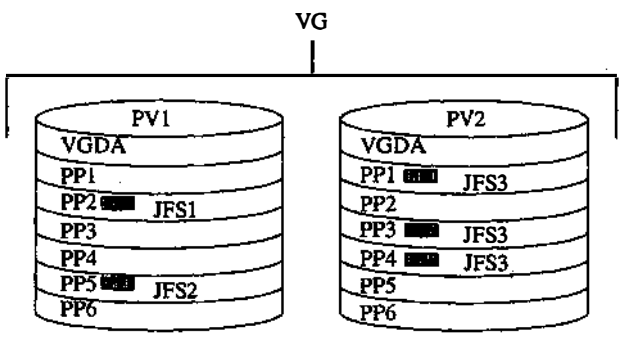

Figure 6.13 The JFS and the LVM.

can simply allocate more physical partitions, as long as free physical partitions exist within the volume group. The partitions allocated to a file system need not be contiguous or even on the same physical disk drive. Best of all, the file system can be extended while it is mounted and in use by users' processes.

While the logical volume manager refers to the 4-megabyte chunks of contiguous disk space as physical partitions, the journaled file system calls them allocation groups. As shown in Fig. 6.13, each allocation group contains its own set of inodes. This is similar to the BSD file system concept of cylinder groups.

Each JFS consists of the following components, as illustrated in Fig. 6.14:

Block 0. The boot block (also called the ipl block), which is not used for anything.

Block 1. The super block, which contains control information for the entire file system.

Block 31. The spare super block copy, used in the event of a corrupted super block. While many UNIX file systems allocated a large number of spare super blocks, the JFS only allocates one. This is because the JFS logging makes it less likely for the super block to become corrupted. Also, as described shortly, the information found in the JFS super block is not as important to the integrity of the file system as the information found in the super block of other UNIX file systems.

Blocks 32 through 63. The inode table for this allocation group.

Blocks 2 through 31, and 64 through the end of the file system. Data blocks.

The header file /usr/include/jfs/filsys.h has some excellent comments describing the design of the JFS. It also contains the definition of the super block structure.

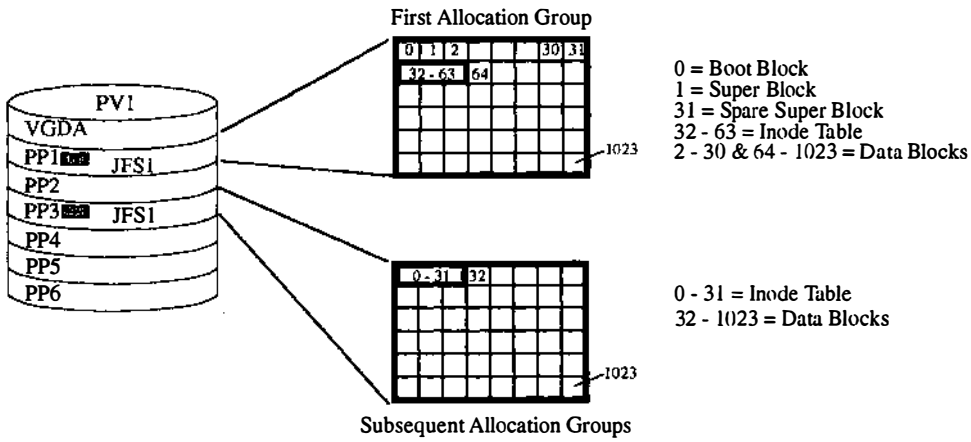

Figure 6.14 JFS block allocation.

The JFS super block

Block one of the JFS is the super block for that file system. The structure of the super block is defined in the header file /usr/include/jfs/filsys.h. The JFS super block is not as "super" as its counterparts in other UNIX-based file systems. Traditionally, the super block contains pointers to the linked lists of free inodes and data blocks for the file system it represents. For this reason, if the super block became corrupted, the file system was trashed. This is why many UNIX-based file systems have spare super block copies scattered throughout the file system. As you will see shortly, IBM has implemented another method of maintaining inode and data block–free lists. The method does not rely on the JFS super block.

The JFS super block contains a few interesting fields, such as:

s_fsize. The file system size (in 512-byte blocks). AIX 3.2 supports a maximum file system size of 2 gigabytes.

s_bsize. The block size for this file system.

s_fname[]. The name of the file system.

s_logdev. A dev_t type (device major and minor numbers) for the JFS log of this file system.

s_ronly. A character field set if the file system is mounted as read-only.

s_time. A time_t type that holds the timestamp of the last super block update.

s_fmod. A character flag that indicates the state of the file system. Values include:

0. File system is clean and unmounted.
1. File system is mounted.
2. File system was mounted when dirty or commit failed.
3. Log redo processing attempted but failed.

Beyond these fields, the JFS super block isn't very interesting.

The inode table and inode allocation

Blocks 32 through 63 of a JFS hold the inode table for the first allocation group. The JFS in AIX 3.2 dynamically allocates another inode table for each new allocation group when the file system is extended. The inode table occupies blocks 0 through 31 of each allocation group after the first allocation group. The JFS attempts to assign data blocks for inodes from the same allocation group whenever possible. Unlike traditional UNIX-based file systems that allow the system administrator to specify the number of inodes for a file system when the file system is created, the JFS in AIX 3.2 defaults to one inode per data block

within the allocation group. Since the size of an allocation group is 4 megabytes, and each data block is 4 kilobytes, the JFS creates 1024 inodes for each allocation group. This is illustrated in Fig. 6.14.

In theory, this inode allocation scheme should mean that a file system will never run out of inodes as long as there are still data blocks because of the one-to-one ratio. In practice, however, it is possible to run out of inodes and still have data blocks left over because zero-length files, such as symbolic links and device special files, require an inode without ever allocating a data block.

> **Author's Note:** The latter example occurred for one of my students who was working with AIX 3.2 on IBM's SP/2 system. She needed to define a large number of device special files in the /dev directory, only to find that the system ran out of inodes in the root file system. The only solution was to extend the root file system to create more inodes.

Reserved inodes

The first 16 inodes of every JFS file system (inodes 0 through 15) are reserved by the JFS. A description of each reserved inode is found in the comments of the /usr/include/jfs/filsys.h header file. Most of these reserved inodes have file names that begin with a "." (dot) character because they are hidden files, but unlike the hidden files found throughout the system directories, these files are "really hidden," as they do not appear in any directory. This is accomplished by handcrafting the inodes so that they do not require a directory entry to support their link count value. Remember that every open file is represented by a segment in the VMM (see Chap. 4). Many of the reserved inodes are associated with files that are only present in the VMM when a file system is mounted and never actually exist on disk.

> **Author's Note:** The /usr/include/jfs/filsys.h header file erroneously comments that inodes 9–16 are reserved for future use, when actually inodes 9–15 are reserved. This error can be tested by creating a new file system and creating a file within the new file systems. The first file created should have an inode number of 16. This just goes to show that you can't always trust the comments!

Inode 0 of the JFS is never used.

Root directory. Inode 2 of the JFS is always used for the root directory of the file system. An interesting thing happens if one performs the ls -ia command in the root file system, then compares the output with that of an ls -ia command in the /home directory. The inode number for /home within the root file system is 2. In other words, since the file system is mounted, the ls command reports the /home directory as the root of the /home file system. In addition, the inode number of "." in the /home directory is also 2 since that is the inode for the root directory of the /home file system, not the /home mount point.

.superblock. Inode 1 of the JFS is reserved for a file named .superblock. This virtual file simply refers to the super block and spare super block. Examining the inode reveals that the file consists of two data blocks, blocks 1 and 31.

.inodes. Inode 3 of the JFS is reserved for a file named .inodes. This file keeps track of all file system data blocks being used to hold inodes. This is similar to a disk or cylinder group map in other UNIX-based file systems.

.indirect. Inode 4 of the JFS is reserved for a virtual file named .indirect. The VMM uses this file to map the pages of indirect blocks for the entire file system. An explanation of indirect blocks is given shortly.

.inodemap. Inode 5 of the JFS is reserved for a virtual file named .inodemap. This file takes the place of the traditional linked list of free inodes via a bit map where each inode of the file system is represented by a bit flag. When an inode is in use (di_nlink>0), the bit flag is turned on for the corresponding bit in the map. When a new file is created in the file system and a new inode is needed, the file system scans the .inodemap segment for the first bit flag that is turned off. The bit flag is turned on and the corresponding inode is assigned to the new file. This technique is much faster than manipulating linked lists.

.diskmap. Inode 6 of the JFS is reserved for a virtual file named .diskmap. As the .inodemap file keeps track of the free and allocated inodes, the .diskmap file keeps track of free and allocated data blocks within the file system. The .diskmap file, however, is not a simple bit map. The file uses a set of hash buckets built on a binary tree principle to point to chains of contiguous free data blocks. This way, when a new file is allocated with a known size, such as when a large file is copied to another file, the JFS can search for a large enough set of contiguous data blocks to allow the file to be stored with the least amount of fragmentation.

.inodex. Inode 7 of the JFS is reserved for a virtual file named .inodex. The file contains information about inode extensions, as used by access control lists (see Sec. 6.2).

.inodexmap. Inode 8 of the JFS is reserved for a virtual file named .inodexmap. The file contains a bit map used to keep track of free and allocated inode extensions. It is similar to the .inodemap file.

Many of these virtual files are created when the file system is mounted.

As mentioned earlier, inodes 9 through 15 of the JFS are reserved for future use. Figure 6.15 recaps the reserved inodes.

0	Not Used
1	.superblock
2	Root directory of the File System
3	.inodes
4	.indirect
5	.inodemap
6	.diskmap
7	.inodex
8	.inodexmap
9-15	Reserved

Figure 6.15 Reserved inodes.

6.5 JFS Storage Schemes

The earlier discussion of inodes alluded to the fact that a file's inode contains the logical disk addresses for the data blocks of that file. One might wonder how the inode, which is only 128 bytes, can hold enough logical disk addresses to accommodate large files. This is done through a series of storage schemes, each designed to efficiently handle files of various sizes. While the JFS method of addressing data blocks is similar to the direct, single indirect, double indirect, and triple indirect schemes used by many UNIX file systems, it differs slightly in its implementation.

The disk inode contains an array of eight logical addresses for the first eight data blocks of a file. The term "logical address" refers to the data blocks' numbers within the file system. The actual disk addresses (i.e., physical sector numbers) are derived by the logical volume manager during disk I/O. The array of eight direct block pointers, called _di_rdaddr[NDADDR], is defined within the dinode structure of the /usr/include/jfs/ino.h header file. NDADDR is defined by AIX 3.2 as eight. Each pointer holds the logical block number for a 4-kilobyte data block. This scheme, as shown in Fig. 6.16, supports a file size of 32 kb. Since support for larger files is required, a file with a size of >32 kb must

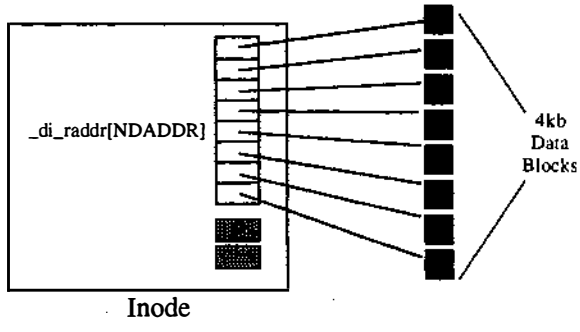

Figure 6.16 Direct data block accessing scheme.

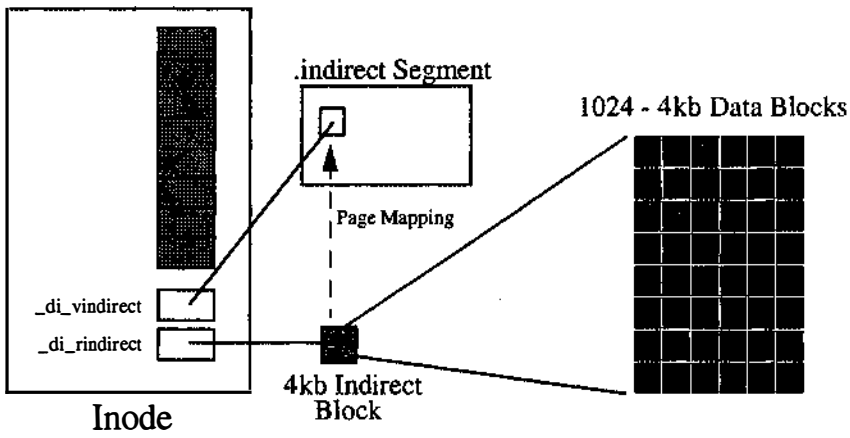

Figure 6.17 Single-indirect data block accessing scheme.

resort to another storage scheme. Figure 6.17 shows how the JFS implements single indirect access. Each inode contains a field named _di_rindirect. This field holds the logical disk address of the file's indirect block. An indirect block is a 4-kilobyte data block that has been converted, by the JFS, to hold up to 1024 4-byte logical disk addresses for data blocks. This allows a file to access up to 4 megabytes of space.

To speed up access to files that use indirect addressing, the JFS works with the VMM to map the indirect block into a page of a VMM segment called .indirect. There is a single .indirect virtual file created for each mounted file system. Each page of the .indirect segment can hold an indirect block from disk. All files larger than 32 kb within the same file system share the use of the .indirect file.

Each inode that has an indirect block uses a field named _di_vindirect to indicate the page number of the memory mapped indirect block within the .indirect segment. For files larger than 4 Mb, the JFS uses a third scheme, known as the double indirect block. In this case, the inode's _di_rindirect field contains the logical disk address of a double indirect block. A double indirect block is a 4-kilobyte data block that has been converted to hold up to 512 8-byte values, each of which consists of two 4-byte pointers. One of the 4-byte pointers from each 8-byte pair holds the logical disk address of an indirect block. The other 4-byte pointer from each 8-byte pair holds the page number of where its corresponding indirect block has been mapped in the .indirect segment. This scheme allows the double indirect block to access up to 512 indirect blocks, each of which can access up to 1024 4-kb data blocks. This provides a maximum access size of 2 gigabytes (512 * 1024 * 4096). Since 2 gigabytes is the largest supported file system in AIX 3.2, this scheme works perfectly. Figure 6.18 shows how a very large file (>4 Mb) might be accessed by the JFS.

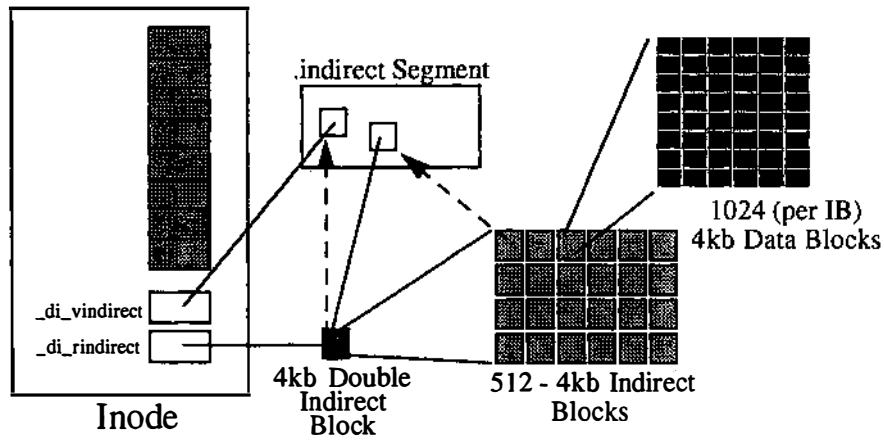

Figure 6.18 Double-indirect data block accessing scheme.

> **Author's Note:** One might be tempted to think that the address scheme described above is the reason that AIX 3.2 has a file size limit of 2 Gb. Actually, the reason for the 2-Gb file size limit is more historical than technical. The field called di_size in the dinode structure defined in /usr/include/jfs/ino.h is a typedef of off_t. The off_t is defined in /usr/include/sys/types.h as a signed long, which means that one of the 32 bits is used to show the sign of the number, leaving 31 bits to hold the number itself; 31 bits can access up to 2 Gb. Your next questions might be "Why is the off_t a signed long? When would I have a negative file size?" The off_t is also used to define the read/write offset within a file (see the manual page for the lseek() subroutine). Since it is possible to seek backward through a file, negative values had to be supported. At the time this strategy was created, no one could implement 2-Gb file sizes because the hardware didn't support it. (Remember, it wasn't that long ago that we were all paying big bucks for 10 Mb hard drives!) Therefore, a maximum file size of 2 Gb seemed more than adequate. Most open system vendors have implemented or are considering schemes that extend the maximum file size beyond 2 Gb. AIX Version 4.2 will support 64-Gb maximum file and file system sizes.

The actual layout of the .indirect segment is not as simple as the example given above. The .indirect segment is described through comments found in the /usr/include/jfs/ino.h header file.

6.6 How the JFS Log Works

As mentioned previously, the goal of the journaled file system is to provide a more robust file system by logging changes made to its own structures and lists. This includes changes made to the file system super block, the inodes, directories, indirect blocks, and free lists of inodes and data blocks.

> **Author's Note:** The term "free list" is used figuratively here. Recall that the JFS uses bit maps to track free and allocated inodes and data blocks.

Whenever a JFS is mounted, the AIX kernel verifies its consistency by examining the log records of that file system. The log records show transactions that were completed (committed), as well as, in the case of a file system crash, incomplete transactions. The JFS reconstructs the committed transactions, bringing the file system structures up to date. Incomplete transactions are discarded, since they would leave the file system structures in a "half-baked" state.

JFS logging is similar to the way many data base programs log transactions. It's important to remember, however, that the JFS does not log the user data bound for the file system data blocks. In other words, when a system crash occurs or someone shuts off the power to the computer without properly bringing down the operating system, data in memory which have not yet been written to disk are lost. The JFS log will make it possible to recover the file system control data, but the user data are not recoverable.

Components of the JFS

The JFS uses a physical disk partition (usually 4 megabytes in size) as a log device. Each volume group (see Chap. 2 for a discussion of volume groups and the logical volume manager) must have a JFS log device. The rootvg's log device is /dev/hd8. A JFS's log device can be specified at the time the file system is created.

The JFS also maintains a log segment (a 256-megabyte segment) in virtual memory for each log device. Pages of this segment are written to the disk log device at regular intervals. The JFS also maintains an inode manager and a lock manager to ensure that in-core inodes are locked while they are updated. (In-core inodes are detailed in Chap. 7.)

A JFS example

To illustrate how the JFS logs changes made to the file system structures, an example of the mkdir command is used. The following list includes just a few of the events that take place in the JFS when a new directory is created by a user:

An inode is allocated for the new directory. The .inodemap segment is updated by setting the bit that represents the inode.

An entry is made for the new directory in the parent directory's data block. The new directory's inode number is associated with the new directory's name. This operation may involve adding a new directory block to the directory.

A data block is allocated to the new directory. The .diskmap segment is updated.

The "." and ".." file names are added to the new directory.

The link count is incremented for the parent directory's inode (due to the ".." file name in the new directory).

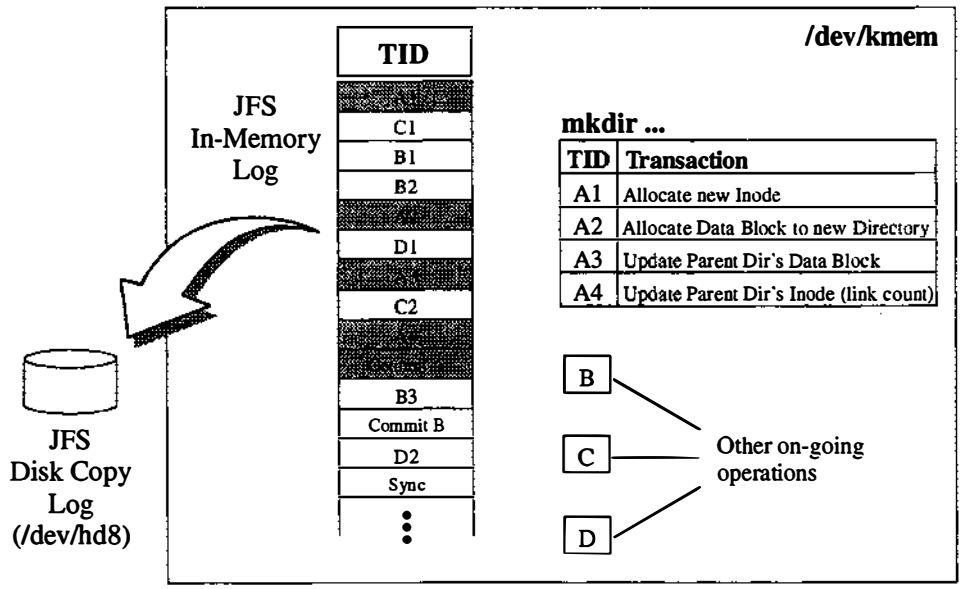

Figure 6.19 A JFS logging example.

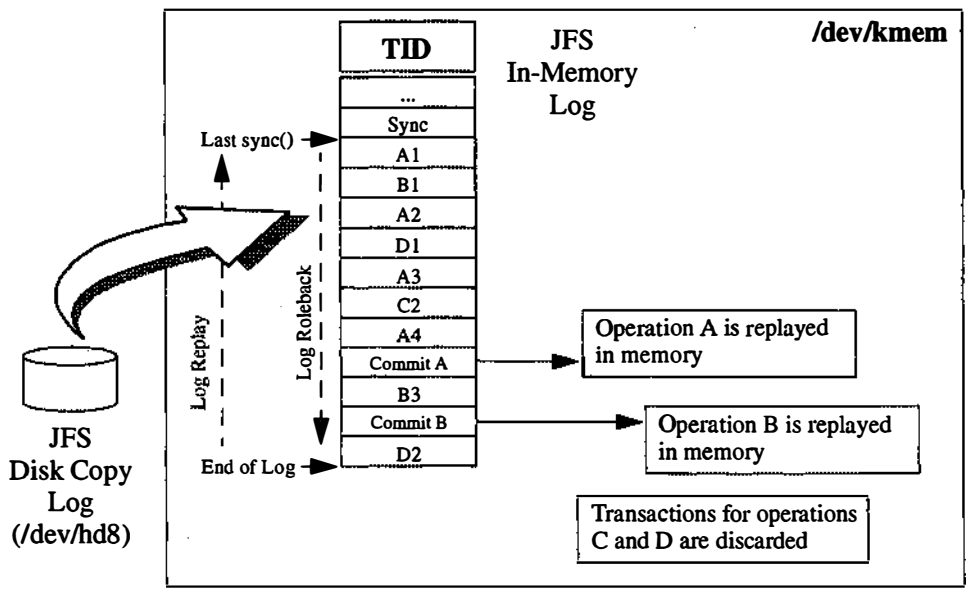

Figure 6.20 The JFS log redo.

The link count is incremented in the new directory's inode (due to the "." file name it now contains).

Author's Note: That's quite a bit of activity, isn't it? On most UNIX-based systems, if the system crashed while in the middle of performing these tasks, the file system would end up in a corrupted state. The JFS attempts to assure that these activities are performed atomically.

It's important to understand that the events listed above happen in memory and are then written to disk some time later. Generally, user data and file system control data are written to disk whenever a sync occurs. The JFS is able to confirm that this has happened by writing a sync record to the log.

Author's Note: Many UNIX-based systems use a sync daemon, which writes all modified file pages and file system data to disk every 30 seconds. The AIX 3.2 syncd process performs a sync once every 60 seconds, by default. The sync time can be changed. The syncd daemon is launched from the /sbin/rc.boot shell script at system start-up. A system administrator can change the parameter to the syncd program from within this script.

As the new directory is being created, the JFS logs each transaction (each event from the list above) in the log segment, as the activity takes place in memory. Once all transactions have been completed in memory and recorded to the log segment, the JFS writes a commit record to the log segment. As mentioned earlier, the log segment pages are written to the disk-based log device at regular intervals. Figure 6.19 illustrates the example so far.

When a file system is mounted, the JFS checks the log entries for the file system. If the log does not end with a sync record, the JFS concludes that the file system was not unmounted cleanly. The JFS performs a log redo, searching back through the log for the previous sync record. The JFS does not care about any transaction above this sync record since the sync caused all changes to be written to disk.

The JFS then performs all transactions for which commit records exist. Any transactions that do not have a commit record are discarded. This should bring the file system structures to a complete and known state. Figure 6.20 illustrates the log redo procedure.

Author's Note: While the JFS is not foolproof, I can attest to the improved file system reliability. I have tried, on occasion, to corrupt a JFS file system by doing things like powering down in the middle of copying a large file. While I have experienced a loss of user data, I have not corrupted a file system. I am not advocating, however, that you try this on your system!

Chapter

7

AIX 3.2 Disk File I/O

This chapter details the kernel components involved in providing local disk file I/O in AIX 3.2. It traces the links of structures from the process to the kernel's file table. Vnodes are introduced, although details of the virtual file system are found in Chap. 8. Gnodes and the in-core inode table are described and the link is made between the file subsystem and the virtual memory management subsystem.

7.1 File I/O Layers

Figure 7.1 shows the layers of local disk file I/O. When a process opens any type of file, a link is established between the process and the file subsystem. The details of the file subsystem are the topic of this chapter. The AIX 3.2 virtual memory manager allocates a segment for each opened file (see Chap. 4). The data blocks of the opened file are mapped to pages in the segment. This technique is simplified by the fact that the sizes of data blocks and VMM pages are both 4 kilobytes.

When a process performs a read operation on an open file, the VMM checks to see if the desired page is already in memory (perhaps the page was recently read by the process or by another process). If the page is found, the process's read request is satisfied without a physical read being performed. If the page is not found in memory, the VMM locates and loads the page via a pagein. A pagein results in a physical read. Write operations work in a similar fashion. This explains the difference between logical I/O, between the process, the file subsystem, and the VMM, and physical I/O, between the VMM and the disk storage subsystem.

For ordinary file I/O, if the desired file page is not in main memory, the VMM issues a request to the logical volume manager (LVM) to fetch the page from its persistent disk location in the file system. The VMM was discussed in Chap. 4

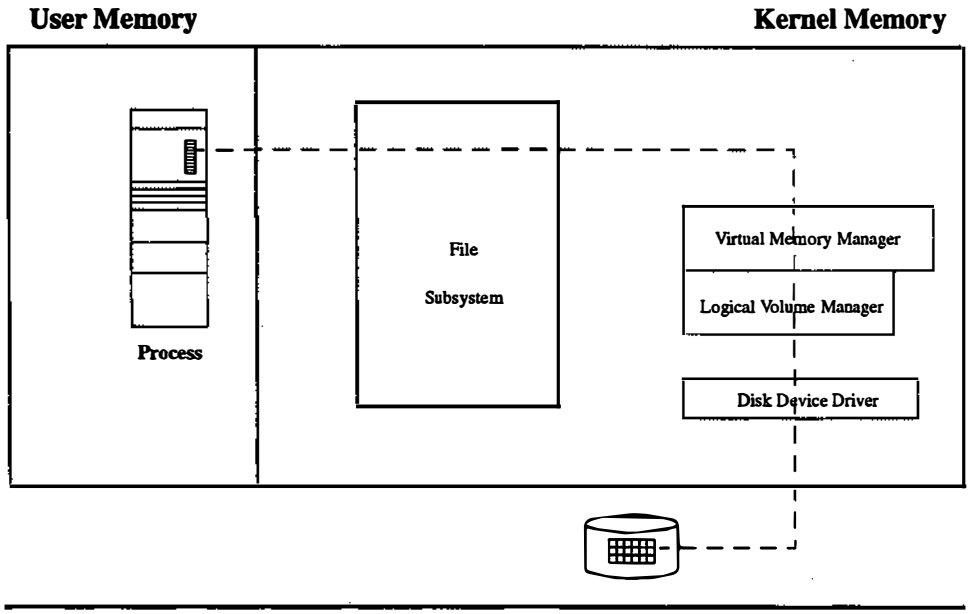

Figure 7.1 File I/O layers.

and the LVM is discussed in Chap. 2. Here, the focus is on the file system components of the AIX 3.2 kernel.

The AIX 3.2 file subsystem can also be divided into three layers, as illustrated in Fig. 7.2. The logical file system deals with file names, directories, inodes, and data blocks. When a process opens a file, the logical file system converts the file name to an inode number and determines the file type. The process's credentials are checked at this level to allow or deny the specified access to the file or directory.

The virtual file system layer is relatively new to most UNIX-based systems. It was created to provide an interface to different file system types, without the process needing to know the details of the file system types. For instance, an application might open, then read from a file named "/custdata/TEXAS/pastdue" without needing to know if the file is coming from a local disk drive, over the network via some network file system utility, or from a CD-ROM. While performing file I/O to and from these file system formats is certainly different for each format, the application need not be concerned with those differences. The workings of the virtual file system (VFS) layer are discussed in detail in Chap. 8. For now, it's enough to say that the VFS layer determines the VFS type of the requested file and performs the appropriate tasks.

The final layer of file I/O involves the kernel components that represent the physical file. These components consist of the process's file descriptor table, the kernel's file table, the in-core inode table, and a few other structures. Figure 7.3 provides an example of how these components are linked when a process opens a file. The vnode is part of the VFS layer, which is actually found in the middle

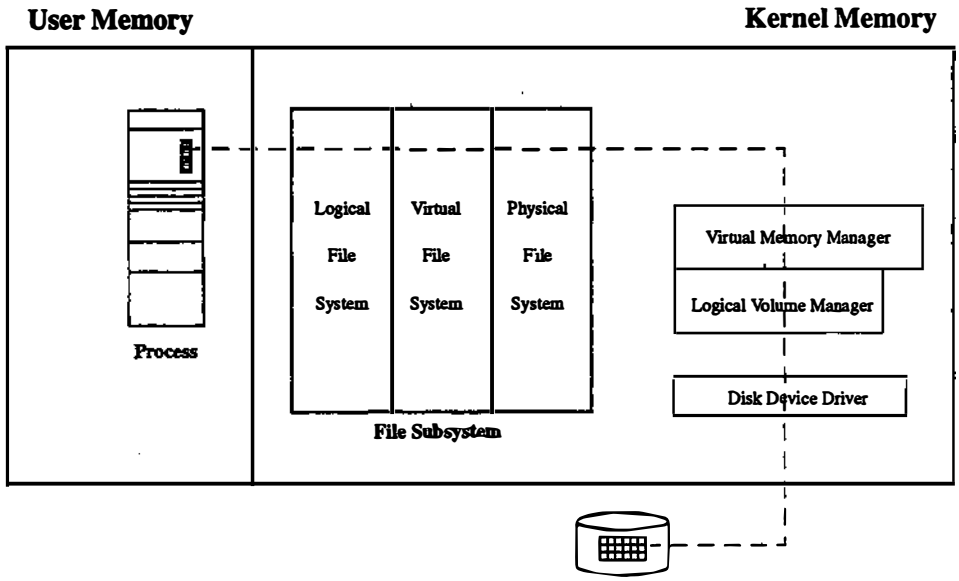

Figure 7.2 The file I/O subsystem.

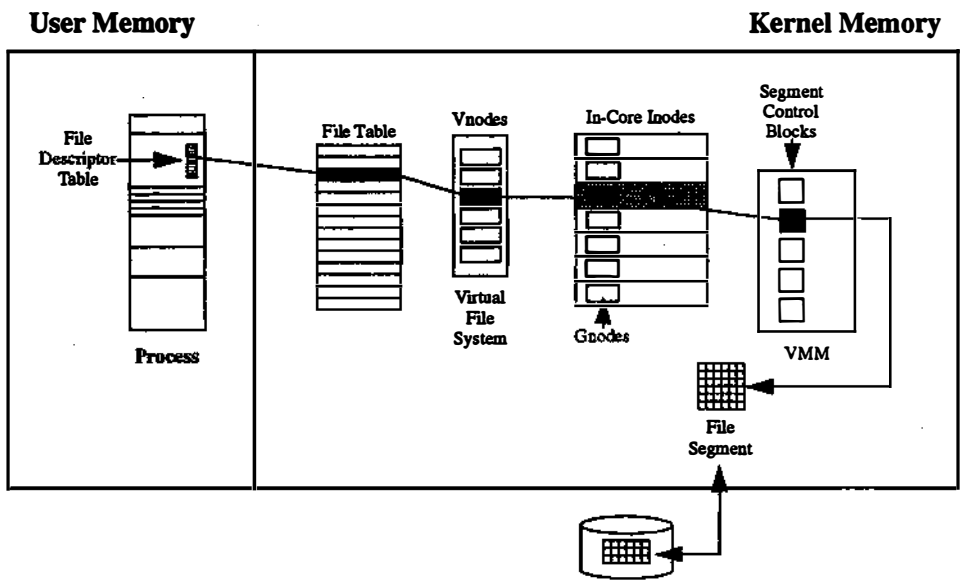

Figure 7.3 File I/O subsystem components.

of the other kernel components. UNIX-based systems that do not implement the VFS layer simply have a pointer from the kernel's file table directly to the in-core inode table.

7.2 The Process File Descriptor Table

Each process has its own file descriptor table (FDT). This is an array found within each process's user area. The FDT is defined as u_ufd[OPEN_MAX] in the user structure found in /usr/include/sys/user.h. OPEN_MAX is defined in /usr/include/sys/limits.h as 2000. Each of the 2000 slots of the u_ufd array consists of a pointer to a struct file, which is a pointer into the kernel's file table. The pointer name is fp. As a C array, the slots are numbered from 0 through 1999. The first three slots of a process's FDT (zero, one, and two), as you might have already guessed, are usually assigned to the process's controlling terminal device file as standard in (0), standard out (1), and standard error (2).

> **Author's Note:** The number of per process file descriptor table entries varies from one UNIX-based system to another. While some systems allow the system administrator to tune the value (the parameter is usually called NOFILES) and some systems allocate chunks of file descriptors dynamically, the size of the per process file descriptor table for AIX 3.2 is static at 2000 slots per process.

File descriptors from the process's FDT are assigned each time a file is opened, as in the case of the open() or creat() subroutines or any of their variations. File descriptors are also assigned by the dup(), socket(), and pipe() subroutines and can be assigned by the fcntl() subroutine. An important rule to remember is that, for most of these subroutines, the system assigns the lowest available file descriptor. As an example, the open() system call returns the lowest available file descriptor. The file descriptor is then used as a handle to the file when performing other file I/O functions. Figure 7.4 provides a short code example. The open() system call returns -1 upon failure and sets the errno variable to indicate the reason for the failure (see /usr/include/sys/errno.h). Reasons for failure might include "No such file" (ENOENT), "Operation not permitted" (EPERM), or "Too many open files" (EMFILE). The last example indicates that the process already has 2000 open files. An open file must be closed in order to free up a file descriptor.

> **Author's Note:** While it may seem unlikely to need more than 2000 files opened by a single process, I have encountered a few students who have hit this limit.

7.3 The Kernel's File Table

The read/write offset

The kernel maintains a single file table. Each slot is defined as a struct file in the /usr/include/sys/file.h header file. One slot is allocated in the file table for each instance of the opening of a file. The key here is to understand that if two

```
#include <fcntl.h>

main()
{
    int fda, fdb, fdc, fdd;
    ...
    if((fda=open("filea",O_RDWR))==-1)
        perror("filea");
    if((fdb=open("fileb",O_RDWR))==-1)
        perror("fileb");
    if((fdc=open("filec",O_RDWR))==-1)
        perror("filec");
    if((fdd=open("filed",O_RDWR))==-1)
        perror("filed");
    ...
}
```

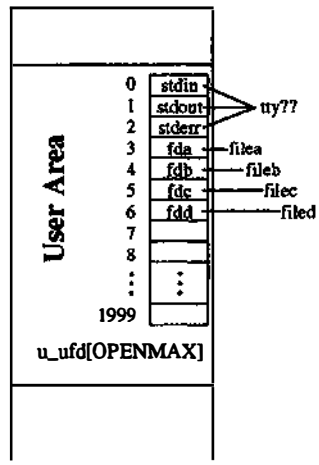

Figure 7.4 The file descriptor table.

processes open the same file at the same time, or if a single process opens the same file more than once, there will be multiple entries in the file table. The reason for this is that the primary purpose of the file table is to hold the read/write offsets for each open file. Since many processes can open the same file at once, yet each process needs to maintain its own location within the file, separate file table entries are allocated. The read/write field is called f_offset and is an off_t data type. This is the same data type used by the disk inode to hold the file's logical size.

> **Author's Note:** The f_offset field is often referred to as the "read/write pointer." Even the comment in the /usr/include/sys/file.h file refers to it as the "read/write character pointer." It's important to realize that this field is not a pointer, in the true C sense, but rather an offset from the first byte of the file.

File table reference counts

Another field in the file table, f_count, indicates the reference count as the number of process file descriptors pointing to the file table slot. Since the kernel allocates one slot in the file table for each instance of an open file, and these slots are pointed to by file descriptors in each process, there is usually a one-to-one relationship between file descriptors and file table slots. There are a few instances, however, when more than one file descriptor points to a single file table slot. The dup() subroutine and its variations duplicate the memory pointer found in one file descriptor to the lowest available file descriptor. This causes the two file descriptors to point to the same file table slot. This technique is

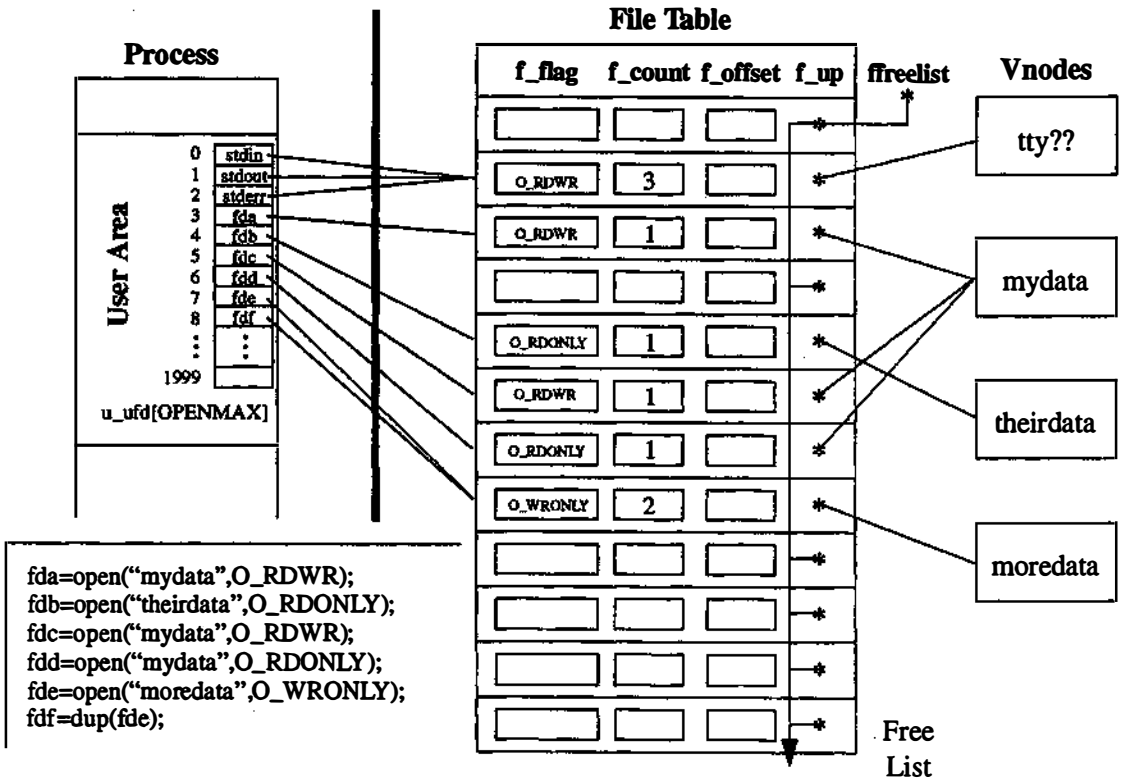

Figure 7.5 The kernel's file table.

used in AIX 3.2 to establish the relationship of standard in, standard out, and standard error. Figure 7.5 shows a code example along with an illustration of the file descriptor table and kernel's file table.

> **Author's Note:** Many UNIX-based systems open si (standard in) as read-only, then open so (standard out) as write-only, then dup so to se (standard error). AIX 3.2 simply opens si for read-write, then dups so and se. While it's obvious that a process would never write to standard in, this technique makes other operations, such as building pipes (see Chap. 10), easier.

Another time, aside from dup(), when file table slots are shared by more than one file descriptor is when a process which has already opened files performs a fork() system call and creates a child process. The child process inherits the parent's file descriptor table; thus the pointers assigned to the file descriptors remain the same. A subroutine called by fork() increments the reference count for each file table slot that is inherited by the child process. Figure 7.6 illustrates an example of inherited file descriptors. After the fork(), the parent and child processes share the read-write offset of each opened file. This technique makes sense when dealing with standard out or standard error, since it is desir-

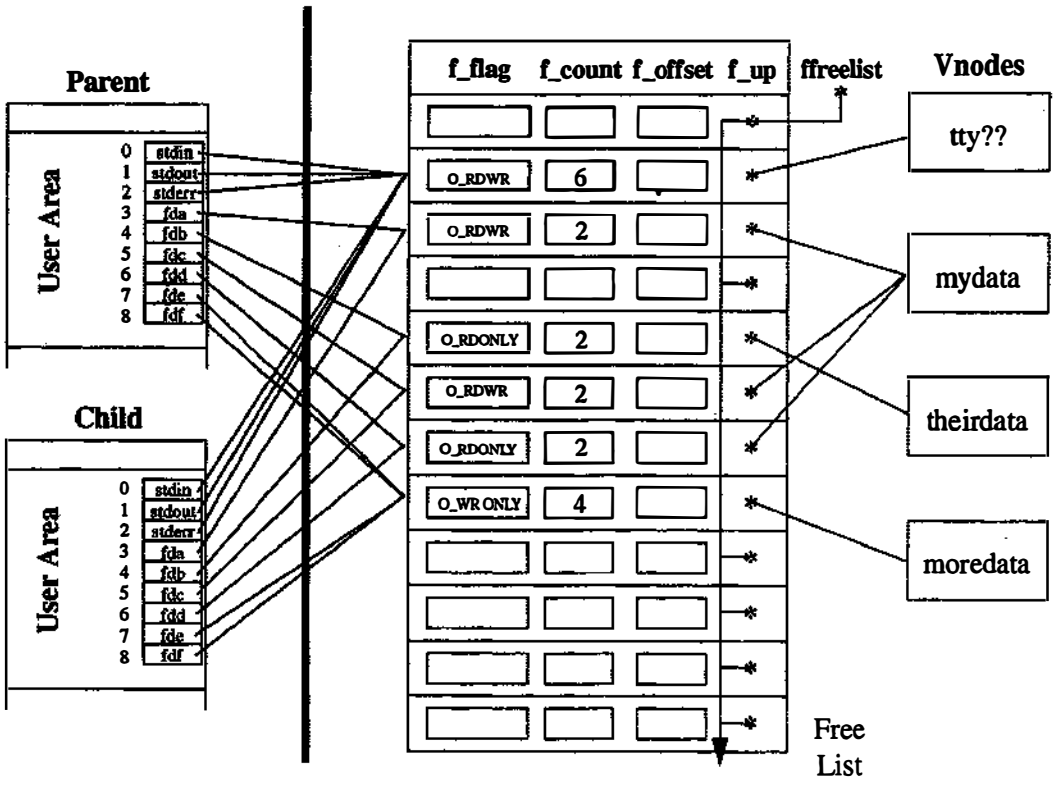

Figure 7.6 The file table after a fork().

able for processes created within the same terminal sessions to adhere to a synchronous data flow of output and errors. After the fork(), files opened by either process are independent of the other process.

Vnode pointer/free list union

Other fields of note in the file table include:

f_flag. A long that indicates how the file was opened (i.e., O_RDWR, O_RDONLY, etc.). See the manual page for the open() subroutine for information on the flags.

f_type. A short that indicates the type of file descriptor associated with the file table slot. The values are defined in /usr/include/sys/file.h as DTYPE_VNODE (1) as an ordinary file, DTYPE_SOCKET (2) as a socket endpoint, or DTYPE_GNODE (3) for a device.

f_up. A union which holds either a pointer to a struct vnode when the file table slot is in-use or a pointer to the next free slot in the file table when the

slot is not in-use. The vnode pointer is called f_uvnode and the free list pointer is called f_unext.

A single vnode is created in the kernel for each opened file, no matter how many times the file is concurrently opened. The term "vnode" comes from "virtual node" and is the key element of the virtual file system layer. Vnodes are mentioned here because they connect the file table to the gnode. Details of the vnode and the virtual file system are provided in Chap. 8. Gnodes are discussed shortly. Figure 7.5 illustrates the relationship of the file table to the vnodes and the file table's free list.

The kernel anchors the file table's free list with a pointer called ffreelist.

The fileops structure

Finally, each file table entry includes a pointer to a struct fileops. The fileops structure is defined directly within the file structure in the /usr/include/sys/file.h header file. The fileops structure contains pointers to functions that manipulate data in the file table, such as functions for reading and writing, which modify the read/write offset.

7.4 The Gnode

As mentioned, an active file table slot points to the vnode associated with an opened file. The vnode then points to a gnode structure. The gnode structure is defined in /usr/include/sys/vnode.h as a struct gnode. The term "gnode" refers to a generic node.

Gnodes are usually contained within another structure. In the case of a local disk file, for instance, the gnode is found within an in-core inode. In-core inodes are used to hold the memory-resident copy of the disk inodes when local disk files have been opened. The kernel maintains a table of in-core inodes. Figure 7.7 illustrates the relationship of vnodes to gnodes and gnodes

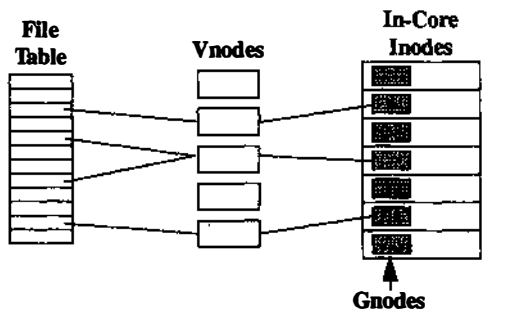

Figure 7.7 Vnodes, gnodes, and in-core inodes.

to in-core inodes. The gnode is explored further after a discussion of the in-core inode table.

7.5 The In-Core Inode Table

When a local disk file is opened, the kernel reads the file's disk inode into memory where the contents of the 128-byte dinode structure are stored within a larger structure called an in-core inode. Figure 7.8 illustrates what happens when a file is opened.

In-core inode allocation

The in-core inode structure is defined in the header file /usr/include/jfs/inode.h. (Recall that the disk inode structure, dinode, is defined in the header file /usr/include/jfs/ino.h.) The goal of AIX 3.2 is to have one in-core inode active for every local disk file that has been opened since the system was booted. In other words, when a file has been closed by all processes that had it open, the file table slot and the vnode are released, but the in-core inode remains intact. The idea is that most systems tend to have the same relatively small set of files opened repeatedly. By keeping the in-core inodes of files that have been closed in the kernel, subsequent opens to the file take less time.

Of course, the kernel cannot allocate new in-core inodes without limit. Therefore, the kernel establishes a set of in-core inodes, known as the in-core inode table (ICIT). Within the ICIT, the kernel maintains a free list and a cached list of in-core inodes. When the free list is exhausted and a new in-core inode is required, the kernel selects the least-recently-used in-core inode and reassigns it to the newly opened file. The size of the ICIT is set at system start-

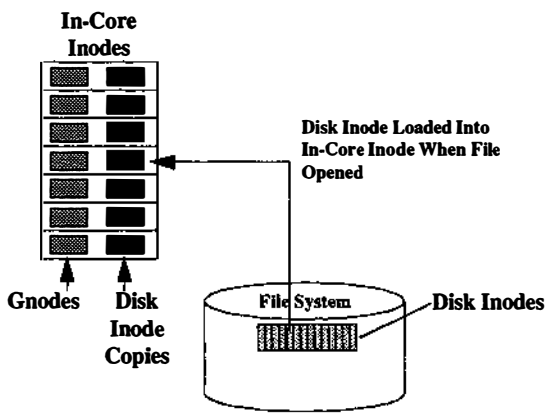

Figure 7.8 Disk inodes and in-core inodes.

up as 4096 + (32 * Memsize_in_megabytes) in-core inodes. As an example, a system with 64 Mb of real memory has an ICIT size of 6144 in-core inodes.

Interesting fields from the in-core inode, as defined in /usr/include/jfs/inode.h, include:

i_forw and i_back. A set of pointers to the next and previous in-core inodes in the hash queue. The hash queues and hashing technique are described shortly.

i_next and i_prev. A set of pointers to the next and previous in-core inode on the free list or cache list.

i_gnode. The entire gnode structure, as defined in /usr/include/sys/vnode.h. Recall that the gnode is usually contained within another structure. For local disk files, the gnode is found within the in-core inode.

i_number. An ino_t data type that holds the disk inode number of the file.

i_dev. A dev_t data type that holds the device major and minor numbers for the file system from which the file comes. The /usr/include/sys/types.h header file includes a macro called makedev() that uses the high-order 16 bits of a 32-bit unsigned long to hold the device's major number, while the low-order 16 bits are used to hold the device's minor number. Major and minor device numbers for file systems can be found by doing a long listing (ls -l) of the /dev directory.

i_locks. A flag used for locking the inode while the kernel is updating it.

i_count. The current reference count for the file (i.e., how many vnodes currently point to the gnode within this inode).

i_dinode. A complete copy of the file's disk inode. By caching the disk inode within the in-core inode, the system need not access the disk inode each time it must be queried or updated.

Searching of the ICIT for a specific in-core inode is done using the file's inode number and the file system's major and minor numbers (i_number and i_dev fields). Searching on the inode number alone would not work as the ICIT contains inodes from many different file systems. As an example, there might be three different inodes #387 in the ICIT from three different file systems.

Figure 7.9 illustrates how in-core inodes are allocated when a file is opened. Three possible scenarios are provided:

Scenario A. A file is opened, causing the kernel to allocate a file table slot and a vnode. Upon searching the list of in-core inodes from previously opened files, the file's in-core inode is found. The in-core inode is put back in use by incrementing the reference count from zero to one. When the file is closed, the in-core inode's reference count is decremented to zero and the inode is placed at the end of the cache list of previously opened files. This means that the least recently opened files are found at the top of the cache list.

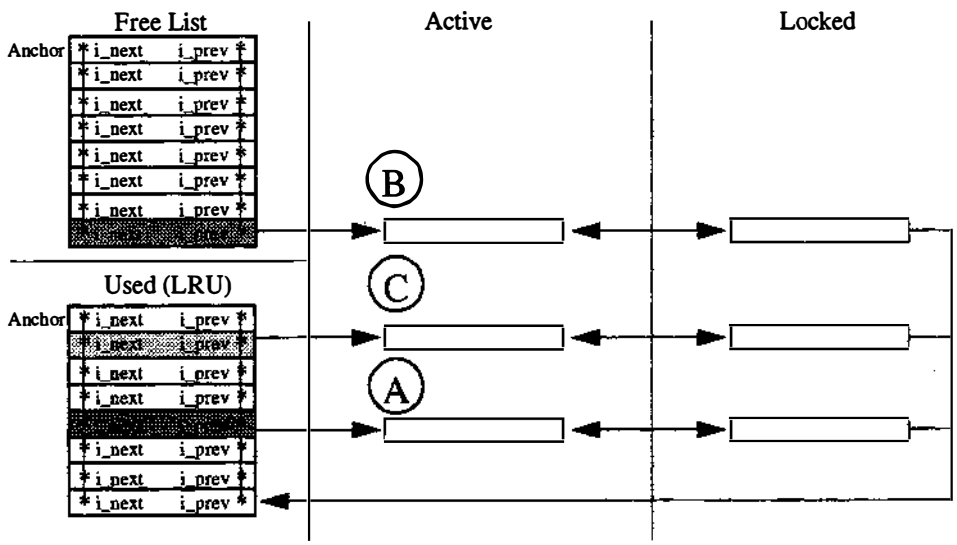

Figure 7.9 Allocation of in-core inodes.

Scenario B. A file is opened, causing the kernel to allocate a file table slot and a vnode. Upon searching the list of in-core inodes from previously opened files, the file's in-core inode cannot be found. This might occur if the file had never been opened since system start-up or it was opened a long time ago and its in-core inode was reassigned to another file. In this case, a new in-core inode is taken from the free list and assigned to the file.

Scenario C. A file is opened, causing the kernel to allocate a file table slot and a vnode. Upon searching the list of in-core inodes from previously opened files, the file's in-core inode cannot be found and there are no more in-core inodes on the free list. In this case, the kernel takes the least recently used in-core inode (from the top of the list of in-core inodes from previously open files) and reassigns it to the new file.

Note that the list of in-core inodes from previously opened files, as well as the free list of in-core inodes, are anchored by dummy in-core inodes.

In-core inode hashing

Over time, the in-core inode table can grow to be very large. Searching the ICIT sequentially would take far too much time. Therefore, in-use in-core inodes are placed in one of 512 hash queues according to the file's inode number. The header file /usr/include/jfs/inode.h defines each hash queue anchor (0-511) as struct hinode, which consists of a pointer forward (hi_forw) and a pointer backward (hi_back). It also defines the number for hashing along with the number of hash queues as NHINO using the macro:

```
#define     NHINO     PAGESIZE/sizeof(struct hinode)
```

...where PAGESIZE is defined as 4096 bytes and the size of the hinode structure is 8 bytes (two pointers).

Figure 7.10 illustrates the hash queues for in-core inodes that are active (reference count > 0) or have been active since system start-up. The formula for calculating an in-core inode's hash queue is also found in the /usr/include/jfs/inode.h header file in a macro:

```
#define     IHASH(X)     (&hinode[(int)(X) & (NHINO-1)])
```

The macro above determines the hash queue by bitwise ANDing the inode number with the value 511. Another way to look at this is that the hash queue is determined by the result of the inode number modulo (remainder when divided by) 512. As an example, inode #387 would be found on hash queue 387, since 387 % 512 = 387 (or 387 | 511 = 387). Inode #518 would be found on hash queue 6, since 518 % 512 = 6. The effect is to simply wrap around to hash queue zero whenever the inode number is evenly divisible by 512.

The hashing described above spreads the in-core inodes out over 512 different doubly linked lists, using the i_forw and i_back pointers. Now, only the in-core inodes for the appropriate queue need to be searched for the required match.

Linking an in-core inode to a file segment

The virtual memory manager allocates a 256-megabyte virtual memory segment for every opened ordinary file, regardless of the file's virtual file system

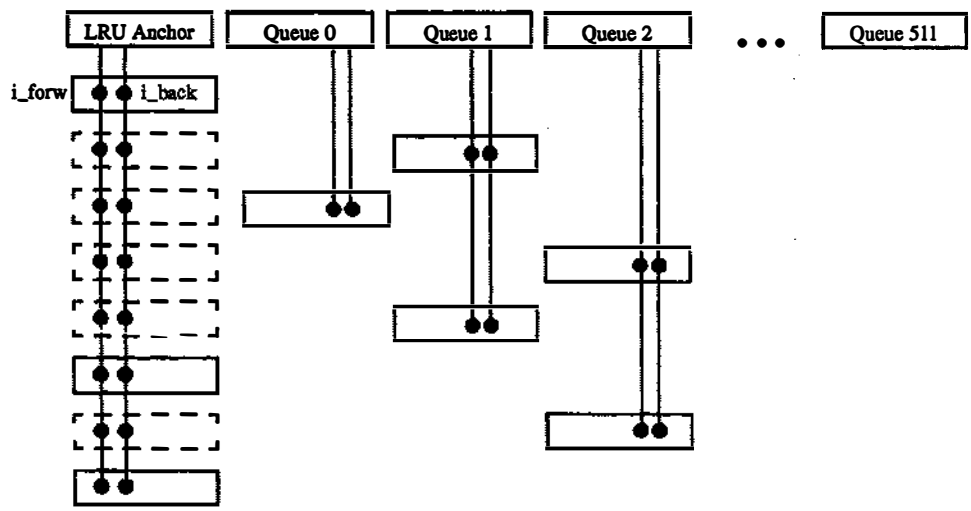

Figure 7.10 Hashing the in-core inodes.

origin (see Chap. 4). In other words, a local disk file (JFS), a remote disk file (NFS), or a CD-ROM file all have segments allocated to hold their pages. A VMM file segment has a unique segment ID number (SID) and an entry in the kernel's segment information table. Each file's gnode contains the segment ID number into which its file pages are mapped.

As mentioned previously, the gnode structure is defined in the /usr/include/sys/vnode.h header file. Interesting fields include:

gn_type. The type of object to which the gnode belongs (regular file, directory, special file, etc.).

gn_seg. An unsigned long that holds the segment ID number into which the file pages are mapped.

gn_vnode. A pointer back to the vnode for this gnode.

gn_filocks. A pointer to a filock structure which serves as the head of the list of read/write locks for this file. (More on file and record locking in the next section of this chapter.)

gn_data. A character pointer (caddr_t is a common type definition from /usr/include/sys/types.h that, as a character pointer, can point to any address in "core") that points to the object in which the gnode is found. For instance, in the case of a local disk file, the gn_data field in the gnode points to the in-core inode which contains the gnode.

The most important field listed above is the gn_seg field. The segment ID number contained in that field allows us to come full circle from where we started in the discussion of VMM file segments in Chap. 4. Figure 7.11 shows how the gnode links the file subsystem to the VMM.

One last note on the in-core inode table: The VMM does not use an external page table (XPT) to track the pages of a persistent storage segment, such as a

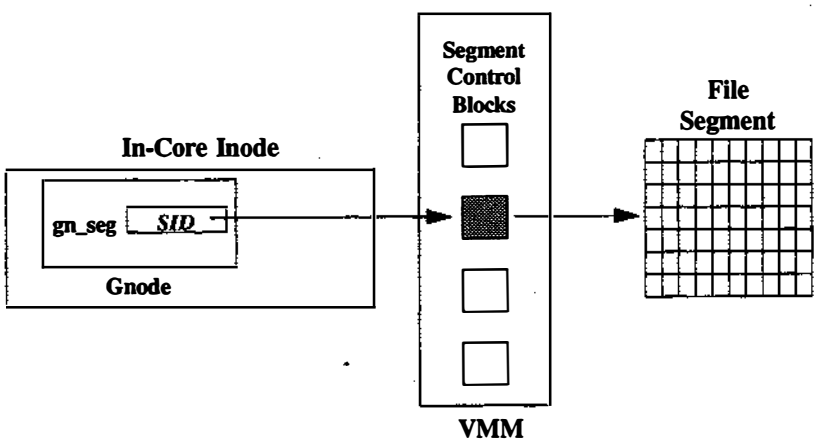

Figure 7.11 In-core inodes and VMM file segments.

segment used to map a local disk file. Instead, the VMM uses the data found in the in-core inode copy of the disk inode for the file. Specifically, the VMM uses the logical disk addresses for the data blocks of the file (see Sec. 4.5). In this way, the in-core inode superimposes itself over an XPT image when presented to the VMM.

7.6 File and Record Locking

New users to UNIX-based systems are often surprised to find that the operating system does not automatically implement some type of file locking scheme. Many have experienced the consequences of two or more users simultaneously updating the same file. UNIX considers file and record locking to be a task better left to the application layer; thus the kernel does no locking unless the application requests it. The kernel does, however, provide data structures and a programming interface for implementing file and record locking.

> **Author's Note:** Simultaneous file updates sometimes occur in my classes where two or more students accidentally log in using the same login account. They set about creating or modifying a shell script or source file, only to find that when they save their work, then edit it later, it has changed. This, of course, happens because some other student has saved to the same file, thus clobbering the edits made by the first student. I explain to them that UNIX does not automatically enforce file or record locking because simultaneous updates might be desired by the users. But I also mention that, if you listen very carefully, you can hear UNIX laughing at you when you do this.

Before explaining how file and record locks are implemented in AIX 3.2, a discussion of lock types must be given. Locks are enforced in one of two ways. Advisory locks are implemented between two or more cooperating processes. Before a cooperating process attempts to read from or write to a region of the file, it checks for the existence of a lock. If a lock already exists, the process can either sleep (block) until the lock becomes available or return an error condition to the application. If the region is not already locked, the process can then set a lock on the region. The kernel guarantees that the lock test and set operation is performed atomically to prevent a race condition. The problem with advisory locks is that the kernel does nothing to prevent a third process which is not part of the cooperating set of processes from simply reading or writing within a locked region of the file. Enforced locks, on the other hand, cause the kernel to block any noncooperating process that attempts to read or write to a locked file region, until the lock is released. By default, AIX 3.2 locks are advisory locks. The kernel will implement enforced locks for any data file that has its SGID bit set. The SGID bit can be set from the shell by using the chmod command or from within a C program by using the chmod() subroutine. Figure 7.12 illustrates how this bit is set. See the manual page for chmod for more information.

In addition to advisory and enforced locks, the kernel supports read and write locks. If a process sets an enforced write lock on a region of a file, other

```
# ls -l
...    ...  ...   ...    ...    ...          ...
-rw-rw-rw-  2   slee  staff  14964  June 13 07:45  ourdata
...    ...  ...   ...    ...    ...          ...
```

SGID bit turned off – Locks are advisory

```
# chmod 2666 mydata
# ls -l
...    ...  ...   ...    ...    ...          ...
-rw-rwSrw-  2   slee  staff  14964  June 13 07:45  ourdata
...    ...  ...   ...    ...    ...          ...
```

SGID bit turned on – Locks are enforced

Figure 7.12 Advisory locks and enforced locks.

processes cannot read or write to that region until the lock is released. If a process sets an enforced read lock on a region of a file, other processes can still read from that region, but they cannot write to that region until the lock is released. The same rules apply for cooperating processes when advisory locks are used. Write locks are usually implemented by a process that is updating a file region to prevent other processes from reading data that are in a transient state. Read locks are usually implemented by a process that is reading a file region and does not want other processes changing the data while it is being read.

To illustrate file and record locking, we'll use an example of a data file, called "ourdata," which has been opened for reading and writing by two different processes. The first process (A) decides to lock bytes 32 through 78 of the file. The second process (B) chooses to lock bytes 121 through 180 of the file. Finally, while still holding its first lock, process A locks bytes 225 to the end of the file. Figure 7.13 shows how the regions of the file are locked. Notice that the last lock goes from byte 225 to 0. The application programming interface allows the specification of 0 to indicate "to the end of the file." This way, the size of the file need not be known. Locking bytes 0 through 0 results in locking the entire file.

The filock and flock structures

The kernel maintains a table, called the lock list, for implementing file and record locking. Lock list entries are struct filock as defined in the header file /usr/include/sys/flock.h. Each filock structure includes an embedded flock structure (also defined in flock.h) to describe the lock instance. The flock structure is called "set" within the filock structure. Interesting fields in the flock structure include:

l_type. A short that holds the lock type (i.e., read or write lock).

l_start. An off_t data type that indicates the offset, in bytes, relative to l_whence for the starting location of the lock.

l_whence. A short integer that holds the prospective from which the l_start determines the starting location of the lock. Possible values are 0 = beginning of file, 1 = current read/write offset, and 2 = end of file. In other words, if a lock is set with l_whence = 0 and l_start = 128, the lock starts at byte 128 of the file. On the other hand, if a lock is set with l_whence = 1, l_start = 128 and the current read/write offset at 200, the lock starts at byte 328.

l_len. An off_t data type that indicates the length, in bytes, of the lock. A zero value means to the end of the file.

l_pid. A pid_t data type that holds the pid of the locking process.

Incidentally, the flock structure is also used by some of the application programming interface routines, which are described shortly. Interesting fields in the filock structure include:

set. The embedded flock structure described above.

prev and next. Pointers to the next and previous locks in the lock list for this file.

The file subsystem keeps track of file locks via a pointer from the gnode to the filock structure in the lock list. The gnode pointer gn_filocks (see /usr/include/sys/vnode.h) provides this link. The example given earlier showed two processes setting three locks on the same file. However, there is only one

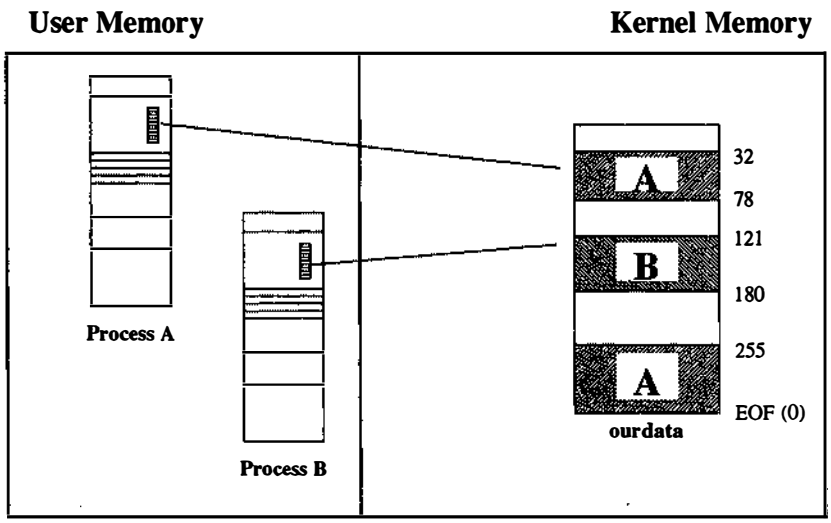

Figure 7.13 File locking example.

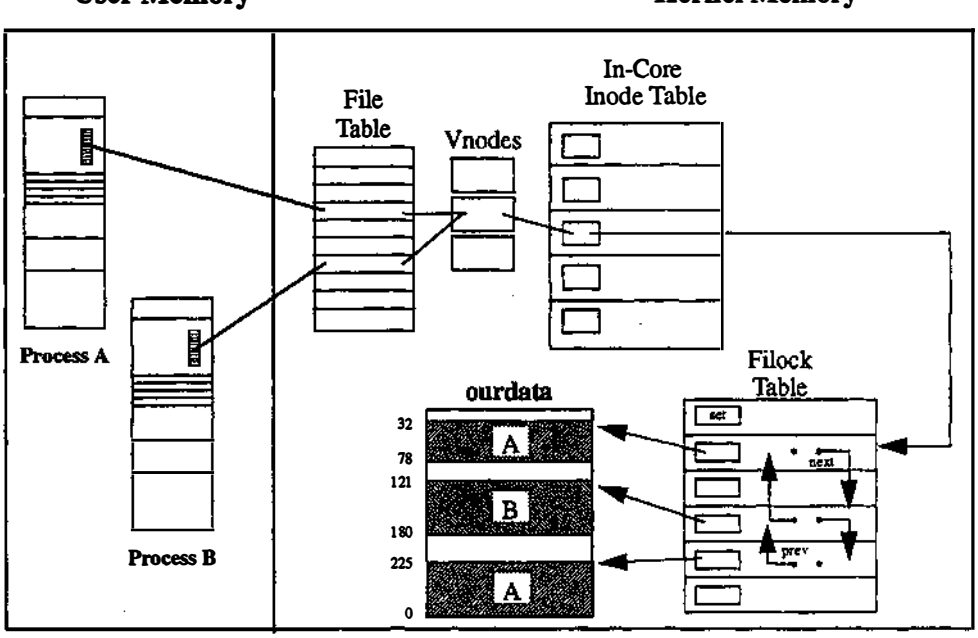

Figure 7.14 JFS block allocation.

pointer from the gnode to the lock list. Multiple locks per file are accomplished by having the gnode point to the filock structure (in the lock list) that describes the first lock on the file. Each filock structure then uses the pointer called next to point to the next filock structure in the lock list that describes a lock for the same file. The next and prev pointers form a doubly linked list of filock structures. Figure 7.14 illustrates how this works.

AIX 3.2 provides four different subroutines for implementing file and record locks:

lockf(). This subroutine comes from System V UNIX and is very simple to use, albeit lacking in capabilities. It includes options for locking and unlocking file regions but does not provide a distinction between read locks and write locks. All locks are assumed write (exclusive) locks. The lockf() subroutine also does not provide a parameter for indicating where the lock should start. The lock region starts at the file descriptor's current read/write offset. Therefore, the application must call lseek() to position the read/write offset prior to calling lockf(). If an attempt is made to lock a region which is already locked, this subroutine will block until the lock is available.

lockfx(). This subroutine takes, as a parameter, a pointer to a flock structure, as defined in /usr/include/sys/flock.h. The application must initialize the values in a flock structure prior to calling lockfx(). The flock fields define

the starting location, size, and type of lock. The lockfx() subroutine includes a parameter to specify the command. Command choices include options to have the process sleep (block) or have the subroutine return a -1 if the lock is not available.

flock(). This subroutine comes from 4.3 BSD. It provides for shared or exclusive locks and instructions on how to act if the lock is not available.

fcntl(). This multipurpose subroutine provides control over opened files. It has many uses, one of which is record locking. The fcntl() subroutine requires a pointer to a flock structure, which must be initialized by the application prior to calling fcntl(). This subroutine includes options for read and write locks and is defined by POSIX.1 and XPG3 standards.

One very nice feature of AIX 3.2 file and record locking is that the four subroutines described above work with one another. Since they are interfaced with a common set of kernel data (i.e., the kernel's lock list), they can be used interchangeably by different processes.

Finally, since only one process can lock a region of a file at a time, file and record locks are not inherited by the child process from the parent process when a fork() occurs. All file and record locks owned by a process are released when the process closes the file or when the process terminates.

7.7 File I/O Subroutines

This section describes many of the common subroutines used by applications to perform file I/O. While code examples are given, the focus is on what happens in the AIX 3.2 kernel when these calls are made. Consult the manual pages or InfoExplorer for details on the syntax and use of each of these subroutines.

The open() and dup() subroutines

The open() subroutine is a system call that opens a file for reading and writing. The application supplies the path name of the file and appropriate flags for access. Unless the O_CREAT flag is given, the kernel assumes that the file exists. If it does not, open() fails. The open() subroutine also fails if the permissions of the file do not allow the user the requested access, or if the process has already opened the maximum number of files allowed per process (2000). If the open() subroutine fails, it returns a value of -1; otherwise it returns the number of the file descriptor assigned to the file, from the process's file descriptor table. Remember, the lowest available file descriptor is always assigned.

The open() subroutine creates a new entry in the kernel's file table and assigns the address of the new entry to the fp pointer in the process's file descriptor table.

If the file has not already been opened by another process, or the calling process, a new vnode is created for the file. The f_uvnode pointer in the file

table entry is set to the address of the new vnode. If the file has already been opened by another process, or the calling process, the f_uvnode pointer in the file table is set to the address of the existing vnode.

Next, for local disk files, the in-core inode table is searched (see Sec. 7.5) to see if the file's in-core inode is already in the cache. This search is unnecessary, of course, if the file is already open. If a cache miss occurs, a new in-core inode is assigned.

Finally, if the file was not already open, the VMM assigns a segment and maps the data blocks of the file to the pages of that segment. Figure 7.15 provides an example. The dup() subroutine copies the pointer stored in a specified file descriptor to the lowest available file descriptor. This allows two or more file descriptors to share a common read/write offset. AIX 3.2 also supports the dup2() subroutine, which allows the calling application to specify not only the desired file descriptor to copy but also the target file descriptor to assign. This is one example of where the lowest available file descriptor need not be assigned.

The read() and write() subroutines

The syntax and operation of the read() and write() subroutines are very similar. The read() system call copies data from a memory mapped file page into a

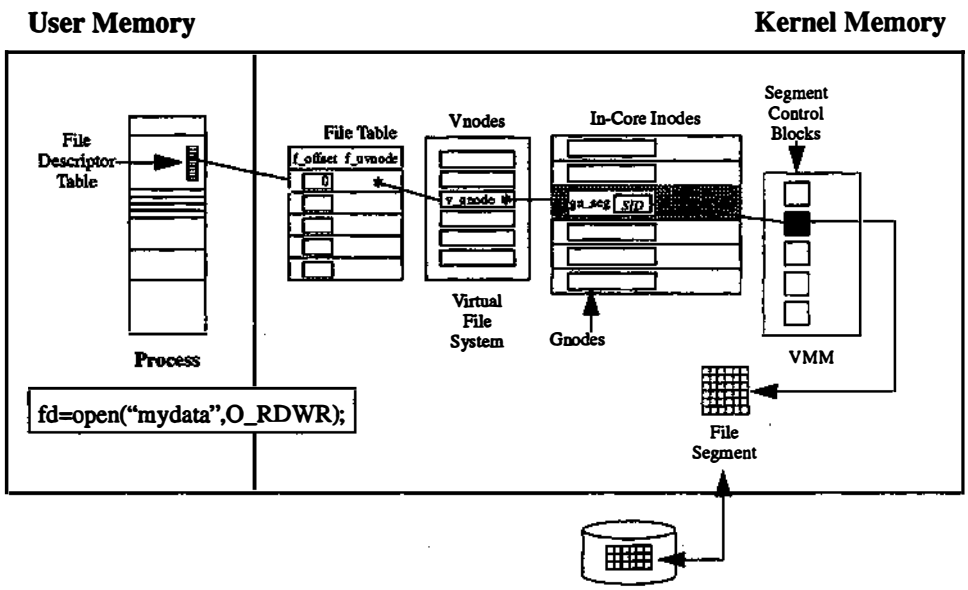

Figure 7.15 The open() subroutine.

specified buffer within the process's data segment. If the file page is not found within real memory, the virtual memory manager instructs the logical volume manager to page in the missing page. The write() system call copies data from a specified buffer within the process's data segment to the memory mapped page of the file. The write() system call is not responsible for actually writing the data to disk. That operation is performed sometime later by the virtual memory manager and the logical volume manager.

The application tells each system call how many bytes to read or write. This causes the read/write offset in the file table to be incremented. The read() system call returns the number of bytes actually read. It returns a zero upon encountering the end of the file. It returns a -1 upon failure. The write() system call return values are the same as for read(). Figure 7.16 illustrates a series of file reads and writes.

This is a good time to discuss the differences between the open(), read(), and write() system calls and the fopen(), fread(), and fwrite() library routines. Programmers tend to use one set over the other for various reasons, but there are some interesting points about how each set of routines operates. First, the open(), read(), and write() subroutines are AIX 3.2 system calls. This means that they execute in system mode (kernel memory) and that a mode switch is required each time one of these calls is made by an application. On the other hand, fopen(), fread(), and fwrite() are library routines whose code becomes part of the application's user mode code at bind time. These routines run in

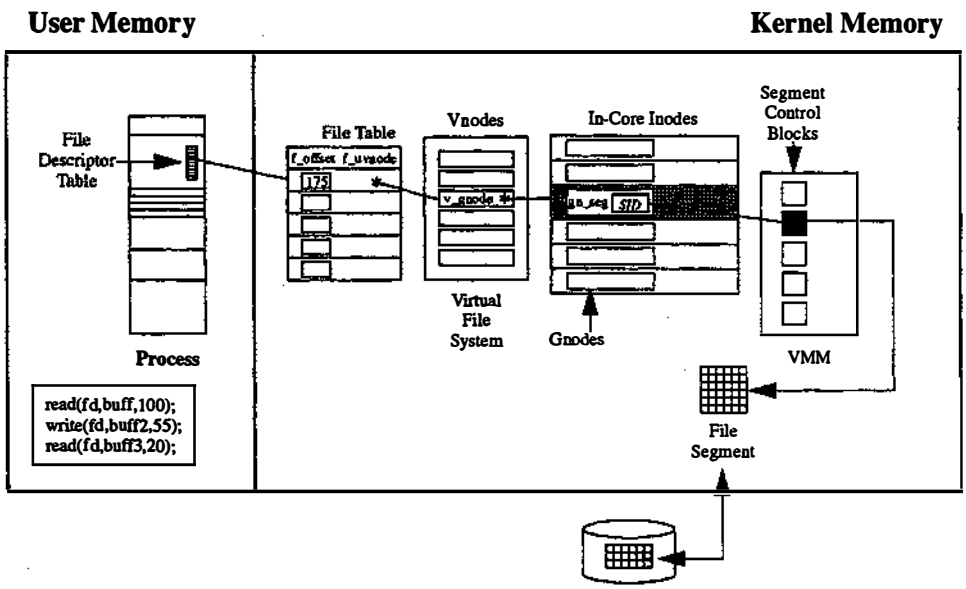

Figure 7.16 Direct data block accessing scheme.

user mode. Obviously, the fopen(), fread(), and fwrite() library routines do perform system calls from time to time, but these calls are made from within the library routines.

When an application issues an fopen() call, a pointer to a struct FILE is returned. The struct FILE defined in the /usr/include/stdio.h header file includes a 4096-byte buffer in the user space of the application for holding the latest page read from the file. The fopen(), fread(), and fwrite() routines tend to be much more efficient for smaller size reads and writes, because of the user mode buffer, than the open(), read(), and write() system calls. Figure 7.17 includes code and a drawing to illustrate how inefficient the open(), read(), and write() system calls can be.

> **Author's Note:** I am using an extreme example here. Any applications programmer who writes a program that reads a million-byte file one byte at a time deserves the performance they get. However, the example still drives home the point of the overhead involved in system calls. I would advise the skeptical reader to take the time to examine data derived from using various read sizes as the discussion continues.

Note that each time through the loop, the application makes a system call, and thus a mode switch. Recall from Chap. 3 the work involved in vectoring to the kernel code of a system call, switching to the per-process kernel stack, then returning to user mode. Now imagine that happening one million times. That's what this program does. It's true that the kernel pages in an entire 4-kb page

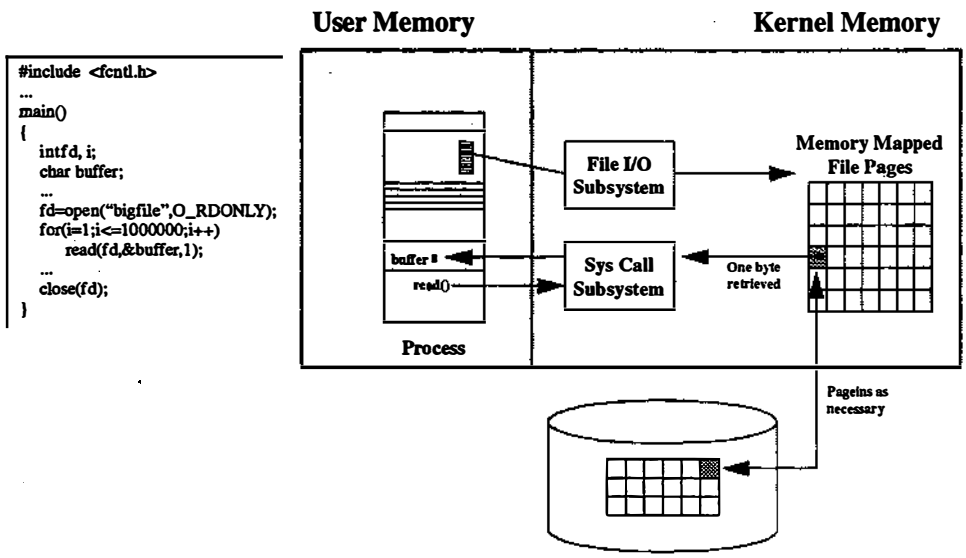

Figure 7.17 Using file I/O system calls.

into the file segment as the file is read (actually, the VMM performs read ahead to help speed the operation when it realizes that the file is being read sequentially). But the read() system call is copying only one character at a time from kernel memory to user memory.

> **Author's Note:** A student once gave me an interesting analogy to the way this program example works, and I've been known to use it from time to time. Imagine that you've gone to your favorite movie theater to see *Rocky XXV*, and you decide to visit the concession stand for a tub-o-popcorn. Upon ordering the popcorn, you are told that the theater has a new policy which will not allow you to take the tub-o-popcorn into the auditorium. You may, however, take a single kernel (no pun intended) of popcorn into the auditorium and leave the tub at the concession stand. Grabbing a single kernel, you rush into the theater, take your seat, and eat the piece of popcorn. You then hurry back out to the concession stand, grab another piece of popcorn, and run back into the auditorium. It would take quite a while to finish the tub, and you'd probably miss a good part of the film. This is certainly not an efficient way of doing things. One can apply this example to a program that uses the read() system call many times to read a large file in small numbers of bytes (again, no pun intended). Each kernel of popcorn represents a character from the file, and the tub represents a page from the VMM file segment. The auditorium represents the system's user memory, and the concession stand represents the kernel memory. Increasing the number of popcorn kernels, say to a handful at a time, helps a little, but not very much. Worst of all, we're probably talking about that new low-fat, air-popped popcorn. Yuk!

Figure 7.18 provides a code example that accomplishes the same task as the program in Fig. 7.17, but it uses the fopen() and fread() library routines. The fread() routine copies an entire 4096 bytes from kernel memory to user memory as needed, then fetches the characters from the FILE buffer upon each call. This means far fewer system calls and much less overhead.

> **Author's Note:** Here we have a theater that allows its patrons to take the entire tub-o-popcorn into the auditorium and eat the popcorn one kernel at a time from the tub.

The point of this example is not to persuade programmers not to use system calls. In fact, if one increases the number of bytes read each time through the loop beyond the 4-kb buffer size of the FILE structure, the read() system call becomes more efficient. The point is to illustrate how an understanding of the operations going on in the kernel can help a programmer make better decisions about which routines to use.

The lseek() subroutine

As mentioned in the previous section, each read() or write() system call performed on a given file increments the read/write offset for that file. Sometimes a program may wish to change the read/write offset without actually requesting any I/O. This can be accomplished with the lseek() subroutine. The lseek()

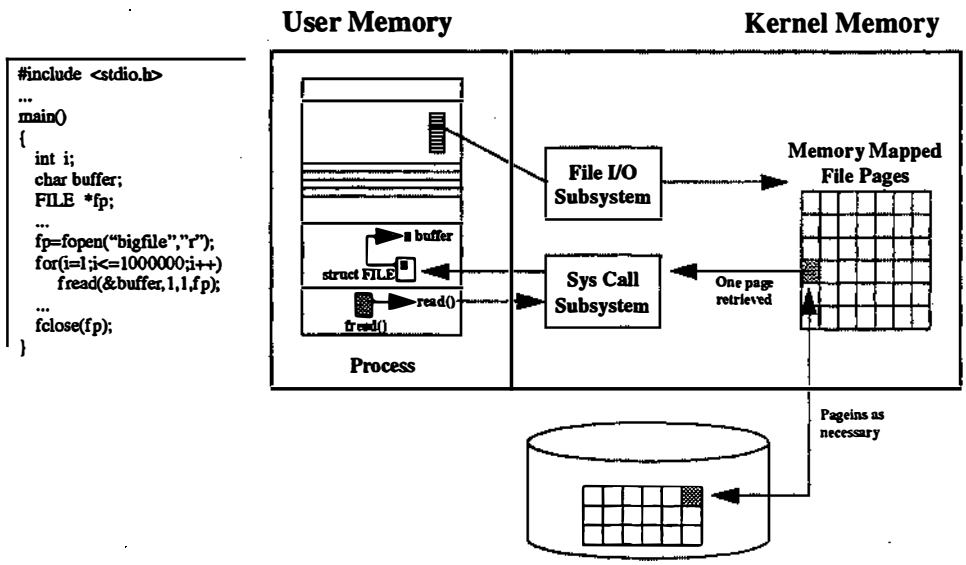

Figure 7.18 Using file I/O library routines.

subroutine simply changes the value of the f_offset field in the file table. It takes three parameters: the file descriptor on which to seek, the number of bytes by which to increment or decrement the read/write offset (this parameter is taken as an off_t data type, which is a signed integer, so negative numbers are supported), and a whence value, which specifies the perspective of the seek. There are three possible whence values, with symbolic constants defined in /usr/include/unistd.h. Table 7.1 describes the whence values. Figure 7.19 gives an example of the lseek() subroutine.

An interesting thing about lseek() is that an application can seek beyond the end of a file. This does not extend the size of the file, however. The file is only

TABLE 7.1 Whence Values for lseek()

Numeric Values	Symbolic Values	Description
0	SEEK_SET	From start of file (absolute position)
1	SEEK_CUR	From current offset (relative position)
2	SEEK_END	From end of file (absolute position)

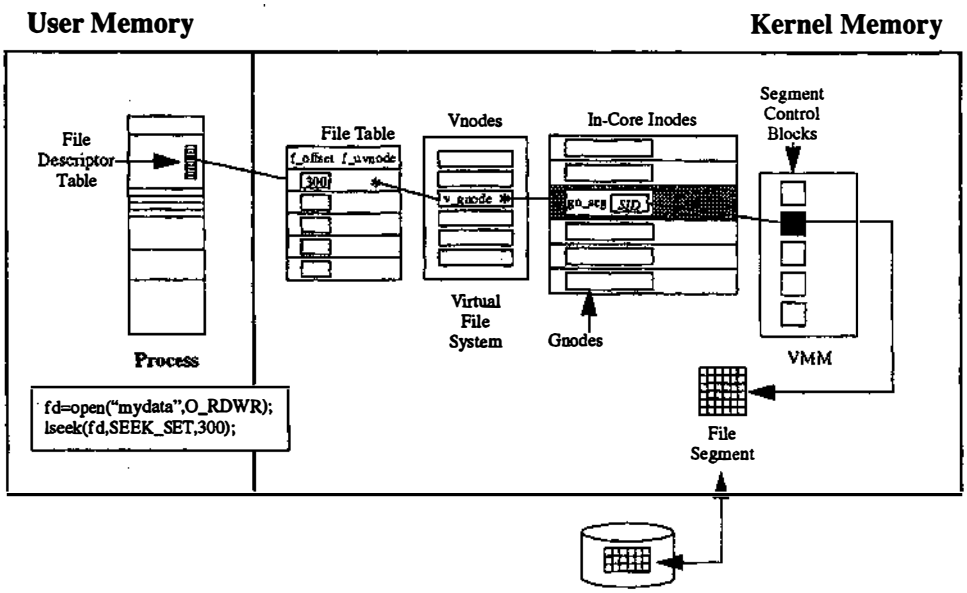

Figure 7.19 The lseek() subroutine.

extended when a write occurs beyond the end of the file. So what happens when an application seeks hundreds of millions of bytes past the end of a file, then writes a small amount of data? AIX creates a sparse file.

The code example in Fig. 7.20 creates a file, then loops 11 times. Each time through the loop, the application writes the word "hello," then uses the lseek() subroutine to increment the read/write offset by 100,000,000 characters. The result is a file that has a logical size of over 1 gigabyte. But AIX does not allocate physical disk data blocks to hold the null data found between the "hello"s. Therefore, the physical size of the file is only twelve 4-kb data blocks (an indirect block is allocated since the size of the file is greater than 4 megabytes).

> **Author's Note:** Notice that the program loops 11 times, but only 10 "gaps" end up in the file. Since there was no write operation after the final lseek(), the extra offset is truncated when the file is closed.

The example above illustrates the difference between the logical size of a file, based on the byte count up to the last location of actual data, and the physical size of a file, which is the actual number of data blocks used by the file. It's possible to store a file with a logical size of 1 gigabyte in a file system whose size is only 4 megabytes. The ls -l command shows the logical size of a file in bytes. Use the ls -s command to see the actual number of data blocks used by a file, but remember that the ls -s command reports in 1-kilobyte block sizes (for

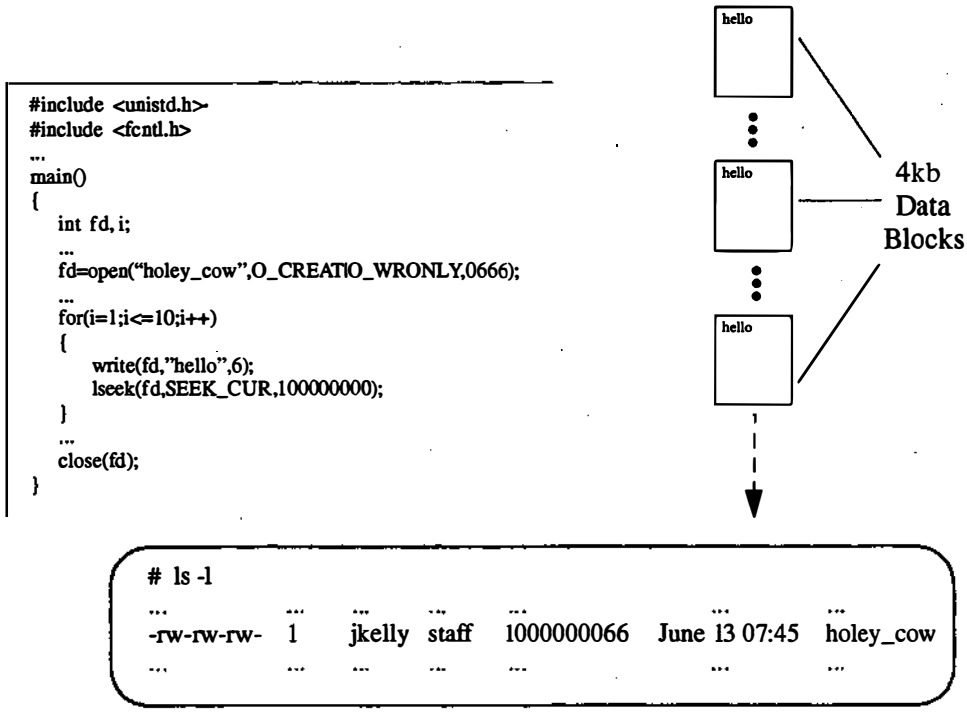

Figure 7.20 Creating a "sparse" file.

POSIX compliance) so the reported number must be divided by 4 for AIX 3.2. A file's disk inode keeps the logical size of the file in the di_size field and the number of data blocks used in the di_nblocks field.

The close() subroutine

The close() subroutine closes an instance of an opened file. It takes a file descriptor number as a parameter and nulls the pointer stored in the fp field of that file descriptor (see /usr/include/sys/user.h), thus dissolving the link between the file descriptor in a process's file descriptor table and the corresponding entry in the kernel's file table. It decrements the value in the f_count (reference count) field of the file table slot (see /usr/include/sys/file.h). If the f_count field reaches zero, the file table slot is freed. In this case, the value of the v_count field in the corresponding vnode is decremented (see /usr/include/sys/vnode.h).

If the v_count value reaches zero, the file has been closed by all processes that once had it opened. The vnode is then freed, and the corresponding in-core inode's i_count is set to zero (see /usr/include/jfs/inode.h). The in-core inode is not freed but is now considered inactive and placed at the end of the least-

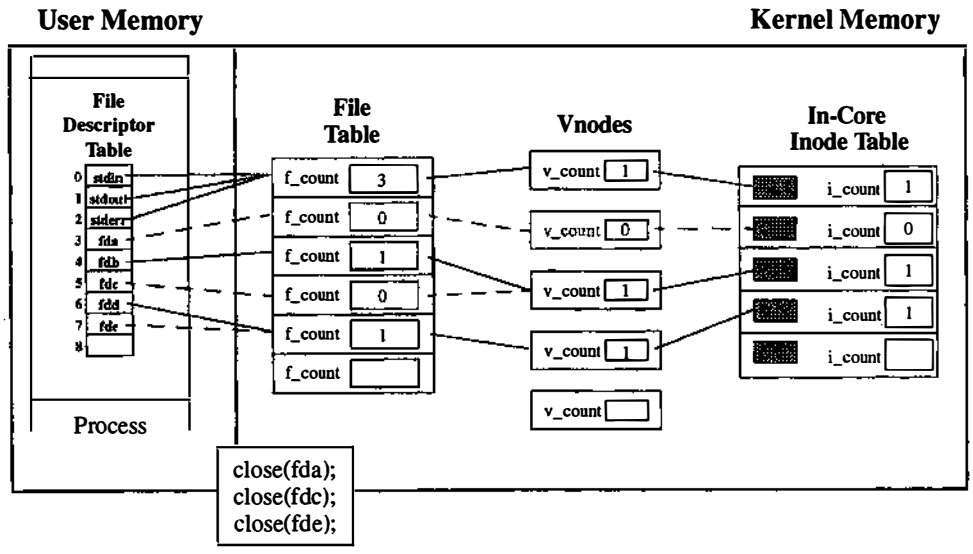

Figure 7.21 Closing files.

recently-used chain (see Sec. 7.5). Figure 7.21 illustrates examples of closed files. All files held open by a process are automatically closed when the process terminates.

The link() and unlink() subroutines

The link() subroutine is similar to the ln command. It links an inode to a directory. The link() subroutine is called by creat() or open() when the O_CREAT flag is used. It can also be called by an application to create a hard link to an existing file. The link() subroutine increments the value of the di_nlink field in the disk inode structure (see /usr/include/jfs/ino.h).

The unlink() subroutine dissolves the link between a directory entry and a disk inode. It decrements the value of the di_nlink field in the disk inode. This is what occurs when a user removes a file via the rm command. If the value of the di_nlink field reaches zero and the file is closed by all processes, the file is considered removed. The inode and all associated data blocks are freed by the system.

Figure 7.22 illustrates an interesting code example that creates a temporary file that is automatically removed when the process terminates. Once unlinked, the file does not appear in any directory but exists as long as it is kept open by the process.

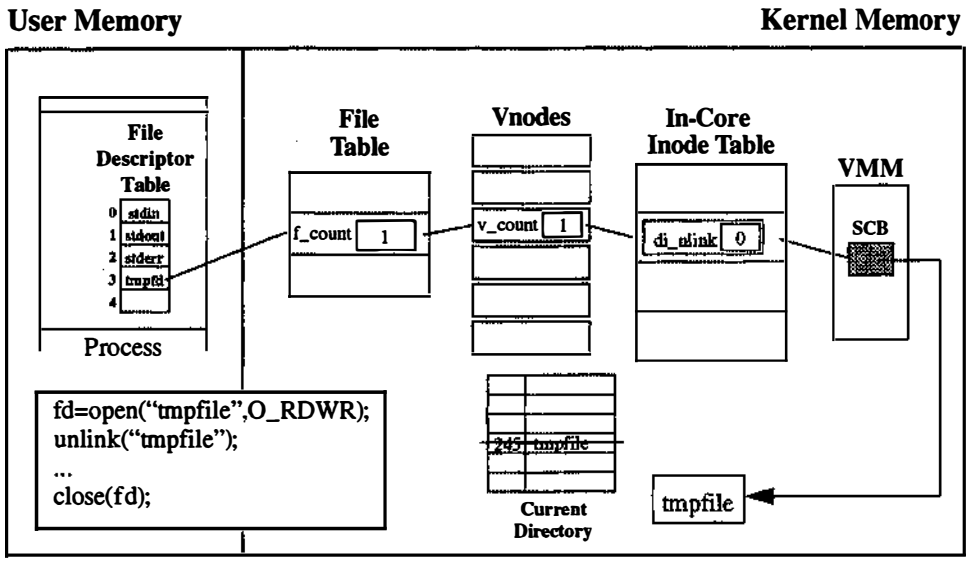

Figure 7.22 The unlink() subroutine.

7.8 Memory Mapped Files

This section describes how AIX 3.2 performs implicit and explicit memory mapping of files. Implicit memory mapping occurs automatically when an ordinary file is opened. Explicit memory mapping occurs when a process calls either shmat() or mmap() to map an already opened ordinary file into the process's user space. This section compares both of these memory mapping techniques to the "traditional" style of using kernel buffers as a disk cache.

Kernel buffers—the traditional approach

Many older-styled UNIX-based systems reserve a portion of real memory for pinned buffers that hold the most recently accessed data blocks from local disk files. The buffers are allocated from kernel memory (/dev/kmem). These buffers were called pinned K-buffers or "pink buffers" in AIX Version 2 on IBM's RT system. The number of buffers is a tunable parameter that must be adjusted to adequately accommodate caching of data blocks while not taking up so much real memory as to hinder the execution of processes. Figure 7.23 illustrates the use of disk cache buffers.

> **Author's Note:** The sar (system activity report) tool on System V UNIX systems provides valuable information on the percentage of read and write cache hits for data blocks found in the disk cache buffers. The sar -b command gives this information. Since AIX 3.2 does not use the disk cache buffers for ordinary file I/O, the

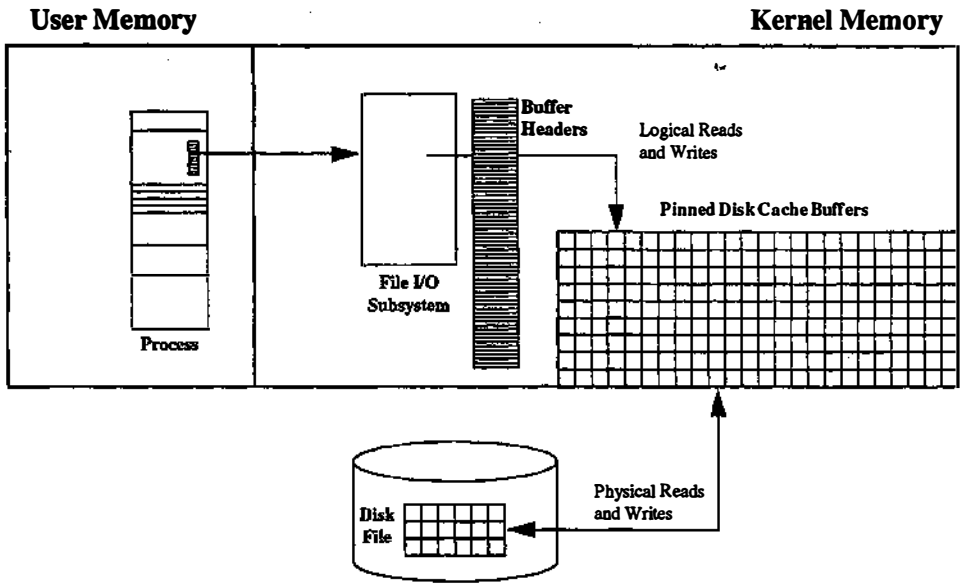

Figure 7.23 "Old style" kernel disk cache buffers.

sar -b command reports 0% read and write cache hits. This is one of many examples where traditional UNIX utilities are obsolete in AIX. Care must be taken when information is obtained from generic documentation about things like performance management. Unless one fully understands the workings of the AIX kernel, faulty assumptions can be made.

AIX 3.2 still has kernel buffers, but their use is limited to holding JFS superblocks during file system mounts and handling buffered I/O of non-JFS logical volumes. An example of the latter occurs when an application opens a logical volume (raw disk partition) or other block device, such as would be the case for a call to open ("/dev/hd15",O_RDWR). This type of operation is typically found in data base applications that request a raw disk partition. AIX 3.2 allocates twenty 4-kb buffers by default, but the number of buffers is tunable via the system management interface tool (SMIT) or the chdev command. AIX also allocates one buffer header for each buffer. To view the number of assigned buffers, select the "System Environments" option from the SMIT main menu, then select the "Change/view characteristics of the operating system" option. Look for the "Buffers=" attribute. It can be changed from this screen.

Users of commercial data base packages or in-house applications that read and write to raw disk partitions should consider tuning the number of kernel buffers to meet their system's needs. Consult the "AIX Performance Tuning Guide" under InfoExplorer for more information on tuning the number of kernel buffers.

Implicit file mapping

Actually, the concept of implicit file mapping has already been discussed in this book, but it is necessary to review implicit file mapping prior to introducing the concept of explicit file mapping. Implicit (or automatic) file mapping occurs in AIX 3.2 whenever an ordinary file is opened. Recall that the virtual memory manager allocates a 256-Mb segment to hold the mapped pages of the file. The segment ID number is held in the gn_seg field of the file's gnode structure (see /usr/include/sys/vnode.h). An entry in the kernel's segment information table keeps track of the segment. Figure 7.24 illustrates implicit file mapping.

The main benefit of file mapping over the traditional kernel buffers is that file mapping allocates virtual segments and pages dynamically as files are opened, then releases them when files are closed. This fluid approach avoids the need to have a static set of pinned buffers taking up a large portion of real memory frames. The system adjusts to the demands for file memory versus computational memory by letting the VMM handle the file I/O.

Explicit file mapping using shmat()

AIX 3.1 and AIX 3.2 allow a process to explicitly memory map a file into that process's own address space by using the shmat() system call.

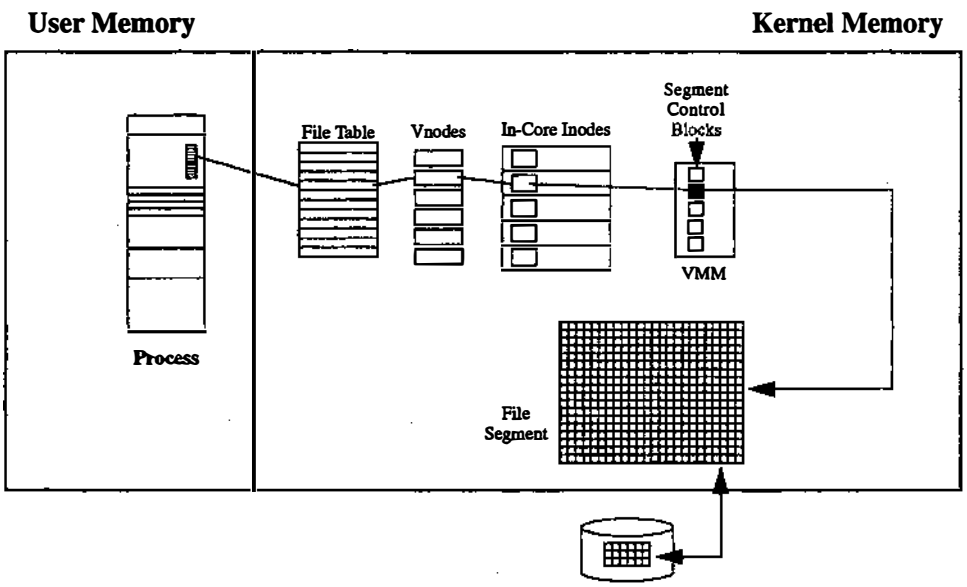

Figure 7.24 Implicit file mapping.

Author's Note: The shmat() system call was introduced with the System V IPCs (interprocess communications) from AT&T. It allows processes to share memory regions (shmat means "shared memory attach"). While AIX uses shmat() for shared memory IPC (see Chap. 10), it also allows shmat() to be used for explicitly mapping files. Most UNIX-based systems use the mmap() system call to perform explicit file mapping and the shmat() system call only for shared memory IPC. AIX 3.1 does not include the mmap() system call, leaving shmat() as the only option for mapping files. AIX 3.2 introduced mmap() to provide portability. The interesting thing is that AIX 3.2 allows a programmer to use shmat() to do shared memory IPC or memory mapped files, or mmap() to do shared memory IPC or memory mapped files. AIX 3.2 programmers concerned with portability and standards should use shmat() only for shared memory IPC and mmap() only for memory mapped files. If portability is not an issue, the choice of which interface to use can be based on the features of each.

The shmat() subroutine maps the entire file segment into one of the calling process's eight available segments (segments 3 through 10). Figure 7.25 illustrates how shmat() maps a file into the process's address space. Regardless of the size of the file that is smaller than a single segment (256 Mb), the entire process segment is used by shmat(). In other words, when using shmat(), only one file can be mapped into any single process segment at a time. This limits the number of files that can be "shmatted" to eight at a time.

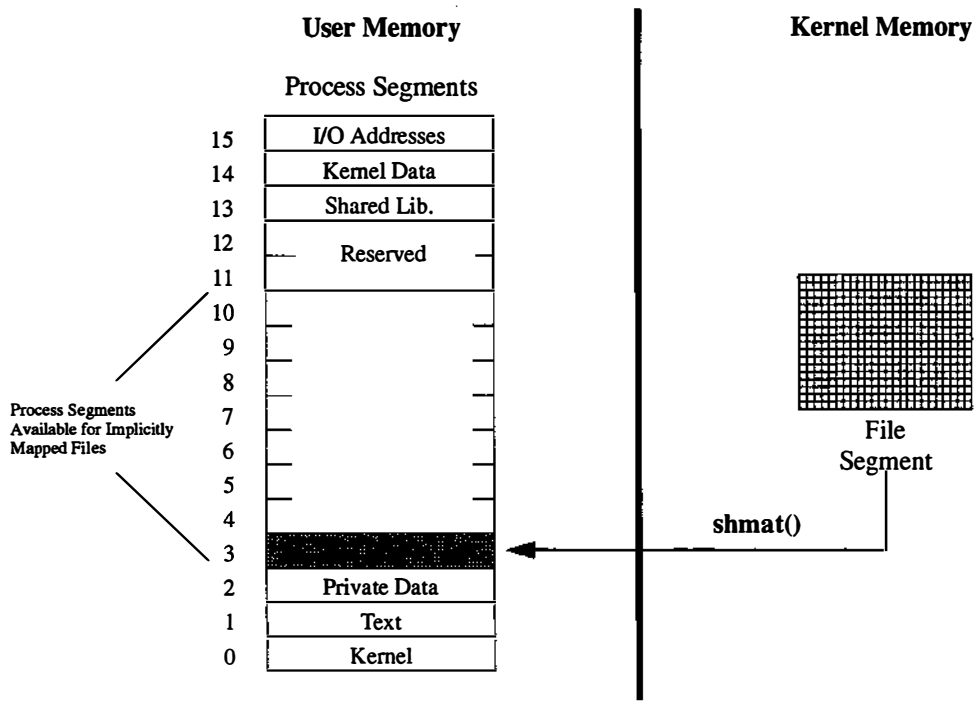

Figure 7.25 Explicit file mapping using shmat().

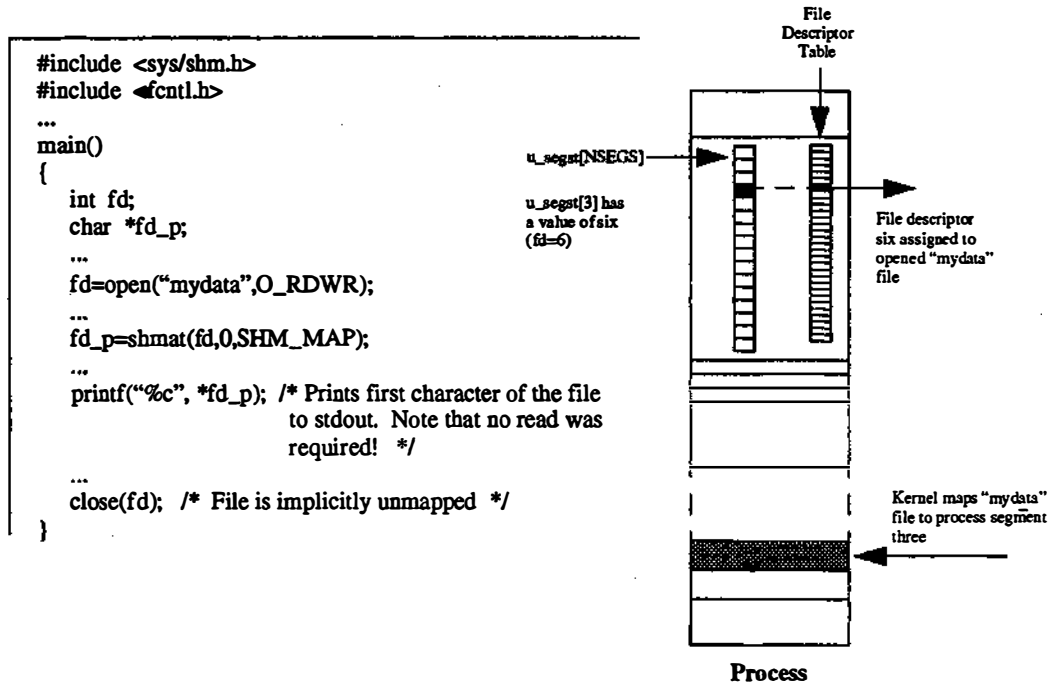

Figure 7.26 An example of explicit file mapping using shmat().

The shmat() subroutine can map files that are larger than a single segment (>256 Mb). It simply uses additional segments from the process's user space.

> **Author's Note:** Some IBM documentation claims that segments 3 through 12 are available for memory mapped files. Other documentation states that segments 11 and 12 are reserved by the operating system. I have found that I can successfully map files to segments 11 and 12 but would advise against it since IBM does not seem to guarantee that this capability will always be supported.

Figure 7.26 shows a code example of explicit file mapping with shmat(). The file is first opened in the usual manner. The shmat() subroutine is then called. The parameters include the file descriptor and the SHM_MAP flag, which tell shmat() that it is mapping a file rather than attaching a shared memory segment. The second parameter of shmat() is very important. It tells shmat() the virtual address within the process at which to start mapping the file (i.e., the starting location of the process's segment). A zero value for this parameter tells shmat() to let the kernel decide which of the process's user segments (3 through 10) to use. The kernel normally selects the lowest available segment.

The state of each of a process's 16 segments is maintained by an array of segstate structures in the process's user area. The array, defined in /usr/

writeit.c

```
#include <fcntl.h>
...
main()
{
   int fd, i;
   char c='X';
   ...
   fd=open("file_w",O_CREATIO_WRONLY,0666);
   ...
   for(i=1;i<=1000000;i++)
         write(fd,&c,1);
   ...
   close(fd);
}
```

```
# time writeit
real    4m57.32s
user    0m26.18s
sys     4m11.42s
```

mapit.c

```
#include <sys/shm.h>
#include <fcntl.h>
...
main()
{
   int fd, i;
   char *fd_p;
   ...
   fd=open("file_m",O_CREATIO_WRONLY,0666);
   ...
   fd_p=shmat(fd,0,SHM_MAP);
   ...
   for(i=1;i<=1000000;i++)
            fd_p++='X';
   ...
   close(fd);
}
```

```
# time mapit
real    0m0.29s
user    0m0.26s
sys     0m0.03s
```

Figure 7.27 Explicit vs. implicit file mapping.

include/sys/user.h, is called u_segst[NSEGS]. Both NSEGS and the segstate structure are defined in /usr/include/sys/seg.h. NSEGS is defined as 16. A segstate structure contains a segflag field, which describes the type of segment, and a union called u_ptrs that holds either a file descriptor in a field called mfileno or a pointer to a shared memory descriptor called shmptr. (See Chap. 10 for information about shared memory descriptors.)

The shmat() system call returns a character pointer to the starting location of the mapped segment. Now the process can access and modify the data of the file without performing read() and write() system calls. It is done directly through the memory pointer. In effect, the file has become part of the process's data. The overall improvement in file I/O can be astonishing. Figure 7.27 shows a side-by-side comparison of implicitly mapped file I/O and explicitly mapped file I/O. The results of timing the applications as each writes 1,000,000 characters to a file, one character at a time, are taken from an actual test case. Your mileage may vary! The main reason for the difference in time comes from the reduction in system calls (note the lower %sys time). The process is performing loads and stores instead of reads and writes.

> **Author's Note:** Yes, folks, I've done it again. I have provided an example of a program that creates a file, then writes one million characters to it, one character at a time. For those of you who didn't catch the example earlier in this chapter where I did a similar thing with read() and fread(), I'll repeat that this example, while

slanted in the favor of explicitly mapped files, is justifiable. Increasing the write buffer size of the writeit.c program does improve its overall response time, but it never matches that of mappit.c. I simply went for greater theatrics here.

Author's Note: When I present this example in my classes, I am sometimes confronted by students who say that this example is not fair because the writeit.c program is writing to a file on disk, while mappit.c is only writing to memory. However, each program is writing to memory, and in each case, the memory contains the mapped pages of a file. In both examples, the data are written to disk at some point by the VMM. In the case of mappit.c, the data are written to the disk file when the file is closed (at program termination).

There is one very important difference in the results of the two files created by the programs in Fig. 7.27. When a page of data is touched in memory for the first time, the kernel zero-fills the entire page. Therefore, the mfile created by mappit.c will be larger than 1,000,000 bytes. It will be zero-filled to the end of the last page. To avoid this minor problem, applications that use shmat() should allocate file space in record sizes that match page boundaries.

A process can still perform read() and write() operations of a file while it is mapped with shmat(). It is also possible for two or more processes to map the same file simultaneously. The file's segment is simply mapped to one of each of the processes' available segments. This results in concepts similar to shared memory, where each process can instantly see the changes made within the segment by other processes.

Explicit file mapping using mmap()

As mentioned earlier, AIX allows explicit file mapping via the shmat() subroutine. The traditional UNIX method for explicit file mapping is provided by the mmap() subroutine. AIX 3.2 added mmap(), along with a number of other memory mapping utilities that work with mmap(), in order to be compatible with other UNIX-based systems. The goal of mmap() is the same as shmat(); however, its capabilities are quite different.

The mmap() subroutine is not based upon segments. Therefore, a file mapped with mmap() uses exactly the amount of virtual memory within the process's user space, as requested. This means that a process can map more than one file into a single segment, which allows a process to map many more files simultaneously with mmap() than with shmat(). The mmap() subroutine also allows the application to map portions of files, instead of requiring the entire file to be mapped. The mmap() subroutine includes options for sharing a mapped file or holding exclusive "rights" to the file.

Figure 7.28 shows an example of using mmap() to explicitly map a file. The first parameter specifies the address of where the mapped file should start. As with shmat(), a zero for this parameter indicates that the kernel should choose the location. This example maps two pages of the file (8192) starting at byte 36,384. The MAP_SHARED option indicates that "writes" made to the file via

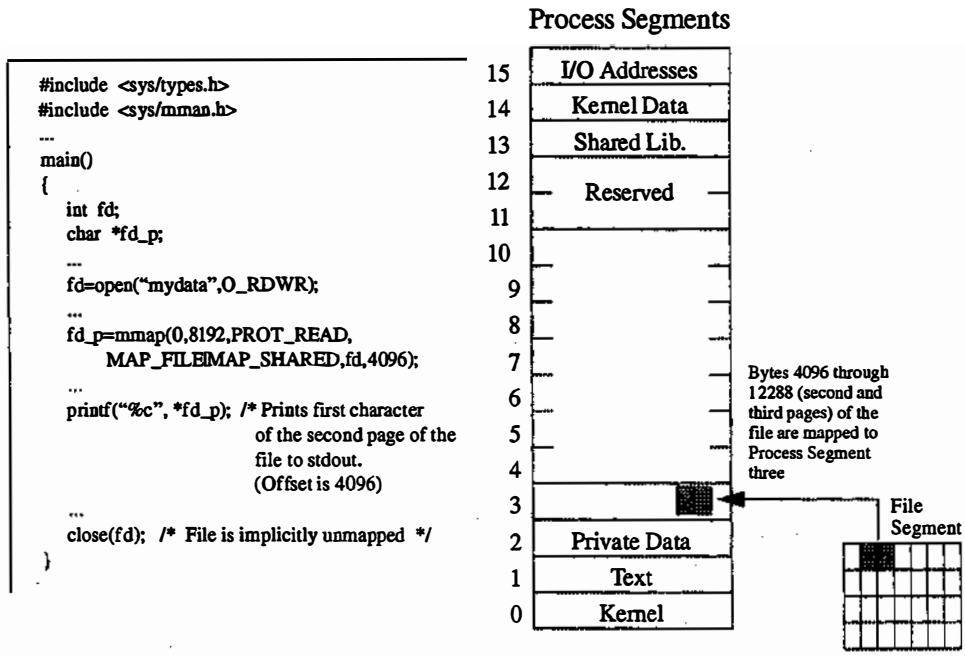

Figure 7.28 Explicitly mapped files using mmap().

TABLE 7.2 shmat() vs. mmap()

shmat()	mmap()
Only one file per segment, therefore, limit of eight files at a time.	Can map multiple files to a single segment, therefore many files can be mapped at a time.
Allows extension (appending) of file up to segment boundary.	Does not allow the extension of a mapped file.
Maps entire file.	Allows portions of files to be mapped, as well as page-level protection of mapped files.
Slightly faster than mmap().	Slightly slower than shmat().
Not portable (see Author's Note at beginning of this section).	Portable for memory mapped files.

pointer assignment are visible to other processes. This is the opposite of MAP_PRIVATE, which causes a private copy of the mapped file region to be created when the file's contents are changed. See the manual page for mmap() or InfoExplorer for a detailed description of the mmap() subroutine. Relatives of mmap() include munmap(), msync(), and mprotect().

A comparison of shmat() and mmap()

Both shmat() and mmap() have their pros and cons. Table 7.2 provides a side-by-side look at these subroutines.

One final note on explicitly mapped files: Neither shmat() nor mmap() allows mapping of special (device) files. Imagine the implications of explicitly mapping /dev/mem!

Chapter

8

Virtual File Systems

In Chap. 7, the virtual file system concept was briefly introduced. As a layer of the file I/O subsystem, the VFS provides applications with access to different types of physical file systems without requiring the application to know the specifics of how the file systems work. It has been mentioned that AIX 3.2 supports three types of virtual file systems: the journaled file system (JFS) for local disk files, the network file system (NFS) for remote file systems, and CD-ROM file systems. Support for other types of file systems can be added, as kernel extensions, to the VFS. Examples include the distributed file system (DFS), the Andrew file system (AFS), and even PC-DOS based file systems.

> **Author's Note:** I know someone who was contemplating implementing an Amiga-style file system for AIX!

Chapter 7 also introduced vnodes (virtual nodes), but only inasmuch as to say that they are found between the kernel's file table and the in-core inode table. This chapter delves much deeper into the structure of the vnode and its relationship to other components of the VFS, such as the vfs structures, gfs structures, vmount table, vnodeops, and vfsops.

8.1 The VFS Layer

Figure 8.1 illustrates how the virtual file system layer fits within the other components of the AIX 3.2 file I/O subsystem. Much of the concept of the VFS layer was created by Bill Joy at Sun Microsystems. Additional work was done by Peter Weinberger at Bell Labs. While you will find support for virtual file systems in other UNIX-based systems, IBM has made a few small changes from the original BSD and SVR4 versions.

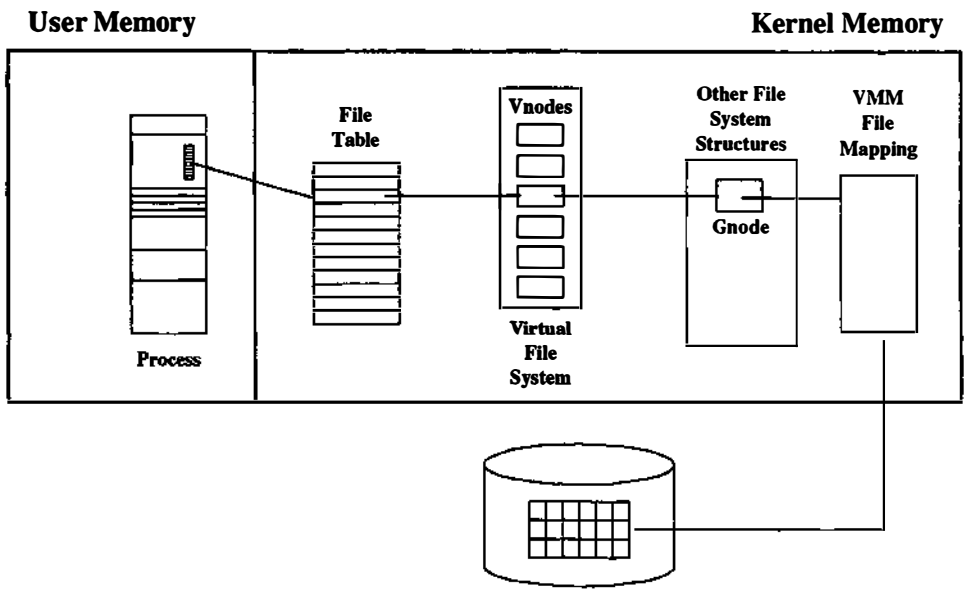

Figure 8.1 The VFS layer.

8.2 Vnodes

The vnode connects the VFS to the file I/O subsystem. There is one vnode structure allocated to each opened file, regardless of the type of file system from which the file comes or how many times the file has been opened. The vnode structure is defined in the /usr/include/sys/vnode.h header file. Interesting fields include:

v_flag. An unsigned short that indicates various states of this vnode or the file system to which this vnode belongs. Flag values are defined in the /usr/include/sys/vnode.h header file and include:

V_ROOT (0x01). This vnode is for the root directory of the VFS.

VFS_UNMOUNTED (0x02). This vnode's VFS has been unmounted.

V_TEXT (0x10). The file of this vnode is currently being executed (program text file).

v_count. An unsigned long that indicates the reference count for this vnode.

v_vfsp. A pointer to the vfs structure (explained shortly) for this vnode. This is one of the most important fields.

v_mvfsp. Another pointer to a vfs structure, this points to the vfs structure for the file system that was mounted over this vnode. If this vnode is not the mount point for a file system, this pointer is null.

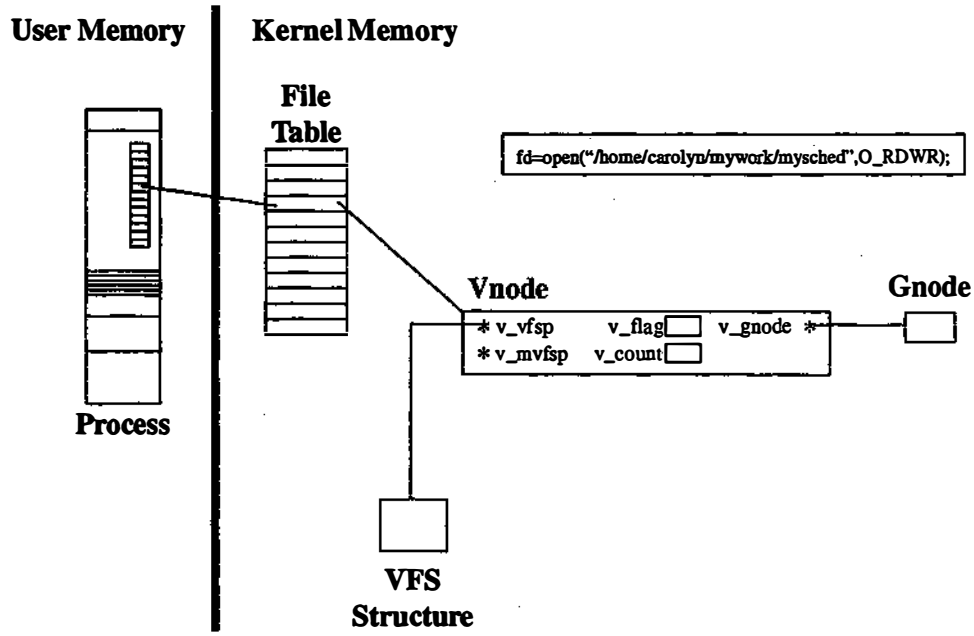

Figure 8.2 The vnode.

v_gnode. A pointer to a gnode structure, the gnode for this vnode (see Chap. 7 for information on gnodes).

There are other fields in the vnode structure, but the ones listed above are the only ones important to the discussion at hand.

Figure 8.2 illustrates the allocation of a vnode for a typical file. As mentioned earlier, the v_vfsp pointer is very important. It points from the vnode to a vfs structure.

8.3 vfs Structures

AIX 3.2 allocates one vfs structure for each mounted file system. Each vnode points to the vfs structure that represents the file system in which the vnode's file is found. The vfs structure is defined in the /usr/include/sys/vfs.h header file. The vfs structure includes:

vfs_next. A pointer to the next vfs structure, as they are maintained in a linked list.

vfs_gfs. A pointer to a gfs structure. This is a very important field and will be explained shortly.

vfs_mntd. A pointer back to the vnode that represents the root directory of the file system of this vfs structure.

vfs_mntdover. A pointer back to the vnode that represents the directory stub (mount point) onto which the file system of the vfs structure is mounted.

vfs_mdata. A pointer to a vmount structure (an entry in the vmount table). The vmount table has one entry for each mounted file system and holds information about the options specified for the mount. The vmount structure is defined in the /usr/include/sys/vmount.h header file.

The most important field in the vfs structure is the vfs_gfs pointer that points to a gfs structure. The gfs structure is described shortly, but the links between the vfs structures and the vnodes is discussed first. Figure 8.3 illustrates the relationships of a vsf and a set of vnodes. To trace the links, start with the vnode allocated to the file /home/carolyn/mywork/mysched. For the example, assume that carolyn's home directory and all directories below it are found in the /home file system.

The vnode for mysched points to a gnode in the in-core inode table for the local disk file. The vnode also points to the vfs structure that represents the /home file system. The vfs structure has three pointers back to vnodes, two of which are illustrated here. One pointer, vfs_mntd, points back to the vnode that represents the root directory of the /home file system. The vnode for the root direc-

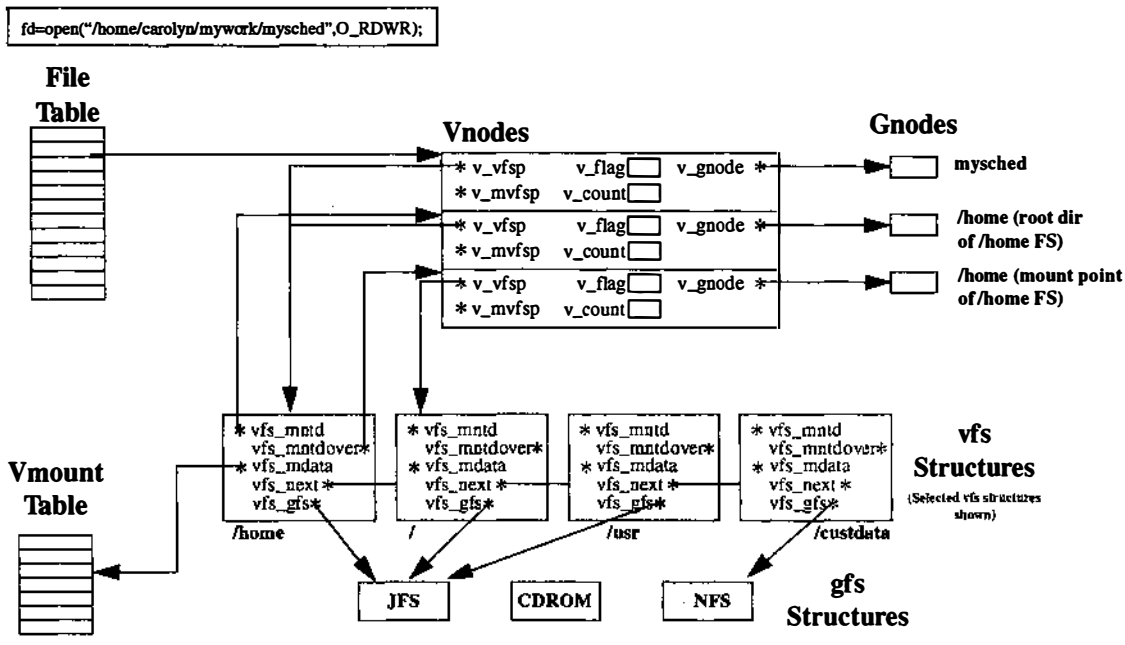

Figure 8.3 Vnodes and vfs structures.

tory points to a gnode in the in-core inode table. This in-core inode holds the disk inode for the root directory of the /home file system. Recall that this inode will always have an index number of 2 (see "Reserved Inodes" in Sec. 6.4).

Another pointer in the vfs structure, vfs_mntdover, points to the vnode that represents the mount point of the /home file system. The mount point is the /home directory within the / (root) file system. Its vnode points to a gnode in another in-core inode. This in-core inode holds the disk inode for the /home directory. Its inode index number is arbitrary. Note that the vnode for the /home directory (within the root file system) points to the vfs structure that represents the root file system. The whole cycle can be repeated from there.

> **Author's Note:** The illustration in Fig. 8.3 represents one of the biggest challenges to most of my students. The important point to remember is that there are two representations of the /home directory. One vnode/gnode/in-core inode combination is the root directory of the /home file system and the other is the mount point of the file system. The mount point is an ordinary directory found in the root file system. Once the concept is clear, the rest of this convoluted drawing is easier to understand.

Another vfs field, vfs_vnodes, points back to a vnode that forms the head of a linked list of vnodes from the same file system. The vnodes maintain the doubly linked list via the v_vfsnext and v_vfsprev pointers defined in the vnode structure (see /usr/include/sys/vnode.h>). Figure 8.4 illustrates this relationship.

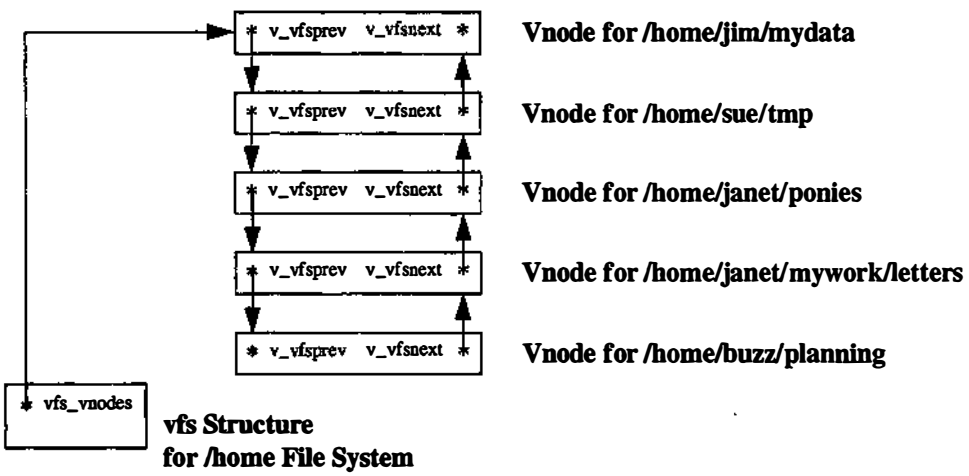

Figure 8.4 Linked list of vnodes for a file system.

8.4 gfs Structures

Each vfs structure points to a gfs structure (via the vfs_gfs pointer). There is one gfs structure in the kernel for each supported virtual file system type. By default, AIX has three gfs structures, one for JFS, one for NFS, and one for CD-ROM. Figure 8.5 shows a typical set of vfs structures and their links to the gfs structures.

The gfs structure is defined in the /usr/include/sys/gfs.h header file. Fields include:

gfs_ops. A pointer to a vfsops structure.

gn_ops. A pointer to a vnodeops structure. These two fields are the most important fields in the gfs structure. They point to structures that contain pointers to functions, which are explained shortly.

gfs_type. An integer that contains a description of the VFS type. The values for this field are found in the /usr/include/sys/vmount.h header file and include the following:

MNT_AIX (0). An old-style AIX file system (pre-JFS).

MNT_NFS (2). Sun's network file system.

MNT_JFS (3). IBM's journaled file system.

MNT_CDROM (5). CD-ROM file system.

IBM reserves gfs_type numbers 0–7. Numbers 8 and above are available for user-added virtual file systems.

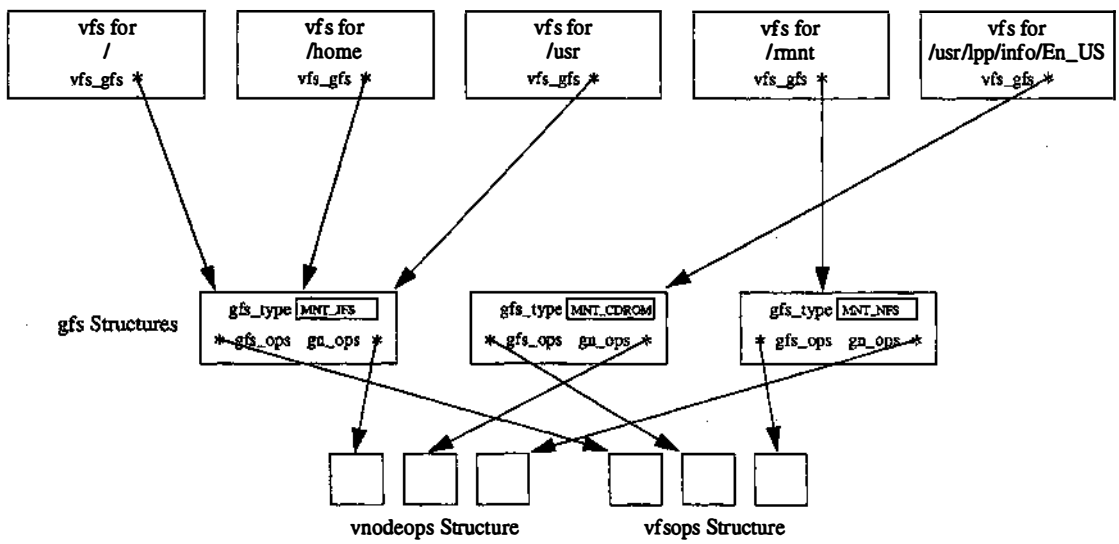

Figure 8.5 vfs and gfs structures.

When a systems programmer adds a virtual file system type to AIX, the kernel extension includes a new gfs structure.

The vnodeops and vfsops

Each gfs structure points to a vnodeops structure. The vnodeops structure is defined in the /usr/include/sys/vnode.h header file. It consists of pointers to functions that perform operations on files. In a sense, the gfs-vnodeops relationship forms a switch table used to vector to the appropriate code for performing an operation on a type of VFS file. In other words, when an application issues an open() request on a file, the open() system call builds the link from the process to the file I/O subsystem. In building that link, the open() system call uses a vnode to establish a relationship with the vfs and gfs structures for the VFS type. The vnodeops structure associated with the gfs contains a pointer to the open() routine specific to the VFS type (see vn_open()). All the application did was open a file. The application did not need to know anything about the VFS type of the file. This brings us to the mission of the VFS layer of the file I/O subsystem, which is to provide the ability to manipulate files from different types of file systems while hiding the details from the application.

Each gfs structure also includes a pointer to a vfsops structure. The vfsops structure is defined in the /usr/include/sys/vfs.h header file. Like the vnodeops structure, the vfsops structure contains pointers to functions. These functions, however, are operations performed on file systems. Figure 8.6 illustrates the switch table concept and wraps up the discussion of the VFS layer of the file I/O subsystem.

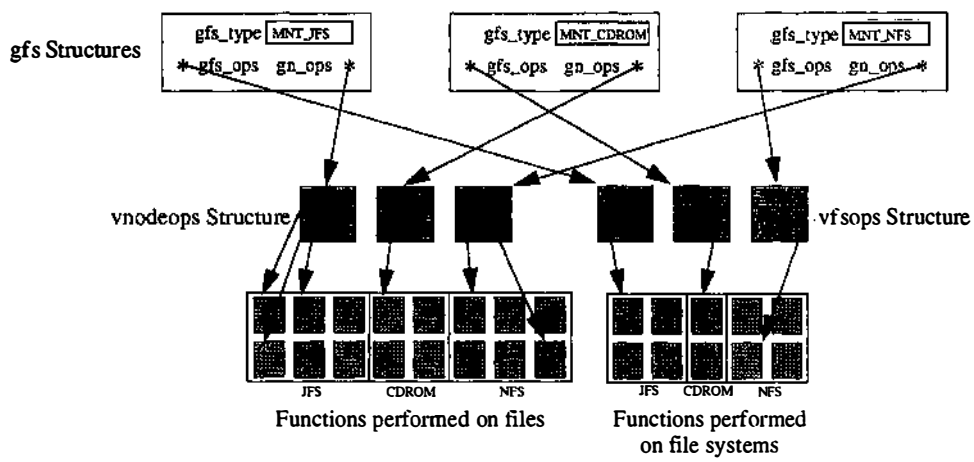

Figure 8.6 vnodeops and vfsops structures.

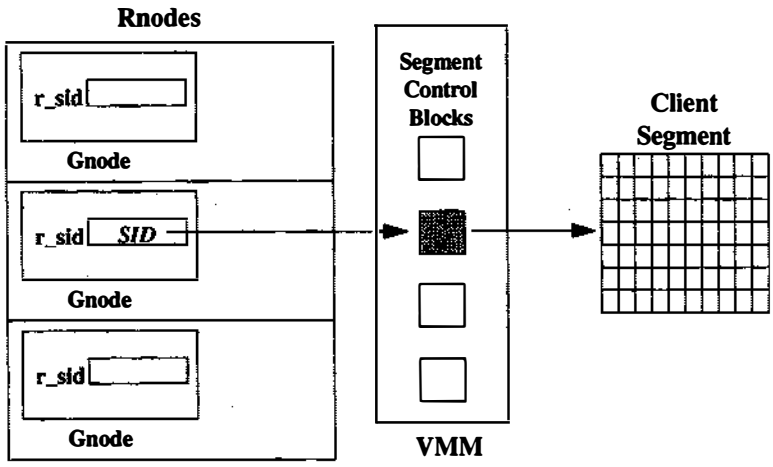

Figure 8.7 The rnode table.

8.5 The rnode Table

Chapter 7 described how vnodes for local disk files point to gnodes which are contained within in-core inodes. For remote files, the NFS client maintains a table of rnodes, which are similar to in-core inodes. The vnode of a remote file points to a gnode contained within an rnode. The rnode structure is defined in the /usr/include/sys/rnode.h header file. Most of the fields in this structure involve NFS internals and are beyond the scope of this book. The r_sid field contains the ID number of the client segment into which the VMM has mapped the remote file. Figure 8.7 illustrates the rnode table.

Chapter

9

The Device I/O Subsystem

Previous chapters have described how AIX 3.2 facilitates file I/O for the storage and retrieval of data and programs. This chapter describes how AIX 3.2 handles I/O directly to and from peripheral devices. It also addresses device configuration and the object data manager (ODM).

From an application's perspective, data are read from and written to devices in the same fashion as an ordinary file. This is because AIX 3.2, as well as other UNIX-based operating systems, treat attached devices as files, giving each device a file name abstraction in the global file system. By convention, the device file names are found in the /dev directory. The device file names are associated with actual files, called special files.

9.1 Components of Device I/O

There are three major components of the AIX 3.2 kernel involved in device I/O: the file I/O subsystem, the device switch table, and the device drivers. The file subsystem includes the special files, mentioned above, as well as vnodes, specnodes, and devnodes. Figure 9.1 illustrates the relationship of the kernel components that facilitate device I/O.

The file subsystem

As mentioned in Chap. 2, devices come in two flavors: block and character. Block devices perform I/O in buffered units for greater efficiency. Examples of block devices include disk and diskette drives. Character devices, such as printers and ASCII terminals, perform I/O one character at a time. Special files come in block and character format to match device types. Figure 9.2 shows the output of a listing of the /dev directory. The first character in each line of out-

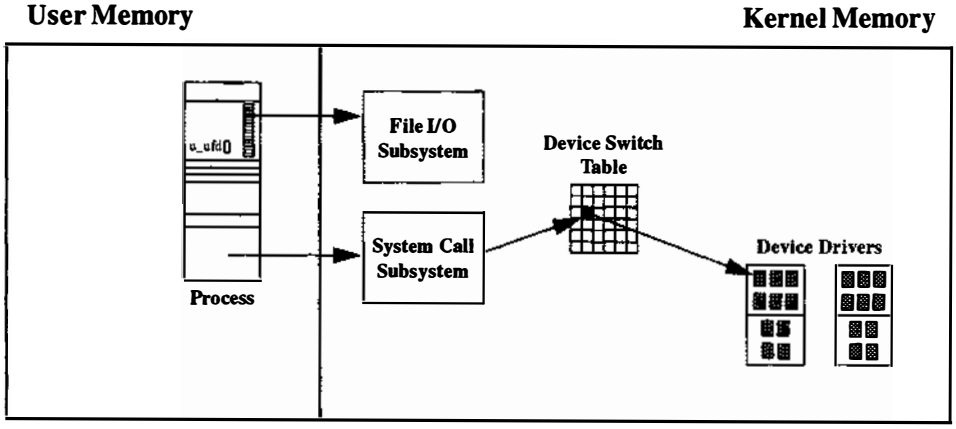

Figure 9.1 Components of device I/O.

```
$ ls -l /dev
crw-rw---- 1 root system 14,0 May 15 09:31 console
...        ... ...  ...    ...  ...       ...
brw-r----- 1 root system 10,0 May 15 09:31 hd1
brw-r----- 1 root system 10,1 May 15 09:31 hd2
...        ... ...  ...    ...  ...       ...
brw-r----- 1 root system 15,0 May 15 09:31 hdisk0
brw-r----- 1 root system 15,1 May 15 09:31 hdisk1
...        ... ...  ...    ...  ...       ...
crw-r----- 1 root system 10,0 May 15 09:31 rhd1
crw-r----- 1 root system 10,1 May 15 09:31 rhd2
...        ... ...  ...    ...  ...       ...
crw-r----- 1 root system 15,0 May 15 09:31 rhdisk0
crw-r----- 1 root system 15,1 May 15 09:31 rhdisk1
...        ... ...  ...    ...  ...       ...
crw-rw---- 1 root system 12,0 May 15 09:31 tty0
crw-rw---- 1 root system 12,1 May 15 09:31 tty1
```

Figure 9.2 Block and character device file names.

put indicates the type of device. Block devices have a "b" as the file type and character devices have a "c." Note that most block devices include character device abstractions. This is because most block devices can function in character mode. For instance, in AIX 3.2, each fixed disk drive has a block device name of "hdisk..." and a character device name of "rhdisk..." (for raw mode).

Block and character special files have zero length. While the file subsystem assigns an inode for a special file, no actual disk space is taken by the file. Special files simply provide handles for interfacing with devices, in the form of

```
fd=open("/home/dave/myfile",O_RDWR);  /* Opening an ordinary file  */
devfd=open("/dev/hdisk0",O_RDWR);  /* Opening a device  */
```

Figure 9.3 Performing device I/O.

file names that can be supplied to the open() system call, as shown in Fig. 9.3. The fact that a device special file is opened like an ordinary file means that an application can perform device I/O without needing to know anything about the device with which it is interfacing.

In place of the size field for special files listed in Fig. 9.2, the ls -l command displays each device's major and minor numbers. A device's major number indicates the device type. All devices that share a common device driver also share a major number. For example, Fig. 9.2 shows ASCII terminals (tty...) as having major number 12. A device's minor number indicates an instance of the device type. For example, if a system included three ASCII terminals, they would all share the same major number, but each terminal would have a different minor number. Figure 9.2 shows the two ASCII terminals with minor numbers of 0 and 1.

> **Author's Note:** You may have noticed that AIX 3.2 uses the same major number for the raw mode interface of block devices (i.e., /dev/hdisk0 and /dev/rhdisk0 both use major number 15 in Fig. 9.2). This is not true with all UNIX-based operating systems.

Specnodes and devnodes

When an application opens a device for I/O, the open() system call causes the kernel to allocate a file descriptor, file table entry, and vnode, as it would for the opening of any ordinary file. From here, however, things are a little different. When the system determines that the file being opened is a special device file, it sets the v_gnode pointer in the vnode to point to a gnode structure contained within a specnode, as illustrated in Fig. 9.4. A second vnode is allocated to maintain the special file's place within the physical file system. The v_gnode pointer in the second vnode points to a gnode contained within an in-core inode. The in-core inode contains the disk inode for the special file. The first vnode is linked to the second vnode via the _v_pfsvnode pointer found in the _v_data union. See the /usr/include/sys/vnode.h header file for more details.

The specnode, which is defined in /usr/include/sys/specnode.h, describes the instance of a special file. AIX 3.2 introduces the specnode for device files and unnamed pipes, which are described in Chap. 10. Interesting fields in the specnode include:

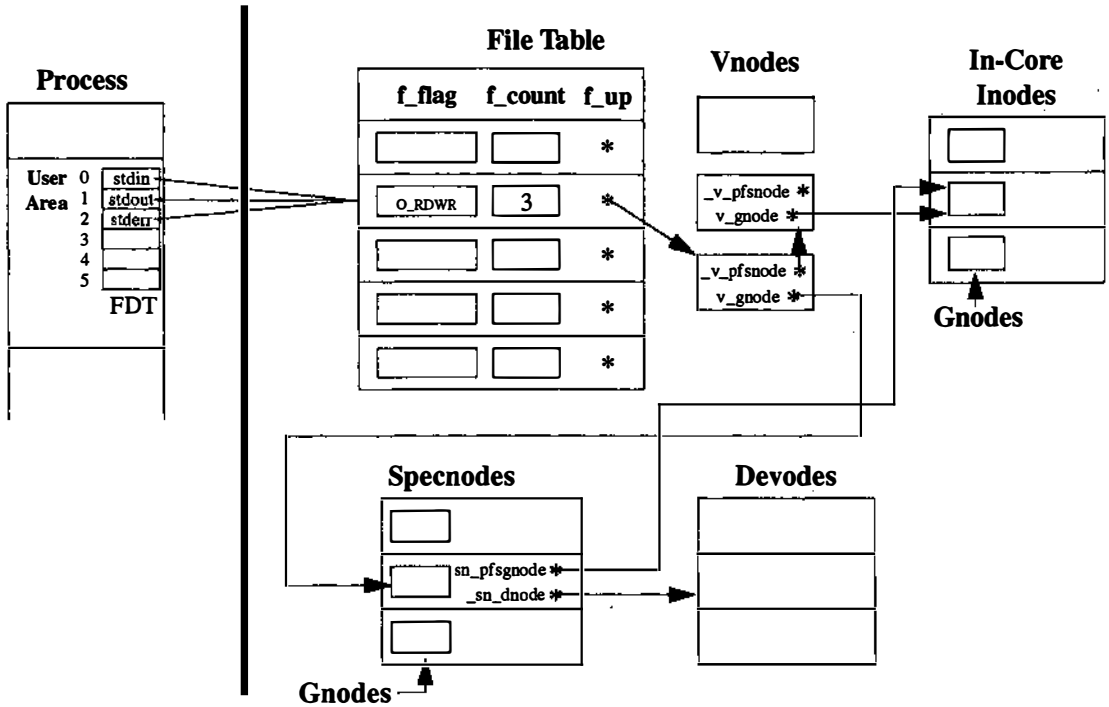

Figure 9.4 Device I/O and the virtual file system.

sn_next. A pointer to another specnode structure, this field builds a linked list of a specnodes that pertain to a specific device.

sn_count. A cnt_t data type (short integer) that holds the reference count for the specnode.

sn_gnode. The embedded gnode structure.

sn_pfsgnode. A pointer to a gnode structure, this field points to the gnode contained within the in-core inode for the device special file, as shown in Fig. 9.4.

sn_data. A union that holds either a pointer to a devnode structure (_sn_dnode) or a pointer to a fifonode structure (_sn_fnode). The pointer to a devnode structure is used when the specnode is associated with a device special file, as described in this chapter. The pointer to a fifonode structure is used when the specnode is associated with an unnamed pipe, as described in Chap. 10.

For device special files, the specnode points to a devnode. The devnode structure is also defined in the /usr/include/sys/specnode.h header file. Interesting fields include:

dv_forw. A pointer to another devnode structure, this field, along with dv_back, builds a doubly linked list of devnode structures that are hashed for faster access.

dv_back. See dv_forw.

dv_dev. A dev_t data type that holds the device's major and minor numbers. See Chap. 7 for an explanation of how the dev_t data type is used.

dv_count. A cnt_t data type (short integer) that holds the reference count for the devnode.

dv_specnodes. A pointer to the specnode that is the head of the linked list of specnodes sharing the devnode.

Device drivers

Device drivers link the AIX 3.2 kernel with physical and logical devices. Each device driver has two halves, commonly referred to as a top half and a bottom half. The top half contains functions called by the kernel on behalf of processes. These functions are routines for tasks performed on devices, such as d_open(), d_close(), d_read(), and d_write(). These routines run in process environment and are therefore preemptable by the AIX 3.2 dispatcher. The code and data of the top half of AIX 3.2 device drivers are also pageable.

The bottom half of an AIX 3.2 device driver contains interrupt handler code, used by the device driver to react to interrupts generated by the device. The bottom half is active only when the system is in interrupt handler environment. The code and data of the bottom half are always pinned. Figure 9.5 illustrates the halves of an AIX device driver.

The device switch table

AIX 3.2 maintains a single device switch table, which is a matrix of pointers to entry points for the functions found in the device driver top halves. The device

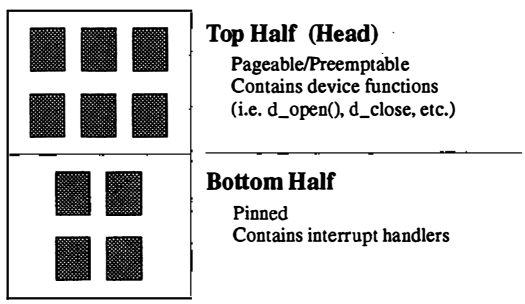

Figure 9.5 AIX 3.2 device drivers.

switch table is actually an array of devsw structures, which is defined in the /usr/include/sys/device.h header file. The devsw structure contains mostly pointers to the device driver functions. There is one devsw structure allocated in the device switch table for each device major number (device driver).

When a process performs an I/O-type operation on a device, the device switch table is consulted to locate the entry point for the desired function. The function is called with the specific device's minor number passed as a parameter, as illustrated in Fig. 9.6.

> **Author's Note:** Most UNIX-based operating systems include two device switch tables: one for block devices and one for character devices. As shown, AIX 3.2 uses a single device switch table, which combines functions for both types of devices.

Figure 9.7 provides an example of how the kernel handles device I/O operations.

Writing device drivers for AIX 3.2

System programmers who write device drivers for AIX 3.2 must take into account the nature of a preemptable kernel. Top half functions and bottom half interrupt handlers should disable interrupts during critical sections of code. Top half functions must lock kernel data structures to assure proper access synchronization. These concepts were discussed in Sec. 5.7. Consult the InfoExplorer on-line documentation for details of writing an AIX 3.2 device driver.

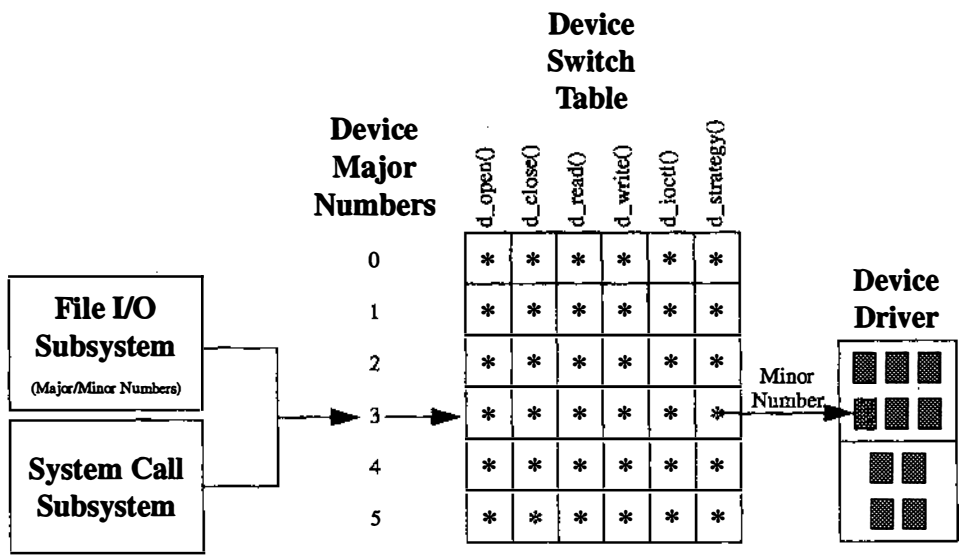

Figure 9.6 The device switch table.

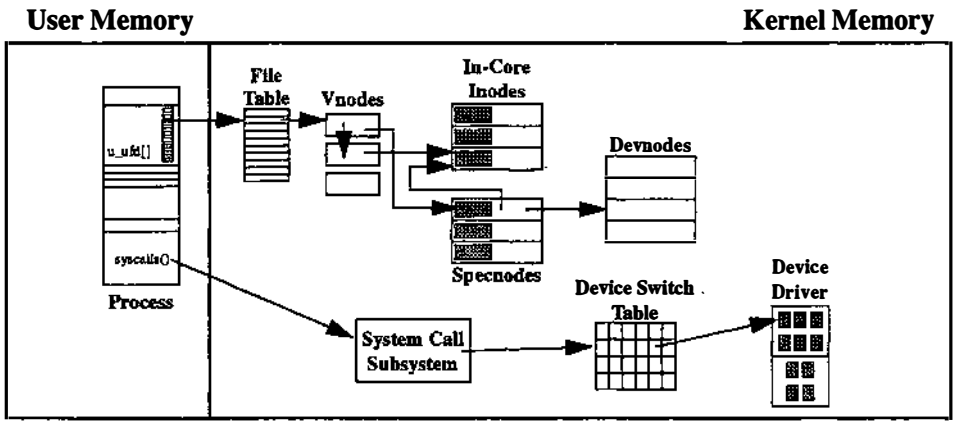

Figure 9.7 Device I/O—the big picture.

9.2 AIX 3.2 Device Configuration

Chapter 2 of this book gave a very brief description of the role of the object data manager in AIX 3.2 device configuration. This section provides greater detail.

Recall that the object data manager is a collection of library routines, commands, and data base files that are used by AIX 3.2 to store configuration data for devices. The data base files, found in or linked to the /etc/objrepos directory, include:

PdDv. Predefined devices: configuration information for all supported devices

PdAt. Predefined attributes: "factory set" defaults for attributes associated with each supported device, as well as ranges of valid values for the attributes

CuDv. Customized devices: devices the system actually has (or thinks it has)

CuAt. Customized attributes: those device attributes that have been modified from the "factory set" default values

Device states

AIX associates a state, or condition, with each device. The three most common device states are "Predefined" (or unknown), "Defined," and "Available" (or configured). All supported devices are in one of these three states. Figure 9.8 illustrates the device states. A device in the predefined state is listed in the PdDv file but not in the CuDv file. In other words, the system does not think it has one of these devices attached. The device is unknown to the system.

A device in the defined state is known to the system. There is an entry in the CuDv file for a defined device, but the device is not available for use. An example of when a device is defined but not available is provided shortly.

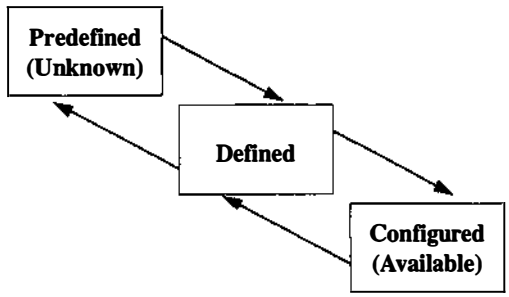

Figure 9.8 Device states.

A device is available when four things have occurred: the driver for the device has been loaded into the kernel, an intr structure for the device has been initialized for interrupt handling, an entry has been made in the device switch table for the device's major number, and an entry in the /dev directory has made the device accessible to applications.

Devices move from state to state with the help of small programs called methods. There are methods for defining and configuring devices, as well as undefining and unconfiguring devices. The methods work with the configuration manager (a program called /etc/cfgmgr) to define and configure devices.

> **Author's Note:** Since the PdDv file contains information for all supported devices, what about nonsupported devices? For instance, if a computer peripheral company develops a new piece of hardware, such as a retina scanner, for the RISC System/6000 or PowerPC, before it can be defined, information about the device must be placed in the PdDv. (There are no retina scanners listed in the PdDv currently; I have checked. Therefore, I like to say that there is a state that precedes "Unknown" which I call "Really Unknown.") The developer of the scanner must not only write the device driver for the scanner but must also include the methods, as well as installation software that updates the PdDv. This would constitute the packaging of the device driver and hardware support software.

Device states become apparent to system administrators at two times. The first is when the system is booted but an external device, such as a CD-ROM unit, is not powered on. The configuration manager is not able to detect the external device, so it only defines the CD-ROM, based on information in the CuDv that indicated that the device was once on the SCSI bus. The device driver is not loaded into the kernel. This technique prevents kernel space from being wasted by drivers for devices that are no longer attached to the system. The problem occurs when the CD-ROM unit is turned on and an attempt is made to use it. Before it can be used, it must be made available. This can be done through an option under the devices menu of SMIT called "Configure Devices Added Since IPL."

Another time understanding device states is important is when a system administrator removes a device. The SMIT dialog screen for removing most devices includes the question "Leave the definition in the data base?" with the possible answers of yes or no. What the configuration manager wants to know is whether the device definition should remain in the CuDv file. Answering "yes" to the SMIT question causes the device driver to be unloaded from the kernel. The /dev entry is also removed. The device moves from the available state to the defined state. Answering "no" to the SMIT question causes the configuration manager to also remove the device's record in the CuDv. If a device is being removed temporarily, it's usually best to leave the definition in the CuDv data base file, as it serves as a place holder for the device.

Figure 9.9 shows how SMIT, the methods, the configuration manager, and the ODM work together to perform device configuration.

9.3 System Start-up

The configuration manager is also involved in the system start-up or IPL (initial program load, for those of you not familiar with IBM-speak). It works with the ODM to define and configure devices in the proper order. There's only one catch. The ODM files are stored in the root file system, which is found in the root volume group, which is on one or more disk drives defined in the ODM. We have a Catch-22! How can the configuration manager access the ODM files when they are stored on a device which requires the ODM files in order to be configured? In fact, how is the configuration manager (/etc/cfgmgr) executed before the disk drives are configured? These questions are answered in this section.

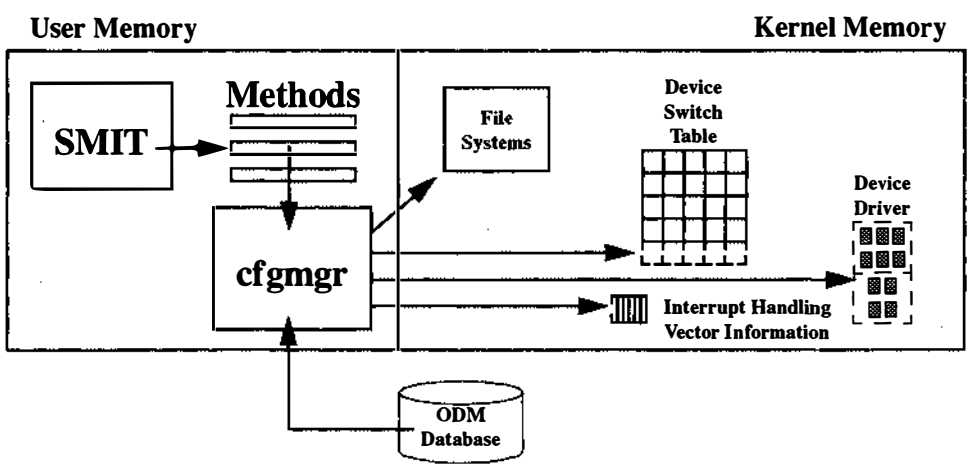

Figure 9.9 AIX 3.2 device configuration.

The boot sequence for the RISC System/6000 includes two phases. The purpose of Phase I is to configure enough devices to vary on the root volume group and mount the root file system. Once that is completed, Phase II configures the remaining devices and puts the system in multiuser mode. There is, however, an important set of events that occur prior to Phase I which shall be referred to as "Phase 0."

Phase 0

When a RISC System/6000 is powered on, its bootstrap program within its read only storage (ROS) initiates two diagnostics tests. The first test is the built-in self test (BIST) and the second test is the power-on self test (POST). These routines test the CPU complex and other system-level components. Upon completion of the POST, the ROS consults the nonvolatile RAM (NVRAM) where two boot lists are stored. The boot lists are maintained in the NVRAM via battery power. Each of the two boot lists corresponds to the two boot modes of the system: NORMAL and SERVICE, which is controlled by a key switch on the front panel of the system unit. The NORMAL mode boot list contains a sequenced list of devices from which the system will attempt to boot when the key is in the NORMAL position. The SERVICE mode boot list contains a sequenced list of devices from which the system will attempt to boot when the key is in the SERVICE position. Figure 9.10 illustrates the NVRAM boot lists for a typical system.

Note that SERVICE mode booting first looks for a floppy diskette, then for a CD-ROM, then for a tape, and finally, if none of these devices are found, it runs the diagnostics manager software found on a fixed disk. This sequence allows the system administrator to boot from one of the diagnostics media or from an installation or mksysb tape. See the manual page for the bosboot command for more information on creating boot records.

The boot list indicates a boot device, which must start with a boot record. For NORMAL mode booting, a fixed disk is usually assigned as the boot device. This decision is made when AIX 3.2 is installed. The boot lists can be viewed

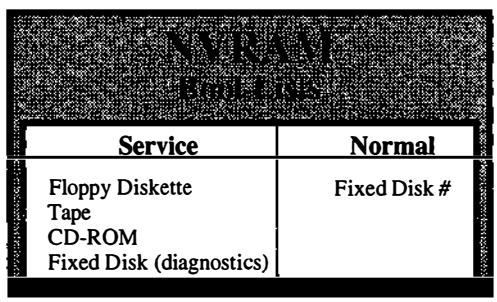

Figure 9.10 The boot list and the NVRAM.

or altered from within the diagnostics program (diag) or by using the bootlist command.

During NORMAL mode booting, the boot device (the first fixed disk drive, in our example), is consulted. The first sector on the drive contains the boot record (or IPL record). The layout of the boot record is defined in the /usr/include/sys/bootrecord.h header file in the ipl_rec_area structure. It contains the physical sector number (PSN) of the start of the boot logical volume (BLV). The BLV is also known as /dev/hd5. The boot logical volume contains boot code, a miniature version of the root file system, and a miniature version of the ODM data base files. The BLV must always reside on the same physical disk drive as the boot record.

The boot code found in the BLV is used to load the kernel and to mount the mini-root file system into memory. This file system is often called the RAM file system. Next, the init process is handcrafted and executed. Phase I can now begin.

Phase 1

The init process runs the configuration manager. It uses the information in the miniature ODM data base to configure the base devices (those devices required to accomplish the goal of Phase 1). It also uses the Config_Rules file in the ODM to determine the correct order for configuring devices. For instance, a SCSI disk drive cannot be configured until the SCSI controller has been con-

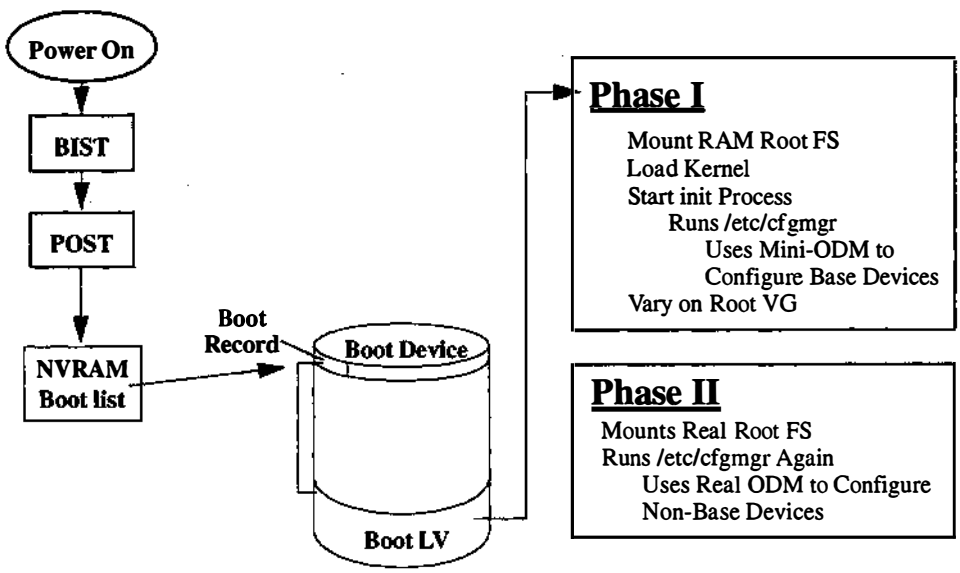

Figure 9.11 The AIX 3.2 boot process.

figured. Once all of the devices associated with the root volume group are configured, the volume group can be varied on (activated). The real root file system can now be mounted and the RAM file system (mini-root) is discarded via a newroot() call. This concludes Phase 1.

Phase 2

The init process runs the configuration manager a second time to configure all other devices. Figure 9.11 illustrates the entire boot process. System administrators need to be aware of a couple of places where problems might occur during the boot process and what steps to take to correct the problems. The boot list and/or BLV can become corrupted. The boot list can be reconstructed by running diagnostics and accessing the "View/Alter Boot list" option under the diagnostics services menu. The BLV can be reconstructed by bringing the system up in maintenance mode and using the bosboot command.

> **Author's Note:** Some of you may be wondering how one boots the system to run diagnostics if the boot list is corrupted. The ROS maintains an intrinsic boot list it uses if the NVRAM is unavailable.

Chapter

10

Interprocess Communication

This chapter describes the kernel activities and components involved in various forms of interprocess communication (IPC). It begins with informal IPC mechanisms, such as signals and pipes, and concludes with more sophisticated facilities, such as the System V IPCs (shared memory, semaphores, and message queues). Each section includes a brief description of the application of the IPC tools in order to help those not familiar with their usage, but detailed discussion of the system call syntax is avoided. Consult the AIX InfoExplorer online documentation for information on system call syntax and usage. Also, the topics discussed in this chapter are, for the most part, generic and applicable to other UNIX-based systems. Therefore, any of the many reference books published on the use of IPCs are recommended for applications programmers.

10.1 An Introduction to Interprocess Communication

An IPC mechanism is any method or tool that allows processes to share information. Some methods involve using facilities that exist for other purposes but which happen to lend themselves to IPC. For example, one process can write data to a hard file; then another process can read the data. This is considered an informal IPC. It also has its drawbacks, as it is slow and requires some form of external synchronization.

Other forms of IPC are designed specifically for the task of allowing processes to share information. These are the formal IPCs, which include their own kernel data structures and application programming interface (API). Table 10.1 lists various types of IPC mechanisms.

TABLE 10.1 AIX IPC Tools

Tool	Use	Benefits	Drawbacks
Disk Files	Store and retrieve data	Easy to code Flexible Any process may participate Files survive power cycles	Very slow Requires disk space
Signals	Notify another process of a condition or situation or to take a specific action	Easy to code Well known Many programming options	Only between processes with same EUID (unless sender has root authority) Limited flexibility Signals might be lost
Exit Status	Message from child to parent	Easy to code	Child must die to participate Only between child and parent Only an integer may be passed
Unnamed Pipes	Data stream flow from one process to another	Easy to code Built-in synchronization I/O redirection Relatively fast	Process must be related Unidirectional
Named Pipes	Data stream flow through FIFO file	Easy to code Built-in synchronization Available to unrelated processes	Can be slow

10.2 AIX 3.2 Signal Management

Signals are common to all UNIX-based systems. There have been, however, various approaches to signal management. The AIX 3.2 signal management subsystem incorporates features from System V and from BSD.

> **Author's Note:** The fact that AIX supports most of both System V and BSD signal APIs, as well as the subroutines defined by POSIX, provides the application programmer with a wide range of options. In fact, one can be overwhelmed by the choices, many of which overlap one another in functionality and features. When in doubt, stick with POSIX.

Signals are sent to a process by another process or by the kernel. An example of a signal sent from one process to another is when a user issues the kill command from the shell prompt to terminate a background process. An example of a signal sent from the kernel to a process is when a process attempts to perform an illegal act, such as a reference to a memory pointer whose value has been set to -1. The kernel uses a signal to terminate the offending process and, in this case, to cause a core dump of the process. (A file named "core," which contains the image of the process at the time of its termination, is created in the current directory to allow the programmer to perform a "postmortem" on the process using a debugger, such as dbx). Signals are also sent from the kernel to

TABLE 10.1 AIX IPC Tools (*Continued*)

Tool	Use	Benefits	Drawbacks
Shared Memory	Common memory region shared by mulitple processes	Small to large amounts of data can be quickly shared Owner can set permissions Can live beyond processes	Hard to synchronize access Segment must be explicitly removed
Semaphores	Notify another process of a condition or situation or serialize access to resources	Sophisticates programming interface Operations performed atomically Owner can set permissions Can live beyond processes	Not useful for sending real data Semaphore sets must be explicitly removed
Message Queues	Messages sent from one process to another	Flexible message structure Message sizes up to 64 kb Can be multiplexed Owner can set permissions Queues can live beyond processes	Relatively slow Queues must be explicitly removed
Sockets	Stream or datagram data transfer between processes on same or different systems	Extensive programming interface Well standardized Flexible application	Complicated programming interface Requires system tuning to optimize performance Possibly heavy memory requirements

a process when driven by terminal control characters. For instance, when a user presses the <CTRL>-<c> key combination, the AIX kernel sends a signal to the current foreground process. This usually results in the termination of the process. Figure 10.1 illustrates how a process receives signals.

Processes can decide how to react to most of the signals they receive. A process can accept a signal with the default consequence, which is usually, but not always, termination. A process can choose to ignore the signal, which means that the process never realizes it received the signal. A process can block (mask) a signal, which holds off the delivery of the signal until the process is ready for it. Finally, a process can catch a signal and branch to a routine called a signal handler where some user-defined action takes place.

AIX 3.2 signals

The AIX 3.2 signal management subsystem is capable of supporting up to 64 different signals, although only 40 are currently defined. Symbolic constants are defined for each signal in the /usr/include/sys/signal.h header file. Table 10.2 lists some of the more interesting signals.

> **Author's Note:** Many of the comments given with the signals defined in /usr/include/sys/signal.h include the symbols *, +, @, and !. The * symbol indicates

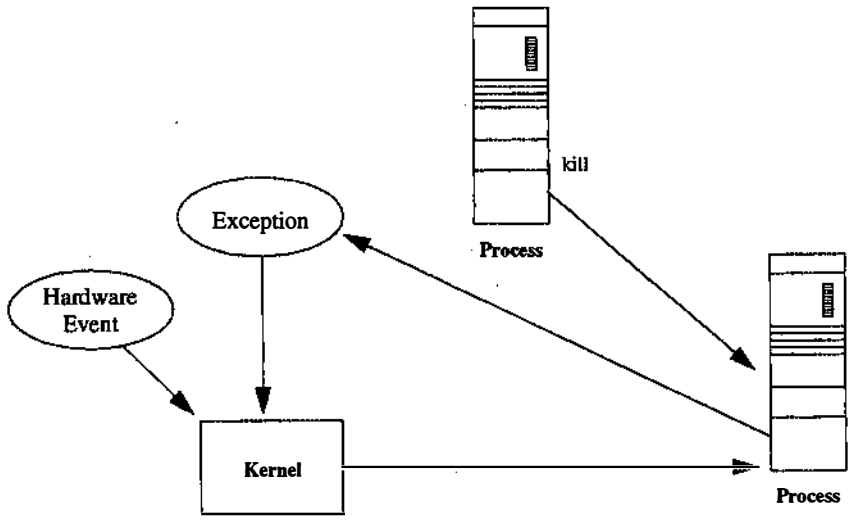

Figure 10.1 Receiving signals.

TABLE 10.2 AIX 3.2 Signals

Signal Number	Signal Name	Description
1	SIGHUP	Terminal disconnect (i.e. <CTRL-d>)
2	SIGINT	Interrupt (i.e. <CTRL-c>)
3	SIGQUIT	Quit (i.e. <CTRL-\>) (core dump)
6	SIGABRT	abort() called (core dump)
9	SIGKILL	Kill (cannot be caught or ignored)
11	SIGSEGV	Segmentation violation
15	SIGTERM	Software kill (i.e. kill command)
17	SIGSTOP	Stop signal
19	SIGCONT	Continue signal
20	SIGCHLD	Death or suspension of child
30	SIGUSR1	User defined
31	SIGUSR2	User defined
33	SIGDANGER	Low paging space

signals that cause a core dump by default. The + symbol indicates signals that are ignored by default. The @ symbol indicates signals that cause the receiving process to be stopped (suspended, not terminated). The ! symbol indicates signals that cause the receiving process to continue from a stopped state. The SIGDANGER signal (33) is unique to AIX 3.2. It is sent to all processes when free paging space is low. By default, processes ignore the SIGDANGER signal, although the comment in the signal.h header file does not include a + symbol.

The /usr/include/sys/signal.h header file also includes defines of symbol names for compatibility with applications written for other UNIX-based systems. For instance, AIX defines signal 20 as SIGCHLD, the signal that is sent to a parent process when a child process terminates or is suspended. Older UNIX-based systems define this signal as SIGCLD, so AIX includes a define of SIGCLD as SIGCHLD.

AIX 3.2 includes two user-definable signals, SIGUSR1 (30) and SIGUSR2 (31). Applications can be written to catch these signals and branch to an appropriate routine. Beware, however, that the default action for these two signals is to terminate the receiving process.

Sending signals

Signals are sent from one process to another via the kill() subroutine. The name "kill" is slightly misleading since not all signals terminate the receiving process, but most signals do terminate the receiver. The kill() subroutine takes two parameters, the PID of the process to which the signal is sent, and the signal value itself. A process can only send signals to another process that shares the same effective user ID (see Chap. 6), unless the sender has root authority.

The kill command, issued from the shell prompt, invokes the kill() subroutine. The kill command can be instructed to send any valid signal. This is done by specifying an option to the command. For example, "kill -9" sends a signal number nine, SIGKILL. AIX 3.2 supports the killpg() subroutine, which delivers a signal to all the processes in a specified process group.

Receiving signals

Signals can arrive at a process at any time, but the signal is only considered "delivered" when the process takes action on the signal. Until the signal is delivered, it is pending. Action is only taken on the signal while the process is running in user mode; however, some system calls, considered "slow" calls because they could block forever, can be interrupted by a signal. Examples of slow system calls include reads and writes to device and fifo files, sleep and pause calls, and some ioctl calls.

As mentioned earlier, processes have four choices of how to handle the reception of most signals. There are numerous subroutines available to the application programmer for defining which signals to ignore, block, and catch. Each subroutine has its own characteristics, and some have drawbacks to be aware

of. Since this book is not intended as a programming guide, the discussion of each subroutine is brief. Programmers should investigate each subroutine, as well as others not listed here, by consulting the manual pages or InfoExplorer.

Some signals, such as SIGKILL (9) and SIGSTOP(17), cannot be caught or ignored.

> **Author's Note:** This is why the "kill -9" command can kill a process when "kill" alone, which sends a SIGTERM (15) signal, may not work. A programmer can choose to ignore the SIGTERM signal, but SIGKILL cannot be ignored. I recommend that users and system administrators always try to terminate a process with "kill" before resorting to "kill -9" since the programmer may have included a handler to catch SIGTERM and gracefully shut down the process. Programmers cannot register signal handlers for SIGKILL.

The signal() subroutine allows a programmer to ignore or catch a signal or restore the default action for the signal. By default, AIX 3.2 implements the System V version of the signal() subroutine, which has one tricky characteristic. If a signal handler is registered via the System V signal() subroutine, then the signal is received, execution branches to the specified handler, but the catch option for the signal is not reregistered. If another signal of the same type is received after the first signal trips the handler, the default action for the signal is invoked, which usually results in the termination of the process. Figure 10.2 illustrates this problem.

Some programmers compensate for this problem by reregistering the handler within the handler. The problem is that a process might receive another signal of the same type before it has a chance to reregister the handler.

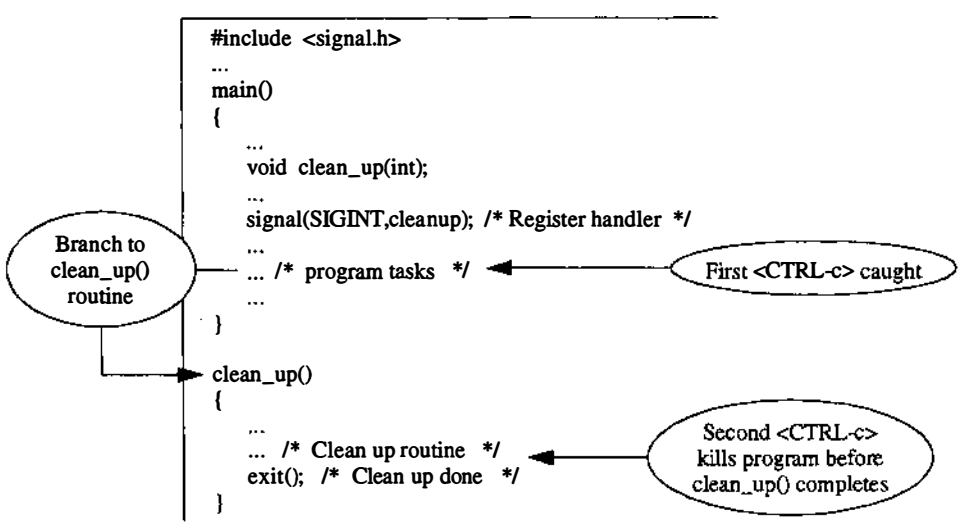

Figure 10.2 Signal catching problem.

Fortunately, AIX 3.2 also supports the BSD version of the signal() subroutine, which automatically reregisters the handler. The BSD version of the signal() subroutine is found in the /lib/libbsd.a library, which can be specified prior to the standard C library when a program is linked.

Even with the BSD version of the signal() subroutine, a problem exists where, if a signal handler is reregistered, a process might experience nested occurrences of the handler. This could cause undesirable results.

Problems like those described above have led to the term "unreliable signals" when referring to these API options. POSIX-compliant programs should use the sigaction() subroutine, which provides reliable signal control. The sigaction() subroutine includes three parameters. The first parameter specifies the signal number for which this action applies. The second parameter is a pointer to a sigaction structure, which is defined in the /usr/include/sys/signal.h header file. The application initializes the sigaction structure with the name of the signal handler routine (sa_handler), the set of signals to block (in addition to those already being blocked by the process), while the handler is invoked (sa_mask) and a flag to control various characteristics of the handler itself (sa_flag). A third, optional, parameter can be used to specify a sigaction structure to hold the previously defined signal action for the specified signal.

Signal masks are used to specify which signals to block. Signal masks are declared as a sigset_t data type as defined in /usr/include/sys/types.h. A sigset is a structure containing two unsigned longs, named losigs and hisigs. This provides 64 bits, one for each supported signal, to use as a mask. AIX 3.2 includes subroutines to manipulate the bits in sigset masks. They include sigemptyset() which zeros all of the bits in the mask, sigfillset() which sets all the bits in the mask, sigaddset() which sets a specified bit in the mask, and sigdelset(), which turns off a specified bit in the mask. The sigismember() subroutine is used to query the status of a bit in the mask. These subroutines are easier to use and read than bitwise operators.

The sigprocmask() subroutine uses a sigset mask to block signals. The same subroutine is used to unblock signals. The sigpending() subroutine allows a process to find out what signals are pending before unblocking. As mentioned earlier, many other subroutines are available for controlling signals.

Signals and the process table

The process table contains many fields used by the signal management subsystem. They are defined in the /usr/include/sys/proc.h header file within a union that is used for either signal information or, once the process becomes a zombie, exit status information. The fields include:

_p_segstate. A character flag that indicates that a signal has arrived or that the process is currently handling a signal.

_p_cursig. The current or last signal taken by the process.

_p_sigignore. A sigset_t data type that defines the signals currently being ignored by the process.

_p_sigcatch. A sigset_t data type that defines the signals currently being caught by the process.

_p_sigmask. A sigset_t data type that defines the signals currently being blocked by the process.

_p_sig. A sigset_t data type that indicates the signals that are pending. This field is returned by the sigpending() subroutine described earlier.

Figure 10.3 illustrates the fields in the process table used by the signal management subsystem.

The signal handling vector

Each process's user area contains an array called u_signal. The definition, found in /usr/include/sys/user.h, is

```
void (*u_signal[NSIG]) (int);
```

The array, which has a size of NSIG (defined as 64 in /usr/include/sys/signal.h), contains pointers to functions. The functions are the signal handler routines registered by the process. Figure 10.4 illustrates the use of the u_signal[] array.

Signals as IPC

While signals are a form of interprocess communication, their use is limited. They can only be implemented between processes that share a common effective user ID (unless the sender is running with root authority, with an effective

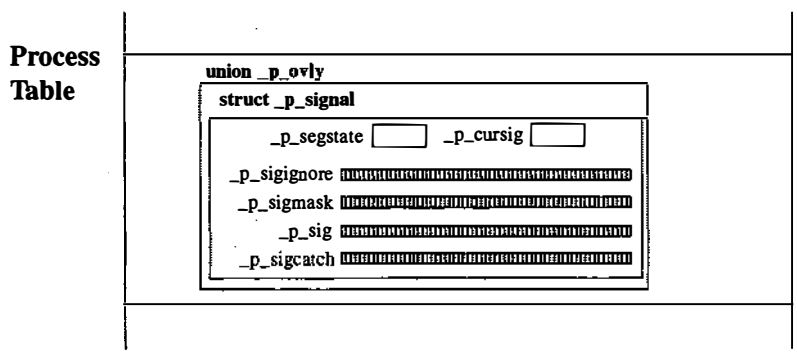

Figure 10.3 Signals and the process table.

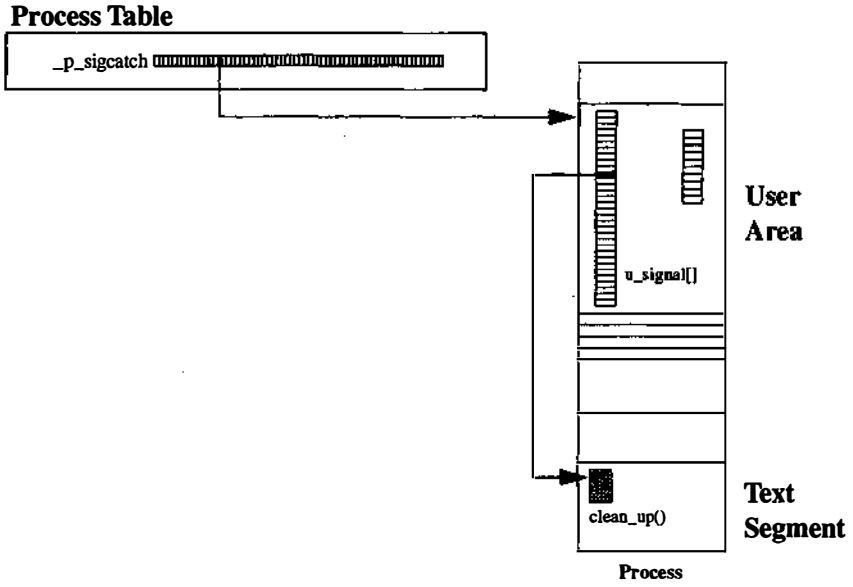

Figure 10.4 The u_signal[] array.

user ID of 0). Signals cannot communicate data but simply notify the receiver of some condition; therefore, the meaning of a signal is very limited. Finally, the receiver of a signal has no way of knowing which process sent it. Signals have a place in the IPC family, but data transfer must be accomplished by some other mechanism.

10.3 Unnamed Pipes

Pipes are a common tool of the shell. Most UNIX users have implemented pipes to forge simple tools, such as "who | wc -l," which reports the number of users logged in. Most users are unaware of how the shell creates a pipe. The section describes unnamed pipes as an IPC mechanism, useful for passing data from one process to another.

When a user issues the command "who | wc -l," the shell creates two processes, which are connected by a pipe. The standard output from the "who" command is redirected through the pipe to become the standard input of the "wc -l" command. Figure 10.5 illustrates how the shell implements a pipe.

The pipe() system call

Unnamed pipes are created by the pipe() subroutine. It allocates the two lowest available file descriptors from the calling process's file descriptor table. The

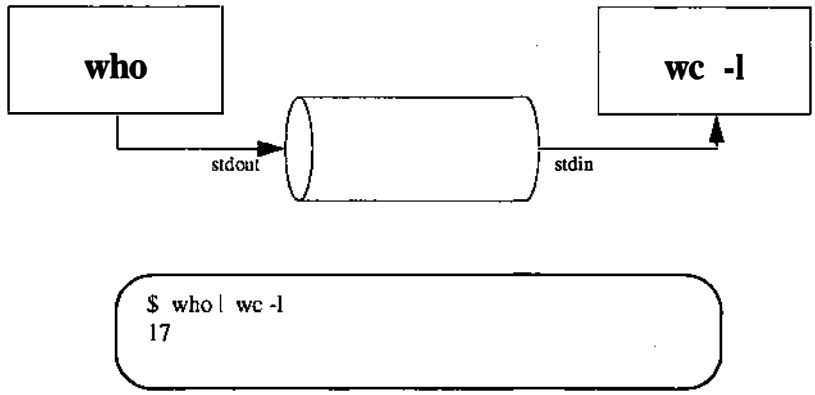

Figure 10.5 Shell pipes.

file descriptors are assigned to an array of two integers that is specified as a parameter to pipe(). The lower of the two file descriptors opens one side of the pipe as read-only. The higher of the two file descriptors opens the other side of the pipe as write-only. Figure 10.6 provides a code example and illustration of the pipe() subroutine.

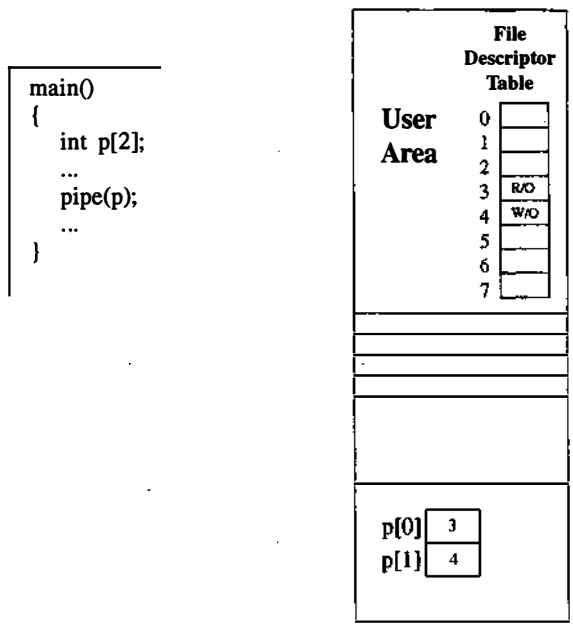

Figure 10.6 The pipe() subroutine.

Unnamed pipes and the file system

Pipes use the file system, as described shortly. Pipes created with the pipe() subroutine, while using the file system, do not have file names. This is why they are called "unnamed pipes."

The file descriptors of a pipe point to entries in the file table. There is one entry in the file table for each side (read and write) of the pipe. Pipes do not use the read/write offset field of the file table (see Chap. 8 for an explanation of the read/write offset). The two file table entries of a pipe point to a common vnode. The vnode of the pipe points to a gnode, which is contained within a specnode structure. The specnode structure (introduced in Chap. 7 and defined in /usr/include/sys/specnode.h) contains a union, called sn_data, that defines two pointers. One pointer, _sn_fnode, is used by unnamed pipes to point to a fifonode. The fifonode is also defined in the /usr/include/sys/specnode.h header file and includes the following fields:

ff_size. An unsigned long that indicates the current number of bytes held in the pipe.

ff_wptr. An unsigned short that indicates the current write offset for the pipe.

ff_rptr. An unsigned short that indicates the current read offset for the pipe.

ff_buf[NFBUF]. An array of caddr_t data types (character pointers). The pointers point to the buffers that hold the data as they move through the pipe. NFBUF is defined as eight.

Figure 10.7 illustrates the file system components employed to implement unnamed pipes in AIX 3.2.

Using unnamed pipes

Figure 10.7 shows what happens when a process calls the pipe() subroutine. The only thing wrong with the example is that now we have a process that can talk to itself through the pipe. This isn't very useful. To truly be called an IPC mechanism, another process must be involved. After building the pipe, the fork() system call is used to create a child process. Recall that a child process inherits its parents' file descriptor table. Therefore, the child inherits both sides of the pipe. Now, each process (parent and child) can close whichever side of the pipe they don't intend to use (unnamed pipes only support unidirectional data transfer). Figure 10.8 shows the final result of the pipe construction. Data can now be sent from the parent process to the child process. The parent process uses the write() subroutine, while the child process uses the read() subroutine.

> **Author's Note:** Unnamed pipes are unidirectional, but the programmer can decide which process does the "talking" and which process does the "listening." The talker closes the read side of the pipe and the listener closes the write side of the pipe.

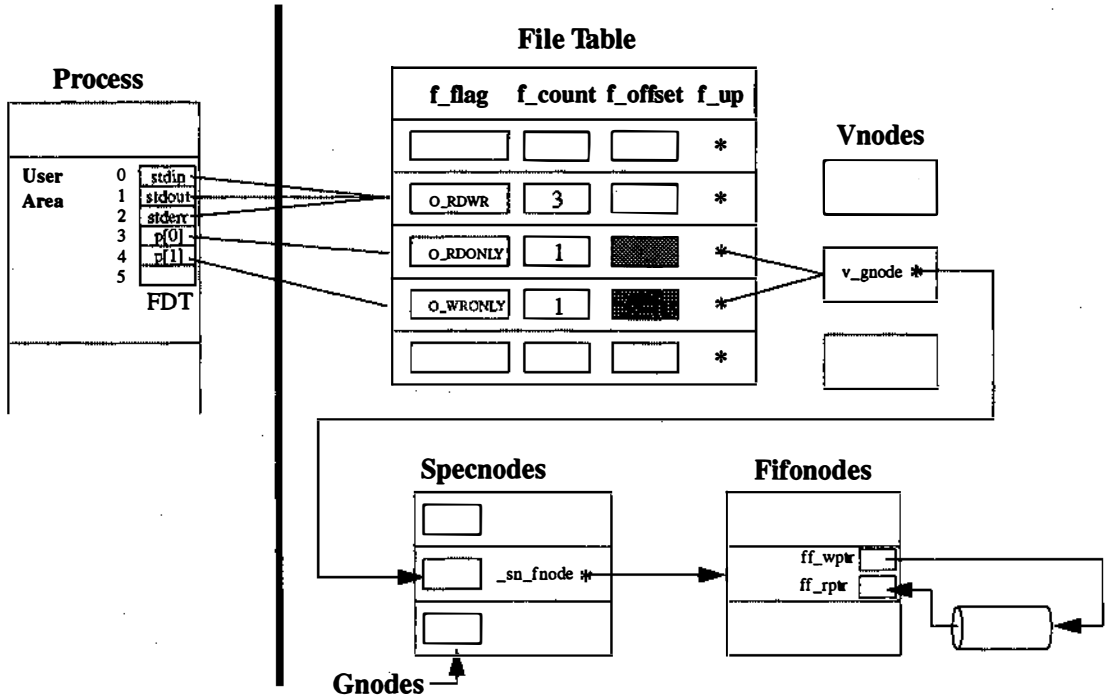

Figure 10.7 Unnamed pipes and the file system.

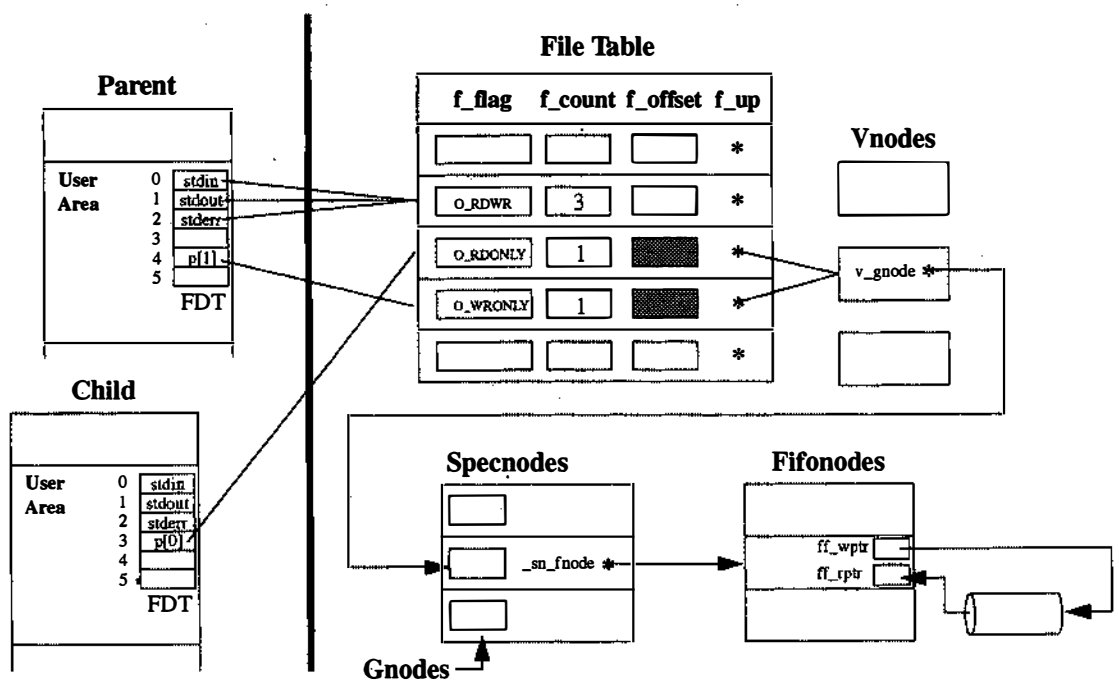

Figure 10.8 A pipe example.

I've noticed that most of my students prefer to designate the parent process as the talker and the child process as the listener. Perhaps their choices are based on personal experience.

One of the best features of unnamed pipes is that the data flow is automatically synchronized. If a pipe reader attempts to read from a pipe before the writer has had a chance to write to the pipe, the reader blocks until there is data in the pipe. On the other hand, if a pipe writer fills the body of a pipe with data and the reader has not pulled any of the data through the read side of the pipe, the writer blocks until there is room in the pipe for more data. The size of the body of the pipe is 32 kilobytes.

To complete the discussion of unnamed pipes, the code example given in Fig. 10.9 shows how pipes can be used to redirect the standard output of one process to the standard input of another process. Notice that the pipe descriptors remain after each process calls execl(). This illustrates why a process's file descriptor table survives exec-type calls.

Named pipes

The biggest problem associated with unnamed pipes is that their existence must be inherited. This limits them to serving only related processes (parent-

```
main()
{
  int rv, p[2];
  ...
  close(0);    /* Close stdin */
  close(1);    /* Close stdout */
  pipe(p);
  rv=fork();
  ...
  if (rv==0) /* Child */
  {
     close(p[1]); /* Close write side of pipe */
     dup(se); /* Reestablish stdin */
     execl("progb","progb",0);
     perror("progb failed");
     exit(1);
  }
  /* Parent */
  close(p[0]); /* Close read side of pipe */
  dup(se); /* Reestablish stdout */
  execl("proga","proga",0);
  perror("proga failed");
  exit(1);
}
```

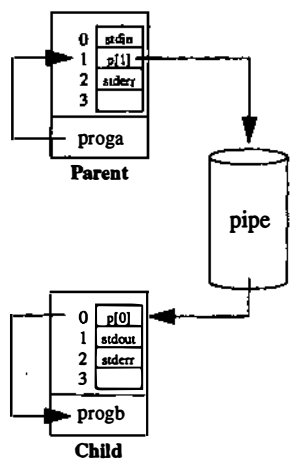

Figure 10.9 Piping stdin and stdout.

child, sibling-sibling, etc.). Since unnamed pipes have no file name or other type of handle, unrelated processes cannot participate. To solve this problem, UNIX-based systems include the concept of named pipes. Named pipes are also known as FIFO files (first-in, first-out) and are implemented as a special file type.

The internal working of named pipes is similar to that of unnamed pipes, but a named pipe has a file name created within a directory. This allows any process to open the named pipe for reading or writing. Data written to a named pipe are held in the FIFO file until another process reads from the named pipe.

In AIX 3.2, named pipes are created via the mkfifo() subroutine. Consult the manual page for mkfifo() or InfoExplorer for detailed information on programming with named pipes.

10.4 System V IPCs—An Introduction

System V IPCs include shared memory, semaphores, and message queues. Each of these three mechanisms has a specific application, yet their interfaces are similar. Each has a set of APIs that allow the creation, use, and control of the mechanism. The kernel maintains three tables of descriptors, one table for each mechanism. Each of the descriptors includes a common structure used for indicating the owner and permissions for the mechanism.

There are many similarities, from an application standpoint, between the way System V IPCs are implemented and the way file I/O is implemented. For instance, each System V IPC mechanism must be created by a process before other processes can use it. While it is in use, it is assigned to a descriptor, similar to a file descriptor, which is used as a handle when accessing the mechanism. The System V IPCs have permissions sets that specify read and write privileges for the IPC owner, the group, and others. Finally, each shared memory segment, semaphore set, or message queue continues to exist, even if all of the processes using them have terminated, until they are explicitly removed.

The ipc_perm structure

As mentioned earlier, the descriptor for each System V IPC mechanism includes a common structure that specifies the owner and group of the mechanism, as well as the permissions for the mechanism. The structure is called ipc_perm and is defined in the /usr/include/sys/ipc.h header file. Fields of the ipc_perm structure include:

uid. A uid_t data type that indicates the owner of the IPC mechanism.

gid. A gid_t data type that indicates the group associated with the IPC mechanism.

cuid. A uid_t data type that indicates the creator of the IPC mechanism. By default, the creator is also the owner; however, subroutines exist to allow the

owner to change ownership of the IPC mechanism. This changes the uid field. The cuid field never changes.

cgid. A gid_t data type that indicates the primary group of the creator of the IPC mechanism at the time the mechanism was created. Subroutines exist to allow the owner to change the group associated with the mechanism. This changes the gid field. The cgid field never changes.

mode. A mode_t data type that holds the permission bits for the IPC mechanism. Permission options for each mechanism type are discussed shortly.

seq. An unsigned short that indicates the slot number from the appropriate descriptor table.

key. A key_t data type (signed long) which holds the key for the IPC mechanism.

A key must be specified for each shared memory segment, semaphore set, or message queue when it is created. The application programmer selects a key that can uniquely identify the mechanism. All processes that intend to use the IPC mechanism must know the key. In a way, a key is similar to a file name.

Care must be taken in selecting a key. If two different applications try to use the same key value when creating separate shared memory segments, for instance, the first process will successfully create the segment, but the second process will fail, because the system deems that a segment associated with the key already exists. For this reason, programmers who choose to hard code key values should avoid obvious keys, such as 12345.

AIX 3.2 provides the ftok() subroutine to generate IPC keys. It takes a file path name and a project ID as parameters. Programs within an application suite can use ftok() to generate a unique key based on the path name of a file that is part of the suite. Another technique is to place the key value in a header file that is included by all programs in the application suite.

System V IPC system calls

The programming interface for creating, accessing, using, and controlling the System V IPC tools provides as consistent a syntax as possible between shared memory, semaphores, and message queues. The system calls shmget(), semget(), and msgget() are used to create or access existing shared memory segments, semaphore sets, and message queues, respectively. The system calls shmctl(), semctl(), and msgctl() are used to control shared memory segments, semaphore sets, and message queues, respectively. "Controlling" means examining the IPC's current settings, changing owners, groups, and permissions, and removing the mechanism. There are other control options that are unique to each mechanism type.

It's the system calls for using the System V IPC mechanisms that vary the most. The shmat() system call is used for attaching a shared memory segment

TABLE 10.3 System V IPC Subroutines

	Creating	Using	Controlling
Shared Memory	shmget()	shmat() shmdt()	shmctl()
Semaphores	semget()	semop()	semctl()
Message Queues	msgget()	msgsnd() msgrcv()	msgctl()

to a process's user space, while shmdt() is used for detaching it when finished. (Recall, from Chap. 7, that AIX 3.2 also allows a programmer to use shmat() to memory map files.) Semaphores are manipulated with the semop() system call. Finally, messages are posted to a message queue via the msgsnd() system call and retrieved from a message queue with msgrcv(). Table 10.3 recaps the primary System V IPC system calls. There are a few other system calls related to System V IPC that are not discussed here. Consult the InfoExplorer on-line documentation for more details.

10.5 AIX 3.2 Shared Memory

Shared memory is the quickest and easiest way for two or more processes to share large amounts of data. AIX 3.2 implements shared memory by allowing processes to attach commonly defined memory regions to their own memory space. A region, in this case, is a 256-megabyte virtual memory segment. Recall from Chap. 4 that each process's virtual memory image includes 16 256-Mb segments (for a 4-gigabyte address space), and that segments 3 through 10 are available for shared memory. Figure 10.10 illustrates the process's virtual memory image and the segments available for shared memory.

Creating shared memory segments—shmget()

To implement shared memory, a process calls shmget(), supplying a key, the size of the region to share (up to 256 Mb), and a flag which must include IPC_CREAT ORed with the desired permissions. The IPC_CREAT constant, along with other flag values, are defined in the /usr/include/sys/shm.h header file. The shmget() subroutine returns an integer which is the identifier for the shared memory instance. The identifier is much like a file descriptor, for it is used by the process, from this point on, to refer to the shared memory segment.

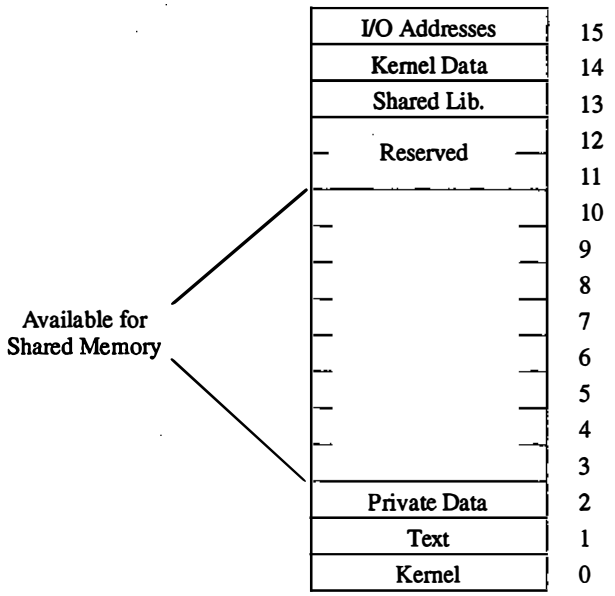

Figure 10.10 Shared memory segments.

> **Author's Note:** I use the term "segment," but the region of memory that is shared between processes has a definable size, which is specified when the region is created (per the shmget() parameter described above). The size limit for shared memory varies from one UNIX-based system to another. The size limit for AIX 3.2 is 256 Mb, or one segment. AIX 3.2 allocates an entire process segment to any shared memory region, regardless of the specified size of the region.

Once a process has created a shared memory segment, other processes call shmget(), specifying the same key, in order to access the shared memory identifier. No other process can call shmget() using the IPC_CREAT flag and a key for a segment that already exists. This results in an error returned from shmget().

> **Author's Note:** IPCs are the backbone of client-server applications. Since the System V IPCs only work between processes on the same system, their use for client-server applications is limited to situations where the client and the server are on the same machine. As a preference, I designate the server process as the creator of the IPC mechanism. For shared memory, I have the server process call shmget() with the IPC_CREAT flag. I have client processes call shmget() without the IPC_CREAT flag. If a client process runs before the server has had a chance to create the shared memory segment, the client's call to shmget() fails. When this happens, I usually have the client print an error message that instructs the user to check to see if the server is running.

Attaching shared memory segments—shmat()

The next step is for each participating process to call shmat(), specifying, as parameters, the shared memory identifier returned from shmget(), an address for mapping the shared memory segment, and any appropriate flags. (See the manual page for shmat() for a list of flags.) Instead of specifying an address for mapping the segment, a programmer can use the value 0, which instructs the kernel to choose a location. This is recommended so that the code is easily ported. The AIX 3.2 kernel usually selects the lowest available segment from those allocated for shared memory (segments 3 through 10). The shmat() routine returns a character pointer to the start of the shared memory segment.

Now the participating processes can assign data to the segment or examine the data in the segment. Since the processes are sharing the memory in real time, changes made by one process are instantly visible to other processes. This illustrates one problem with shared memory IPC. The programmer must implement some form of synchronization to prevent processes from updating the same memory at the same time. This problem can be solved many ways. One method of implementing synchronization for shared memory access is the use of semaphores, as discussed shortly.

Detaching shared memory segments—shmdt()

When a process concludes its use of a shared memory segment, the process should detach the segment. This is done by calling shmdt(). The shmdt() call takes the starting address of the segment (as returned from shmat()) for its only parameter. Once a process detaches a segment, another segment can be attached to the same location. A process automatically detaches all shared memory segments when the process terminates. Keep in mind that detaching a shared memory segment, even when done by the creator of the segment or the last process to have it attached, does not remove the segment from the system.

Removing a shared memory segment—shmctl()

A shared memory segment is removed from the system when a process with the same effective user ID as the creator or owner of the segment, or a process running with an effective user ID of 0 (root authority) calls the shmctl() subroutine, providing, as parameters, the shared memory identifier and the IPC_RMID flag. This causes the shared memory segment to be removed when all participating processes have detached it. The IPC_RMID symbolic constant is defined in the /usr/include/sys/shm.h header file. As mentioned previously, if no process explicitly removes the shared memory segment, it continues to exist even after all participating processes have terminated.

> **Author's Note:** This is true for all of the System V IPC mechanisms. Actually, it may be desirable to have another process, many days later perhaps, issue a shmget() call with the appropriate key and attach the segment.

The system maintains a reference count for attached shared memory segments. Since the segment won't actually be removed until the last process detaches it, the shmctl() call with the IPC_RMID flag can be called any time. This does, however, mean that if the reference count of processes with the segment attached ever reaches zero, the segment is lost forever.

> **Author's Note:** As stated earlier, the System V IPCs are similar to file and file I/O activity. As a recap, the shared memory segment is created by some process, just as a file is created. Processes attach the segment, just as processes open files. A descriptor is associated with the shared memory segment, much as a file descriptor is associated with an open file. (There is one significant difference here, however. With files, the descriptor is returned when the file is opened. With shared memory, the descriptor, or identifier, is returned by shmget(), not shmat().) When a process detaches a segment, the effect is similar to when a process closes a file, even to the point that all attached shared memory segments are automatically detached when a process terminates. Finally, some process is responsible for removing the shared memory segment from the system. This is similar to a process removing a file from the system.

Figure 10.11 provides a code example of shared memory IPC.

Program A

```
#include <sys/ipc.h>
#include <sys/shm.h>
...
main()
{
   int shmid, i;
   char *shm_start, *shm_p;
   int mykey=9087;
   ...
   if((shmid=shmget(mykey,256,IPC_CREAT|0666))<0)
        { perror(shmget failed"); exit(1); }
   ...
   if((shm_start=shm_p=shmat(shmid,0,0))==0)
        { perror("shmat failed"); exit(2); }
   ...
   for(i=1;i<=100;i++)
        *shm_p++ = 'X';
   ...
   shmdt(shm_start);
   ...
}
```

Program B

```
#include <sys/ipc.h>
#include <sys/shm.h>
...
main()
{
   int shmid, i;
   char *shm_start, *shm_p;
   int mykey=9087;
   ...
   if((shmid=shmget(mykey,0,0))<0)
        { perror(shmget failed"); exit(1); }
   ...
   if((shm_start=shm_p=shmat(shmid,0,0))==0)
        { perror("shmat failed"); exit(2); }
   ...
   for(i=1;i<=100;i++)
        printf("%c",*shm_p++);
   ...
   shmdt(shm_start);
   ...
   shmctl(shmid,IPC_RMID);
}
```

Figure 10.11 Using shared memory.

The shmid_ds table

The kernel maintains a table of shared memory ID data structures. The table is made up of shmid_ds structures as defined in the /usr/include/sys/shm.h header file. One slot in this table is allocated for each shared memory segment active on the system. Interesting fields from the shmid_ds structure include:

shm_perm. An embedded ipc_perm structure from /usr/include/sys/ipc.h. This structure contains the user IDs and group IDs of the creator and owner of the shared memory segment, along with the permissions and key value. The ipc_perm structure also holds the slot number for the shmid_ds table.

shm_segsz. An integer that holds the defined size of the shared memory segment.

shm_lpid. A pid_t data type that holds the process ID number of the process that performed the last operation on the segment.

shm_cpid. A pid_t data type that holds the process ID number of the creator of the shared memory segment. Recall that the ipc_perm structure holds the user ID of the creator.

shm_nattch. An unsigned short that holds the number of current attaches for the segment. This field serves as a reference count.

shm_atime. A time_t data type that holds the timestamp for the last shmat() performed on the segment.

shm_dtime. A time_t data type that holds the timestamp for the last shmdt() performed on the segment.

shm_ctime. A time_t data type that holds the timestamp of the last modification made to the shmid_ds structure itself.

shm_handle. A vmhandle_t that holds the segment ID number for the shared memory segment. This field establishes a link to the virtual memory manager.

Author's Note: The data type vmhandle_t always represents a segment ID number in AIX 3.2.

The u_segst array

As mentioned in the discussion of explicitly memory mapped files in Chap. 7, each process's user area (as defined in /usr/include/sys/user.h) includes an array called u_segst[NSEGS]. Its contents describe the use of each of the process's 16 segments (NSEG is defined as 16 in the /usr/include/sys/seg.h header files). If a segment is used for shared memory IPC, the value of the corresponding u_segst element is the address of the shmid_ds structure assigned

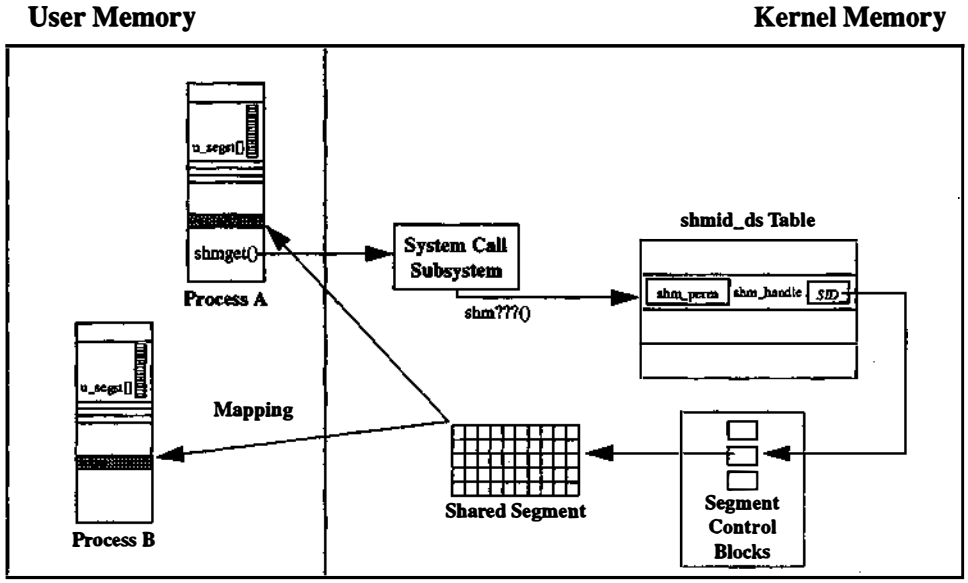

Figure 10.12 Kernel structures for shared memory.

to the segment. The address is stored in the shmptr pointer found within the u_ptrs union of the segstate structure (see /usr/include/sys/seg.h).

Figure 10.12 illustrates the structures used to define a shared memory segment.

The shminfo structure

The /usr/include/sys/shm.h header file defines a structure called shminfo. This structure defines the parameters for shared memory implementation.

> **Author's Note:** The parameters defined in the shminfo structure are tunable in most UNIX-based systems. They are not tunable in AIX 3.2 but should be adequate for most applications.

The shminfo structure includes the following fields. The values were determined by examining these fields with the crash facility.

shmmax. An integer that defines the maximum size of a shared memory region. The value is 268,435,456 (which is the size of an AIX 3.2 segment).

shmmin. An integer that defines the minimum size of a shared memory region. The value is 1.

shmmni. An integer that defines the maximum number of available shared memory identifiers. The value is 4096.

10.6 AIX 3.2 Semaphores

A semaphore is an IPC mechanism that is usually used to relay some condition to all participating processes. For instance, a semaphore can be used to synchronize access to some resource, such as a file or device.

Types of semaphores

Semaphores are implemented in sets. A set is an array of one or more semaphore values. The number of semaphore values in a semaphore set is established when the set is created. Each semaphore value can be initialized to any positive number. Most applications use semaphore sets that consist of a single semaphore value.

There are two types of semaphores, binary semaphores and positive semaphores. With binary semaphores, the semaphore value is initialized to 1. When a process wants to access the resource associated with the semaphore, it tests for a positive semaphore value. If the test is true, the value is decremented to zero, indicating that the process has control of the resource. When the process is finished accessing the resource, it increments the value of the semaphore. With positive semaphores, the semaphore value is initialized to some positive value, which indicates the number of parallel resources available. Each process wishing to access one of the parallel resources tests the semaphore for a positive value. If the test is true, the process decrements the value. When the process is finished accessing the parallel resource, it increments the semaphore value. Test and set operations on semaphores are guaranteed to be performed atomically by the kernel. This prevents a situation where a process receives a true result when testing for a positive semaphore value, only to be preempted before having a chance to decrement the value. Then, while preempted, another process tests and sets the semaphore value, resulting in a race condition.

One interesting aspect of semaphores is the possibility that a process could test and set a semaphore value, then terminate before resetting (releasing) the semaphore. To prevent this, a flag, called SEM_UNDO can be specified when the semaphore is set. The kernel maintains an adjustment value for "undoing" the semaphore.

Processes can also decide how to react when a semaphore test fails (i.e., when the semaphore value is not positive). By default, the process will block until the semaphore value becomes positive. The process can, however, include the IPC_NOWAIT flag when testing the semaphore. The IPC_NOWAIT flag causes a semaphore test to return immediately with a failure condition if the semaphore value is not positive.

Creating a semaphore set—semget()

A semaphore set is created by a call to the semget() subroutine, which includes, as parameters, the designated key for the set, the number of semaphore values to allocate to the set, and flags, which must specify IPC_CREAT "ORed" with

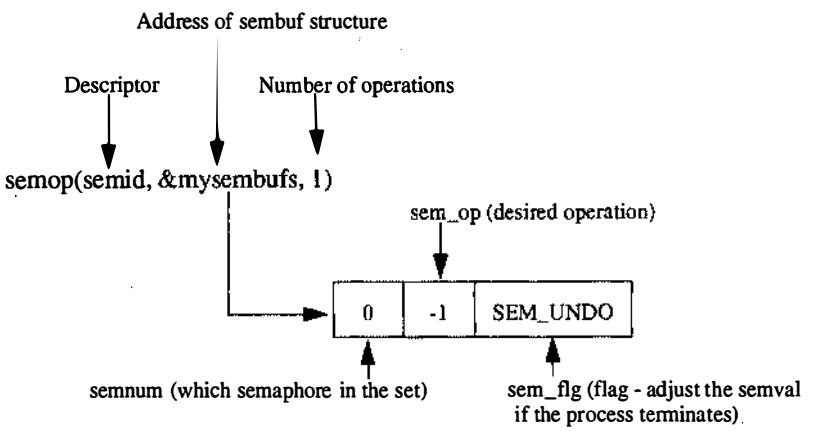

Figure 10.13 The sembuf structure and the semop() subroutine.

the permissions for the set. Other flag options allow the creating process to initialize the value(s) of the semaphore set. Consult the manual page for the semget() subroutine for more details.

The semget() subroutine returns a semaphore set identifier. This identifier is used with subsequent calls when accessing the semaphore set.

Once the semaphore set has been created, other processes can access the set by calling semget() with the same key, but without the IPC_CREAT flag. (See shmget() from Sec. 10.5 for details on creating and accessing the System V IPCs.) When calling semget(), the parameter used to specify the number of semaphores in the set should be zero for noncreating processes.

Semaphore operations—semop()

The semop() subroutine is used to perform operations on semaphores. This is the call that processes use to test and set the semaphore value, as well as unset the semaphore value. The semop() subroutine takes, as parameters, the semaphore set identifier (returned from the semget() call), a pointer to a single sembuf structure or an array of sembuf structures, and an integer which indicates the number of operations to be performed atomically by this semop() call. A single sembuf structure is used when there is only one operation for the semop() call to perform. If there are multiple operations to be performed by the semop() call, the pointer parameter points to an array of sembuf structures. Figure 10.13 illustrates how the semop() subroutine uses sembuf structures.

The sembuf structure, defined in the /usr/include/sys/sem.h header file, has three fields:

sem_num. An unsigned short that indicates to which semaphore in the set this operation applies. This is the index number for the semaphore within

the set. For single semaphore sets, which are most common, the sem_num value is 0.

sem_flg. A short that contains flags specified by the programmers. Flags include SEM_UNDO and IPC_NOWAIT, both of which are described below. (The sem_flg is actually the third of the three parameters.)

sem_op. A short integer which indicates the operation to perform. The following operations are supported.

If the sem_op value is a positive number, that number is added to the semaphore value. This represents releasing the semaphore. If the SEM_UNDO flag is set in the sem_flg field, the value of sem_op is subtracted from the adjustment field for the semaphore.

If the sem_op value is a negative number, the kernel verifies that the semaphore's value is greater than or equal to the absolute value of sem_op. If it is, then the absolute value of sem_op is subtracted from the semaphore's value. This represents gaining access to the resource. The semaphore value must remain greater than or equal to zero. If the SEM_UNDO flag is set in the sem_flg field, the value of sem_op is added to the adjustment field for the semaphore.

If the semaphore's value is less than the absolute value of the sem_op field, the resource is not available. If the IPC_NOWAIT flag is set in the sem_flg field, the semop() call returns as failed. The process can then take whatever action the programmer deems appropriate. If the IPC_NOWAIT flag is not specified in the sem_flg field, the process sleeps until either the semaphore's value becomes greater than or equal to the absolute value of sem_op, the semaphore set is removed, or a signal is received to end the sleep.

If the sem_op value is zero and the semaphore's value is zero, the operation is completed. This represents a test of the semaphore value for zero, which is sometimes used by applications that proceed when the semaphore value is zero. If the sem_op value is zero and the semaphore's value is not zero, the process blocks until the semaphore value is zero, or the semaphore is removed from the system, or a signal is received to end the sleep. If the IPC_NOWAIT flag is set in the sem_flg field and the semaphore's value is not zero, the semop() call fails.

Controlling the semaphore set—semctl()

Semaphore sets are controlled with the semctl() subroutine. This subroutine includes commands for examining the semaphore set characteristics (IPC_STAT), changing the owner, group, or permissions of the semaphore set (IPC_SET), and removing the semaphore set (IPC_RMID). In addition, commands exist for retrieving and initializing one or more of the semaphore values in the set. As with shared memory, only a process with the same effective user ID of the creator or owner of a semaphore set can remove that semaphore set, unless the process is running with an effective user ID or 0 (root authority).

The semid_ds table

The kernel maintains a table of semaphore set ID data structures. The table is made up of semid_ds structures as defined in the /usr/include/sys/sem.h header file. One slot in this table is allocated for each semaphore set active on the system. Interesting fields from the semid_ds structure include:

sem_perm. An embedded ipc_perm structure from /usr/include/sys/ipc.h. This structure contains the user IDs and group IDs of the creator and owner of the semaphore set, along with the permissions and key value. The ipc_perm structure also holds the slot number for the semid_ds table.

sem_base. A pointer to the first semaphore in the set. This field points to a sem structure, which represents one semaphore in a set. The sem structure, which is also defined in the /usr/include/sys/sem.h header file, is described shortly.

sem_nsems. An unsigned short that holds the number of semaphores in the set. This indicates the number of sem structures in the array.

sem_otime. A time_t data type that holds the timestamp of the last semop() call for this set.

sem_ctime. A time_t data type that holds the timestamp of the last change made to the semid_ds entry.

The sem structure

Each semaphore in a set is represented by a sem structure, which contains the following fields:

semval. An unsigned short that holds the value of the semaphore. This is the field that was described in the previous section when the semop() subroutine was discussed.

sempid. A pid_t data type that holds the process ID number of the process that performed the last semop() call on this semaphore.

semncnt. An unsigned short that indicates the number of processes waiting for the semaphore value to be greater than its current value.

semzcnt. An unsigned short that indicates the number of processes waiting for the semaphore value to equal zero.

The sem_undo structure

The /usr/include/sys/sem.h header file includes the definition of a sem_undo structure. There is one chain of sem_undo structures allocated in the kernel for each active process on the system. The u_semundo pointer, found in each process's user area (see /usr/include/sys/user.h) links the process to its chain of sem_undo structures. The number of sem_undo structures linked into a

process's chain is determined by the number of semop() calls, with the SEM_UNDO flag set made by the process. The sem_undo structure contains the following fields:

un_np. A pointer to the next sem_undo structure in the chain for this process

un_cnt. A short integer that holds the number of active entries in the chain

un_aoe. A short integer (adjust on exit) that holds the adjustment value

un_num. An unsigned short that holds the semaphore number within a set

un_id. An integer that holds the semaphore set identifier

The last three fields, un_aoe, un_num, and un_id, are defined within a structure called un_ent. An array of un_ent structures is allocated to the sem_undo structure so that each element of the array contains the adjustment information for each semop() call that included the SEM_UNDO flag.

Figure 10.14 illustrates the kernel components for a couple of semaphore sets.

The seminfo structure

The /usr/include/sys/sem.h header file defines a structure called seminfo. This structure defines the parameters for semaphore implementation.

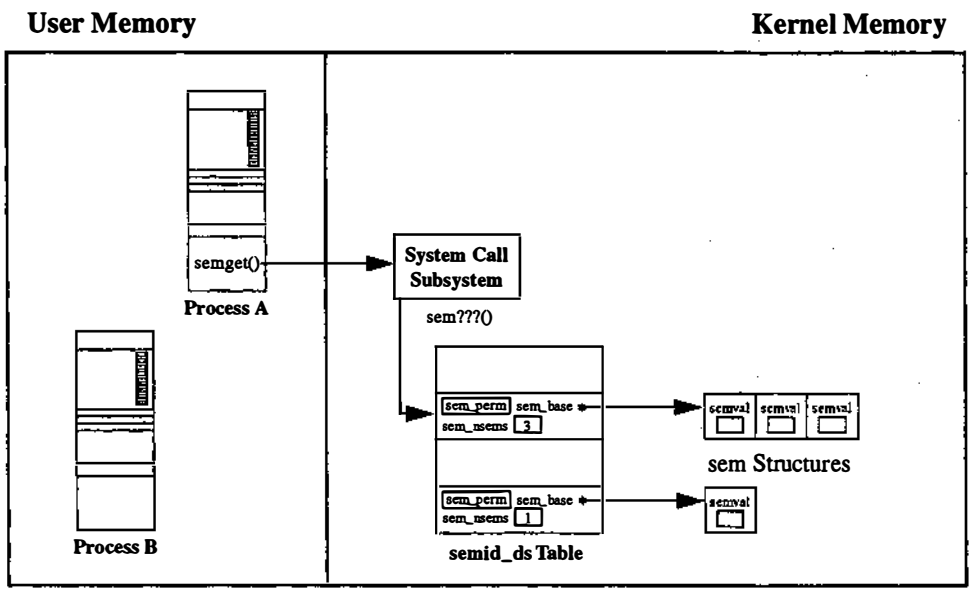

Figure 10.14 Kernel structures for semaphores.

Author's Note: As with shminfo, the parameters defined in the seminfo structure are tunable in most UNIX-based systems. They are not tunable in AIX 3.2 but should be adequate for most applications.

The seminfo structure includes the following fields. The values were determined by examining these fields with the crash facility.

semmni. An integer that defines the maximum number of available semaphore identifiers. The value is 4096.

semmsl. An integer that defines the maximum number of semaphores per set. The value is 102400.

semopm. An integer that defines the maximum number of operations per semop() call. The value is 1024.

semume. An integer that defines the maximum number of undo entries per process. The value is 1024.

semusz. An integer that defines, in bytes, the size of the undo structure. The value is 8208.

semvmx. An integer that defines the maximum semaphore value. The value is 32767.

semaem. An integer that defines the maximum "adjust on exit" value. The value is 16384.

A semaphore example

Figure 10.15 provides a code example of semaphores. Note the use of the SEM_UNDO flag.

10.7 AIX 3.2 Message Queues

The third, and final, System V IPC is message queues. A message queue is like a bulletin board on which a process can post a message. Another process can later "pick up" the message, thus removing the message from the queue. Message queues offer a great deal of flexibility and capacity for data sharing. A single message can be up to 64 kilobytes in size. Messages can be "addressed" for an intended receiver, which allows a single message queue to serve any number of processes.

Creating a message queue—msgget()

A message queue is created with a call to msgget(). The msgget() subroutine takes two parameters: the designated key for the message queue, and a flag. The process that creates the message queue must include the IPC_CREAT flag "ORed" with the desired permissions for the queue. All other processes must

Semaphore Routines

```
#include <sys/ipc.h>
#include <sys/sem.h>
...
int semid;  /* semaphore descriptor */
struct sembuf getit={0,-1,SEM_UNDO};
struct sembuf dropit={0,1,0};
...
sem_create()
{
    int mykey=5485;
    ...
    if((semid=semget(mykey,1,IPC_CREAT|0666))<0)
    perror(semget failed");
    ...
}
```

```
sem_get()
{
    semop(semid,&getit,1);
}

sem_drop
{
    semop(semid, &dropit, 1);
}
```

Figure 10.15 A semaphore example.

call the msgget() subroutine, providing the designated key but not specifying the IPC_CREAT flag. (See shmget() from Sec. 10.5 for details on creating and accessing the System V IPCs.)

The msgget() call returns an identifier for the message queue. The identifier is used with subsequent system calls as a handle for the message queue. Consult the manual page for msgget() for details.

Message structures—the msgbuf structure

Before discussing how messages are sent and received, it is important to understand the structure of a message, as expected by the kernel. The /usr/include/sys/msg.h header file defines a msgbuf structure that represents an example of a message. Actually, the kernel only cares that the first field in the msgbuf structure contains a mtyp_t data type (integer) called "mtype" that specifies the message type. Beyond that, the programmer is free to customize the message body to suit the needs of the application. Figure 10.16 illustrates the msgbuf template and how it might be customized for a client-server application.

Sending messages—msgsnd()

The msgsnd() subroutine is used to post a message to a queue. It takes, as parameters, the identifier of the message queue, as returned from the msgget() call, a pointer to the customized msgbuf structure that contains the outgoing message, the size of the message, and a flag value. Flag options include

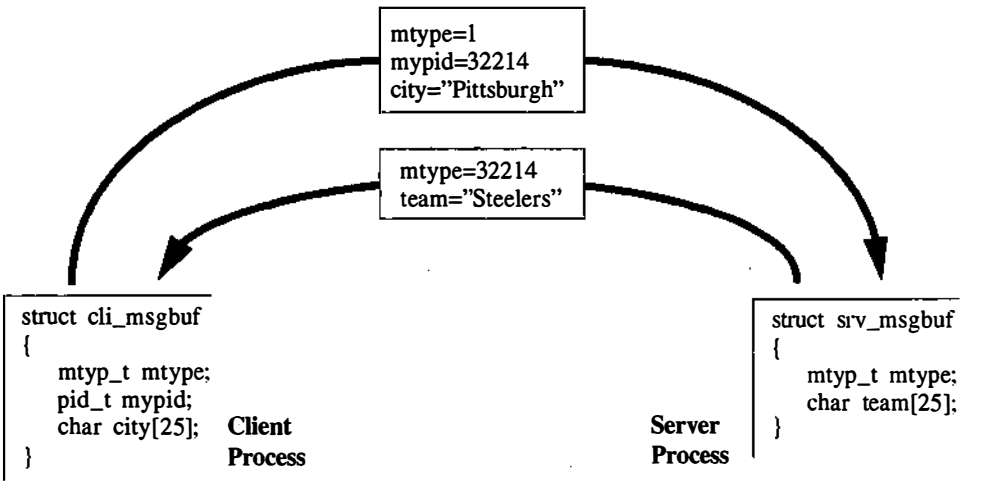

Figure 10.16 The msgbuf structure.

IPC_NOWAIT, which specifies that if there is not enough space on the message queue for the message, or if the queue has reached its maximum allowed number of posted messages, the msgsnd() call will fail and return to the process. The default action for either condition is to cause the process to block until there is sufficient space on the queue for the message or the current count of messages on the queue is no longer at the maximum allowed.

Receiving messages—msgrcv()

Processes pull messages from a message queue by calling msgrcv(). The msgrcv() subroutine takes, as parameters, the message queue identifier returned from msgget(), a pointer to the customized msgbuf structure that will hold the incoming message, the expected size of the message, an integer to indicate the message type, and a flag value. Flag options include IPC_NOWAIT, which specifies that if there are no messages of the type specified, the msgrcv() will fail and return to the process. The default action for msgrcv() when there are no messages of the specified type on the queue is to block.

Message queues work in a fifo (first in, first out) fashion. If a process calls msgrcv() and specifies a message type as zero, the oldest message of any type is pulled from the queue and received by the process. If a process calls msgrcv() and specifies a message type other than zero, the oldest message of the type specified is pulled from the queue and received by the process.

The size of any message is specified by the sending process when the message is posted to a queue. If a receiver specifies a message size that is smaller

than the actual message, the msgrcv() call fails and an error is returned to the receiving process. The receiving process can include the MSG_NOERROR flag for msgrcv(), which means that a message that is larger than the size specified by the receiver will be truncated to the specified size. The receiving process is not made aware that the message has been truncated.

Controlling message queues—msgctl()

As with shared memory and semaphores, message queues must be explicitly removed when they are no longer required. The msgctl() subroutine is called, with the IPC_RMID flag, to remove a message queue. Only a process with the same effective user ID as the creator or owner of the message queue, or a process with an effective user ID of 0 (root authority) may remove a message queue from the system.

Other commands for msgctl() include IPC_STAT, which allows the characteristics of a message queue to be queried, and IPC_SET, which allows the characteristics of a message queue to be changed. Message queue characteristics include the owner, group, and permissions.

A message queue example

Figure 10.17 provides a code example of a client-server application using message queues. Since the application supports multiple clients, it uses a multiplexed message queue.

The msqid_ds table

The kernel maintains a table of message queue ID data structures. The table is made up of msqid_ds structures as defined in the /usr/include/sys/msg.h header file. One slot in this table is allocated for each message queue active on the system. Interesting fields from the msqid_ds structure include:

msg_perm. An embedded ipc_perm structure from /usr/include/sys/ipc.h. This structure contains the user IDs and group IDs of the creator and owner of the message queue, along with the permissions and key value. The ipc_perm structure also holds the slot number for the msqid_ds table.

msg_first. A pointer to the msg structure of the first message on the queue. Each posted message includes an msg structure (described shortly) which serves as a header for the message.

msg_last. A pointer to the msg structure of the last message on the queue.

msg_cbytes. An unsigned short that indicates the current number of bytes on the queue.

msg_qnum. An unsigned short that indicates the number of messages current on the queue.

Server

```
#include <sys/types.h>
#include <sys/types.h>
#include <sys/msg.h>
...
main()
{
   int msgid, msgkey=6792;
   struct
   {
      mtyp_t mtype;
      pid_t clipid;
      char city[25];
   } srv_msgin;
   struct
   {
      mtyp_t mtype;
      char team[25];
   } srv_msgout;
   msgid=msgget(msgkey,IPC_CREAT|0666);
   msgrcv(msgid,&srv_msgin,sizeof(srv_msgin),1,0);
   ... /* Look up team for city */
   ... /* Populate srv_msgout.team with
          results of lookup and srv_msgout.mtype
          with value of srv_msgin.mypid */
   msgsnd(msgid,&srv_msgout,sizeof(srv_msgout),0);
   ...
   msgctl(msgid, IPC_RMID);
   ...
}
```

Client

```
#include <sys/types.h>
#include <sys/types.h>
#include <sys/msg.h>
...
main()
{
   int msgid, msgkey=6792;
   struct
   {
      mtyp_t mtype;
      pid_t mypid;
      char city[25];
   } cli_msgout;
   struct
   {
      mtyp_t mtype;
      char team[25];
   } cli_msgin;
   msgid=msgget(msgkey,0);
   ...
   ... /* Ask user for city */
   ... /* Populate cli_msgout.city with dsired city,
          cli_mtype with 1, and cli_msgout.mypid
          with return from getpid() */
   msgsnd(msgid,&cli_msgout,sizeof(cli_msgout),0);
   msgrcv(msgid,&cli_msgin,sizeof(cli_msgin,getpid(),0);
   printf("The team is %s.\n",cli_msgin.team);
   ...
}
```

Figure 10.17 A message queue example.

msg_qbytes. An unsigned short that indicates the maximum number of bytes on the queue.

msg_lspid. A pid_t data type that indicates the process ID number of the process that issued the last msgsnd() call on the queue.

msg_lrpid. A pid_t data type that indicates the process ID number of the process that issued the last msgrcv() call on the queue.

msg_stime. A time_t data type that holds the timestamp of the last msgsnd() call issued on the queue.

msg_rtime. A time_t data type that holds the timestamp of the last msgrcv() call issued on the queue.

msg_ctime. A time_t data type that holds the timestamp of the last change made to the msqid_ds structure.

Additional fields in the msqid_ds structure keep track of processes waiting to send messages to or receive messages from the queue.

The msg structure

Each message on a queue is represented by an msg structure, as defined in the /usr/include/sys/msg.h header file. The msg structures for a queue are linked together, and as seen in the previous section, the msqid_ds structure for the queue points to the msg structures for first (oldest) and last (newest) messages on the queue.

The msg structure includes the following fields:

msg_next. A pointer to the msg structure for the next message on the queue

msg_attr. A pointer to an msg_hdr structure, as defined below

msg_ts. An unsigned short that indicates the message text size

msg_spot. A character pointer to the actual message

Each msg structure points to an msg_hdr, also defined in /usr/include/sys/msg.h, that includes the following fields:

mtype. An mtyp_t data type (integer) that indicates the message type.

MSGX. A symbolic constant defined just above the msg_hdr structure in the /usr/include/sys/msg.h header file. It includes the following four fields:

mtime. A time_t data type that indicates the time that the message was sent.

muid. A uid_t data type that holds the effective user ID number of the author of the message.

mgid. A gid_t data type that holds the effective group ID number of the author of the message.

mpid. A pid_t data type that holds the process ID number of the author of the message.

Figure 10.18 illustrates the kernel components involved in message queues.

The msginfo structure

The /usr/include/sys/msg.h header file defines a structure called msginfo. This structure defines the parameters for message queue implementation.

> **Author's Note:** As with shminfo and seminfo, the parameters defined in the msginfo structure are tunable in most UNIX-based systems. They are not tunable in AIX 3.2 but should be adequate for most applications.

The msginfo structure includes the following fields. The values were determined by examining these fields with the crash facility.

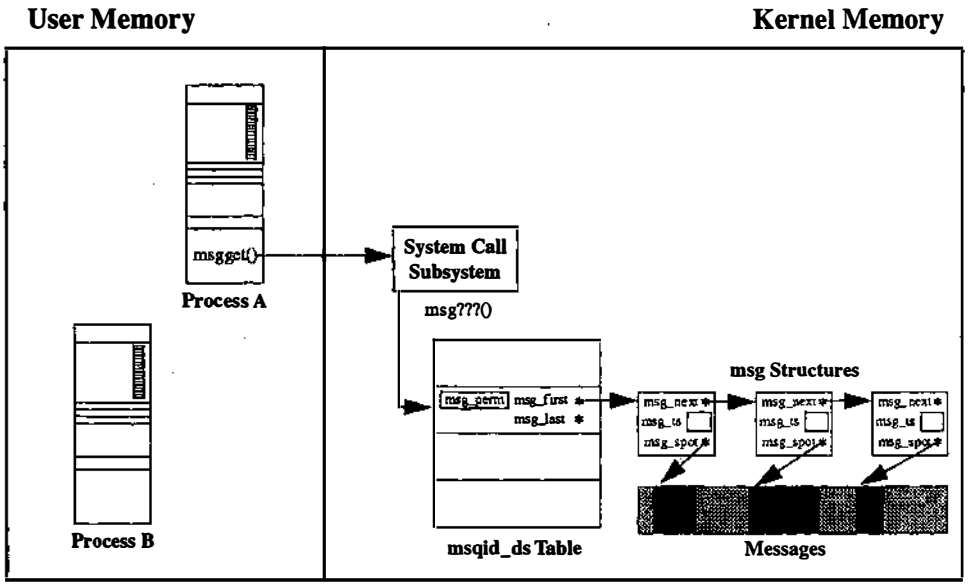

Figure 10.18 Kernel structure for message queues.

msgmax. An integer that defines the maximum message size. The value is 65,536.

msgmnb. An integer that defines the maximum number of bytes on a queue. The value is 65,536.

msgmni. An integer that defines the maximum number of message queue identifiers. The value is 4096.

msgmnm. An integer that defines the maximum number of messages allowed per queue. The value is 8192.

Author's Note: AIX 3.2 includes two commands that can be executed from the shell to manage System V IPC mechanisms. The ipcs command displays information about shared memory segments, semaphores, and message queues. The ipcrm command is used to remove specified shared memory segments, semaphores, and message queues. Consult the manual pages for each of these commands or the InfoExplorer on-line documentation for further information on ipcs and ipcrm.

Index

ABOUT THE AUTHOR

David A. Kelly's activities have put him at the very forefront of AIX developments. He is a principal of Kelly/Lloyd Associations in Plano, Texas, a firm that specializes in developing and presenting AIX seminars. He has more than 12 years of direct involvement in the UNIX environment and has developed highly successful courses for such technology leaders as IBM, AT&T, EDS, and many others.