System Interface Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-4750–05
June 2000

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Adobe PostScript

Please
Recycle

Contents

Preface

Purpose

Read this guide for information about system interfaces provided by SunOS™ libraries. Rather than teaching you to write programs, this guide supplements programming texts by concentrating on other elements that are part of getting programs into operation.

Audience and Prerequisite Knowledge

This guide addresses programmers. Expert programmers, such as those developing system software, might find that this guide lacks the depth of information they need. Expert programmers should see the *Solaris 8 Reference Manual Collection*.

Knowledge of terminal use, of a UNIX system editor, and of the UNIX system directory and file structure is assumed. Read the *OpenWindows User's Guide* to review these basic tools and concepts.

The C Connection

The SunOS system supports many programming languages. Nevertheless, the relationship between this operating system and C has been and remains very close.

Most of the code in the operating system is written in the C language. So, while this guide is intended to be useful to you no matter what language you are using, most of the examples assume you are programming in C.

Hardware and Software Dependency

Except for hardware-specific information such as addresses, most of the text in this book applies to any computer running the Solaris 8 operating environment and compatible versions.

If commands work differently in your system environment, your system might be running a different software release. If some commands do not seem to exist, they might be in packages that are not installed on your system—talk to your system administrators to find out what commands you have available.

Command References

When a command is mentioned in a section of the text for the first time, a reference to the manual section where the command is formally described is included in parentheses: `command`(section). Numbered sections are in the *Solaris 8 Reference Manual Collection*.

For example, “See priocntl(2)” tells you to look at the `priocntl` page in section 2 of the *Solaris 8 Reference Manual Collection*.

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system might produce slightly different output. Some displays depend on a particular machine configuration that might differ from yours.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	**Meaning**	**Example**
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	**Prompt**
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to the API

A goal of Sun Microsystems is to define the architectural interfaces of the Solaris™ operating environment. There are two reasons:

- The system interface is an effective "contract" with customers. It is completely specified in *Solaris 8 Reference Manual Collection*.
- The interfaces in a particular version of the product are preserved in future versions of the product, which maintains upward compatibility in subsequent releases of the Solaris operating environment.

Programming Interface

Solaris offers many kinds of "interfaces", such as the programming interface, elements of the user interface, protocols, and rules about naming and the locations of objects in the file system. One of the most important interfaces to the system is the programming interface — the one offered to developers. The programming interface has two major parts: one seen by developers of applications, which is the API, and one seen by developers of system components, such as device drivers and platform support modules, which is the SPI (system programming interface).

Each programming interface to Solaris is also "visible" to the developer at two levels: source level and binary. When you see the acronyms API and SPI, this indicates the source level programming interface to the system. The terms Application Binary Interface (ABI) and System Binary Interface (SBI) indicate the binary interfaces corresponding to the respective source level programming interfaces. (Because the phrase "the ABI" can be confused with other binary interfaces, the "Solaris ABI" is refered to only by name.)

Interface Functions

The SunOS 5.0 through 5.8 functions discussed in this manual are the interfaces between the services provided by the kernel and application programs. The functions described in *man pages section 2: System Calls*, *man pages section 3: Basic Library Functions*, *man pages section 3: Curses Library Functions*, *man pages section 3: Extended Library Functions*, *man pages section 3: Library Interfaces and Headers*, *man pages section 3: Networking Library Functions*, *man pages section 3: Threads and Realtime Library Functions*, are an application's interface to the SunOS 5.0 through 5.8 operating system. These functions are how an application uses facilities such as the file system, interprocess communication primitives, and multitasking mechanisms. This manual is one of a set that describe major elements of the API. Other manuals in the set are *STREAMS Programming Guide*, *Multithreaded Programming Guide*, *Network Interface Guide*, and so forth.

When you use the library routines described in sections 2 and 3 of the *Solaris 8 Reference Manual Collection*, the details of their implementation are transparent to the program. For example, the function `read` underlies the `fread` implementation in the standard C library.

A C program is automatically linked to the invoked functions when you compile the program. The procedure might be different for programs written in other languages. See the *Linker and Libraries Guide* for more information.

Libraries

Solaris provides both static and dynamic implementations of libraries. Static libraries do not provide an interface; they provide only an implementation. The application programming interface of Solaris is made available to developers through the shared libraries (also called shared objects). In the runtime environment, a dynamic executable and shared objects are processed by the runtime linker to produce a runnable process. The official API to the system is the interface between an application and the dynamic shared libraries.

Static Libraries

The traditional, static, implementation of libraries (`.a` files or archives), do not separate the application programming interface from its implementation (the contents of the library). When an application is linked to a static library, the object code that implements that library is bound into the executable object that results from the build. The source-level programming interface to the library can be preserved, but the application must be relinked to produce an executable that runs on a later version of an operating system. Future binary compatibility is assured only when shared libraries are used.

The presence of static libraries is a historical artifact and there is no mechanism to define their interfaces in a way that is separate from their implementation. For this reason, use of static libraries should be avoided by new applications.

Dynamic libraries

Unlike the static libraries, shared libraries do separate the application programming interface from the implementation. The interface is bound to an implementation of the library only at runtime. This allows SMI to evolve the library's implementation - such as changing internal interfaces, while maintaining the API and preserving binary compatibility with applications built against it.

Interface Taxonomy

The interface taxonomy classifies commitment level of an interface. The commitment level identifies who can, or how to, use the interface. Definitions:

Open specification	An interface specification that customers can use freely (build products that use this implementation of the interface). Others are free to provide alternative implementations without licensing or legal restrictions.
Closed specification	An interface specification is not published.
Compatible change	A change to an interface or its implementation that has no effect on previously valid programs.
Incompatible change	A change to an interface or its implementation that makes previously valid programs invalid. This might include bug fixes or performance degradation. This does not include programs that depend on unspecified "artifacts of the implementation."

Standard Classification

Specification	Open
Incompatible Change	Major release (X.0)
Examples	POSIX, ANSI-C, Solaris ABI, SCD, SVID, XPG, X11, DKI, Ethernet

Standard interfaces are those whose specification is controlled by a group outside of Sun. This includes standards such as POSIX and ANSI C, as well as industry specifications from groups such as X/Open, the MIT X-Consortium, and the OMG.

Public Classification

Specification	Open
Incompatible Change	Major release (X.0)
Examples	Sun DDI, XView, ToolTalk, NFS protocol, Sbus, OBP

These are interfaces whose specification is completely under Sun Microsystem's control. The specification of these interfaces and will remain compatible with the published specification.

Obsolete Classification

Specification	None
Incompatible Change	Minor release (.X.0)
Examples	RFS

An interface no longer in general use. An existing interface can be downgraded from some other status (such as Public or Standard) to Obsolete through a standard program to communicate the change in commitment to customers.

A change in commitment requires one year's notice to the customer base and the Sun product development community of the intended obsoleting of the interface. A full year must elapse before delivering a product that contains a change incompatible with the present status of the interface.

Acceptable ways to notify customers includes letters to customers on support contracts, release notes or product documentation, or announcements to customer forums appropriate for the interface in question.

The notice of obsolescence is considered to be "public" information in that it is freely available to the customers. This should not require specific actions to "publish" the information, such as press releases or similar forms of publicity.

CHAPTER **2**

Processes

This chapter describes processes and the library functions that operate on them.

Overview

Executing a command starts a process that is numbered and tracked by the operating system. Processes are always generated by other processes. For example, log in to your system running a shell, then use an editor such as `vi`(1). Take the option of invoking the shell from `vi`(1). Execute `ps`(1) and you see a display resembling this (which shows the results of `ps -f`):

```
UID     PID     PPID    C       STIME           TTY     TIME    CMD
abc     24210   1       0       06:13:14        tty29   0:05    -sh
abc     24631   24210   0       06:59:07        tty29   0:13    vi c2
abc     28441   28358   80      09:17:22        tty29   0:01    ps -f
abc     28358   24631   2       09:15:14        tty29   0:01    sh -i
```

User `abc` has four processes active. The process ID (PID) and parent process ID (PPID) columns show that the shell that started when user `abc` logged on is process `24210`; its parent is the initialization process (process ID `1`). Process `24210` is the parent of process `24631`, and so on.

A program might need to run one or more other programs based on conditions it encounters. Reasons that you might not want to create one large executable include:

- You might want to execute two, or more, of the modules concurrently.

- The load module might get too big to fit in the maximum process size for your system.
- You might not have control over the object code of all the other modules you want to include.

The fork(2) and exec(2) functions let you create a new process (a copy of the creating process) and start a new executable in place of the running one.

Functions

The functions listed in Table 2–1 are used to control user processes:

TABLE 2–1 Process Functions

Function Name	Purpose
fork(2)	Create a new process

exec(2)
execl(2)
execv(2)
execle(2)
execve(2)
execlp(2)
execvp(2)
	Execute a program

TABLE 2–1 Process Functions *(continued)*

Function Name	**Purpose**
`exit`(2) `_exit`(2)	Terminate a process
`wait`(2)	Wait for a child process to stop or terminate
`dladdr`(3DL)	Translate address to symbolic information
`dlclose`(3DL)	Close a shared object
`dlerror`(3DL)	Get diagnostic information
`dlopen`(3DL)	Open a shared object
`dlsym`(3DL)	Get the address of a symbol in a shared object
`setuid`(2) `setgid`(2)	Set user and group IDs
`setpgrp`(2)	Set process group ID
`chdir`(2) `fchdir`(2)	Change working directory

TABLE 2–1 Process Functions *(continued)*

Function Name	Purpose
`chroot`(2)	Change root directory
`nice`(2)	Change priority of a process
`getcontext`(2) `setcontext`(2)	Get and set current user context
`getgroups`(2) `setgroups`(2)	Get or set supplementary group access list IDs
`getpid`(2) `getpgrp`(2) `getppid`(2) `getpgid`(2)	Get process, process group, and parent process IDs
`getuid`(2) `geteuid`(2) `getgid`(2) `getegid`(2)	Get real user, effective user, real group, and effective group IDs

TABLE 2–1 Process Functions *(continued)*

Function Name	Purpose
pause(2)	Suspend process until signal
priocntl(2)	Control process scheduler
setpgid(2)	Set process group ID
setsid(2)	Set session ID
waitid(2)	Wait for a child process to change state

Spawning New Processes

fork(2)

fork(2) creates a new process that is an exact copy of the calling process. The new process is the child process; the old process is the parent process. The child gets a new, unique process ID. fork(2) returns a 0 to the child process and the child's process ID to the parent. The returned value is how a forked program determines whether it is the parent process or the child process.

The new process created by fork(2) or exec(2) function inherits all open file descriptors from the parent including the three standard files: stdin, stdout, and stderr. When the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork(2).

The following code is an example of a call to fork(2) and the subsequent actions:

```
pid_t  pid;

 pid = fork();
 switch (pid) {
  case -1:   /* fork failed */
   perror ("fork");
   exit (1);
  case 0:   /* in new child process */
   printf ("In child, my pid is: %d\n", getpid(); );
   do_child_stuff();
   exit (0);
  default:   /* in parent, pid contains PID of child */
   printf ("In parent, my pid is %d, my child is %d\n", getpid(), pid);
   break;
 }

 /* Parent process code */
 ...
```

If the parent and the child process both read input from a stream, whatever is read by one process is lost to the other. So, once something has been delivered from the input buffer to a process, the buffer pointer has moved on.

Note - An obsolete practice is to use fork(2) and exec(2) to start another executable, then wait for the new process to die. In effect, a second process is created to perform a subroutine call. It is much more efficient to use dlopen(3DL), dlsym(3DL), and dlclose(3DL) as described in "Runtime Linking " on page 25 to make a subroutine temporarily resident in memory.

exec(2)

exec(2) is the name of a family of functions that includes execl(2), execv(2), execle(2), execve(2), execlp(2), and execvp(2). All load a new process over the calling process, but with different ways of pulling together and presenting the arguments of the function. For example, execl(2) could be used like this

```
execl("/usr/bin/prog2", "prog2", progarg1, progarg2, (char (*)0));
```

The execl argument list is:

/usr/bin/prog2	The path name of the new program file
prog2	The name the new process gets in its argv[0]
progarg1, progarg2	The arguments to prog2 as char (*)s
(char (*)0)	A null char pointer to mark the end of the arguments

There is no return from a successful execution of any variation of exec(2); the new process overlays the process that calls exec(2). The new process also takes over the process ID and other attributes of the old process. If a call to exec(2) fails, control is returned to the calling program with a return value of –1. You can check errno to learn why it failed.

Runtime Linking

An application can extend its address space during execution by binding to additional shared objects. There are several advantages in this delayed binding of shared objects:

- Processing a shared object when it is required, rather than during the initialization of an application, may greatly reduce start-up time. Also, the shared object may not be required during a particular run of the application, for example, objects containing help or debugging information.
- The application may choose between a number of different shared objects depending on the exact services required; for example, networking protocols.
- Any shared objects added to the process address space during execution may be freed after use.

  The following is a typical scenario that an application may perform to access an additional shared object:

- A shared object is located and added to the address space of a running application using `dlopen`(3DL). Any dependencies of shared object are also located and added at this time. For example:

```
#include        <stdio.h>
    #include        <dlfcn.h>

    main(int argc, char ** argv)
    {
     void *  handle;
     .....
     if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
      (void) printf("dlopen: %s\n", dlerror());
      exit (1);
     } .....
```

- The added shared objects are relocated and any initialization sections in the new shared objects are called.
- The application locates symbols in the added shared objects using `dlsym`(3DL). The application can then reference the data or call the functions defined by these new symbols. Continuing the preceding example:

```
 if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
        ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
     (void) printf("dlsym: %s\n", dlerror());
     exit (1);
 }
```

- After the application has finished with the shared objects, the address space is freed using `dlclose`(3DL). Any termination sections within the shared objects being freed are called at this time. For example:

```
 if (dlcose (handle) != 0) {
     (void) printf("dlclose: %s\n", dlerror());
     exit (1);
    }
```

- Any error conditions that occur as a result of using these runtime linker interface routines can be displayed using dlerror(3DL).

  The services of the runtime linker are defined in the header file <dlfcn.h> and are made available to an application by the shared library libdl.so.1. For example:

  ```
  $ cc -o prog main.c -ldl
  ```

  Here the file main.c can refer to any of the dlopen(3DL) family of routines, and the application prog will be bound to these routines at runtime.

  For a thorough discussion of application directed runtime linking, see the *Linker and Libraries Guide*. See dladdr(3DL), dlclose(3DL), dlerror(3DL), dlopen(3DL), and dlsym(3DL) for use details.

Process Scheduling

The UNIX system scheduler determines when processes run. It maintains process priorities based on configuration parameters, process behavior, and user requests. It uses these priorities to assign processes to the CPU.

Scheduler functions give users varying degrees of control over the order in which certain processes run and the amount of time each process can use the CPU before another process gets a chance.

By default, the scheduler uses a time-sharing policy. A time-sharing policy adjusts process priorities dynamically in an attempt to give good response time to interactive processes and good throughput to CPU-intensive processes.

The scheduler also provides an alternate real-time scheduling policy. Real-time scheduling allows users to set fixed priorities—priorities that the system does not change. The highest priority real-time user process always gets the CPU as soon as it can be run, even if other system processes are also eligible to be run. A program can therefore specify the exact order in which processes run. You can also write a program so that its real-time processes have a guaranteed response time from the system.

For most versions of SunOS 5.0 through 5.8, the default scheduler configuration works well and no real-time processes are needed: administrators need not change configuration parameters and users need not change scheduler properties of their processes. However, for some programs with strict timing constraints, real-time processes are the only way to guarantee that the timing requirements are met.

For more information, see priocntl(1), priocntl(2), and dispadmin(1M) . For a fuller discussion of this subject, see Chapter 3.

Error Handling

Functions that do not conclude successfully almost always return a value of –1 to your program. (For a few functions in *man Pages(2): System Calls*, there are calls for which no return value is defined, but these are the exceptions.) In addition to the –1 that is returned to the program, the unsuccessful function places an integer in an externally declared variable, errno. In a C program, you can determine the value in errno if your program contains the following statement:

```
#include <errno.h>
```

The value in errno is not cleared on successful calls, so check it only if the function returns -1. Since some functions return -1 but do not set errno refer to the man page for the function to be sure that errno contains a valid value. See error descriptions in Intro().

Use the C language function perror(3C) to print an error message on stderr based on the value of errno, or strerror(3C) to obtain the corresponding printable string.

CHAPTER **3**

Process Scheduler

This chapter describes the scheduling of processes. See the *Multithreaded Programming Guide* for a description of multithreaded scheduling. This chapter is for programmers who need more control over the order of process execution than default scheduling provides.

Overview of the Scheduler

When a process is created, it is assigned a Light Weight Process (LWP). (If the process is multithreaded, it might be assigned more LWPs.) An LWP is the object that is actually scheduled by the UNIX system scheduler, which determines when processes run. The scheduler maintains process priorities based on configuration parameters, process behavior, and user requests. It uses these priorities to let processes run.

The default scheduling is a time-sharing policy. This policy adjusts process priorities dynamically to balance the response time of interactive processes and the throughput of processes that use a lot of CPU time.

The SunOS 5.8 scheduler also provides a real-time scheduling policy. Real-time scheduling lets users set fixed priorities of specific processes. The highest-priority real-time user process always gets the CPU as soon as the process is runnable, even if system processes are runnable.

A program can be written so that its real-time processes have a guaranteed response time from the system. See Chapter 8for detailed information.

The control of process scheduling provided by real-time scheduling is rarely needed and can cause more problems than it solves. However, when the requirements for a program include strict timing constraints, real-time processes might be the only way to satisfy those constraints.

Note - Careless use of real-time processes can have a dramatic negative effect on the performance of time-sharing processes.

Because changes in scheduler administration can affect scheduler behavior, programmers might also need to know something about scheduler administration. The manual pages affecting scheduler administration are:

- `dispadmin`(1M) tells how to change scheduler configuration in a running system.
- `ts_dptbl`(4) and `rt_dptbl`(4) describe the time-sharing and real-time parameter tables that are used to configure the scheduler.

Figure 3–1 shows how the SunOS 5.8 process scheduler works:

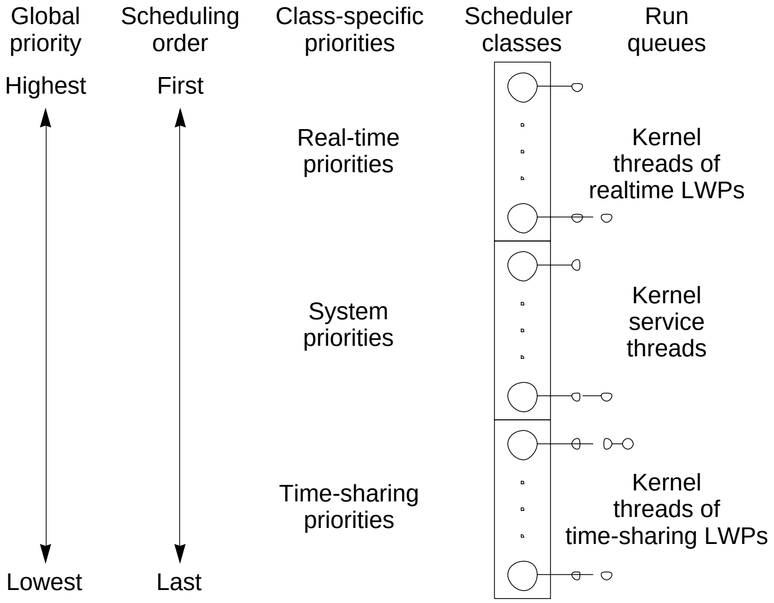

Figure 3–1 SunOS 5.8 Process Scheduler

When a process is created, it inherits its scheduling parameters, including scheduling class and a priority within that class. A process changes class only by user request. The system manages the priority of a process based on user requests and the policy associated with the scheduler class of the process.

In the default configuration, the initialization process belongs to the time-sharing class. So, all user login shells begin as time-sharing processes.

The scheduler converts class-specific priorities into global priorities. The global priority of a process determines when it runs—the scheduler always runs the runnable process with the highest global priority. Numerically higher priorities run first. Once the scheduler assigns a process to the CPU, the process runs until it sleeps, uses its time slice, or is preempted by a higher-priority process. Processes with the same priority run round-robin.

All real-time processes have higher priorities than any kernel process, and all kernel processes have higher priorities than any time-sharing process.

Note - As long as there is a runnable real-time process and assuming a single processor system, no kernel process and no time-sharing process runs.

Administrators specify default time slices in the configuration tables and users can assign per-process time slices to real-time processes.

You can display the global priority of a process with the `-cl` options of the `ps`(1) command. You can display configuration information about class-specific priorities with the `priocntl`(1) command and the `dispadmin`(1M) command.

The following sections describe the scheduling policies of the three default classes.

Time-Sharing Class

The goal of the time-sharing policy is to provide good response time to interactive processes and good throughput to CPU-bound processes. The scheduler switches CPU allocation often enough to provide good response time, but not so often that it spends too much time on switching. Time slices are typically a few hundred milliseconds.

The time-sharing policy changes priorities dynamically and assigns time slices of different lengths. The scheduler raises the priority of a process that sleeps after only a little CPU use (a process sleeps, for example, when it starts an I/O operation such as a terminal read or a disk read). Frequent sleeps are characteristic of interactive tasks such as editing and running simple shell commands. The time-sharing policy lowers the priority of a process that uses the CPU for long periods without sleeping.

The default time-sharing policy gives larger time slices to processes with lower priorities. A process with a low priority is likely to be CPU-bound. Other processes get the CPU first, but when a low-priority process finally gets the CPU, it gets a bigger chunk of time. If a higher-priority process becomes runnable during a time slice, however, it preempts the running process.

Global process priorities and user-supplied priorities are in ascending order: numerically higher priorities run first. The user priority runs from the negative of a configuration-dependent maximum to the positive of that maximum. A process inherits its user priority. Zero is the default initial user priority.

The "user priority limit" is the configuration-dependent maximum value of the user priority. You can set a user priority to any value below the user priority limit. With appropriate permission, you can raise the user priority limit. Zero is the default user priority limit.

You can lower the user priority of a process to give the process reduced access to the CPU or, with the appropriate permission, raise the user priority to get better service. Because you cannot set the user priority above the user priority limit, you must raise the user priority limit before you raise the user priority, if both have their default values at zero.

An administrator configures the maximum user priority independent of global time-sharing priorities. In the default configuration, for example, a user can set a user priority only in the range from –20 to +20, but 60 time-sharing global priorities are configured.

The scheduler manages time-sharing processes using configurable parameters in the time-sharing parameter table `ts_dptbl`(4). This table contains information specific to the time-sharing class.

System Class

The system class uses a fixed-priority policy to run kernel processes such as servers and housekeeping processes like the paging daemon. The system class is reserved to the kernel. Users can neither add nor remove a process from the system class. Priorities for system class processes are set up in the kernel code. Once established, the priorities of system processes do not change. (User processes running in kernel mode are not in the system class.)

Real-time Class

The real-time class uses a fixed-priority scheduling policy so that critical processes run in predetermined order. Real-time priorities never change except when a user requests a change. Privileged users can use the `priocntl`(1) command or the `priocntl`(2) function to assign real-time priorities.

The scheduler manages real-time processes using configurable parameters in the real-time parameter table `rt_dptbl`(4). This table contains information specific to the real-time class.

Commands and Functions

Figure 3–2 illustrates the default process priorities.

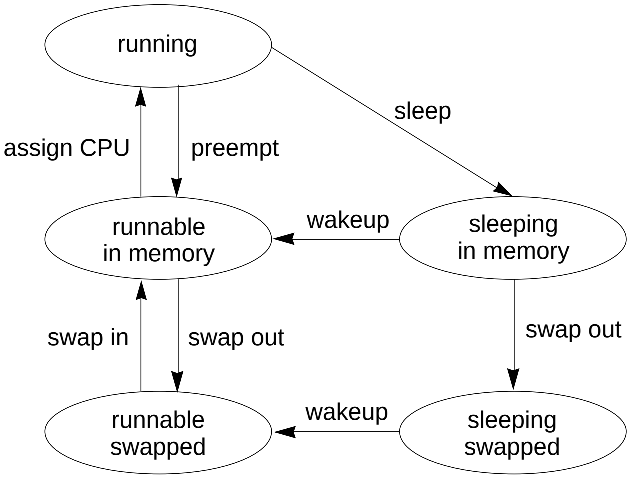

Figure 3–2 Process Priorities (Programmer's View)

A process priority has meaning only in the context of a scheduler class. You specify a process priority by specifying a class and a class-specific priority value. The class and class-specific value are mapped by the system into a global priority that the system uses to schedule processes.

A system administrator's view of priorities is different from that of a user or programmer. When configuring scheduler classes, an administrator deals directly with global priorities. The system maps priorities supplied by users into these global priorities. See *System Administration Guide, Volume I* for more information about priorities.

The `ps`(1)command with `-cel` options reports global priorities for all active processes. The `priocntl`(1) command reports the class-specific priorities that users and programmers use.

The `priocntl`(1) command and the `priocntl`(2) and `priocntlset`(2) functions set or retrieve scheduler parameters for processes. Setting priorities is generally the same for all three functions:

- Specify the target processes.
- Specify the scheduler parameters you want for those processes.
- Do the command or function to set the parameters for the processes.

These IDs are basic properties of UNIX processes. (See `Intro`() .) The class ID is the scheduler class of the process. `priocntl`(2) works only for the time-sharing and the real-time classes, not for the system class.

The `priocntl`(1) Command

The `priocntl`(1) utility performs four different control functions on the scheduling of a process:

`priocntl -l`	displays configuration information
`priocntl -d`	displays the scheduling parameters of processes
`priocntl -s`	sets the scheduling parameters of processes
`priocntl -e`	executes a command with the specified scheduling parameters

The following are some examples of using `priocntl`(1).

The output of the `-l` option for the default configuration is:

```
$ priocntl -d -i all
CONFIGURED CLASSES
==================

SYS (System Class)

TS (Time Sharing)
Configured TS User Priority Range -20 through 20

RT (Real Time)
Maximum Configured RT Priority: 59
```

An example of displaying information on all processes:

```
$ priocntl -d -i all
```

An example of displaying information on all time-sharing processes:

```
$ priocntl -d -i class TS
```

An example of displaying information on all processes with user ID 103 or 6626:

```
$ priocntl -d -i uid 103 6626
```

An example of making the process with ID 24668 a real-time process with default parameters:

```
$ priocntl -s -c RT -i pid 24668
```

An example of making 3608 RT with priority 55 and a one-fifth second time slice:

```
$ priocntl -s -c RT -p 55 -t 1 -r 5 -i pid 3608
```

An example of changing all processes into time-sharing processes:

```
$ priocntl -s -c TS -i all
```

For uid 1122, reduce TS user priority and user priority limit to -10:

```
$ priocntl -s -c TS -p -10 -m -10 -i uid 1122
```

An example of starting a real-time shell with default real-time priority:

```
$ priocntl -e -c RT /bin/sh
```

An example of running make with a time-sharing user priority of -10:

```
$ priocntl -e -c TS -p -10 make bigprog
```

`priocntl`(1) subsumes the function of `nice`(1). `nice` works only on time-sharing processes and uses higher numbers to assign lower priorities. The example above is equivalent to using `nice`(1) to set an "increment" of 10:

```
$ nice -10 make bigprog
```

The `priocntl`(2) Function

`priocntl`(2) gets or sets the scheduling parameters of a process or set of processes much as the `priocntl`(1) utility does for a process. An invocation of `priocntl`(2) can act on a LWP, on a single process, or on a group of processes. A group of processes can be identified by parent process, process group, session, user, group, class, or all active processes. The manual page contains the details of its use.

An example of using `priocntl`(2) to do the equivalent of `% priocntl -l` is in Appendix A.

The `PC_GETCLINFO` command gets a scheduler class name and parameters given the class ID. This command makes it easy to write programs that make no assumptions about what classes are configured. An example of using `priocntl`(2) with `PC_GETCLINFO` to get the class name of a process based on the process ID is in Code Example A–2.

The `PC_SETPARMS` command sets the scheduler class and parameters of a set of processes. The `idtype` and `id` input arguments specify the processes to be changed. Code Example A–3provides an example of using `priocntl`(2) with the `PC_SETPARMS` command to convert a time-share process into a real-time process.

The `priocntlset`(2) Function

The `priocntlset`(2) function changes scheduler parameters of a set of processes, like `priocntl`(2). `priocntlset`(2) has the same command set as `priocntl`(2). The `cmd` and `arg` input arguments are the same. But while `priocntl`(2) applies to a set of processes specified by a single `idtype/id` pair, `priocntlset`(2) applies to a set of processes that results from a logical combination of two `idtype/id` pairs. Again, refer to the manual page for details.

An example of using `priocntlset`(2) to change the priority of a real-time processes without changing time-sharing processes with the same user ID to real-time processes is in Code Example A–4.

Interaction With Other Functions

Kernel Processes

The kernel's daemon and housekeeping processes are assigned to the system scheduler class. Users can neither add processes to nor remove processes from this class, nor can they change the priorities of these processes. The command `ps -cel` lists the scheduler class of all processes. Processes in the system class are identified by a SYS entry in the CLS column.

`fork`(2) and `exec`(2)

Scheduler class, priority, and other scheduler parameters are inherited across the `fork`(2) and `exec`(2) functions.

`nice`(2)

The `nice`(1) command and the `nice`(2) function work as in previous versions of the UNIX system. They let you change the priority of a time-sharing process. Use lower numeric values to assign higher time-sharing priorities with these functions.

To change the scheduler class of a process or to specify a real-time priority, you must use one of the priocntl functions. Use higher numeric values to assign higher priorities with the `priocntl`(2) functions.

init(1M)

The init(1M) process is a special case to the scheduler. To change the scheduling properties of init(1M), init must be the only process specified by idtype and id or by the procset structure.

Performance

Because the scheduler determines when and for how long processes run, it has an overriding importance in the performance and perceived performance of a system.

By default, all user processes are time-sharing processes. A process changes class only by a priocntl(2) call.

All real-time process priorities have a higher priority than any time-sharing process. As long as any real-time process is runnable, no time-sharing process or system process ever runs. So if a real-time application fails to relinquish control of the cpu occasionally, it can completely lock out other users and essential kernel housekeeping.

Besides controlling process class and priorities, a real-time application must also control several other factors that influence its performance. The most important factors in performance are CPU power, amount of primary memory, and I/O throughput. These factors interact in complex ways. The sar(1) command has options for reporting on all performance factors.

Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. Here’s a simplified overview of UNIX process states and the transitions between states:

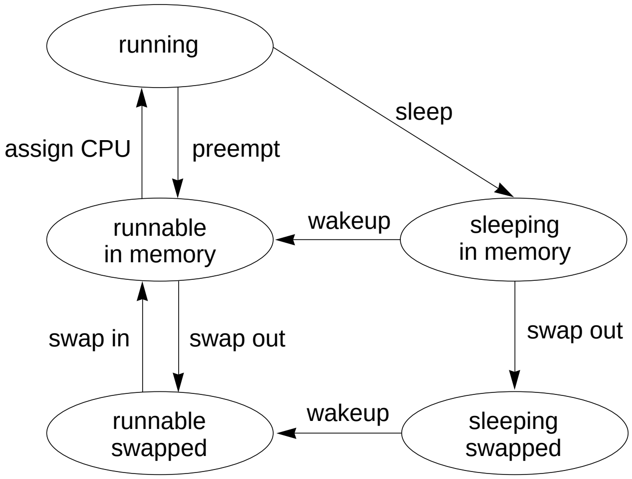

Figure 3–3 Process State Transition Diagram

An active process is normally in one of the five states in the diagram. The arrows show how it changes states.

- A process is running if it is assigned to a CPU. A process is preempted—that is, removed from the running state—by the scheduler if a process with a higher priority becomes runnable. A process is also preempted if it consumes its entire time slice and a process of equal priority is runnable.
- A process is runnable in memory if it is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if it is in primary memory but is waiting for a specific event before it can continue execution. For example, a process is sleeping if it is waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, the process is sent a wake up; if the reason for its sleep is gone, the process becomes runnable.
- A process is runnable and swapped if it is not waiting for a specific event but has had its whole address space written to secondary memory to make room in primary memory for other processes.
- A process is sleeping and swapped if it is both waiting for a specific event and has had its whole address space written to secondary memory to make room in primary memory for other processes.

  If a machine does not have enough primary memory to hold all its active processes, it must page or swap some address space to secondary memory.

- When the system is short of primary memory, it writes individual pages of some processes to secondary memory but still leaves those processes runnable. When a process runs, if it accesses those pages, it must sleep while the pages are read back into primary memory.
- When the system gets into a more serious shortage of primary memory, it writes all the pages of some processes to secondary memory and marks those processes

as swapped. Such processes get back into a state where they can be scheduled only by being chosen by the system scheduler daemon process, then read back into memory.

Both paging and swapping cause delay when a process is ready to run again. For processes that have strict timing requirements, this delay can be unacceptable.

To avoid swapping delays, real-time processes are never swapped, though parts of them can be paged. A program can prevent paging and swapping by locking its text and data into primary memory. For more information, see `memcntl`(2). How much memory can be locked is limited by how much memory is configured. Also, locking too much can cause intolerable delays to processes that do not have their text and data locked into memory.

Trade-offs between performance of real-time processes and performance of other processes depend on local needs. On some systems, process locking might be required to guarantee the necessary real-time response.

Software Latencies

See “Dispatch Latency” on page 85for information about latencies in real-time applications.

CHAPTER 4

Asynchronous Notification

This chapter describes asynchronous notification mechanisms. These are signals, which come in two flavors, and polling, which comes in three flavors. We describe what they can do for an application and what they can do to an application.

Signals

Signals are software generated interrupts that are sent to a process when an event happens. Signals can be synchronously generated by an error in an application, such as SIGFPE and SIGSEGV, but most signals are asynchronous. Signals can be posted to a process when the system detects a software event, such as a user entering an *interrupt* or *stop* or a *kill* request from another process. Signals can be sent by the kernel when a hardware event such as a bus error or an illegal instruction is encountered. Signals can be sent by the kernel to note the completion of an input/output request.

The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future. Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored. Each signal has a default action that is one of the following:

- The signal is discarded after being received.
- The process is terminated after the signal is received.
- A core file is written, then the process is terminated.
- Stop the process after the signal is received.

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

The set of signals is defined in the header `<signal.h>`.

Signal Processing

When a signal is delivered to a process, it is added to a set of signals pending for the process. If the signal is not blocked for the process, it is delivered. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated, and the signal handler is invoked.

In BSD signal semantics, when a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler (or until a mask modifying interface is called). This mask blocks the signal that has just been delivered from interrupting the process again while the handler for the signal is running.

System V signal semantics do not provide this protection, letting the same signal interrupt the handler that processes the signal. This requires that a signal handler be reentrant.

All signals have the same priority. If a signal handler blocks the signal that invoked it, other signals can still be delivered to the process.

Signals are not stacked. There is no way for a signal handler to record how many times a signal has actually been delivered.

Blocking

A global signal *mask* defines the set of signals that are blocked from being delivered to a process. A process's signal mask copies its initial state from the signal mask of the parent process. The mask can be changed with calls to `sigaction`(2), `sigblock`(3UCB), `sigsetmask`(3UCB), `sigprocmask`(2), `sigsetops`(3C), `sighold`(3C), or `sigrelse`(3C) (see `signal`(3C)).

Handling

An application program can specify a function called a *signal handler* to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to *catch* the signal. A process can deal with a signal in one of the following ways:

- The process can let the default action happen.
- The process can block the signal (some signals cannot be ignored).
- The process can catch the signal with a handler.

Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself.

Installing a Handler

The functions `signal`(3C), `sigset`(3C), `signal`(3UCB), and `sigvec`(3UCB) can all be used to install a signal handler. All return the previous action for the signal. There is one major difference between the four interfaces: `signal`(3C) results in System V signal semantics (the same signal can interrupt its handler); `sigset`(3C), `signal`(3UCB), and `sigvec`(3UCB) all result in BSD signal semantics (the signal is blocked until its handler returns). In addition, both `signal`(3C) and `signal`(3UCB) can flag that the signal is to be ignored or that the default action be restored. Code Example 4–1 illustrates a simple handler and its installation.

CODE EXAMPLE 4–1 Signal handler installation

```
#include <stdio.h>
#include <signal.h>
#define TRUE 1

void sigcatcher()
{
  printf ("PID %d caught signal.\n", getpid());
}

main()
{
  pid_t ppid;

  signal (SIGINT, sigcatcher);
  if (fork() == 0) {
   sleep( 5 );
   ppid = getppid();
   while( TRUE )
    if (kill( ppid, SIGINT) == -1 )
     exit( 1 );
  }
  pause();
}
```

Trying to install a handler or set SIG_IGN for the signals SIGKILL or SIGSTOP results in an error. Trying to set SIG_IGN for the signal SIGCONT also results in an error, since it is ignored by default.

After a signal handler is installed, it remains so until it is explicitly replaced by another call to signal(3C), sigset(3C), signal(3UCB), or sigvec(3UCB).

Catching SIGCHLD

When a child process stops or terminates, SIGCHLD is sent to the parent process. The default response to the signal is to ignore it. The signal can be caught and the exit status from the child process can be obtained by immediately calling wait(2) and wait3(3C). This allows zombie process entries to be removed as quickly as possible. Code Example 4–2demonstrates installing a handler that catches SIGCHLD.

CODE EXAMPLE 4–2 Catching SIGCHLD

```
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/resource.h>

void proc_exit()
{
  int wstat;
  union wait wstat;
  pid_t pid;

  while (TRUE) {
   pid = wait3 (&wstat, WNOHANG, (struct rusage *)NULL );
   if (pid == 0)
    return;
   else if (pid == -1)
    return;
   else
    printf ("Return code: %d\n", wstat.w_retcode);
  }
}
main ()
{
  signal (SIGCHLD, proc_exit);
  switch (fork()) {
   case -1:
    perror ("main: fork");
    exit (0);
   case 0:
    printf ("I’m alive (temporarily)\n");
    exit (rand());
   default:
    pause();
  }
}
```

SIGCHLD catchers are usually set up as part of process initialization. They must be set before a child process is forked. A typical SIGCHLD handler retrieves the child process's exit status.

CHAPTER **5**

Input/Output Interfaces

This chapter introduces file input/output operations as provided on systems that do not provide virtual memory services. It discusses the improved input/output method provided by the virtual memory facilities. It also describes the older, heavyweight method of file and record locking.

Files and I/O

Files that are organized as a sequence of data are called *regular* files. These can contain ASCII text, text in some other binary data encoding, executable code, or any combination of text, data, and code. The file has two components:

- The control data, called the *inode*. These data include the file type, the access permissions, the owner, the file size, and the location(s) of the data blocks.
- The file contents: a nonterminated sequence of bytes.

SunOS provides three basic forms of file input/output interfaces.

- The traditional, raw, style of file I/O is described in “Basic File I/O” on page 48.
- The second form is the standard file I/O. The standard I/O buffering provides an easier interface and improved efficiency to an application run on a system without virtual memory. In an application running in a virtual memory environment, such as the Solaris 8 operating environment, standard file I/O is a very inefficient anachronism.
- The third form of file I/O is provided by the memory mapping interface described in “Memory Management Interfaces” on page 61. Mapping files is the most efficient and powerful form of file I/O for most applications run in the Solaris 8 environment.

Basic File I/O

The functions listed in Table 5–1perform basic operations on files:

TABLE 5–1 Basic File I/O Functions

Function Name	Purpose
open(2)	Open a file for reading or writing
close(2)	Close a file descriptor
read(2)	Read from a file
write(2)	Write to a file
creat(2)	Create a new file or rewrite an existing one
unlink(2)	Remove a directory entry
lseek(2)	Move read/write file pointer

The following code sample demonstrates the use of the basic file I/O interface. read(2) and write(2) both transfer no more than the specified number of bytes, starting at the current offset into the file. The number of bytes actually transferred is returned. The end of a file is indicated, on a read(2), by a return value of zero.

CODE EXAMPLE 5–1

```
#include   <fcntl.h>
#define   MAXSIZE   256

main()
{
  int  fd;
  ssize_t n;
  char  array[MAXSIZE];
  fd = open ("/etc/motd", O_RDONLY);
  if (fd == -1) {
   perror ("open");
   exit (1);
  }
  while ((n = read (fd, array, MAXSIZE)) > 0)
   if (write (1, array, n) != n)
    perror ("write");
  if (n == -1)
   perror ("read");
  close (fd);
}
```

Always close(2) a file when you are done reading or writing it, but never close(2) a file descriptor that you did not open(2).

Offset into an open file are changed by read(2)s, write(2)s, or by calls to lseek(2). Some examples of using lseek(2) are:

```
off_t  start, n;
 struct  record  rec;

 /* record current offset in start */
 start = lseek (fd, 0L, SEEK_CUR);

 /* go back to start */
 n = lseek (fd, -start, SEEK_SET);
 read (fd, &rec, sizeof (rec));

 /* rewrite previous record */
 n = lseek (fd, -sizeof (rec), SEEK_CUR);
 write (fd, (char *&rec, sizeof (rec));
```

Advanced File I/O

Advanced file I/O functions create and remove directories and files, create links to existing files, and obtain or modify file status information.

TABLE 5–2 Advanced File I/O Functions

Function Name	**Purpose**
link(2)	Link to a file
access(2)	Determine accessibility of a file
mknod(2)	Make a special or ordinary file
chmod(2)	Change mode of file
chown(2), lchown(2), fchown(2)	Change owner and group of a file
utime(2)	Set file access and modification times

TABLE 5–2 Advanced File I/O Functions *(continued)*

Function Name	Purpose
`stat`(2), `lstat`(2), `fstat`(2)	Get file status
`fcntl`(2)	Perform file control functions
`ioctl`(2)	Control device
`fpathconf`(2)	Get configurable path name variables
`opendir`(3C), `readdir`(3C), `closedir`(3C)	Perform directory operations
`mkdir`(2)	Make a directory
`readlink`(2)	Read the value of a symbolic link
`rename`(2)	Change the name of a file
`rmdir`(2)	Remove a directory
`symlink`(2)	Make a symbolic link to a file

File System Control

File system control functions let you to control various aspects of the file system:

TABLE 5–3 File System Control Functions

Function Name	Purpose
`ustat`(2)Get file system statistics	Get file system statistics
`sync`(2)	Update super block
`mount`(2)	Mount a file system
`statvfs`(2), `fstatvfs`(2)	Get file system information
`sysfs`(2)	Get file system type information

File and Record Locking

You don't need to use traditional file I/O to do locking of file elements. The lighter weight synchronization mechanisms described in *Multithreaded Programming Guide* can be used effectively with mapped files and are much more efficient than the old style file locking described in this section.

You lock files, or portions of files, to prevent the errors that can occur when two or more users of a file try to update information at the same time.

File locking and record locking are really the same thing, except that file locking blocks access to the whole file, while record locking blocks access to only a specified segment of the file. (In the SunOS 5.0 through 5.8 system, all files are a sequence of bytes of data: a record is a concept of the programs that use the file.)

Supported File Systems

Both advisory and mandatory locking are supported on the following types of file systems:

- `ufs`—the default disk-based file system
- `fifofs`—a pseudo file system of named pipe files that give processes common access to data
- `namefs`—a pseudo file system used mostly by STREAMS for dynamic mounts of file descriptors on top of file
- `specfs`—a pseudo file system that provides access to special character and block devices

  Only advisory file locking is supported on NFS.

  File locking is not supported for the `proc` and `fd` file systems.

Choosing a Lock Type

Mandatory locking suspends a process until the requested file segments are free. Advisory locking returns a result indicating whether the lock was obtained or not: processes can ignore the result and do the I/O anyway. You cannot use both mandatory and advisory file locking on the same file at the same time. The mode of a file at the time it is opened determines whether locks on a file are treated as mandatory or advisory.

Of the two basic locking calls, `fcntl`(2) is more portable, more powerful, and less easy to use than `lockf`(3C). `fcntl`(2) is specified in Posix 1003.1 standard. `lockf`(3C) is provided to be compatible with older applications.

Terminology

Some useful definitions for reading the rest of this section are shown below:

record	An arbitrary sequence of bytes in a file. The UNIX operating system supports no record structure. Programs that use the files can impose any arbitrary record structure.
cooperating processes	Two or more processes using some mechanism to regulate access to a shared resource.
read lock	Lets other processes also apply a read lock and/or perform reads, and blocks other processes from writing or applying a write lock.
write lock	Blocks all other process from reading, writing, or applying any lock.
advisory lock	Returns an error without blocking to a process that does not hold the lock. Advisory locking is not enforced on `creat`(2), `open`(2), `read`(2), or `write`(2) operations.
mandatory lock	Blocks execution of processes that do not hold the lock. Access to locked records is enforced on `creat`(2), `open`(2), `read`(2), and `write`(2) operations.

Opening a File for Locking

A lock can only be requested on a file with a valid open descriptor. For read locks, the file must be opened with at least read access. For write locks, the file must also be opened with write access. In this example, a file is opened for both read and write access.

```
...
  filename = argv[1];
  fd = open (filename, O_RDWR);
  if (fd < 0) {
   perror(filename);
   exit(2);
  }
  ...
```

Setting a File Lock

To lock an entire file, set the offset to zero and set the size to zero.

There are several ways to set a lock on a file. Choice of method depends on how the lock interacts with the rest of the program, performance, and portability. This example uses the POSIX standard-compatible `fcntl`(2) function. It tries to lock a file until one of the following happens:

- The file is successfully locked

- There is an error
- MAX_TRY is exceeded, and the program gives up trying to lock the file

```
#include <fcntl.h>

    ...
     struct flock lck;

    ...
     lck.l_type = F_WRLCK; /* setting a write lock */
     lck.l_whence = 0; /* offset l_start from beginning of file */
     lck.l_start = (off_t)0;
     lck.l_len = (off_t)0; /* until the end of the file */
     if (fcntl(fd, F_SETLK, &lck) <0) {
      if (errno == EAGAIN || errno == EACCES) {
       (void) fprintf(stderr, "File busy try again later!\n");
       return;
      }
      perror("fcntl");
      exit (2);
     }
     ...
```

Using fcntl(2), you can set the type and start of the lock request by setting a few structure variables.

Note - Mapped files cannot be locked with flock(3UCB). See mmap(2). However, the multithread-oriented synchronization mechanisms (in either POSIX or Solaris styles) can be used with mapped files. See mutex(3THR), condition(3THR), semaphore(3THR), and rwlock(3THR).

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except that the starting point and length of the lock segment is not set to zero.

Plan a failure response for when you cannot obtain all the required locks. Contention for data is why you use record locking, so different programs might:

- Wait a certain amount of time, then try again
- Abort the procedure and warn the user
- Let the process sleep until signaled that the lock has been freed
- Do some combination of the above

This example shows locking a record using fcntl(2).

```
{
     struct flock lck;
       ...
     lck.l_type = F_WRLCK; /* setting a write lock */
     lck.l_whence = 0; /* offset l_start from beginning of file */
     lck.l_start = here;
     lck.l_len = sizeof(struct record);

     /* lock "this" with write lock */
     lck.l_start = this;
     if (fcntl(fd, F_SETLKW, &lck) < 0) {
      /* "this" lock failed. */
      return (-1);
    ...
   }
```

The next example shows the `lockf`(3C) function.

```
#include <unistd.h>

   {
    ...
     /* lock "this" */
     (void) lseek(fd, this, SEEK_SET);
     if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
      /* Lock on "this" failed. Clear lock on "here". */
      (void) lseek(fd, here, 0);
      (void) lockf(fd, F_ULOCK, sizeof(struct record));
      return (-1);
   }
```

Locks are removed the same way they are set—only the lock type is different (`F_ULOCK`). An unlock cannot be blocked by another process and affects only locks placed by the calling process. The unlock affects only the segment of the file specified in the preceding locking call.

Getting Lock Information

You can determine which process, if any, is holding a lock. Use this as a simple test or to find locks on a file. A lock is set, as in the previous examples, and `F_GETLK` is used in `fcntl`(2). The next example finds and prints indentifying data on all the locked segments of a file.

CODE EXAMPLE 5–2

```
struct flock lck;

  lck.l_whence = 0;
  lck.l_start = 0L;
  lck.l_len = 0L;
  do {
   lck.l_type = F_WRLCK;
   (void) fcntl(fd, F_GETLK, &lck);
   if (lck.l_type != F_UNLCK) {
    (void) printf("%d %d %c %8ld %8ld\n", lck.l_sysid, lck.l_pid,
      (lck.l_type == F_WRLCK) ? ’W’ : ’R’, lck.l_start, lck.l_len);
    /* If this lock goes to the end of the address space, no
     * need to look further, so break out. */
    if (lck.l_len == 0) {
    /* else, look for new lock after the one just found. */
      lck.l_start += lck.l_len;
    }
   }
  } while (lck.l_type != F_UNLCK);
```

fcntl(2) with the F_GETLK command can sleep while waiting for a server to respond, and it can fail (returning ENOLCK) if there is a resource shortage on either the client or server.

lockf(3C) with the F_TEST command can be used to test if a process is holding a lock. This function does not return information about where the lock is and which process owns it.

```
(void) lseek(fd, 0, 0L);
 /* set the size of the test region to zero (0). to test until the
    end of the file address space. */
 if (lockf(fd, (off_t)0, SEEK_SET) < 0) {
  switch (errno) {
   case EACCES:
   case EAGAIN:
    (void) printf("file is locked by another process\n");
    break;
   case EBADF:
    /* bad argument passed to lockf */
    perror("lockf");
    break;
   default:
    (void) printf("lockf: unexpected error <%d>\n", errno);
    break;
  }
```

Forking and Locks

When a process forks, the child receives a copy of the file descriptors that the parent opened. Locks are not inherited by the child because they are owned by a specific process. The parent and child share a common file pointer for each file. Both processes can try to set locks on the same location in the same file. This problem happens with both lockf(3C) and fcntl(2). If a program holding a record lock forks, the child process should close the file and reopen it to set a new, separate file pointer.

Deadlock Handling

The UNIX locking facilities provide deadlock detection/avoidance. Deadlocks can happen only when the system is about to put a record locking function to sleep. A search is made to determine whether process A will wait for a lock that B holds while B is waiting for a lock that A holds. If a potential deadlock is detected, the locking function fails and sets errno to indicate deadlock. Processes setting locks using F_SETLK do not cause a deadlock because they do not wait when the lock cannot be granted immediately.

Selecting Advisory or Mandatory Locking

For mandatory locks, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Set a mandatory lock as follows.

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

 int mode;
 struct stat buf;
  ...
  if (stat(filename, &buf) < 0) {
   perror("program");
   exit (2);
  }
  /* get currently set mode */
  mode = buf.st_mode;
  /* remove group execute permission from mode */
  mode &= ~(S_IEXEC>>3);
   /* set ’set group id bit’ in mode */
  mode |= S_ISGID;
  if (chmod(filename, mode) < 0) {
   perror("program");
   exit(2);
  }
  ...
```

Files to be record locked should never have any execute permission set. This is because the operating system ignores record locks when executing a file.

The chmod(1) command can also be used to set a file to permit mandatory locking. For example:

```
$ chmod +l file
```

This command sets the O20n0 permission bit in the file mode, which indicates mandatory locking on the file. If n is even, the bit is interpreted as enabling mandatory locking. If n is odd, the bit is interpreted as “set group ID on execution.”

The ls(1) command shows this setting when you ask for the long listing format with the –l option:

```
$ ls -l file
```

displays following information:

```
-rw---l--- 1 user group size mod_time file
```

The letter "l" in the permissions indicates that the set-group-ID bit is on, so mandatory locking is enabled, along with the normal semantics of set group ID.

Cautions About Mandatory Locking

- Mandatory locking works only for local files. It is not supported when accessing files through NFS.
- Mandatory locking protects only the segments of a file that are locked. The remainder of the file can be accessed according to normal file permissions.
- If multiple reads or writes are needed for an atomic transaction, the process should explicitly lock all such segments before any I/O begins. Advisory locks are sufficient for all programs that perform in this way.
- Arbitrary programs should not have unrestricted access permission to files on which record locks are used.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Terminal I/O

Terminal I/O functions deal with a general terminal interface for controlling asynchronous communications ports, as shown in the table below. Also see `termios`(3C) and `termio`(7I).

TABLE 5–4 Terminal I/O Functions

Function Name	**Purpose**
`tcgetattr`(3C), `tcsetattr`(3C)	Get and set terminal attributes
`tcsendbreak`(3C), `tcdrain`(3C), `tcflush`(3C), `tcflow`(3C)	Perform line control functions
`cfgetospeed`(3C), `cfgetispeed`(3C) `cfsetispeed`(3C), `cfsetospeed`(3C)	Get and set baud rate

TABLE 5–4 Terminal I/O Functions *(continued)*

Function Name	**Purpose**
`tcsetpgrp`(3C)	Get and set terminal foreground process group ID
`tcgetsid`(3C)	Get terminal session ID

CHAPTER **6**

Memory Management

This chapter describes SunOS 5.0 through 5.8 virtual memory from the application developer's viewpoint. It identifies the capabilities that virtual memory makes available to application developers that are not found in systems with fixed memory. And it describes the interfaces provided in SunOS 5.0 through 5.8 to use and control these capabilities.

Overview of Virtual Memory

In a system with fixed memory (non-virtual), the address space of a process occupies and is limited to a portion of the system's main memory.

In SunOS 5.0 through 5.8 virtual memory, the actual address space of a process occupies a file in the swap partition of disk storage (the file is called the backing store). Pages of main memory buffer the active (or recently active) portions of the process address space to provide code for the CPU(s) to execute and data for the program to process.

A page of address space is loaded when an address that is not currently in memory is accessed by a CPU, causing a *page fault*. Since execution cannot continue until the page fault is resolved, by reading the referenced address segment into memory, the process sleeps until the page has been read.

The most obvious difference between the two memory systems for the application developer is that virtual memory lets applications occupy much larger address spaces. Less obvious advantages of virtual memory are much simpler and more efficient file I/O and very efficient sharing of memory between processes.

Address Spaces and Mapping

Since backing store files (the process address space) exist only in swap storage, they are not included in the UNIX named file space. (This makes backing store files inaccessible to other processes.) However, a simple extension allows the logical insertion of all, or part, of one or more, named files in the backing store and treats the result as a single address space. This is called *mapping.*

With mapping, any part of any readable or writable file can be logically included in a process's address space. Like any other portion of the process's address space, no page of the file is actually loaded into memory until a page fault forces this action. Pages of memory are written to the file only if their contents have been modified. So, reading from and writing to files is completely automatic and very efficient.

More than one process can map a single named file. This provides very efficient memory sharing between processes. All or part of other files can also be shared between processes.

Not all named file system objects can be mapped. Devices that cannot be treated as storage, such as terminal and network device files, are examples of objects that cannot be mapped.

A process address space is defined by all of the files (or portions of files) mapped into the address space. Each mapping is sized and aligned to the page boundaries of the system on which the process is executing. No memory is associated with processes themselves.

A process page maps to only one object at a time, although an object address can be the subject of many process mappings. The notion of a "page" is not a property of the mapped object. Mapping an object provides only the potential for a process to read or write the object's contents.

Mapping makes the object's contents directly addressable by a process. Applications can access the storage resources they use directly rather than indirectly through read and write. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the read, modify buffer, write cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Because the file system name space includes any directory trees that are connected from other systems by NFS, any networked file can also be mapped into a process's address space.

Coherence

Whether to share memory or to share data contained in the file, when multiple processes map a file simultaneously there can be problems with simultaneous access to data elements. Such processes can cooperate through any of the synchronization mechanisms provided in SunOS 5.0 through 5.8. Because they are very light weight, the most efficient synchronization mechanisms in SunOS 5.0 through 5.8 are those provides in the threads library. See mutex(3THR), condition(3THR), rwlock(3THR), and semaphore(3THR) in the manual pages for details on their use.

Memory Management Interfaces

The virtual memory facilities are used and controlled through several sets of functions. This section summarizes these calls and provides examples of their use.

Creating and Using Mappings

mmap(2) establishes a mapping of a named file system object (or part of one) into a process address space. It is the basic memory management function and it is very simple. First open(2) the file, then mmap(2) it with appropriate access and sharing options and go about your business.

The mapping established by mmap(2) replaces any previous mappings for specified address range.

The flags MAP_SHARED and MAP_PRIVATE specify the mapping type, and one of them must be specified. MAP_SHARED specifies that writes modify the mapped object. No further operations on the object are needed to make the change. MAP_PRIVATE specifies that an initial write to the mapped area creates a copy of the page and all writes reference the copy. Only modified pages are copied.

A mapping type is retained across a fork(2).

The file descriptor used in a mmap(2) call need not be kept open after the mapping is established. If it is closed, the mapping remains until the mapping is undone by munmap(2) or be replacing it with a new mapping.

If a mapped file is shortened by a call to truncate, an access to the area of the file that no longer exists causes a SIGBUS signal.

Mapping /dev/zero gives the calling program a block of zero-filled virtual memory of the size specified in the call to mmap(2). The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address that the system chooses.

```
int fd;
caddr_t result;

if ((fd = open("/dev/zero", O_RDWR)) == -1)
 return ((caddr_t)-1);
result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
(void) close(fd);
```

Some devices or files are useful only if accessed by mapping. An example of this is frame buffer devices used to support bit-mapped displays, where display management algorithms function best if they can operate randomly on the addresses of the display directly.

Removing Mappings

munmap(2) removes all mappings of pages in the specified address range of the calling process. munmap(2) has no affect on the objects that were mapped.

Cache Control

The SunOS 5.0 through 5.8 virtual memory system is a cache system, in which processor memory buffers data from file system objects. There are interfaces to control or interrogate the status of the cache.

mincore(2)

mincore(2) determines the residency of the memory pages in the address space covered by mappings in the specified range. Because the status of a page can change after mincore(2) checks it, but before mincore(2) returns the data, returned information can be outdated. Only locked pages are guaranteed to remain in memory.

mlock(3C) and munlock(3C)

mlock(3C) causes the pages in the specified address range to be locked in physical memory. References to locked pages (in this or other processes) do not result in page faults that require an I/O operation. This operation ties up physical resources and can disrupt normal system operation, so, use of mlock(3C) is limited to the superuser. The system lets only a configuration dependent limit of pages be locked in memory. The call to mlock(3C) fails if this limit is exceeded.

munlock(3C) releases the locks on physical pages. If multiple mlock(3C) calls are made on an address range of a single mapping, a single munlock(3C) call is release the locks. However, if different mappings to the same pages are mlock(3C)ed, the pages are not unlocked until the locks on all the mappings are released.

Locks are also released when a mapping is removed, either through being replaced with an mmap(2) operation or removed with munmap(2).

A lock is transferred between pages on the "copy-on-write" event associated with a MAP_PRIVATE mapping, thus locks on an address range that includes MAP_PRIVATE mappings are retained transparently along with the copy-on-write redirection (see mmap(2) above for a discussion of this redirection).

mlockall(3C) and munlockall(3C)

mlockall(3C)and munlockall(3C) are similar to mlock(3C) and munlock(3C), but they operate on entire address spaces. mlockall(3C) sets locks on all pages in the address space and munlockall(3C) removes all locks on all pages in the address space, whether established by mlock(3C) or mlockall(3C).

msync(3C)

msync(3C) causes all modified pages in the specified address range to be flushed to the objects mapped by those addresses. It is similar to fsync(3C) for files.

Other Memory Control Functions

sysconf(3C)

sysconf(3C) returns the system dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set.

mprotect(2)

mprotect(2) assigns the specified protection to all pages in the specified address range. The protection cannot exceed the permissions allowed on the underlying object.

brk(2) and sbrk(2)

brk(2) and sbrk(2) are called to add storage to the data segment of a process.

A process can manipulate this area by calling brk(2) and sbrk(2):

```
caddr_t
 brk(caddr_t addr);

 caddr_t
 sbrk(intptr_t incr);
```

brk(2) identifies the lowest data segment location not used by the caller as *addr* (rounded up to the next multiple of the system page size).

sbrk(2), the alternate function, adds *incr* bytes to the caller data space and returns a pointer to the start of the new data area.

CHAPTER **7**

Interprocess Communication

This chapter is for programmers who develop multiprocess applications.

The Solaris 8 and compatible operating environments has a large variety of mechanisms for concurrent processes to exchange data and synchronize execution. These mechanisms include:

- Pipes: anonymous data queues
- Named pipes: data queues with file names
- System V message queues, semaphores, and shared memory
- POSIX message queues, semaphores, and shared memory
- Signals: software generated interrupts
- Sockets
- Mapped memory and files (see “Memory Management Interfaces” on page 61)

All of these mechanisms except mapped memory are introduced in this chapter.

Pipes

A pipe between two processes is a pair of files that is created in a parent process. It connects the resulting processes when the parent process forks. A pipe has no existence in any file name space, so it is said to be anonymous. It is most common for a pipe to connect only two processes, although any number of child processes can be connected to each other and their parent by a single pipe.

A pipe is created in the process that becomes the parent by a call to `pipe`(2). The call returns two file descriptor in the array passed to it. After forking, both processes use p[0] to read from and p[1] to write to. The processes actually read from and write to a circular buffer that is managed for them.

Since on a fork(2) the per-process open file table is duplicated, each process has two readers and two writers. The extra readers and writers must be closed if the pipe is to function properly. For example, no end-of-file indication would ever be returned if the other end of a reader is also open for writing by the same process. The following code shows pipe creation, a fork, and clearing the duplicate pipe ends.

```
#include <stdio.h>
#include <unistd.h>
 ...
  int p[2];
 ...
  if (pipe(p) == -1) exit(1);
  switch( fork() )
  {
   case 0:      /* in child */
    close( p[0] );
    dup2( p[1], 1);
    close P[1] );
    exec( ... );
    exit(1);
   default:      /* in parent */
    close( p[1] );
    dup2( P[0], 0 );
    close( p[0] );
    break;
  }
  ...
```

Table 7–1 shows the results of reads from and writes to a pipe under certain conditions.

TABLE 7–1 Read / write results in a pipe

Attempt	Conditions	Result
read	Empty pipe, writer attached	Read blocked
write	Full pipe, reader attached	Write blocked
read	Empty pipe, no writer attached	EOF returned
write	No reader	SIGPIPE

Blocking can be prevented by calling fcntl(2) on the descriptor to set FNDELAY. This causes an error return (-1) from the I/O call with errno set to EWOULDBLOCK.

Named Pipes

Named pipes function much like pipes, but are created as named entities in a file system. This allows the pipe to be opened by any processes with no requirement that they be related by forking. A named pipe is created by a call to mknod(2). Then any process with appropriate permission can read from or write to a named pipe.

In the open(2) call, the process opening the pipe blocks until another process also opens the pipe.

To open a named pipe without blocking, the O_NDELAY mask (found in <sys/fcntl.h>) can be or-ed with the selected file mode mask on the call to open(2). If no other process is connected to the pipe when open(2) is called, -1 is returned with errno set to EWOULDBLOCK.

Sockets

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see <sys/socket.h>), of which only the UNIX and Internet domains are normally used in SunOS 5.8 and compatible operating environments.

Sockets can be used to communicate between processes on a single system, like other forms of IPC. The UNIX domain (AF_UNIX) provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. UNIX domain sockets are further described in “UNIX Domain Sockets” in *Network Interface Guide*. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain (AF_INET). Internet domain communication uses the TCP/IP internet protocol suite. Internet domain sockets are described in “Socket Interfaces” in *Network Interface Guide*.

POSIX IPC

POSIX interprocess communication is a variation of System V interprocess communication. It was introduced in Solaris 7. Like System V objects, POSIX IPC objects have read and write (but not execute) permissions for the owner, the owner's group, and for others. There is no way for the owner of a POSIX IPC object to assign a different owner.

Unlike the System V IPC interfaces, the POSIX IPC interfaces are all multithread safe.

POSIX Messages

The POSIX message queue interfaces are:

`mq_open`(3RT)	Connects to, and optionally creates, a named message queue
`mq_close`(3RT)	Ends the connection to an open message queue
`mq_unlink`(3RT)	Ends the connection to an open message queue and causes the queue to be removed when the last process closes it
`mq_send`(3RT)	Places a message in the queue
`mq_receive`(3RT)	Receives (removes) the oldest, highest priority message from the queue
`mq_notify`(3RT)	Notifies a process or thread that a message is available in the queue
`mq_setattr`(3RT), `mq_getattr`(3RT)	Set or get message queue attributes

POSIX Semaphores

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to twenty five semaphores.

The POSIX semaphore interfaces are

sem_open(3RT)	Connects to, and optionally creates, a named semaphore
sem_init(3RT)	Initializes a semaphore structure (internal to the calling program, so not a named semaphore)
sem_close(3RT)	Ends the connection to an open semaphore
sem_unlink(3RT)	Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it
sem_destroy(3RT)	Initializes a semaphore structure (internal to the calling program, so not a named semaphore).
sem_getvalue(3RT)	Copies the value of the semaphore into the specified integer
sem_wait(3RT), sem_trywait(3RT)	Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process
sem_post(3RT)	Increments the count of the semaphore

POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory (see “Creating and Using Mappings” on page 61). The major differences are to use shm_open(3RT) to open the shared memory object (instead of calling open(2)) and use shm_unlink(3RT) to close and delete the object (instead of calling close(2) which does not remove the object). The options in shm_open(3RT) are substantially fewer than the number of options provided in open(2).

System V IPC

The Solaris 8 and compatible operating environments provides an InterProcess Communication (IPC) package that supports three types of interprocess communication that are more versatile than pipes and named pipes.

- Messages allow processes to send formatted data streams to arbitrary processes.
- Semaphores allow processes to synchronize execution.
- Shared memory allows processes to share parts of their virtual address space.

See the `ipcrm`(1), `ipcs`(1), `Intro`(), `msgctl`(2), `msgget`(2), `msgrcv`(2), `msgsnd`(2), `semget`(2), `semctl`(2), `semop`(2), `shmget`(2), `shmctl`(2), `shmop`(2), and `ftok`(3C) manual pages for more information about System V IPC.

Permissions

Messages, semaphores, and shared memory have read and write permissions (but no execute permission) for the owner, group, and others the same as ordinary files. Like files, the creating process identifies the default owner. Unlike files, the creator can assign ownership of the facility to another user; it can also revoke an ownership assignment.

IPC Functions, Key Arguments, and Creation Flags

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a `key_t` *key* argument. The *key* is an arbitrary value or one that can be derived from a common seed at runtime. One way is with `ftok`(3C), which converts a file name to a key value that is unique within the system.

Functions that initialize or get access to messages, semaphores, or shared memory return an ID number of type int. IPC functions that perform read, write, and control operations use this ID.

If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process.

When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the function tries to create the facility, if it does not exist already.

When called with both the `IPC_CREAT` and `IPC_EXCL` flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with IPC_EXCL in effect, only the first attempt succeeds.

If neither of these flags is given and the facility already exists, the functions to get access return the ID of the facility. If IPC_CREAT is omitted and the facility is not already initialized, the calls fail.

These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist:

```
msqid = msgget(ftok("/tmp", ’A’), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key (’A’) based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

System V Messages

Before a process can send or receive a message, the queue must be initialized through the msgget(2) function. The owner or creator of a queue can change its ownership or permissions using msgctl(2). Also, any process with permission to do so can use msgctl(2) for control operations.

IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length.

Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Operations to send and receive messages are performed by the msgsnd(2) and msgrcv(2) functions, respectively. When a message is sent, its text is copied to the message queue. The msgsnd(2) and msgrcv(2) functions can be performed as either blocking or non-blocking operations. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initializing a Message Queue

The msgget(2) function initializes a new message queue. It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflg argument must be an octal integer with settings for the queue’s permissions and control flags.

The MSGMNI kernel configuration option determines the maximum number of unique message queues that the kernel will support. msgget(2) fails when this limit is exceeded. The following code illustrates msgget(2):

```
#include <sys/ipc.h>
 #include <sys/msg.h>

 ...
  key_t key;  /* key to be passed to msgget() */
  int msgflg, /* msgflg to be passed to msgget() */
    msqid; /* return value from msgget() */
  ...
  key = ...
  msgflg = ...
  if ((msqid = msgget(key, msgflg)) == -1)
  {
   perror("msgget: msgget failed");
   exit(1);
  } else
   (void) fprintf(stderr, "msgget succeeded");
  ...
```

Controlling Message Queues

The msgctl(2) function alters the permissions and other characteristics of a message queue. The msqid argument must be the ID of an existing message queue. The cmd argument is one of the following:

IPC_STAT	Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.
IPC_SET	Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.
IPC_RMID	Remove the message queue specified by the msqid argument.

The following code illustrates msgctl(2) with all its various flags:

```
#include   <sys/types.h>
 #include   <sys/ipc.h>
 #include   <sys/msg.h>

  ...
  if (msgctl(msqid, IPC_STAT, &buf) == -1)  {
   perror("msgctl: msgctl failed");
   exit(1);
  }
  ...
  if (msgctl(msqid, IPC_SET, &buf) == -1) {
   perror("msgctl: msgctl failed");
   exit(1);
  }
  ...
```

Sending and Receiving Messages

The msgsnd(2) and msgrcv(2) functions send and receive messages, respectively. The msqid argument must be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The msgsz argument specifies the length of the message in bytes. Various control flags can be passed in the msgflg argument.

The following code illustrates msgsnd(2) and msgrcv(2):

```
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>

 ...
  int              msgflg;    /* message flags for the operation */
  struct msgbuf    *msgp;     /* pointer to the message buffer */
  size_t           msgsz;     /* message size */
 size_t           maxmsgsize;
  long             msgtyp;    /* desired message type */
  int              msqid      /* message queue ID to be used */
  ...
  msgp = malloc(sizeof(struct msgbuf) - sizeof (msgp->mtext)
       + maxmsgsz);
  if (msgp == NULL) {
   (void) fprintf(stderr, "msgop: %s %ld byte messages.\n",
     "could not allocate message buffer for", maxmsgsz);
   exit(1);
   ...
   msgsz = ...
   msgflg = ...
   if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
    perror("msgop: msgsnd failed");
   ...
   msgsz = ...
   msgtyp = first_on_queue;
   msgflg = ...
   if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
    perror("msgop: msgrcv failed");
   ...
```

System V Semaphores

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments. Semaphores can be operated on as individual units or as elements in a set.

Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter weight semaphores are available in the threads library (see semaphore(3THR)) . Also, POSIX semaphores are the most current implementation of System V semaphores (see “POSIX Semaphores” on page 69). Threads library semaphores must be used with mapped memory (see “Memory Management Interfaces” on page 61).

A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements. The semaphore set must be initialized using semget(2). The semaphore creator can change its ownership or permissions using semctl(2). Any process with permission can use semctl(2) to do control operations.

Semaphore operations are performed by the semop(2) function. This function takes a pointer to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. Operations to increment or decrement a semaphore require write permission.

When an operation fails, none of the semaphores is altered. The process blocks (unless the IPC_NOWAIT flag is set), and remains blocked until:

- The semaphore operations can all finish, so the call succeeds
- The process receives a signal
- The semaphore set is removed

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a semop(2) call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the SEM_UNDO control flag makes semop(2) allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state.

If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with SEM_UNDO in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state.

When performing a semaphore operation with SEM_UNDO in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancel to zero. When the undo structure reaches zero, it is removed.

Using SEM_UNDO inconsistently can lead to excessive resource consumption because allocated undo structures might not be freed until the system is rebooted.

Initializing a Semaphore Set

semget(2) initializes or gains access to a semaphore. When the call succeeds, it returns the semaphore ID (semid). The key argument is a value associated with the semaphore ID. The nsems argument specifies the number of elements in a semaphore array. The call fails when nsems is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed. The semflg argument specifies the initial access permissions and creation control flags.

The SEMMNI system configuration option determines the maximum number of semaphore arrays allowed. The SEMMNS option determines the maximum possible number of individual semaphores across all semaphore sets. Because of fragmentation between semaphore sets, it might not be possible to allocate all available semaphores.

The following code illustrates semget(2).

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>
...
  key_t  key;         /* key to pass to semget() */
  int    semflg;      /* semflg to pass to semget() */
  int    nsems;       /* nsems to pass to semget() */
  int    semid;       /* return value from semget() */
  ...
  key = ...
  nsems = ...
  semflg = ...
  ...
  if ((semid = semget(key, nsems, semflg)) == -1) {
   perror("semget: semget failed");
   exit(1);
  } else
   exit(0);
 ...
```

Controlling Semaphores

semctl(2) changes permissions and other characteristics of a semaphore set. It must be called with a valid semaphore ID. The semnum value selects a semaphore within an array by its index. The *cmd* argument is one of the following control flags.

GETVAL — Return the value of a single semaphore.

SETVAL — Set the value of a single semaphore. In this case, arg is taken as arg.val, an int.

GETPID — Return the PID of the process that performed the last operation on the semaphore or array.

GETNCNT — Return the number of processes waiting for the value of a semaphore to increase.

GETZCNT — Return the number of processes waiting for the value of a particular semaphore to reach zero.

GETALL — Return the values for all semaphores in a set. In this case, arg is taken as arg.array, a pointer to an array of unsigned shorts.

SETALL — Set values for all semaphores in a set. In this case, arg is taken as arg.array, a pointer to an array of unsigned shorts.

IPC_STAT — Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by arg.buf, a pointer to a buffer of type semid_ds.

IPC_SET — Set the effective user and group identification and permissions. In this case, arg is taken as arg.buf.

IPC_RMID — Remove the specified semaphore set.

A process must have an effective user identification of owner, creator, or superuser to perform an IPC_SET or IPC_RMID command. Read and write permission is required as for the other control commands.

The following code illustrates semctl(2):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
...
  register int    i;
 ...
  i = semctl(semid, semnum, cmd, arg);
  if (i == -1) {
   perror("semctl: semctl failed");
   exit(1);
...
```

Semaphore Operations

semop(2) performs operations on a semaphore set. The semid argument is the semaphore ID returned by a previous semget(2) call. The sops argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- The semaphore number
- The operation to be performed
- Control flags, if any

The sembuf structure specifies a semaphore operation, as defined in <sys/sem.h>. The nsops argument specifies the length of the array, the maximum size of which is determined by the *SEMOPM* configuration option; this is the maximum number of operations allowed by a single semop(2) call, and is set to 10 by default.

The operation to be performed is determined as follows:

- Positive integer increments the semaphore value by that amount.

- Negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether IPC_NOWAIT is in effect.
- Value of zero means to wait for the semaphore value to reach zero.

The two control flags that can be used with semop(2) are shown below:

IPC_NOWAIT
Can be set for any operations in the array. Makes the function return without changing any semaphore value if any operation for which IPC_NOWAIT is set cannot be performed. The function fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

SEM_UNDO
Allows individual operations in the array to be undone when the process exits.

The following code illustrates the semop(2) function:

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
...
  int    i;   /* work area */
  int    nsops; /* number of operations to do */
  int    semid; /* semid of semaphore set */
  struct sembuf *sops; /* ptr to operations to perform */
  ...
  if ((i = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
  } else
   (void) fprintf(stderr, "semop: returned %d\n", i);
 ...
```

System V Shared Memory

In the SunOS 5.8 operating system, the most efficient way to implement shared memory applications is to rely on the mmap(2) function and on the system’s native virtual memory facility. See Chapter 6.

SunOS 5.8 also supports System V shared memory, which is a less efficient way to let multiple processes attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using shmget(2). This call is also used to get the ID of an existing shared segment. The creating process sets the permissions and the size in bytes for the segment.

The original owner of a shared memory segment can assign ownership to another user with shmctl(2). It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl`(2).

Once created, a shared segment can be attached to a process address space using `shmat`(2). It can be detached using `shmdt`(2). The attaching process must have the appropriate permissions for `shmat`(2). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process.

A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the `shmid`. The structure definition for the shared memory segment control structure can be found in `<sys/shm.h>`.

Accessing a Shared Memory Segment

`shmget`(2) is used to obtain access to a shared memory segment. When the call succeeds, it returns the shared memory segment ID (*shmid*). The following code illustrates `shmget`(2):

```
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
  key_t    key;       /* key to be passed to shmget() */
  int      shmflg;    /* shmflg to be passed to shmget() */
  int      shmid;     /* return value from shmget() */
  size_t   size;      /* size to be passed to shmget() */
  ...
  key = ...
  size = ...
  shmflg) = ...
  if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed");
   exit(1);
  } else {
   (void) fprintf(stderr,
      "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
 ...
```

Controlling a Shared Memory Segment

`shmctl`(2) is used to alter the permissions and other characteristics of a shared memory segment. The `cmd` argument is one of following control commands:

SHM_LOCK
Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.

SHM_UNLOCK
Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.

IPC_STAT
Return the status information contained in the control structure and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.

IPC_SET
Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.

IPC_RMID
Remove the shared memory segment. The process must have an effective ID of owner, creator or superuser to perform this command.

The following code illustrates shmctl(2):

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
...
 int  cmd;  /* command code for shmctl() */
 int  shmid; /* segment ID */
 struct shmid_ds shmid_ds; /* shared memory data structure to
          hold results */
  ...
  shmid = ...
  cmd = ...
  if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
   perror("shmctl: shmctl failed");
   exit(1);
  ...
```

Attaching and Detaching a Shared Memory Segment

shmat() and shmdt() (see shmop(2)) are used to attach and detach shared memory segments. shmat(2) returns a pointer to the head of the shared segment. shmdt(2) detaches the shared memory segment located at the address indicated by *shmaddr*. The following code illustrates calls to shmat(2) and shmdt(2):

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

static struct state { /* Internal record of attached segments. */
  int  shmid;  /* shmid of attached segment */
  char  *shmaddr; /* attach point */
  int  shmflg;  /* flags used on attach */
  } ap[MAXnap];   /* State of current attached segments. */
  int  nap;   /* Number of currently attached segments. */
 ...
  char    *addr;   /* address work variable */
  register int    i;   /* work area */
  register struct state   *p;    /* ptr to current state entry */
 ...
  p = &ap[nap++];
  p->shmid = ...
  p->shmaddr = ...
  p->shmflg = ...
  p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
  if(p->shmaddr == (char *)-1) {
   perror("shmat failed");
   nap--;
  } else
    (void) fprintf(stderr, "shmop: shmat returned %p\n",
      p->shmaddr);
  ...
  i = shmdt(addr);
  if(i == -1) {
   perror("shmdt failed");
  } else {
   (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
   for (p = ap, i = nap; i--; p++) {
    if (p->shmaddr == addr) *p = ap[--nap];
   }
  }
  ...
```

CHAPTER **8**

Real-time Programming and Administration

This chapter describes writing and porting real-time applications to run under Solaris SunOS 5.0 through 5.8. This chapter is written for programmers experienced in writing real-time applications and administrators familiar with real-time processing and the Solaris system.

Basic Rules of Real-time Applications

Real-time response is guaranteed when certain conditions are met. This section identifies these conditions and some of the more significant design errors that can cause problems or disable a system.

Most of the potential problems described here can degrade the response time of the system. One of the potential problems can freeze a workstation. Other, more subtle, mistakes are priority inversion and system overload.

A Solaris real-time process:

- Runs in the RT scheduling class, as described in "Scheduling" on page 85.
- Locks down all the memory in its process address space, as described in "Memory Locking" on page 96.
- Is from a statically-linked program or from a program in which all dynamic binding is completed early, as described in "Shared Libraries " on page 83.

Real-time operations are described in this chapter in terms of single-threaded processes, but the description can also apply to multithreaded processes (for detailed information about multithreaded processes, see the *Multithreaded Programming Guide*). To guarantee real-time scheduling of a thread, it must be created as a bound thread, and the thread's LWP must be run in the RT scheduling class. The locking of memory and early dynamic binding is effective for all threads in a process.

When a process is the highest priority real-time process, it:

- Acquires the processor within the guaranteed dispatch latency period of becoming runnable (see"Dispatch Latency" on page 85)
- Continues to run for as long as it remains the highest priority runnable process

A real-time process can lose control of the processor or can be unable to gain control of the processor because of other events on the system. These events include external events (such as interrupts), resource starvation, waiting on external events (synchronous I/O), and preemption by a higher priority process.

Real-time scheduling generally does not apply to system initialization and termination services such as `open`(2) and `close`(2).

Degrading Response Time

The problems described in this section all increase the response time of the system to varying extents. The degradation can be serious enough to cause an application to miss a critical deadline.

Real-time processing can also significantly impact the operation of aspects of other applications active on a system running a real-time application. Since real-time processes have higher priority, time-sharing processes can be prevented from running for significant amounts of time. This can cause interactive activities, such as displays and keyboard response time, to be noticeably slowed.

System Response Time

System response under SunOS 5.0 through 5.8 provides no bounds to the timing of I/O events. This means that synchronous I/O calls should never be included in any program segment whose execution is time critical. Even program segments that permit very large time bounds must not perform synchronous I/O. Mass storage I/O is such a case, where causing a read or write operation hangs the system while the operation takes place.

A common application mistake is to perform I/O to get error message text from disk. This should be done from an independent nonreal-time process or thread.

Interrupt Servicing

Interrupt priorities are independent of process priorities. Prioritizing processes does not carry through to prioritizing the services of hardware interrupts that result from the actions of the processes. This means that interrupt processing for a device controlled by a real-time process is not necessarily done before interrupt processing for another device controlled by a timeshare process.

Shared Libraries

Time-sharing processes can save significant amounts of memory by using dynamically linked, shared libraries. This type of linking is implemented through a form of file mapping. Dynamically linked library routines cause implicit reads.

Real-time programs can use shared libraries, yet avoid dynamic binding, by setting the environment variable `LD_BIND_NOW` to a non-`NULL` value when the program is invoked. This forces all dynamic linking to be bound before the program begins execution. See the *Linker and Libraries Guide* for more information.

Priority Inversion

A time-sharing process can block a real-time process by acquiring a resource that is required by a real-time process. Priority inversion is a condition that occurs when a higher priority process is blocked by a lower priority process. The term *blocking* describes a situation in which a process must wait for one or more processes to relinquish control of resources. If this blocking is prolonged, even for lower level resources, deadlines might be missed.

By way of illustration, consider the case in Figure 8–1where a high priority process wanting to use a shared resource gets blocked when a lower priority process holds the resource, and the lower priority process is preempted by an intermediate priority process. This condition can persist for a long time, arbitrarily long, in fact, since the amount of time the high priority process must wait for the resource depends not only on the duration of the critical section being executed by the lower priority process, but on the duration until the intermediate process blocks. Any number of intermediate processes can be involved.

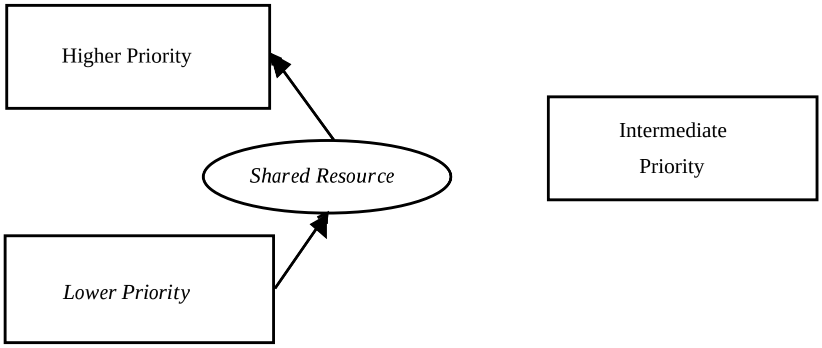

Figure 8–1 Unbounded Priority Inversion

This issue and the methods of dealing with it are described in “Mutual Exclusion Lock Attributes” in *Multithreaded Programming Guide*.

Sticky Locks

A page is permanently locked into memory when its lock count reaches 65535 (0xFFFF). The value 0xFFFF is implementation-defined and might change in future releases. Pages locked this way cannot be unlocked.

Runaway Real-time Processes

Runaway real-time processes can cause the system to halt or can slow the system response so much that the system appears to halt.

Note - If you have a runaway process on a SPARC system, press Stop-A. You might have to do this more than one time. If this doesn’t work, or on non-SPARC systems, turn the power off, wait a moment, then turn it back on.

When a high priority real-time process does not relinquish control of the CPU, there is no simple way to regain control of the system until the infinite loop is forced to terminate. Such a runaway process does not respond to control-C. Attempts to use a shell set at a higher priority than that of a runaway process do not work.

I/O Behavior

Asynchronous I/O

There is no guarantee that asynchronous I/O operations will be done in the sequence in which they are queued to the kernel. Nor is there any guarantee that asynchronous operations will be returned to the caller in the sequence in which they were done.

If a single buffer is specified for a rapid sequence of calls to aioread(3AIO), there is no guarantee about the state of the buffer between the time that the first call is made and the time that the last result is signaled to the caller.

An individual aio_result_t structure can be used only for one asynchronous read or write at a time.

Real-time Files

SunOS 5.0 through 5.8 provides no facilities to ensure that files will be allocated as physically contiguous.

For regular files, the `read`(2) and `write`(2) operations are always buffered. An application can use `mmap`(2) and `msync`(3C) to effect direct I/O transfers between secondary storage and process memory.

Scheduling

Real-time scheduling constraints are necessary to manage data acquisition or process control hardware. The real-time environment requires that a process be able to react to external events in a bounded amount of time. Such constraints can exceed the capabilities of a kernel designed to provide a "fair" distribution of the processing resources to a set of time-sharing processes.

This section describes the SunOS 5.0 through 5.8 real-time scheduler, its priority queue, and how to use system calls and utilities that control scheduling.

Dispatch Latency

The most significant element in scheduling behavior for real-time applications is the provision of a real-time scheduling class. The standard time-sharing scheduling class is not suitable for real-time applications because this scheduling class treats every process equally and has a limited notion of priority. Real-time applications require a scheduling class in which process priorities are taken as absolute and are changed only by explicit application operations.

The term dispatch latency describes the amount of time it takes for a system to respond to a request for a process to begin operation. With a scheduler written specifically to honor application priorities, real-time applications can be developed with a bounded dispatch latency.

Figure 8–2 illustrates the amount of time it takes an application to respond to a request from an external event.

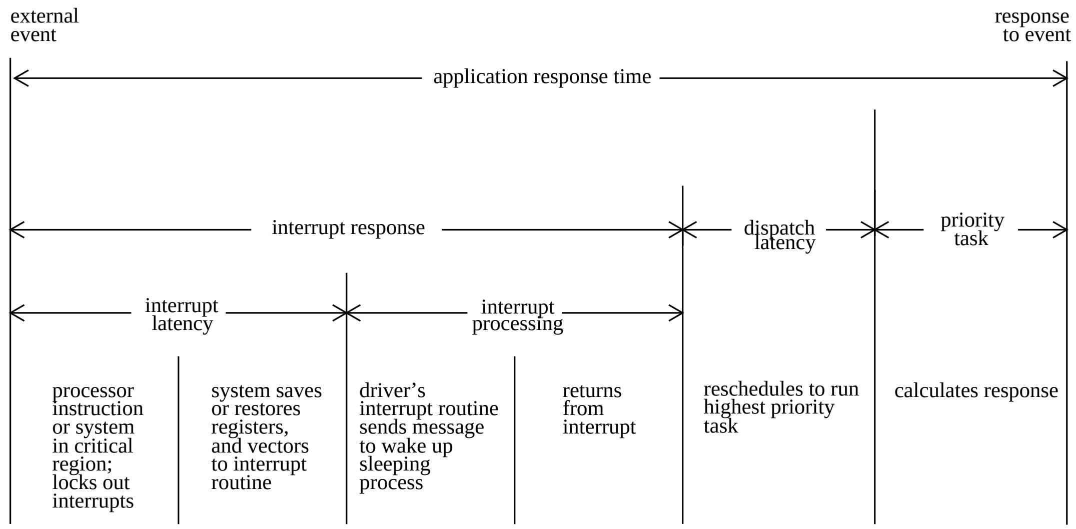
external event
response to event
application response time
interrupt response
dispatch latency
priority task
interrupt latency
interrupt processing
processor instruction or system in critical region; locks out interrupts
system saves or restores registers, and vectors to interrupt routine
driver's interrupt routine sends message to wake up sleeping process
returns from interrupt
reschedules to run highest priority task
calculates response

Figure 8–2 Application Response Time.

The overall application response time is composed of the interrupt response time, the dispatch latency, and the time it takes the application itself to determine its response.

The interrupt response time for an application includes both the interrupt latency of the system and the device driver's own interrupt processing time. The interrupt latency is determined by the longest interval that the system must run with interrupts disabled; this is minimized in SunOS 5.0 through 5.8 using synchronization primitives that do not commonly require a raised processor interrupt level.

During interrupt processing, the driver's interrupt routine wakes up the high priority process and returns when finished. The system detects that a process with higher priority than the interrupted process in now dispatchable and arranges to dispatch that process. The time to switch context from a lower priority process to a higher priority process is included in the dispatch latency time.

Figure 8–3 illustrates the internal dispatch latency/application response time of a system, defined in terms of the amount of time it takes for a system to respond to an internal event. The dispatch latency of an internal event represents the amount of time required for one process to wake up another higher priority process, and for the system to dispatch the higher priority process.

The application response time is the amount of time it takes for a driver to wake up a higher priority process, have a low priority process release resources, reschedule the higher priority task, calculate the response, and dispatch the task.

Note - Interrupts can arrive and be processed during the dispatch latency interval. This processing increases the application response time, but is not attributed to the dispatch latency measurement, and so is not bounded by the dispatch latency guarantee.

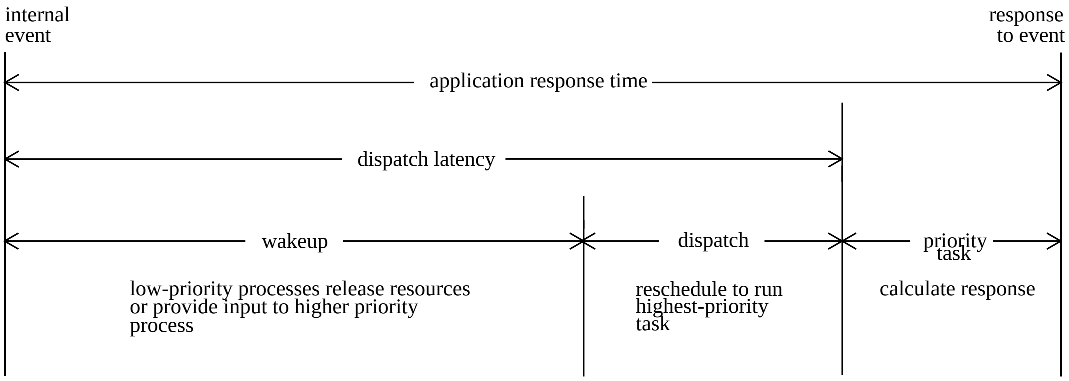
internal event
response to event
application response time
dispatch latency
wakeup
dispatch
priority task
low-priority processes release resources or provide input to higher priority process
reschedule to run highest-priority task
calculate response

Figure 8–3 Internal Dispatch Latency

With the new scheduling techniques provided with real-time SunOS 5.0 through 5.8, the system dispatch latency time is within specified bounds. As you can see in the table below, dispatch latency improves with a bounded number of processes.

TABLE 8–1 Real-time System Dispatch Latency with SunOS 5.0 through 5.8

Workstation	Bounded Number of Processes	Arbitrary Number of Processes
SPARCstation 2	<0.5 milliseconds in a system with fewer than 16 active processes	1.0 milliseconds
SPARCstation 5	<0.3 millisecond	0.3 millisecond
Ultra 1-167	<0.15 millisecond	<0.15 millisecond

Tests for dispatch latency and experience with such critical environments as manufacturing and data acquisition have proven that SunOS 5.8 is an effective platform for the development of real-time applications. (These examples are not of current products.)

Scheduling Classes

The SunOS 5.0 through 5.8 kernel dispatches processes by priority. The scheduler (or dispatcher) supports the concept of scheduling classes. Classes are defined as Real-time (RT), System (sys), and Time-Sharing (TS). Each class has a unique scheduling policy for dispatching processes within its class.

The kernel dispatches highest priority processes first. By default, real-time processes have precedence over sys and TS processes, but administrators can configure systems so that TS and RT processes have overlapping priorities.

Figure 8–4 illustrates the concept of classes as viewed by the SunOS 5.0 through 5.8 kernel.

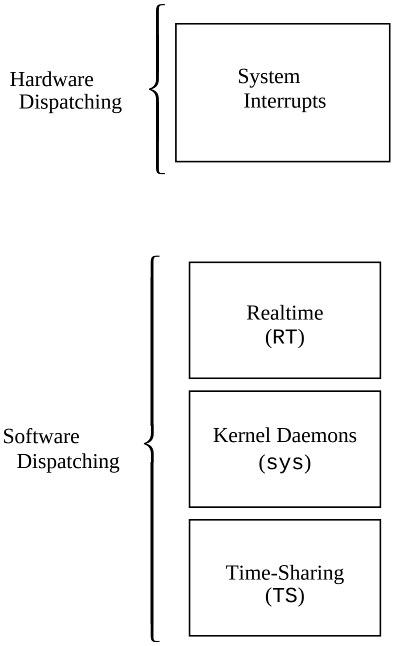

Figure 8–4 Dispatch Priorities for Scheduling Classes

At highest priority are the hardware interrupts; these cannot be controlled by software. The interrupt processing routines are dispatched directly and immediately from interrupts, without regard to the priority of the current process.

Real-time processes have the highest default software priority. Processes in the RT class have a priority and *time quantum* value. RT processes are scheduled strictly on the basis of these parameters. As long as an RT process is ready to run, no SYS or TS process can run. Fixed priority scheduling allows critical processes to run in a predetermined order until completion. These priorities never change unless an application changes them.

An RT class process inherits the parent's time quantum, whether finite or infinite. A process with a finite time quantum runs until the time quantum expires or the process terminates, blocks (while waiting for an I/O event), or is preempted by a higher priority runnable real-time process. A process with an infinite time quantum ceases execution only when it terminates, blocks, or is preempted.

The SYS class exists to schedule the execution of special system processes, such as paging, STREAMS, and the swapper. It is not possible to change the class of a process to the SYS class. The SYS class of processes has fixed priorities established by the kernel when the processes are started.

At lowest priority are the time-sharing (TS) processes. TS class processes are scheduled dynamically, with a few hundred milliseconds for each time slice. The TS scheduler switches context in round-robin fashion often enough to give every process an equal opportunity to run, depending upon its time slice value, its process history (when the process was last put to sleep), and considerations for CPU utilization. Default time-sharing policy gives larger time slices to processes with lower priority.

A child process inherits the scheduling class and attributes of the parent process through `fork`(2). A process' scheduling class and attributes are unchanged by `exec`(2).

Different algorithms dispatch each scheduling class. Class dependent routines are called by the kernel to make decisions about CPU process scheduling. The kernel is class-independent, and takes the highest priority process off its queue. Each class is responsible for calculating a process' priority value for its class. This value is placed into the dispatch priority variable of that process.

As Figure 8–5 illustrates, each class algorithm has its own method of nominating the highest priority process to place on the global run queue.

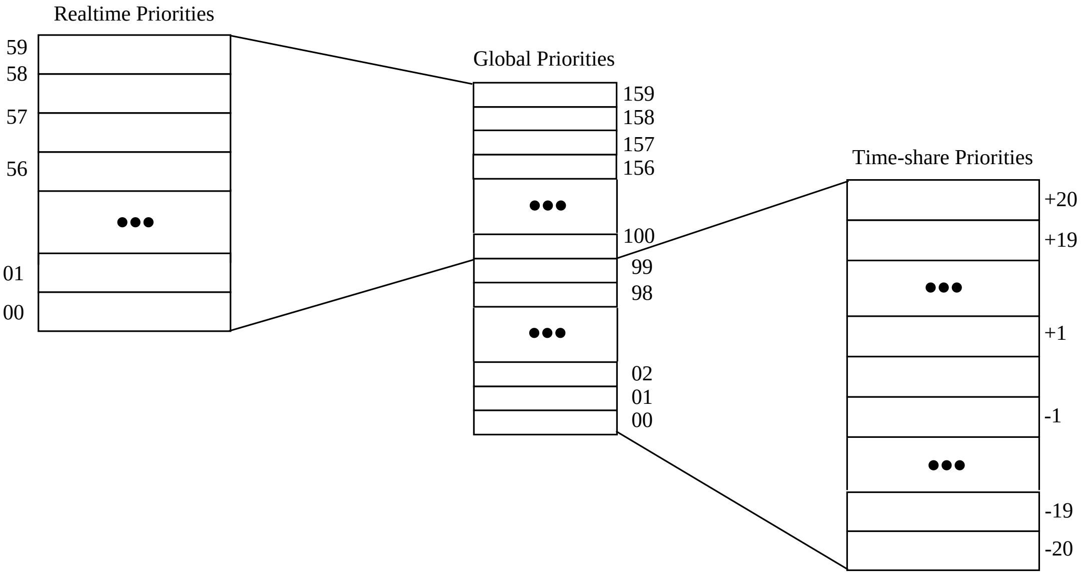

Figure 8–5 The Kernel Dispatch Queue

Each class has a set of priority levels that apply to processes in that class. A class-specific mapping maps these priorities into a set of global priorities. It is not required that a set of global scheduling priority maps start with zero, nor that they be contiguous.

By default, the global priority values for time-sharing (TS) processes range from -20 to +20, mapped into the kernel from 0-40, with temporary assignments as high as 99.

The default priorities for real-time (RT) processes range from 0-59, and are mapped into the kernel from 100 to 159. The kernel's class-independent code runs the process with the highest global priority on the queue.

Dispatch Queue

The dispatch queue is a linear-linked list of processes with the same global priority. Each process is invoked with class specific information attached to it. A process is dispatched from the kernel dispatch table based upon its global priority.

Dispatching Processes

When a process is dispatched, the process' context is mapped into memory along with its memory management information, its registers, and its stack. Then execution begins. Memory management information is in the form of hardware registers containing data needed to perform virtual memory translations for the currently running process.

Preemption

When a higher priority process becomes dispatchable, the kernel interrupts its computation and forces the context switch, preempting the currently running process. A process can be preempted at any time if the kernel finds that a higher priority process is now dispatchable.

For example, suppose that process A performs a read from a peripheral device. Process A is put into the sleep state by the kernel. The kernel then finds that a lower priority process B is runnable, so process B is dispatched and begins execution. Eventually, the peripheral device interrupts, and the driver of the device is entered. The device driver makes process A runnable and returns. Rather than returning to the interrupted process B, the kernel now preempts B from processing and resumes execution of the awakened process A.

Another interesting situation occurs when several processes contend for kernel resources. When a lower priority process releases a resource for which a higher priority real-time process is waiting, the kernel immediately preempts the lower priority process and resumes execution of the higher priority process.

Kernel Priority Inversion

Priority inversion occurs when a higher priority process is blocked by one or more lower priority processes for a long time. The use of synchronization primitives such as mutual-exclusion locks in the SunOS 5.0 through 5.8 kernel can lead to priority inversion.

A process is *blocked* when it must wait for one or more processes to relinquish resources. If blocking continues, it can lead to missed deadlines, even for low levels of utilization.

The problem of priority inversion has been addressed for mutual-exclusion locks for the SunOS 5.0 through 5.8 kernel by implementing a basic priority inheritance policy. The policy states that a lower priority process inherits the priority of a higher priority process when the lower priority process blocks the execution of the higher priority process. This places an upper bound on the amount of time a process can remain blocked. The policy is a property of the kernel's behavior, not a solution that a programmer institutes through system calls or function execution. User-level processes can still exhibit priority inversion, however.

User Priority Inversion

This issue and the means to deal with it are discussed in "Mutual Exclusion Lock Attributes" in *Multithreaded Programming Guide*.

Function Calls That Control Scheduling

priocntl(2)

Control over scheduling of active classes is done with priocntl(2). Class attributes are inherited through fork(2) and exec(2), along with scheduling parameters and permissions required for priority control. This is true for both the RT and the TS classes.

The priocntl(2) function is the interface for specifying a real-time process, a set of processes, or a class to which the system call applies. priocntlset(2) also provides the more general interface for specifying an entire set of processes to which the system call applies.

The command arguments of priocntl(2) can be one of: PC_GETCID, PC_GETCLINFO, PC_GETPARMS, or PC_SETPARMS. The real or effective ID of the calling process must match that of the affected processes, or must have super-user privilege.

PC_GETCID	This command takes the name field of a structure that contains a recognizable class name (RT for real-time and TS for time-sharing). The class ID and an array of class attribute data are returned.
PC_GETCLINFO	This command takes the ID field of a structure that contains a recognizable class identifier. The class name and an array of class attribute data are returned.
PC_GETPARMS	This command returns the scheduling class identifier and/or the class specific scheduling parameters of one of the specified processes. Even though idtype & id might specify a big set, PC_GETPARMS returns the parameter of only one process. It is up to the class to select which one.
PC_SETPARMS	This command sets the scheduling class and/or the class specific scheduling parameters of the specified process or processes.

sched_get_priority_max(3RT)

Returns the maximum values for the specified policy.

sched_get_priority_min(3RT)

Returns the minimum values for the specified policy (see sched_get_priority_max(3R)).

sched_rr_get_interval(3RT)

Updates the specified timespec structure to the current execution time limit (see sched_get_priority_max(3RT)).

sched_setparam(3RT), sched_getparam(3RT)

Sets or gets the scheduling parameters of the specified process.

sched_yield(3RT)

Blocks the calling process until it returns to the head of the process list.

Utilities That Control Scheduling

The administrative utilities that control process scheduling are dispadmin() and priocntl(1). Both these utilities support the priocntl(2) system call with compatible options and loadable modules. These utilities provide system administration functions that control real-time process scheduling during runtime.

priocntl(1)

The priocntl(1) command sets and retrieves scheduler parameters for processes.

dispadmin()

The dispadmin() utility displays all current process scheduling classes by including the -l command line option during runtime. Process scheduling can also be changed for the class specified after the -c option, using RT as the argument for the real-time class.

The options shown in Table 8–2are also available.

TABLE 8–2 Class Options for the dispadmin(1M) Utility

Option	Meaning
-l	Lists scheduler classes currently configured
-c	Specifies the class whose parameters are to be displayed or changed
-g	Gets the dispatch parameters for the specified class
-r	Used with –g, specifies time quantum resolution
-s	Specifies a file where values can be located

A class specific file containing the dispatch parameters can also be loaded during runtime. Use this file to establish a new set of priorities replacing the default values established during boot time. This class specific file must assert the arguments in the format used by the -g option. Parameters for the RT class are found in the rt_dptbl(4), and are listed in the example at the end of this section.

To add an RT class file to the system, the following modules must be present:

- An rt_init() routine in the class module that loads the rt_dptbl(4)
- An rt_dptbl(4) module that provides the dispatch parameters and a routine to return pointers to config_rt_dptbl
- The dispadmin() executable

1. **Load the class specific module with the following command, where *module_name* is the class specific module:**

```
# modload /kernel/sched/module_name
```

2. **Invoke the dispadmin() command:**

```
# dispadmin -c RT -s file_name
```

The file must describe a table with the same number of entries as the table that is being overwritten.

Configuring Scheduling

Associated with both scheduling classes is a parameter table, `rt_dptbl`(4), and `ts_dptbl`(4). These tables are configurable by using a loadable module at boot time, or with `dispadmin`() during runtime.

Dispatcher Parameter Table

The in-core table for real-time establishes the properties for RT scheduling. The `rt_dptbl`(4) structure consists of an array of parameters, `struct rt_dpent_t`, one for each of the n priority levels. The properties of a given priority level are specified by the ith parameter structure in the array, `rt_dptbl[i]`.

A parameter structure consists of the following members (also described in the `/usr/include/sys/rt.h` header file).

`rt_globpri`	The global scheduling priority associated with this priority level. The `rt_globpri` values cannot be changed with `dispadmin`().
`rt_quantum`	The length of the time quantum allocated to processes at this level in ticks (see "Timestamp Functions" on page 107). The time quantum value is only a default or starting value for processes at a particular level. The time quantum of a realtime process can be changed by using the `priocntl`(1) command or the `priocntl`(2) system call.

Reconfiguring `config_rt_dptbl`

A real-time administrator can change the behavior of the real-time portion of the scheduler by reconfiguring the `config_rt_dptbl` at any time. One method is described in rt_dptbl(4) in the section titled “REPLACING THE RT_DPTBL LOADABLE MODULE.”

A second method for examining or modifying the real-time parameter table on a running system is through using the dispadmin() command. Invoking dispadmin() for the real-time class allows retrieval of the current rt_quantum values in the current config_rt_dptbl configuration from the kernel’s in-core table. When overwriting the current in-core table, the configuration file used for input to dispadmin() must conform to the specific format described in rt_dptbl(4).

Following is an example of prioritized processes rtdpent_t with their associated time quantum config_rt_dptbl[] value as they might appear in config_rt_dptbl[]:

```
rtdpent_t  rt_dptbl[] = {    129,    60,
  /* prilevel Time quantum */       130,    40,
  100,    100,           131,    40,
  101,    100,           132,    40,
  102,    100,           133,    40,
  103,    100,           134,    40,
  104,    100,           135,    40,
  105,    100,           136,    40,
  106,    100,           137,    40,
  107,    100,           138,    40
  108,    100,           139,    40,
  109,    100,           140,    20,
  110,    80,            141,    20,
  111,    80,            142,    20,
  112,    80,            143,    20,
  113,    80,            144,    20,
  114,    80,            145,    20,
  115,    80,            146,    20,
  116,    80,            147,    20,
  117,    80,            148,    20,
  118,    80,            149,    20,
  119,    80,            150,    10,
  120,    60,            151,    10,
  121,    60,            152,    10,
  122,    60,            153,    10,
  123,    60,            154,    10,
  124,    60,            155,    10,
  125,    60,            156,    10,
  126,    60,            157,    10,
  126,    60,            158,    10,
  127,    60,            159,    10,
  128,    60,          }
```

Memory Locking

Locking memory is one of the most important issues for real-time applications. In a real-time environment, a process must be able to guarantee continuous memory residence to reduce latency and to prevent paging and swapping.

This section describes the memory locking mechanisms available to real-time applications in SunOS 5.0 through 5.8.

Overview

Under SunOS 5.0 through 5.8, the memory residency of a process is determined by its current state, the total available physical memory, the number of active processes, and the processes' demand for memory. This is appropriate in a time-share environment, but it is often unacceptable for a real-time process. In a real-time environment, a process must guarantee a memory residence for all or part of itself to reduce its memory access and dispatch latency.

For real-time in SunOS 5.0 through 5.8, memory locking is provided by a set of library routines that allow a process running with superuser privileges to lock specified portions of its virtual address space into physical memory. Pages locked in this manner are exempt from paging until they are unlocked or the process exits.

There is a system-wide limit on the number of pages that can be locked at any time. This is a tunable parameter whose default value is calculated at boot time. It is based on the number of page frames less another percentage (currently set at ten percent).

Locking a Page

A call to `mlock`(3C) requests that one segment of memory be locked into the system's physical memory. The pages that make up the specified segment are faulted in and the lock count of each is incremented. Any page with a lock count greater than 0 is exempt from paging activity.

A particular page can be locked multiple times by multiple processes through different mappings. If two different processes lock the same page, the page remains locked until both processes remove their locks. However, within a given mapping, page locks do not nest. Multiple calls of locking functions on the same address by the same process are removed by a single unlock request.

If the mapping through which a lock has been performed is removed, the memory segment is implicitly unlocked. When a page is deleted through closing or truncating the file, it is also unlocked implicitly.

Locks are not inherited by a child process after a fork(2) call is made. So, if a process with memory locked forks a child, the child must perform a memory locking operation in its own behalf to lock its own pages. Otherwise, the child process incurs copy-on-write page faults, which are the usual penalties associated with forking a process.

Unlocking a Page

To unlock a page of memory, a process requests that a segment of locked virtual pages be released by a call to munlock(3C). The lock counts of the specified physical pages are decremented. Once the lock count of a page has been decremented to 0, the page is swapped normally.

Locking All Pages

A superuser process can request that all mappings within its address space be locked by a call to mlockall(3C). If the flag MCL_CURRENT is set, all the existing memory mappings are locked. If the flag MCL_FUTURE is set, every mapping that is added to or that replaces an existing mapping is locked into memory.

Sticky Locks

A page is permanently locked into memory when its lock count reaches 65535 (0xFFFF). The value 0xFFFF is implementation defined and might change in future releases. Pages locked in this manner cannot be unlocked. Reboot the system to recover.

High Performance I/O

This section describes I/O with realtime processes. In SunOS 5.0 through 5.8, the libraries supply two sets of functions and calls to perform fast, asynchronous, I/O operations. The POSIX asynchronous I/O interfaces are the new standard. For robustness, SunOS also provides file and in-memory synchronization operations and modes to prevent information loss and data inconsistency.

Standard UNIX I/O is synchronous to the application programmer. An application that calls read(2) or write(2) usually waits until the system call has finished.

Real-time applications need asynchronous, bounded I/O behavior. A process that issues an asynchronous I/O call proceeds without waiting for the I/O operation to complete. The caller is notified when the I/O operation has finished. In the mean time the process does something useful.

Asynchronous I/O can be used with any SunOS file. Files are opened in the synchronous way and no special flagging is required. An asynchronous I/O transfer has three elements: call, request, and operation. The application calls an asynchronous I/O function, the request for the I/O is placed on a queue, and the call returns immediately. At some point, the system dequeues the request and initiates the I/O operation.

Asynchronous and standard I/O requests can be intermingled on any file descriptor. The system maintains no particular sequence of read and write requests. The system arbitrarily resequences all pending read and write requests. If a specific sequence is required for the application, the application must insure the completion of prior operations before issuing the dependent requests.

POSIX Asynchronous I/O

POSIX asynchronous I/O is performed using `aiocb` structures. An `aiocb` control block identifies each asynchronous I/O request and contains all of the controlling information. A control block can be used for only one request at a time and can be reused after its request has been completed.

A typical POSIX asynchronous I/O operation is initiated by a call to `aio_read`(3RT) or `aio_write`(3RT). Either polling or signals can be used to determine the completion of an operation. If signals are used for operation completion, each operation can be uniquely tagged and the tag is returned in the `si_value` component of the generated signal (see `siginfo`(3HEAD)).

`aio_read`(3RT)

`aio_read`(3RT) is called with an asynchronous I/O control block to initiate a read operation.

`aio_write`(3RT)

`aio_write`(3RT) is called with an asynchronous I/O control block to initiate a write operation.

`aio_return`(3RT) and `aio_error`(3RT)

`aio_return`(3RT) and `aio_error`(3RT) are called to obtain return and error values. respectively, after an operation is known to have been completed.

aio_cancel(3RT)

aio_cancel(3RT) is called with an asynchronous I/O control block to cancel pending operations. It can be used to cancel a specific request, if the control block specifies one, or all of the requests pending for the specified file descriptor.

aio_fsync(3RT)

aio_fsync(3RT) queues an asynchronous fsync(3C) or fdatasync(3RT) request for all of the pending I/O operations on the specified file.

aio_suspend(3RT)

aio_suspend(3RT) suspends the caller as though one, or more, of the preceding asynchronous I/O requests had been made synchronously.

Solaris Asynchronous I/O

Notification (SIGIO)

When an asynchronous I/O call returns successfully, the I/O operation has only been queued, waiting to be done. The actual operation also has a return value and a potential error identifier, the values that would have been returned to the caller as the result of a synchronous call. When the I/O is finished, the return value and error value are stored at a location given by the user at the time of the request as a pointer to an aio_result_t. The structure of the aio_result_t is defined in <sys/asynch.h>:

```
typedef struct aio_result_t {
  ssize_t aio_return; /* return value of read or write */
  int   aio_errno;  /* errno generated by the IO */
 } aio_result_t;
```

When aio_result_t has been updated, a SIGIO signal is delivered to the process that made the I/O request.

Note that a process with two or more asynchronous I/O operations pending has no certain way to determine which request, if any, is the cause of the SIGIO signal. A process receiving a SIGIO should check all its conditions that could be generating the SIGIO signal.

aioread(3AIO)

aioread(3AIO) is the asynchronous version of read(2). In addition to the normal read arguments, aioread(3AIO) takes the arguments specifying a file position and the address of an aio_result_t structure in which the system stores the result information about the operation. The file position specifies a seek to be performed within the file before the operation. Whether the aioread(3AIO) call succeeds or fails, the file pointer is updated.

aiowrite(3AIO)

aiowrite(3AIO) is the asynchronous version of write(2). In addition to the normal write arguments, aiowrite(3AIO) takes arguments specifying a file position and the address of an aio_result_t structure in which the system is to store the resulting information about the operation.

The file position specifies a seek to be performed within the file before the operation. If the aiowrite(3AIO) call succeeds, the file pointer is updated to the position that would have resulted in a successful seek and write. The file pointer is also updated when a write fails to allow for subsequent write requests.

aiocancel(3AIO)

aiocancel(3AIO) attempts to cancel the asynchronous request whose aio_result_t structure is given as an argument. An aiocancel(3AIO) call succeeds only if the request is still queued. If the operation is in progress, aiocancel(3AIO) fails.

aiowait(3AIO)

A call to aiowait(3AIO) blocks the calling process until at least one outstanding asynchronous I/O operation is completed. The timeout parameter points to a maximum interval to wait for I/O completion. A timeout value of zero specifies that no wait is wanted. aiowait(3AIO) returns a pointer to the aio_result_t structure for the completed operation.

poll(2)

To synchronously determine the completion of an asynchronous I/O event rather than depend on a SIGIO interrupt, use poll(2). You can also poll to determine the origin of a SIGIO interrupt.

Use of poll(2) for very large numbers of files is slow. This problem is resolved by poll(7D).

poll(7D)

/dev/poll provides a highly scalable way of polling a large number of file descriptors. This is provided through a new set of APIs and a new driver, /dev/poll. The /dev/poll API is an alternative to, not a replacement of poll(2). poll(7D) provides details and examples of the /dev/poll API. When used properly, the /dev/poll API scales much better than poll(2). It is especially suited for applications that satisfy the following criteria:

- Applications repeatedly poll a large number of file descriptors.
- The polled file descriptors are relatively stable, that is they are not constantly closed and reopened.
- The set of file descriptors which actually have polled events pending is small, comparing to the total number of file descriptors being polled.

close(2)

Files are closed by calling close(2). close(2) cancels any outstanding asynchronous I/O request that can be. close(2) waits for an operation that cannot be cancelled (see " aiocancel(3AIO)" on page 100). When close(2) returns, there is no asynchronous I/O pending for the file descriptor. Only asynchronous I/O requests queued to the specified file descriptor are cancelled when a file is closed. Any I/O pending requests for other file descriptors are not cancelled.

Synchronized I/O

Applications might need to guarantee that information has been written to stable storage, or that file updates are performed in a particular order. Synchronized I/O provides for these needs.

Modes of Synchronization

Under SunOS 5.0 through 5.8, for a write operation, data is successfully transferred to a file when the system ensures that all written data is readable after any subsequent open of the file in the absence of a failure of the physical storage medium (even one that follows a system or power failure). For a read operation data is successfully transferred when an image of the data on the physical storage medium is available to the requesting process. An I/O operation is complete when either the associated data has been successfully transferred or the operation has been diagnosed as unsuccessful.

An I/O operation has reached synchronized I/O data integrity completion when:

For reads, the operation has been completed or diagnosed unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests are successfully transferred prior to reading the data.

For writes, the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred, and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (access time, modification time, status change time) are not transferred prior to returning to the calling process.

Synchronized I/O file integrity completion is identical to synchronized I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including access time, modification time, status change time) must be successfully transferred prior to returning to the calling process.

Synchronizing a File

The `fsync`(3C) and `fdatasync`(3RT) functions explicitly synchronize a file to secondary storage.

`fsync`(3C) guarantees the function is synchronized at the I/O file integrity completion level, while `fdatasync`(3RT) guarantees the function is synchronized at the I/O data integrity completion level.

Applications can synchronize each I/O operation before the operation completes. Setting the `O_DSYNC` flag on the file description by `open`(2) or `fcntl`(2) ensures that all I/O writes (`write`(2) and `aiowrite`(3AIO)) have reached I/O data completion before the operation is indicated as completed. Setting the `O_SYNC` flag on the file description ensures that all I/O writes have reached completion before the operation is indicated as completed. Setting the `O_RSYNC` flag on the file description ensures that all I/O reads `read`(2) and `aio_read`(3RT)) have reached the same level of completion as request for writes by the setting, `O_DSYNC` or `O_SYNC`, on the descriptor.

Interprocess Communication

This section describes the interprocess communication (IPC) functions of SunOS 5.0 through 5.8 as they relate to real-time processing. Signals, pipes, FIFOs (named pipes), message queues, shared memory, file mapping, and semaphores are described here. For more information about the libraries, functions, and routines useful for interprocess communication, see Chapter 7.

Overview

Real-time processing often requires fast, high-bandwidth interprocess communication. The choice of which mechanisms should be used can be dictated by functional requirements, and the relative performance will depend upon application behavior.

The traditional method of interprocess communication in UNIX is the pipe. Unfortunately, pipes can have framing problems. Messages can become intermingled by multiple writers or torn apart by multiple readers.

IPC messages mimic the reading and writing of files. They are easier to use than pipes when more than two processes must communicate by using a single medium.

The IPC shared semaphore facility provides process synchronization. Shared memory is the fastest form of interprocess communication. The main advantage of shared memory is that the copying of message data is eliminated. The usual mechanism for synchronizing shared memory access is semaphores.

Signals

Signals can be used to send a small amount of information between processes. The sender can use `sigqueue`(3RT) to send a signal together with a small amount of information to a target process.

The target process must have the `SA_SIGINFO` bit set for the specified signal (see `sigaction`(2)), for subsequent occurrences of a pending signal to be queued also.

The target process can receive signals either synchronously or asynchronously. Blocking a signal (see `sigprocmask`(2)) and calling either `sigwaitinfo`(3RT) or `sigtimedwait`(3RT), causes the signal to be received synchronously, with the value sent by the caller of `sigqueue`(3RT) stored in the `si_value` member of the `siginfo_t` argument. Leaving the signal unblocked causes the signal to be delivered to the signal handler specified by `sigaction`(2), with the value appearing in the `si_value` of the `siginfo_t` argument to the handler.

Only a fixed number of signals with associated values can be sent by a process and remain undelivered. Storage for {`SIGQUEUE_MAX`} signals is allocated at the first call to `sigqueue`(3RT). Thereafter, a call to `sigqueue`(3RT) either successfully enqueues at the target process or fails within a bounded amount of time.

Pipes

Pipes provide one-way communication between processes. Processes must have a common ancestor in order to communicate with pipes. Data passed through a pipe is treated as a conventional UNIX byte stream. See “Pipes” on page 65for more information about pipes.

Named Pipes

SunOS 5.0 through 5.8 provides named pipes or FIFOs. The FIFO is more flexible than the pipe because it is a named entity in a directory. Once created, a FIFO can be opened by any process that has legitimate access to it. Processes do not have to share a parent and there is no need for a parent to initiate the pipe and pass it to the descendants. See "Named Pipes" on page 67for more information.

Message Queues

Message queues provide another means of communicating between processes that also allows any number of processes to send and receive from a single message queue. Messages are passed as blocks of arbitrary size, not as byte streams. Message queues are provided in both System V and POSIX versions. See "System V Messages" on page 71 and "POSIX Messages" on page 68 for more information.

Semaphores

The semaphore is a mechanism to synchronize access to shared resources. Semaphores are also provided in both System V and POSIX styles. The System V semaphores are very flexible and very heavy weight. The POSIX semaphores are quite light weight. See "System V Semaphores" on page 73 and "POSIX Semaphores" on page 69for more information.

Note that using semaphores can cause priority inversions unless these are explicitly avoided by the techniques mentioned earlier in this chapter.

Shared Memory

The fastest way for processes to communicate is directly, through a shared segment of memory. A common memory area is added to the address space of sharing processes. Applications use stores to send data and fetches to receive communicated data. SunOS 5.0 through 5.8 provides three mechanisms for shared memory: memory mapped files, described in "Memory Management Interfaces" on page 61, System V IPC shared memory, and POSIX shared memory.

The major difficulty with shared memory is that results can be wrong when more than two processes are trying to read and write in it at the same time. See "Shared Memory Synchronization" on page 105for more information.

Memory Mapped Files

The mmap(2) interface connects a shared memory segment to the caller's address space. The caller specifies the shared segment by address and length. The caller must also specify access protection flags and how the mapped pages are managed. mmap(2) can also be used to map a file or a segment of a file to a process's memory. This technique is very convenient in some applications, but it is easy to forget that any store to the mapped file segment results in implicit I/O. This can make an otherwise bounded process have unpredictable response times. msync(3C) forces immediate or eventual copies of the specified memory segment to its permanent storage location(s). See "Memory Management Interfaces" on page 61for more information.

Fileless Memory Mapping

The zero special file, /dev/zero(4S), can be used to create an unnamed, zero initialized memory object. The length of the memory object is the least number of pages that contain the mapping. The object can be shared only by descendants of a common ancestor process.

System V IPC Shared Memory

A shmget(2) call can be used to create a shared memory segment or to obtain an existing shared memory segment. shmget(2) returns an identifier that is analogous to a file identifier. A call to shmat(2) makes the shared memory segment a virtual segment of the process memory much like mmap(2). See "System V Shared Memory" on page 77.

POSIX Shared Memory

POSIX shared memory is a variation of System V shared memory and provides similar capabilities with some minor variations. See "POSIX Shared Memory" on page 69for more information.

Shared Memory Synchronization

In sharing memory, a portion of memory is mapped into the address space of one or more processes. No method of coordinating access is automatically provided, so nothing prevents two processes from writing to the shared memory at the same time in the same place. So, it is typically used with semaphores or another mechanism used to synchronize processes. System V and POSIX semaphores both can be used for this purpose. Mutual exclusion locks, reader/writer locks, semaphores, and conditional variables provided in the multithread library can also be used for this purpose.

Choice of IPC and Synchronization Mechanisms

Applications can have specific functional requirements that determine which IPC mechanism to use. If one of several mechanisms can be used, the application writer determines which mechanism performs best for the application. The SunOS 5.0 through 5.8 interprocess communication facilities are sensitive to application behavior. Determine which mechanism provides the best response capabilities by measuring the throughput capacity of each mechanism for the particular combination of message sizes used in the application.

Asynchronous Networking

This section introduces asynchronous network communication, using sockets or Transport-Level Interface (TLI) for real-time applications. Asynchronous networking with sockets is done by setting an open socket, of type SOCK_STREAM, to asynchronous and non blocking (see "Asynchronous Socket I/O in "Advanced Topics" in *Network Interface Guide*). Asynchronous network processing of TLI events is supported using a combination of STREAMS asynchronous features and the non-blocking mode of the TLI library routines (see "Asynchronous Networking" in *Network Interface Guide*).

For more information on the Transport-Level Interface, see "Socket Interfaces" in *Network Interface Guide*.

Modes of Networking

Both sockets and Transport-Level Interface provide two modes of service: *connection-mode* and *connectionless-mode*.

Connection-Mode Service

Connection-mode service is circuit-oriented and enables the transmission of data over an established connection in a reliable, sequenced manner. It also provides an identification procedure that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless-Mode Service

Connectionless-mode service is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. All information required to deliver a unit of data, including the destination address, is passed by the sender to the transport provider, together with the data, in a single service request. Connectionless-mode service is attractive for applications that involve short-term request/response interactions and do not require guaranteed, in-sequence delivery of data. It is generally assumed that connectionless transports are unreliable.

Timers

This section describes the timing facilities available for real-time applications under SunOS 5.0 through 5.8. Real-time applications that use these mechanisms require detailed information from the manual pages of the routines listed in this section.

The timing functions of SunOS 5.0 through 5.8 fall into two separate areas of functionality: *timestamps* and *interval* timers. The timestamp functions provide a measure of elapsed time and allow the application to measure the duration of a state or the time between events. Interval timers allow an application to wake up at specified times and to schedule activities based on the passage of time. Although an application can poll a timestamp function to schedule itself, such an application would monopolize the processor to the detriment of other system functions.

Timestamp Functions

Two functions provide timestamps. The `gettimeofday`(3C) function provides the current time in a *timeval* structure, representing the time in seconds and microseconds since midnight, Greenwich Mean Time, on January 1, 1970. The `clock_gettime(3R)` function, with a clockid of `CLOCK_REALTIME`, provides the current time in a timespec structure, representing in seconds and nanoseconds the same time interval returned by `gettimeofday`(3C).

SunOS 5.0 through 5.8 uses a hardware periodic timer. For some workstations, this is the sole timing information, and the accuracy of timestamps is limited to the resolution of that periodic timer. For other platforms, a timer register with a resolution of one microsecond allows SunOS 5.0 through 5.8 to provide timestamps accurate to one microsecond.

Interval Timer Functions

Real-time applications often schedule actions using interval timers. Interval timers can be either of two types: a *one-shot* type or a *periodic* type.

A one-shot is an armed timer that is set to an expiration time relative to either current time or an absolute time. The timer expires once and is disarmed. Such a timer is useful for clearing buffers after the data has been transferred to storage, or to time-out an operation.

A periodic timer is armed with an initial expiration time (either absolute or relative) and a repetition interval. Each time the interval timer expires it is reloaded with the repetition interval and rearmed. This timer is useful for data logging or for servo-control. In calls to interval timer functions, time values smaller than the resolution of the system hardware periodic timer are rounded up to the next multiple of the hardware timer interval (typically 10 ms).

There are two sets of timer interfaces in SunOS 5.0 through 5.8. The `setitimer`(2) and `getitimer`(2) interfaces operate fixed set timers, called the BSD timers, using the timeval structure to specify time intervals. The POSIX timers, `timer_create`(3RT), operate the POSIX clock, `CLOCK_REALTIME`. POSIX timer operations are expressed in terms of the timespec structure.

The functions `getitimer`(2) and `setitimer`(2) retrieve and establish, respectively, the value of the specified BSD interval timer. There are three BSD interval timers available to a process, including a real-time timer designated `ITIMER_REAL`. If a BSD timer is armed and allowed to expire, the system sends a signal appropriate to the timer to the process that set the timer.

`timer_create`(3RT) can create up to `TIMER_MAX` POSIX timers. The caller can specify what signal and what associated value are sent to the process when the timer expires. `timer_settime`(3RT) and `timer_gettime`(3RT) retrieve and establish respectively the value of the specified POSIX interval timer. Expirations of POSIX timers while the required signal is pending delivery are counted, and `timer_getoverrun`(3RT) retrieves the count of such expirations. `timer_delete`(3RT) deallocates a POSIX timer.

Code Example 8–1 illustrates how to use `setitimer`(2) to generate a periodic interrupt, and how to control the arrival of timer interrupts.

CODE EXAMPLE 8–1 Controlling Timer Interrupts

```
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>

#define TIMERCNT 8

void timerhandler();
int  timercnt;
struct  timeval alarmtimes[TIMERCNT];

main()
{
 struct itimerval times;
 sigset_t sigset;
 int  i, ret;
 struct sigaction act;
 siginfo_t si;

 /* block SIGALRM */
 sigemptyset (&sigset);
 sigaddset (&sigset, SIGALRM);
 sigprocmask (SIG_BLOCK, &sigset, NULL);

 /* set up handler for SIGALRM */
 act.sa_action = timerhandler;
 sigemptyset (&act.sa_mask);
 act.sa_flags = SA_SIGINFO;
 sigaction (SIGALRM, &act, NULL);
 /*
  * set up interval timer, starting in three seconds,
  * then every 1/3 second
  */
 times.it_value.tv_sec = 3;
 times.it_value.tv_usec = 0;
 times.it_interval.tv_sec = 0;
 times.it_interval.tv_usec = 333333;
 ret = setitimer (ITIMER_REAL, &times, NULL);
 printf ("main:setitimer ret = %d\n", ret);

 /* now wait for the alarms */
 sigemptyset (&sigset);
 timerhandler (0, si, NULL);
 while (timercnt < TIMERCNT) {
  ret = sigsuspend (&sigset);
 }
 printtimes();
}

void timerhandler (sig, siginfo, context)
 int  sig;
 siginfo_t *siginfo;
 void  *context;
{
 printf ("timerhandler:start\n");
 gettimeofday (&alarmtimes[timercnt], NULL);
 timercnt++;
 printf ("timerhandler:timercnt = %d\n", timercnt);
}

printtimes ()
{
 int i;

 for (i = 0; i < TIMERCNT; i++) {
  printf("%ld.%0l6d\n", alarmtimes[i].tv_sec,
    alarmtimes[i].tv_usec);
 }
}
```

APPENDIX **A**

Full Code Examples

The following program is equivalent to the priocntl(1) -l utility invocation. It gets and prints the range of valid priorities for the time-sharing and real-time scheduler classes

CODE EXAMPLE A–1 Use of priocntl(2) to Display Valid Priorities

```
 /*
 * Get scheduler class IDs and priority ranges.
 */

#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
main ()
{
 pcinfo_t   pcinfo;
 tsinfo_t   *tsinfop;
 rtinfo_t  * rtinfop;
 short   maxtsupri, maxrtpri;

 /* time sharing */
  (void) strcpy (pcinfo.pc_clname, "TS");
  if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
   perror ("PC_GETCID failed for time-sharing class");
   exit (1);
  }
  tsinfop = (struct tsinfo *) pcinfo.pc_clinfo;
  maxtsupri = tsinfop->ts_maxupri;
  (void) printf("Time sharing: ID %ld, priority range -%d
      through %d\n",
   pcinfo.pc_cid, maxtsupri, maxtsupri);

 /* real time */
```

```
  (void) strcpy(pcinfo.pc_clname, "RT");
  if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
   perror ("PC_GETCID failed for realtime  class");
   exit (2);
  }
 rtinfop = (struct rtinfo *) pcinfo.pc_clinfo;
 maxrtpri = rtinfop->rt_maxpri;
 (void) printf("Real time: ID %ld, priority range 0 through
%d\n",
   pcinfo.pc_cid, maxrtpri);
 return (0);
}
```

The following screen shows the output of this program, called `getcid` in this example.

```
$ getcid
Time sharing: ID 1, priority range -20 through 20
Real time: ID 2, priority range 0 through 59
```

The following example uses `PC_GETCLINFO` to get the class name of a process based on the process ID.

CODE EXAMPLE A–2 Use of `priocntl`(2) to Get a Class Name

```
/* Get scheduler class name given process ID. */
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main (argc, argv)
 int argc;
 char *argv[];
{
 pcinfo_t   pcinfo;
 id_t   pid, classID;
 id_t   getclassID();

 if ((pid = atoi(argv[1])) <= 0) {
  perror ("bad pid");
  exit (1);
 }
 if ((classID = getclassID(pid)) == -1) {
  perror ("unknown class ID");
  exit (2);
 }
 pcinfo.pc_cid = classID;
 if (priocntl (0L, 0L, PC_GETCLINFO, &pcinfo) == -1L) {
  perror ("PC_GETCLINFO failed");
  exit (3);
 }
 (void) printf("process ID %d, class %s\n", pid,
  pcinfo.pc_clname);
}

/*
 * Return scheduler class ID of process with ID pid.
 */

getclassID (pid)
 id_t pid;
{
 pcparms_t   pcparms;

 pcparms.pc_cid = PC_CLNULL;
 if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
  return (-1);
 }
 return (pcparms.pc_cid);
}
```

The following program takes a process ID as input, makes the process a real-time process with the highest valid priority minus 1, and gives it the default time slice for that priority. The program calls the `schedinfo` function to get the real-time class ID and maximum priority.

CODE EXAMPLE A–3 Use of `priocntl`(2) to Convert a Specified Process to Real-time

```
/*
 * Input arg is proc ID. Make process a realtime
 * process with highest priority minus 1.
 */
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main (argc, argv)
 int argc;
 char *argv[];
{
 pcparms_t  pcparms;
 rtparms_t  *rtparmsp;
 id_t  pid, rtID;
 id_t  schedinfo();
 short  maxrtpri;
 if ((pid = atoi(argv[1])) <= 0) {
  perror ("bad pid");
  exit (1);
 }

 /* Get highest valid RT priority. */
 if ((rtID = schedinfo ("RT", &maxrtpri)) == -1) {
  perror ("schedinfo failed for RT");
  exit (2);
 }

 /* Change proc to RT, highest prio - 1, default time slice */
 pcparms.pc_cid = rtID;
 rtparmsp = (struct rtparms *) pcparms.pc_clparms;
 rtparmsp->rt_pri = maxrtpri - 1;
 rtparmsp->rt_tqnsecs = RT_TQDEF;

 if (priocntl(P_PID, pid, PC_SETPARMS, &pcparms) == -1) {
  perror ("PC_SETPARMS failed");
  exit (3);
 }
}

/*
 * Return class ID and maximum priority.
 * Input argument name is class name.
 * Maximum priority is returned in *maxpri.
 */

id_t
schedinfo (name, maxpri)
 char *name;
 short *maxpri;
{
 pcinfo_t   info;
 tsinfo_t   *tsinfop;
 rtinfo_   *rtinfop;

 (void) strcpy(info.pc_clname, name);
 if (priocntl (0L, 0L, PC_GETCID, &info) == -1L) {
  return (-1);
 }
 if (strcmp(name, "TS") == 0) {
  tsinfop = (struct tsinfo *) info.pc_clinfo;
  *maxpri = tsinfop->ts_maxupri;
 } else if (strcmp(name, "RT") == 0) {
  rtinfop = (struct rtinfo *) info.pc_clinfo;
  *maxpri = rtinfop->rt_maxpri;
 } else {
  return (-1);
 }
 return (info.pc_cid);
}
```

Here is a situation where `priocntlset`(2) is useful: suppose a program had both real-time and time-sharing processes that ran under a single user ID. If the program wanted to change the priority of only its real-time processes without changing the time-sharing processes to real-time processes, it could do so as shown in Code Example A–4.

CODE EXAMPLE A–4 Use of `priocntlset`(2) to Change a Process Priority

```
/*
 * Change real-time priorities of this uid
 * to highest realtime priority minus 1.
 */
```

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main (argc, argv)
 int argc;
 char *argv[];
{
 procset_t    procset;
 pcparms_t    pcparms;
 struct rtparms  *rtparmsp;
 id_t    rtclassID;
 id_t    schedinfo();
 short    maxrtpri;

 /* left set: select processes with same uid as this process */
 procset.p_lidtype = P_UID;
 procset.p_lid = getuid();

 /* get info on realtime class */
 if ((rtclassID = schedinfo ("RT", &maxrtpri)) == -1) {
  perror ("schedinfo failed");
  exit (1);
 }

...
}

/*
 * Return class ID and maximum priority.
 * Input argument name is class name.
 * Maximum priority is returned in *maxpri.
 */

id_t
schedinfo (name, maxpri)
 char *name;
 short *maxpri;
{
 pcinfo_t   info;
 tsinfo_t   *tsinfop;
 rtinfo_   *rtinfop;

 (void) strcpy(info.pc_clname, name);
 if (priocntl (0L, 0L, PC_GETCID, &info) == -1L) {
  return (-1);
 }
 if (strcmp(name, "TS") == 0) {
  tsinfop = (struct tsinfo *) info.pc_clinfo;
  *maxpri = tsinfop->ts_maxupri;
 } else if (strcmp(name, "RT") == 0) {
  rtinfop = (struct rtinfo *) info.pc_clinfo;
  *maxpri = rtinfop->rt_maxpri;
 } else {
  return (-1);
 }
 return (info.pc_cid);
}
```

CODE EXAMPLE A–5 Sample Program to Illustrate msgget(2)

```
/*
 * msgget.c: Illustrate the msgget() function.
 * This is a simple exerciser of the msgget() function. It prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

extern void  exit();
extern void  perror();

main()
{
 key_t  key;  /* key to be passed to msgget() */
 int  msgflg,  /* msgflg to be passed to msgget() */
   msqid;  /* return value from msgget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%ol", &key);
 (void) fprintf(stderr, "\nExpected flags for msgflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter msgflg value: ");
 (void) scanf("%i", &msgflg);

 (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx, %#o)\n",
 key, msgflg);
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
   "msgget: msgget succeeded: msqid = %d\n", msqid);
  exit(0);
 }
}
```

CODE EXAMPLE A–6 Sample Program to Illustrate `msgctl`(2)

```
/*
 * msgctl.c:  Illustrate the msgctl() function.
 *
 * This is a simple exerciser of the msgctl() function.  It allows
 * you to perform one control operation on one message queue.  It
 * gives up immediately if any control operation fails, so be
careful
 * not to set permissions to preclude read permission; you won’t
be
 * able to reset the permissions with this code if you do.
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission
for \
     yourself, this program will fail frequently!";

main()
{
 struct msqid_ds buf;     /* queue descriptor buffer for IPC_STAT
          and IP_SET commands */
 int    cmd,  /* command to be given to msgctl() */
     msqid;  /* queue ID to be given to msgctl() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the msqid and cmd arguments for the msgctl() call. */
 (void) fprintf(stderr,
  "Please enter arguments for msgctls() as requested.");
 (void) fprintf(stderr, "\nEnter the msqid: ");
 (void) scanf("%i", &msqid);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\nEnter the value for the command: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_SET:
   /* Modify settings in the message queue control structure.
*/
   (void) fprintf(stderr, "Before IPC_SET, get current
values:");
   /* fall through to IPC_STAT processing */
  case IPC_STAT:
   /* Get a copy of the current message queue control
    * structure and show it to the user. */
   do_msgctl(msqid, IPC_STAT, &buf);
   (void) fprintf(stderr,
   "msg_perm.uid = %d\n", buf.msg_perm.uid);
   (void) fprintf(stderr,
   "msg_perm.gid = %d\n", buf.msg_perm.gid);
   (void) fprintf(stderr,
   "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
   (void) fprintf(stderr,
   "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
   (void) fprintf(stderr, "msg_perm.mode = %#o, ",
   buf.msg_perm.mode);
   (void) fprintf(stderr, "access permissions = %#o\n",
   buf.msg_perm.mode & 0777);
   (void) fprintf(stderr, "msg_cbytes = %d\n",
       buf.msg_cbytes);
   (void) fprintf(stderr, "msg_qbytes = %d\n",
       buf.msg_qbytes);
   (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
   (void) fprintf(stderr, "msg_lspid = %d\n",
       buf.msg_lspid);
   (void) fprintf(stderr, "msg_lrpid = %d\n",
       buf.msg_lrpid);
   (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
   ctime(&buf.msg_stime) : "Not Set\n");
   (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
   ctime(&buf.msg_rtime) : "Not Set\n");
   (void) fprintf(stderr, "msg_ctime = %s",
       ctime(&buf.msg_ctime));
   if (cmd == IPC_STAT)
    break;
   /*  Now continue with IPC_SET. */
   (void) fprintf(stderr, "Enter msg_perm.uid: ");
   (void) scanf ("%hi", &buf.msg_perm.uid);
   (void) fprintf(stderr, "Enter msg_perm.gid: ");
   (void) scanf("%hi", &buf.msg_perm.gid);
   (void) fprintf(stderr, "%s\n", warning_message);
   (void) fprintf(stderr, "Enter msg_perm.mode: ");
   (void) scanf("%hi", &buf.msg_perm.mode);
   (void) fprintf(stderr, "Enter msg_qbytes: ");
   (void) scanf("%hi", &buf.msg_qbytes);
   do_msgctl(msqid, IPC_SET, &buf);
   break;
  case IPC_RMID:
  default:
   /* Remove the message queue or try an unknown command. */
   do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
   break;
 }
 exit(0);
}
```

```
/*
 * Print indication of arguments being passed to msgctl(), call
 * msgctl(), and report the results. If msgctl() fails, do not
 * return; this example doesn’t deal with errors, it just reports
 * them.
 */
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds    *buf;  /* pointer to queue descriptor buffer */
int    cmd,  /* command code */
     msqid;  /* queue ID */
{
 register int rtrn;  /* hold area for return value from msgctl() */

 (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
   msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
 rtrn = msgctl(msqid, cmd, buf);
 if (rtrn == -1) {
  perror("msgctl: msgctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "msgctl: msgctl returned %d\n",
      rtrn);
 }
}
```

CODE EXAMPLE A–7 Sample Program to Illustrate `msgsnd`(2) and `msgrcv`(2)

```
/*
 * msgop.c: Illustrate the msgsnd() and msgrcv() functions.
 *
 * This is a simple exerciser of the message send and receive
 * routines. It allows the user to attempt to send and receive
 * as many messages as wanted to or from one message queue.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

static int ask();
extern void exit();
extern char *malloc();
extern void perror();

char first_on_queue[] = "- first message on queue",
 full_buf[] = "Message buffer overflow. Extra message text\
      discarded.";

main()
{
 register int  c;   /* message text input */
 int     choice; /* user’s selected operation code */
 register int  i;   /* loop control for mtext */
 int     msgflg; /* message flags for the operation */
 struct msgbuf  *msgp;  /* pointer to the message buffer */
 int     msgsz;  /* message size */
 long    msgtyp; /* desired message type */
 int     msqid,  /* message queue ID to be used */
      maxmsgsz, /* size of allocated message buffer */
      rtrn;  /* return value from msgrcv or msgsnd */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Get the message queue ID and set up the message buffer. */
 (void) fprintf(stderr, "Enter msqid: ");
 (void) scanf("%i", &msqid);
 /*
  * Note that <sys/msg.h> includes a definition of struct
msgbuf
  * with the mtext field defined as:
  *   char mtext[1];
  * therefore, this definition is only a template, not a structure
  * definition that you can use directly, unless you want only to
  * send and receive messages of 0 or 1 byte. To handle this,
  * malloc an area big enough to contain the template - the size
  * of the mtext template field + the size of the mtext field
  * wanted. Then you can use the pointer returned by malloc as a
  * struct msgbuf with an mtext field of the size you want. Note
  * also that sizeof msgp-mtext is valid even though msgp isn’t
  * pointing to anything yet. Sizeof doesn’t dereference msgp, but
  * uses its type to figure out what you are asking about.
  */
 (void) fprintf(stderr,
  "Enter the message buffer size you want:");
 (void) scanf("%i", &maxmsgsz);
 if (maxmsgsz < 0) {
  (void) fprintf(stderr, "msgop: %s\n",
    "The message buffer size must be >= 0.");
  exit(1);
 }
 msgp = (struct msgbuf *)malloc(sizeof(struct msgbuf)
    - sizeof msgp-mtext + maxmsgsz);
 if (msgp == NULL) {
  (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
    "could not allocate message buffer for", maxmsgsz);
  exit(1);
 }
 /* Loop through message operations until the user is ready to
  quit. */
 while (choice = ask()) {
  switch (choice) {
  case 1: /* msgsnd() requested: Get the arguments, make the
    call, and report the results. */
   (void) fprintf(stderr, "Valid msgsnd message %s\n",
    "types are positive integers.");
   (void) fprintf(stderr, "Enter msgp->mtype: ");
   (void) scanf("%li", &msgp->mtype);
   if (maxmsgsz) {
    /* Since you’ve been using scanf, you need the loop
       below to throw away the rest of the input on the
       line after the entered mtype before you start
```

```
     reading the mtext. */
  while ((c = getchar()) != ’\n’ && c != EOF);
  (void) fprintf(stderr, "Enter a %s:\n",
      "one line message");
  for (i = 0; ((c = getchar()) != ’\n’); i++) {
   if (i >= maxmsgsz) {
    (void) fprintf(stderr, "\n%s\n", full_buf);
    while ((c = getchar()) != ’\n’);
    break;
   }
   msgp->mtext[i] = c;
  }
  msgsz = i;
 } else
  msgsz = 0;
 (void) fprintf(stderr,"\nMeaningful msgsnd flag is:\n");
 (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
  IPC_NOWAIT);
 (void) fprintf(stderr, "Enter msgflg: ");
 (void) scanf("%i", &msgflg);
 (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
  "msgop: Calling msgsnd", msqid, msgsz, msgflg);
 (void) fprintf(stderr, "msgp->mtype = %ld\n",
     msgp->mtype);
 (void) fprintf(stderr, "msgp->mtext = \"");
 for (i = 0; i < msgsz; i++)
  (void) fputc(msgp->mtext[i], stderr);
  (void) fprintf(stderr, "\"\n");
  rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
  if (rtrn == ->1)
   perror("msgop: msgsnd failed");
  else
   (void) fprintf(stderr,
      "msgop: msgsnd returned %d\n", rtrn);
  break;
case 2: /* msgrcv() requested: Get the arguments, make the
     call, and report the results. */
 for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
    (void) scanf("%i", &msgsz))
  (void) fprintf(stderr, "%s (0 <= msgsz <= %d): ",
      "Enter msgsz", maxmsgsz);
 (void) fprintf(stderr, "msgtyp meanings:\n");
 (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
 (void) fprintf(stderr, "\t>0 %s of given type\n",
  first_on_queue);
 (void) fprintf(stderr, "\t<0 %s with type <= |msgtyp|\n",
     first_on_queue);
 (void) fprintf(stderr, "Enter msgtyp: ");
 (void) scanf("%li", &msgtyp);
 (void) fprintf(stderr,
     "Meaningful msgrcv flags are:\n");
 (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
     MSG_NOERROR);
 (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
     IPC_NOWAIT);
 (void) fprintf(stderr, "Enter msgflg: ");
 (void) scanf("%i", &msgflg);
 (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
     "msgop: Calling msgrcv", msqid, msgsz,
     msgtyp, msgflg);
   rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
   if (rtrn == -1)
    perror("msgop: msgrcv failed");
   else {
    (void) fprintf(stderr, "msgop: %s %d\n",
        "msgrcv returned", rtrn);
    (void) fprintf(stderr, "msgp->mtype = %ld\n",
        msgp->mtype);
    (void) fprintf(stderr, "msgp->mtext is: \"");
    for (i = 0; i < rtrn; i++)
     (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
   }
   break;
  default:
   (void) fprintf(stderr, "msgop: operation unknown\n");
   break;
  }
 }
 exit(0);
}

/*
 * Ask the user what to do next. Return the user's choice code.
 * Don't return until the user selects a valid choice.
 */
static
ask()
{
 int response; /* User's response. */

 do {
  (void) fprintf(stderr, "Your options are:\n");
  (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
  (void) fprintf(stderr, "\tmsgsnd =\t1\n");
  (void) fprintf(stderr, "\tmsgrcv =\t2\n");
  (void) fprintf(stderr, "Enter your choice: ");

  /* Preset response so "^D" will be interpreted as exit. */
  response = 0;
  (void) scanf("%i", &response);
 } while (response < 0 || response > 2);

 return(response);
}
```

CODE EXAMPLE A–8 Sample Program to Illustrate `semget`(2)

```
/*
 * semget.c: Illustrate the semget() function.
 *
 * This is a simple exerciser of the semget() function. It prompts
 * for the arguments, makes the call, and reports the results.
*/
```

```
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to pass to semget() */
 int  semflg;   /* semflg to pass to semget() */
 int  nsems;   /* nsems to pass to semget() */
 int  semid;   /* return value from semget() */

 (void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 (void) fprintf(stderr, "Enter nsems value: ");
 (void) scanf("%i", &nsems);
 (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter semflg value: ");
 (void) scanf("%i", &semflg);
 (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %
     %#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
  perror("semget: semget failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "semget: semget succeeded: semid = %d\n",
   semid);
  exit(0);
 }
}
```

CODE EXAMPLE A–9 Sample Program to Illustrate `semctl`(2)

```
/*
 * semctl.c:   Illustrate the semctl() function.
 *
 * This is a simple exerciser of the semctl() function. It lets you
 * perform one control operation on one semaphore set. It gives up
 * immediately if any control operation fails, so be careful not to
 * set permissions to preclude read permission; you won’t be able to
 * reset the permissions with this code if you do.
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <time.h>

struct semid_ds semid_ds;

static void  do_semctl();
static void  do_stat();
extern char  *malloc();
extern void   exit();
extern void   perror();

char    warning_message[] = "If you remove read permission\
    for yourself, this program will fail frequently!";

main()
{
 union semun arg;   /* union to pass to semctl() */
 int    cmd,   /* command to give to semctl() */
     i,    /* work area */
     semid,   /* semid to pass to semctl() */
     semnum;  /* semnum to pass to semctl() */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "Enter semid value: ");
 (void) scanf("%i", &semid);

 (void) fprintf(stderr, "Valid semctl cmd values are:\n");
 (void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
 (void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
 (void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
 (void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
 (void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
 (void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
 (void) fprintf(stderr, "\nEnter cmd: ");
 (void) scanf("%i", &cmd);

 /* Do some setup operations needed by multiple commands. */
 switch (cmd) {
  case GETVAL:
  case SETVAL:
  case GETNCNT:
  case GETZCNT:
   /* Get the semaphore number for these commands. */
   (void) fprintf(stderr, "\nEnter semnum value: ");
   (void) scanf("%i", &semnum);
   break;
  case GETALL:
  case SETALL:
   /* Allocate a buffer for the semaphore values. */
   (void) fprintf(stderr,
    "Get number of semaphores in the set.\n");
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   if (arg.array =
    (ushort *)malloc((unsigned)
     (semid_ds.sem_nsems * sizeof(ushort)))) {
    /* Break out if you got what you needed. */
    break;
   }
   (void) fprintf(stderr,
    "semctl: unable to allocate space for %d values\n",
    semid_ds.sem_nsems);
   exit(2);
 }

 /* Get the rest of the arguments needed for the specified
    command. */
 switch (cmd) {
  case SETVAL:
   /* Set value of one semaphore. */
   (void) fprintf(stderr, "\nEnter semaphore value: ");
   (void) scanf("%i", &arg.val);
   do_semctl(semid, semnum, SETVAL, arg);
   /* Fall through to verify the result. */
   (void) fprintf(stderr,
    "Do semctl GETVAL command to verify results.\n");
  case GETVAL:
   /* Get value of one semaphore. */
   arg.val = 0;
   do_semctl(semid, semnum, GETVAL, arg);
   break;
  case GETPID:
   /* Get PID of last process to successfully complete a
      semctl(SETVAL), semctl(SETALL), or semop() on the
      semaphore. */
   arg.val = 0;
   do_semctl(semid, 0, GETPID, arg);
   break;
  case GETNCNT:
   /* Get number of processes waiting for semaphore value to
      increase. */
   arg.val = 0;
   do_semctl(semid, semnum, GETNCNT, arg);
   break;
  case GETZCNT:
   /* Get number of processes waiting for semaphore value to
      become zero. */
   arg.val = 0;
   do_semctl(semid, semnum, GETZCNT, arg);
   break;
  case SETALL:
   /* Set the values of all semaphores in the set. */
   (void) fprintf(stderr,
       "There are %d semaphores in the set.\n",
       semid_ds.sem_nsems);
   (void) fprintf(stderr, "Enter semaphore values:\n");
   for (i = 0; i < semid_ds.sem_nsems; i++) {
    (void) fprintf(stderr, "Semaphore %d: ", i);
    (void) scanf("%hi", &arg.array[i]);
   }
   do_semctl(semid, 0, SETALL, arg);
   /* Fall through to verify the results. */
   (void) fprintf(stderr,
    "Do semctl GETALL command to verify results.\n");
  case GETALL:
   /* Get and print the values of all semaphores in the
      set.*/
   do_semctl(semid, 0, GETALL, arg);
   (void) fprintf(stderr,
       "The values of the %d semaphores are:\n",
       semid_ds.sem_nsems);
   for (i = 0; i < semid_ds.sem_nsems; i++)
    (void) fprintf(stderr, "%d ", arg.array[i]);
   (void) fprintf(stderr, "\n");
   break;
  case IPC_SET:
   /* Modify mode and/or ownership. */
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   (void) fprintf(stderr, "Status before IPC_SET:\n");
   do_stat();
   (void) fprintf(stderr, "Enter sem_perm.uid value: ");
   (void) scanf("%hi", &semid_ds.sem_perm.uid);
   (void) fprintf(stderr, "Enter sem_perm.gid value: ");
   (void) scanf("%hi", &semid_ds.sem_perm.gid);
   (void) fprintf(stderr, "%s\n", warning_message);
   (void) fprintf(stderr, "Enter sem_perm.mode value: ");
   (void) scanf("%hi", &semid_ds.sem_perm.mode);
   do_semctl(semid, 0, IPC_SET, arg);
   /* Fall through to verify changes. */
   (void) fprintf(stderr, "Status after IPC_SET:\n");
  case IPC_STAT:
   /* Get and print current status. */
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   do_stat();
   break;
  case IPC_RMID:
   /* Remove the semaphore set. */
   arg.val = 0;
   do_semctl(semid, 0, IPC_RMID, arg);
   break;
  default:
   /* Pass unknown command to semctl. */
   arg.val = 0;
   do_semctl(semid, 0, cmd, arg);
   break;
 }
 exit(0);
}

/*
 * Print indication of arguments being passed to semctl(), call
 * semctl(), and report the results. If semctl() fails, do not
 * return; this example doesn’t deal with errors, it just reports
 * them.
 */
static void
do_semctl(semid, semnum, cmd, arg)
union semun  arg;
int  cmd,
  semid,
  semnum;
{
 register int      i;   /* work area */

 void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d,",
    semid, semnum, cmd);
 switch (cmd) {
  case GETALL:
   (void) fprintf(stderr, "arg.array = %#x)\n",
       arg.array);
   break;
  case IPC_STAT:
  case IPC_SET:
   (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
   break;
  case SETALL:
   (void) fprintf(stderr, "arg.array = [", arg.buf);
   for (i = 0;i < semid_ds.sem_nsems;) {
    (void) fprintf(stderr, "%d", arg.array[i++]);
    if (i < semid_ds.sem_nsems)
      (void) fprintf(stderr, ", ");
   }
   (void) fprintf(stderr, "])\n");
   break;
  case SETVAL:
  default:
   (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
   break;
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
 }
 (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}

/*
 * Display contents of commonly used pieces of the status
structure.
 */
static void
do_stat()
{
 (void) fprintf(stderr, "sem_perm.uid = %d\n",
       semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "sem_perm.gid = %d\n",
       semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "sem_perm.cuid = %d\n",
       semid_ds.sem_perm.cuid);
 (void) fprintf(stderr, "sem_perm.cgid = %d\n",
       semid_ds.sem_perm.cgid);
 (void) fprintf(stderr, "sem_perm.mode = %#o, ",
       semid_ds.sem_perm.mode);
 (void) fprintf(stderr, "access permissions = %#o\n",
       semid_ds.sem_perm.mode & 0777);
 (void) fprintf(stderr, "sem_nsems = %d\n",
semid_ds.sem_nsems);
 (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
      ctime(&semid_ds.sem_otime) : "Not Set\n");
 (void) fprintf(stderr, "sem_ctime = %s",
       ctime(&semid_ds.sem_ctime));
}
```

CODE EXAMPLE A–10 Sample Program to Illustrate semop(2)

```
/*
 * semop.c: Illustrate the semop() function.
 *
 * This is a simple exerciser of the semop() function. It lets you
 * to set up arguments for semop() and make the call. It then reports
 * the results repeatedly on one semaphore set. You must have read
 * permission on the semaphore set or this exerciser will fail. (It
 * needs read permission to get the number of semaphores in the set
 * and to report the values before and after calls to semop().)
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

static int      ask();
extern void     exit();
extern void     free();
extern char     *malloc();
extern void     perror();

static struct semid_ds  semid_ds;            /* status of semaphore set */

static char    error_mesg1[] = "semop: Can’t allocate space for %d\
        semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can’t allocate space for %d\
        sembuf structures. Giving up.\n";

main()
{
 register int    i;   /* work area */
 int    nsops;   /* number of operations to do */
 int    semid;   /* semid of semaphore set */
 struct sembuf    *sops;   /* ptr to operations to perform */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Loop until the invoker doesn’t want to do anymore. */
 while (nsops = ask(&semid, &sops)) {
  /* Initialize the array of operations to be performed.*/
  for (i = 0; i < nsops; i++) {
   (void) fprintf(stderr,
     "\nEnter values for operation %d of %d.\n",
       i + 1, nsops);
   (void) fprintf(stderr,
     "sem_num(valid values are 0 <= sem_num < %d): ",
     semid_ds.sem_nsems);
   (void) scanf("%hi", &sops[i].sem_num);
   (void) fprintf(stderr, "sem_op: ");
   (void) scanf("%hi", &sops[i].sem_op);
   (void) fprintf(stderr,
     "Expected flags in sem_flg are:\n");
   (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
     IPC_NOWAIT);
   (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
     SEM_UNDO);
   (void) fprintf(stderr, "sem_flg: ");
   (void) scanf("%hi", &sops[i].sem_flg);
  }

  /* Recap the call to be made. */
  (void) fprintf(stderr,
     "\nsemop: Calling semop(%d, &sops, %d) with:",
     semid, nsops);
  for (i = 0; i < nsops; i++)
  {
   (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
      sops[i].sem_num);
   (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
   (void) fprintf(stderr, "sem_flg = %#o\n",
      sops[i].sem_flg);
  }

  /* Make the semop() call and report the results. */
  if ((i = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
  } else {
   (void) fprintf(stderr, "semop: semop returned %d\n", i);
  }
 }
}

/*
 * Ask if user wants to continue.
 *
 * On the first call:
 * Get the semid to be processed and supply it to the caller.
 * On each call:
 *  1. Print current semaphore values.
 *  2. Ask user how many operations are to be performed on the next
 *     call to semop. Allocate an array of sembuf structures
 *     sufficient for the job and set caller-supplied pointer to
that
 *     array. (The array is reused on subsequent calls if it is big
 *     enough. If it isn’t, it is freed and a larger array is
 *     allocated.)
 */
static
ask(semidp, sopsp)
int      *semidp;  /* pointer to semid (used only the first time) */
struct sembuf   **sopsp;
{
 static union semun arg;   /* argument to semctl */
 int      i;    /* work area */
 static int   nsops = 0; /* size of currently allocated sembuf array */
 static int   semid = -1; /* semid supplied by user */
 static struct sembuf *sops;   /* pointer to allocated array */

 if (semid < 0) {
  /* First call; get semid from user and the current state of
     the semaphore set. */
  (void) fprintf(stderr,
    "Enter semid of the semaphore set you want to use: ");
  (void) scanf("%i", &semid);
  *semidp = semid;
  arg.buf = &semid_ds;
  if (semctl(semid, 0, IPC_STAT, arg) == -1) {
   perror("semop: semctl(IPC_STAT) failed");
   /* Note that if semctl fails, semid_ds remains filled
      with zeros, so later test for number of semaphores will
      be zero. */
   (void) fprintf(stderr,
     "Before and after values are not printed.\n");
  } else {
   if ((arg.array = (ushort *)malloc(
    (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
       == NULL) {
    (void) fprintf(stderr, error_mesg1,
       semid_ds.sem_nsems);
    exit(1);
   }
  }
 }
 /* Print current semaphore values. */
 if (semid_ds.sem_nsems) {
  (void) fprintf(stderr,
      "There are %d semaphores in the set.\n",
      semid_ds.sem_nsems);
  if (semctl(semid, 0, GETALL, arg) == -1) {
   perror("semop: semctl(GETALL) failed");
  } else {
   (void) fprintf(stderr, "Current semaphore values are:");
   for (i = 0; i < semid_ds.sem_nsems;
    (void) fprintf(stderr, " %d", arg.array[i++]));
   (void) fprintf(stderr, "\n");
  }
 }
 /* Find out how many operations are going to be done in the
next
    call and allocate enough space to do it. */
 (void) fprintf(stderr,
     "How many semaphore operations do you want %s\n",
     "on the next call to semop()?");
 (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
 i = 0;
 if (scanf("%i", &i) == EOF || i == 0)
```

```
  exit(0);
 if (i > nsops) {
  if (nsops)
   free((char *)sops);
  nsops = i;
  if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
   sizeof(struct sembuf)))) == NULL) {
   (void) fprintf(stderr, error_mesg2, nsops);
   exit(2);
  }
 }
 *sopsp = sops;
 return (i);
}
```

CODE EXAMPLE A–11 Sample Program to Illustrate `shmget`(2)

```
/*
 * shmget.c: Illustrate the shmget() function.
 *
 * This is a simple exerciser of the shmget() function. It
prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

extern void    exit();
extern void    perror();

main()
{
 key_t key;   /* key to be passed to shmget() */
 int  shmflg;  /* shmflg to be passed to shmget() */
 int  shmid;   /* return value from shmget() */
 int  size;   /* size to be passed to shmget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the key. */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* Get the size of the segment. */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);
 /* Get the shmflg value. */
 (void) fprintf(stderr,
    "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* Make the call and report the results. */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
  perror("shmget: shmget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
     "shmget: shmget returned %d\n", shmid);
  exit(0);
 }
}
```

CODE EXAMPLE A–12 Sample Program to Illustrate `shmctl`(2)

```
/*
 * shmctl.c: Illustrate the shmctl() function.
 *
 * This is a simple exerciser of the shmctl() function. It lets you
 * to perform one control operation on one shared memory segment.
 * (Some operations are done for the user whether requested or
not.
 * It gives up immediately if any control operation fails. Be
careful
 * not to set permissions to preclude read permission; you won’t
be
 *able to reset the permissions with this code if you do.)
*/

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>
#include    <time.h>

static void   do_shmctl();
extern void   exit();
extern void   perror();

main()
```

```
{
 int     cmd;  /* command code for shmctl() */
 int     shmid;  /* segment ID */
 struct shmid_ds shmid_ds; /* shared memory data structure to
          hold results */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
     "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get shmid and cmd. */
 (void) fprintf(stderr,
     "Enter the shmid for the desired segment: ");
 (void) scanf("%i", &shmid);
 (void) fprintf(stderr, "Valid shmctl cmd values are:\n");
 (void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
 (void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
 (void) fprintf(stderr, "Enter the desired cmd value: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_STAT:
   /* Get shared memory segment status. */
   break;
  case IPC_SET:
   /* Set owner UID and GID and permissions. */
   /* Get and print current values. */
   do_shmctl(shmid, IPC_STAT, &shmid_ds);
   /* Set UID, GID, and permissions to be loaded. */
   (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.uid);
   (void) fprintf(stderr, "Enter shm_perm.gid: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.gid);
   (void) fprintf(stderr,
    "Note: Keep read permission for yourself.\n");
   (void) fprintf(stderr, "Enter shm_perm.mode: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.mode);
   break;
  case IPC_RMID:
   /* Remove the segment when the last attach point is
      detached. */
   break;
  case SHM_LOCK:
   /* Lock the shared memory segment. */
   break;
  case SHM_UNLOCK:
   /* Unlock the shared memory segment. */
   break;
  default:
   /* Unknown command will be passed to shmctl. */
   break;
 }
 do_shmctl(shmid, cmd, &shmid_ds);
 exit(0);
}

/*
 * Display the arguments being passed to shmctl(), call shmctl(),
 * and report the results. If shmctl() fails, do not return; this
 * example doesn’t deal with errors, it just reports them.
 */
static void
do_shmctl(shmid, cmd, buf)
int    shmid,   /* attach point */
     cmd;   /* command code */
struct shmid_ds *buf;   /* pointer to shared memory data structure */
{
 register int rtrn;   /* hold area */

 (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d, buf)\n",
  shmid, cmd);
 if (cmd == IPC_SET) {
  (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
     buf->shm_perm.uid);
  (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
     buf->shm_perm.gid);
  (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
     buf->shm_perm.mode);
 }
 if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
  perror("shmctl: shmctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
      "shmctl: shmctl returned %d\n", rtrn);
 }
 if (cmd != IPC_STAT && cmd != IPC_SET)
  return;

 /* Print the current status. */
 (void) fprintf(stderr, "\nCurrent status:\n");
 (void) fprintf(stderr, "\tshm_perm.uid = %d\n",
      buf->shm_perm.uid);
 (void) fprintf(stderr, "\tshm_perm.gid = %d\n",
      buf->shm_perm.gid);
 (void) fprintf(stderr, "\tshm_perm.cuid = %d\n",
      buf->shm_perm.cuid);
 (void) fprintf(stderr, "\tshm_perm.cgid = %d\n",
      buf->shm_perm.cgid);
 (void) fprintf(stderr, "\tshm_perm.mode = %#o\n",
      buf->shm_perm.mode);
 (void) fprintf(stderr, "\tshm_perm.key = %#x\n",
      buf->shm_perm.key);
 (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
 (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
 (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
 (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
 (void) fprintf(stderr, "\tshm_atime = %s",
   buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_dtime = %s",
   buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_ctime = %s",
      ctime(&buf->shm_ctime));
}
```

CODE EXAMPLE A–13 Sample Program to Illustrate `shmat`(2) and `shmdt`(2)

```
/*
 * shmop.c: Illustrate the shmat() and shmdt() functions.
 *
 * This is a simple exerciser for the shmat() and shmdt() system
 * calls. It allows you to attach and detach segments and to
 * write strings into and read strings from attached segments.
 */

#include   <stdio.h>
#include   <setjmp.h>
#include   <signal.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

#define   MAXnap  4 /* Maximum number of concurrent attaches. */

static   ask();
static void  catcher();
extern void  exit();
static   good_addr();
extern void  perror();
extern char  *shmat();

static struct state { /* Internal record of currently attached segments. */
 int  shmid;    /* shmid of attached segment */
 char *shmaddr;   /* attach point */
 int  shmflg;   /* flags used on attach */
} ap[MAXnap];    /* State of current attached segments. */

static int  nap;  /* Number of currently attached segments. */
static jmp_buf segvbuf; /* Process state save area for SIGSEGV catching. */

main()
{
 register int   action;  /* action to be performed */
 char     *addr;   /* address work area */
 register int   i;    /* work area */
 register struct state *p;   /* ptr to current state entry */
 void     (*savefunc)(); /* SIGSEGV state hold area */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 while (action = ask()) {
  if (nap) {
   (void) fprintf(stderr,
      "\nCurrently attached segment(s):\n");
   (void) fprintf(stderr, " shmid address\n");
   (void) fprintf(stderr, "------ ----------\n");
   p = &ap[nap];
   while (p-- != ap) {
    (void) fprintf(stderr, "%6d", p->shmid);
    (void) fprintf(stderr, "%#11x", p->shmaddr);
    (void) fprintf(stderr, " Read%s\n",
     (p->shmflg & SHM_RDONLY) ?
     "-Only" : "/Write");
   }
  } else
   (void) fprintf(stderr,
    "\nNo segments are currently attached.\n");
  switch (action) {
  case 1:   /* Shmat requested. */
   /* Verify that there is space for another attach. */
   if (nap == MAXnap) {
    (void) fprintf(stderr, "%s %d %s\n",
       "This simple example will only allow",
       MAXnap, "attached segments.");
    break;
   }
   p = &ap[nap++];
   /* Get the arguments, make the call, report the
    results, and update the current state array. */
   (void) fprintf(stderr,
    "Enter shmid of segment to attach: ");
   (void) scanf("%i", &p->shmid);

   (void) fprintf(stderr, "Enter shmaddr: ");
   (void) scanf("%i", &p->shmaddr);
   (void) fprintf(stderr,
    "Meaningful shmflg values are:\n");
   (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
    SHM_RDONLY);
   (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
    SHM_RND);
   (void) fprintf(stderr, "Enter shmflg value: ");
   (void) scanf("%i", &p->shmflg);

   (void) fprintf(stderr,
    "shmop: Calling shmat(%d, %#x, %#o)\n",
    p->shmid, p->shmaddr, p->shmflg);
   p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
   if(p->shmaddr == (char *)-1) {
    perror("shmop: shmat failed");
    nap--;
   } else {
    (void) fprintf(stderr,
     "shmop: shmat returned %#8.8x\n",
     p->shmaddr);
   }
   break;

  case 2:   /* Shmdt requested. */
   /* Get the address, make the call, report the results,
    and make the internal state match. */
   (void) fprintf(stderr,
    "Enter detach shmaddr: ");
   (void) scanf("%i", &addr);

   i = shmdt(addr);
   if(i == -1) {
    perror("shmop: shmdt failed");
   } else {
    (void) fprintf(stderr,
     "shmop: shmdt returned %d\n", i);
    for (p = ap, i = nap; i--; p++) {
     if (p->shmaddr == addr)
      *p = ap[--nap];
    }
   }
   break;
  case 3: /* Read from segment requested. */
   if (nap == 0)
    break;

   (void) fprintf(stderr, "Enter address of an %s",
    "attached segment: ");
   (void) scanf("%i", &addr);

   if (good_addr(addr))
    (void) fprintf(stderr, "String @ %#x is ‘%s’\n",
     addr, addr);
   break;

  case 4: /* Write to segment requested. */
   if (nap == 0)
    break;

   (void) fprintf(stderr, "Enter address of an %s",
    "attached segment: ");
   (void) scanf("%i", &addr);

   /* Set up SIGSEGV catch routine to trap attempts to
    write into a read->only attached segment. */
   savefunc = signal(SIGSEGV, catcher);

   if (setjmp(segvbuf)) {
    (void) fprintf(stderr, "shmop: %s: %s\n",
     "SIGSEGV signal caught",
     "Write aborted.");
   } else {
    if (good_addr(addr)) {
     (void) fflush(stdin);
     (void) fprintf(stderr, "%s %s %#x:\n",
      "Enter one line to be copied",
      "to shared segment attached @",
      addr);
     (void) gets(addr);
    }
   }
   (void) fflush(stdin);

   /* Restore SIGSEGV to previous condition. */
   (void) signal(SIGSEGV, savefunc);
   break;
  }
 }
 exit(0);
 /*NOTREACHED*/
}
/*
** Ask for next action.
*/
static
```

```
ask()
{
 int  response;   /* user response */
 do {
   (void) fprintf(stderr, "Your options are:\n");
   (void) fprintf(stderr, "\t^D = exit\n");
   (void) fprintf(stderr, "\t 0 = exit\n");
   (void) fprintf(stderr, "\t 1 = shmat\n");
   (void) fprintf(stderr, "\t 2 = shmdt\n");
   (void) fprintf(stderr, "\t 3 = read from segment\n");
   (void) fprintf(stderr, "\t 4 = write to segment\n");
   (void) fprintf(stderr,
    "Enter the number corresponding to your choice: ");

   /* Preset response so "^D" will be interpreted as exit. */
   response = 0;
   (void) scanf("%i", &response);
 } while (response < 0 || response > 4);
 return (response);
}
/*
** Catch signal caused by attempt to write into shared memory
segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
{
 longjmp(segvbuf, 1);
 /*NOTREACHED*/
}
/*
** Verify that given address is the address of an attached
segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char *address;
{
 register struct state          *p;   /* ptr to state of attached
segment */

 for (p = ap; p != &ap[nap]; p++)
   if (p->shmaddr == address)
    return(1);
 return(0);
}
```

The next example demonstrates how to insert an entry into a doubly linked list that is stored in a file of list element records. For the example, assume that the record after which the new record is to be inserted already has a read lock on it. The lock on this record must be changed or promoted to a write lock so that the record can be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. When processes with pending write locks are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Changing a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the lockf(3C) function does not have read locks, lock promotion does not apply to that call.

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command were used instead, the fcntl(2) functions would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error-return sections.

CODE EXAMPLE A–14 Example of Record Locking With Lock Promotion

```
 struct record {
 ...     /* data portion of record */
 off_t prev;  /* index to previous record in the list */
 off_t next;  /* index to next record in the list */
};

/* Lock promotion using fcntl(2): When this routine is entered it is
 * assumed that there are read locks on "here" and "next." If write
 * locks on "here" and "next" are obtained;
 *     Set a write lock on "this."
 *     Return index to "this" record.
 * If any write lock is not obtained;
 *     Restore read locks on "here" and "next."
 *     Remove all other locks.
 *     Return a -1.
 */

off_t
set3lock (this, here, next)
off_t this, here, next;
{
 struct flock lck;
 lck.l_type = F_WRLCK; /* setting a write lock */
 lck.l_whence = 0; /* offset l_start from beginning of file */
 lck.l_start = here;
 lck.l_len = sizeof(struct record);

 /* promote lock on "here" to write lock */
 if (fcntl(fd, F_SETLKW, &lck) < 0) {
      return (-1);
 }
 /* lock "this" with write lock */
 lck.l_start = this;
 if (fcntl(fd, F_SETLKW, &lck) < 0) {
  /* "this" lock failed; demote "here" lock to read lock. */
  lck.l_type = F_RDLCK;
  lck.l_start = here;
  (void) fcntl(fd, F_SETLKW, &lck);
  return (-1);
 /* promote lock on "next" to write lock */
 lck.l_start = next;
 if (fcntl(fd, F_SETLKW, &lck) < 0) {
  /* "next" lock failed; demote lock on "here" to read lock, */
  lck.l_type = F_RDLCK;
  lck.l_start = here;
  (void) fcntl(fd, F_SETLK, &lck);
  /* and remove lock on "this". */
  lck.l_type = F_UNLCK;
  lck.l_start = this;
  (void) fcntl(fd, F_SETLK, &lck);
  return (-1); /* cannot set lock, try again or quit */
 }

 return (this);
}
```

CODE EXAMPLE A–15 Record Write Locks With `lockf`(3C)

```
/* lockf(3C)
 * When this routine is entered, it is assumed that there are no
 * locks on "here" and "next". If locks are obtained: set a lock
 * on "this"; return index to "this" record. If any lock is not
 * obtained: remove all other locks; return a -1.
 */
#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;
{
 /* lock "here" */
 (void) lseek(fd, here, 0);
 if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
  return (-1);
 }
 /* lock "this" */
 (void) lseek(fd, this, SEEK_SET);
 if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
  /* Lock on "this" failed. Clear lock on "here". */
  (void) lseek(fd, here, 0);
  (void) lockf(fd, F_ULOCK, sizeof(struct record));
  return (-1);
 }
 /* lock "next" */
 (void) lseek(fd, next, 0);
 if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
  /* Lock on "next" failed. Clear lock on "here". */
  (void) lseek(fd, here, 0);
  (void) lockf(fd, F_ULOCK, sizeof(struct record));
  /* and remove lock on "this". */
  (void) lseek(fd, this, 0);
  (void) lockf(fd, F_ULOCK, sizeof(struct record));
  return (-1);    /* cannot set lock, try again or quit */
 }
 return (this);
}
```

APPENDIX **B**

System Interface Guide, Solaris 8 6/00 Updates

Incorporated Bug Fixes

A variety of typographical errors in text and source code examples was corrected.

Index