System Interfaces Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

Tables

Figures

Code Samples

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 2.5 Reference Manual AnswerBook*.

Knowledge of terminal use, of a UNIX system editor, and of the UNIX system directory and file structure is assumed. Read the *Solaris 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 always 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 2.x operating system.

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.

Typeface Conventions

The following conventions are used in this guide:

- Prompts and error messages from the system are printed in `listing type like this`.
- Information you type as a command or in response to prompts is shown in **`boldface listing type like this`**. Type everything shown in boldface exactly as it appears in the text.
- Parts of a command shown in *italic text like this* refer to a variable that you have to substitute from a selection. It is up to you to make the correct substitution.
- Output to the screen by the system or an application are in `currier`, inputs from the keyboard are in **`currier bold`**:

```
$pwd
/home/traveler/scotty
```

- You are expected to press the RETURN key after entering a command or menu choice, so the RETURN key is not explicitly shown in these cases. If, however, you are expected to press RETURN without typing any text, the notation is shown.
- Control characters are shown by the string "CTRL-" followed by the appropriate character, such as D (this is known as CTRL-D). To enter a control character, hold down the key marked CTRL (or CONTROL) and press the D key.

- The default prompt signs for an ordinary user and `root` are the dollar sign or percent sign ($ or %) and the number sign (#). When the # prompt is used in an example, the command illustrated can be executed only by `root`.

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 2.5 Reference Manual AnswerBook.*

For example, “See `priocntl(2)`” tells you to look at the `priocntl` page in section 2 of the *Solaris 2.5 Reference Manual AnswerBook.*

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.

Introduction to the API 1

A SunSoft goal is to define the Architectural Interfaces of Solaris. There are two reasons:

- The system interface is an effective “contract” with our customers. We tell our customers exactly what we offer them to use, and help ensure that only the official (intended) interface is used.
- We use the definition to ensure that we honor this contract. Interfaces we offer in a particular version of the product are preserved in future versions of the product. Thus, we maintain upward compatibility in subsequent releases of Solaris.

The Programming Interface

Solaris offers many kinds of “interface”, 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 we call the API, and one seen by developers of system components such as device drivers and platform support modules, which we call the SPI (system programming interface).

Each programming interface to Solaris is also “visible” to the developer at two levels, source level and binary. When we use the acronyms API and SPI, we indicate the source level programming interface to the system. We use the terms Application Binary Interface (ABI) and System Binary Interface (SBI) to indicate the binary interfaces corresponding to the respective source level programming interfaces. (Because the phrase “the ABI” can be confused with other binary interfaces, we refer to the “Solaris ABI” only by name.)

Interface Functions

The Solaris 2.x functions discussed in this manual are the interfaces between the services provided by the kernel and application programs. The functions described in Sections 2 and 3 of the *Solaris 2.5 Reference Manual AnswerBook* are an application’s interface to the Solaris 2.x 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*, *Transport Interfaces Programming Guide*, etc.

When you use the library routines described in sections 2 and 3 of the *Solaris 2.5 Reference Manual AnswerBook*, 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 resulting from the build. The source-level programming interface to the library may 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 only assured 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 SunSoft 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 may, or how to, use the interface. Definitions:.

Open specification	An interface specification that we publish, which customers can use freely (build products that use our implementation of the interface). Others are free to provide alternative implementations without licensing or legal restrictions.
Closed specification	An interface specification that we do not publish. On which we do not want customers to build products and of which we do not want others to build alternative implementations.
Compatible change	A change to an interface or its implementation has no effect on previously valid programs.
Incompatible change	A change to an interface or its implementation that makes previously valid programs invalid. This may include bug fixes or performance degradation. This does not include programs which 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, VMEbus, 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's control. We publish the specification of these interfaces and commit to remain compatible with them.

Classification

Specification:	Open
Incompatible Change:	minor release (X.Y)
Examples:	VFS interface, vm kernel interfaces, libkvm

Uncommitted interfaces are available for use by customers, but they lack the commitment that comes with a Public interface. We often publish specifications of the current versions of these interfaces in a "this is how it works in this release, but we may change it next release" form. Use these interfaces at your own risk in experiments.

Some of these interfaces are ones we would like to elevate to “Public” status, but at the moment we don’t feel confident enough in them to commit to the compatibility constraints of a Public interface. The VFS and vm interfaces are examples of this.

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 proactive 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 means of customer notice 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. It is not intended that this require specific actions to “publish” the information, such as press releases or similar forms of publicity.

Processes 2

Overview

When you execute a command, you start a process that is numbered and tracked by the operating system. A flexible feature of the operating system is that processes are generated by other processes. For example, log in to your system running a shell, then use an editor such as `vi`. Take the option of invoking the shell from `vi`. Execute the `ps` command and you will see a display resembling this (which shows the results of a `ps –f` command):

```
UID   PID    PPID   C  STIME     TTY    TIME   COMD
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
```

Here, user `abc` has four processes active. When you trace the chain shown in the process ID (PID) and parent process ID (PPID) columns, you see that the shell that was 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.

The four processes in the example are shell-level commands, but you can start new processes from your own program.

Overlooking the case where your program is interactive and contains many choices for the user, it might need to run one or more other programs based on conditions it encounters in its own processing. The reasons why it might not be practical to create one large executable include:

- 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.

With the “fork(2)” on page 9 and “exec(2)” on page 10 functions you can create a new process (copy of the creating process) and make a process start a new executable in place of the running one.

Functions

These functions are used to control user processes:

Table 2-1 Process Functions

Function Name	Purpose
`fork`	Create a new process

`exec`
`execl`
`execv`
`execle`
`execve`
`execlp`
`execvp`
	Execute a program

`exit` `_exit`	Terminate a process
`wait`	Wait for a child process to stop or terminate
`dladdr`	Translate address to symbolic information
`dlclose`	Close a shared object
`dlerror`	Get diagnostic information
`dlopen`	Open a shared object
`dlsym`	Get the address of a symbol in a shared object

Table 2-1 Process Functions (Continued)

Function Name	**Purpose**
`setuid` `setgid`	Set user and group IDs
`setpgrp`	Set process group ID
`chdir` `fchdir`	Change working directory
`chroot`	Change root directory
`nice`	Change priority of a process
`getcontext` `setcontext`	Get and set current user context
`getgroups` `setgroups`	Get or set supplementary group access list IDs

`getpid`
`getpgrp`
`getppid`
`getpgid`
	Get process, process group, and parent process IDs

`getuid`
`geteuid`
`getgid`
`getegid`
	Get real user, effective user, real group, and effective group IDs

`pause`	Suspend process until signal
`priocntl`	Control process scheduler
`setpgid`	Set process group ID
`setsid`	Set session ID
`waitid`	Wait for a child process to change state

Spawning new processes

`fork(2)`

The `fork` call creates a new process that is an exact copy of the calling process. The new process is called the child process; the creator is called the parent process. The child gets a new, unique process ID. When the `fork` function has finished successfully, it returns a 0 to the child process and the child's process ID to the parent. The returned value is how an executable determines whether it is the parent process or the child process.

Leaving out the possibility of named files, the new process created by the `fork` or `exec` function has the three standard files that are automatically opened: `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`.

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

Note – An obsolete practice is to use `fork()` and `exec()` 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()`, `dlsym()`, and `dlclose()` as described in "Runtime Linking" on page 11.

---

exec*(2)*

`exec` is the name of a family of functions that includes `execl`, `execv`, `execle`, `execve`, `execlp`, and `execvp`. They all transform the calling process into a new process, but with different ways of pulling together and presenting the arguments of the function. For example, `execl` 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 process 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.

See `exec`(2) for more details.

The key point about the exec family is that there is no return from a successful execution; the new process overlays the process that makes the exec call. The new process also takes over the process ID and other attributes of the old process. If the call to exec is unsuccessful, control is returned to your program with a return value of –1. You can check errno to learn why it failed.

```
main()
{
    pid_t   pid;

    pid = fork;
    switch (pid) {
        case -1:    /* fork failed */
            perror ("fork");
            exit (1);
        case 0:     /* in new chiild 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 */
    ...
}
```

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, help or debugging information.
- The application may choose between a number of different shared objects depending on the exact services required, for example, a networking protocol.
- Any shared objects added to the process address space during execution may be freed after use.

A typical scenario that an application may perform to access an additional shared object is:

- A shared object is located and added to the address space of a running application using dlopen(3X). Any dependencies of shared object are also located and added as 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 object(s) are relocated, and any initialization sections in the new shared object(s) are called.
- The application locates symbols in the added shared object(s) using dlsym(3X). 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 object(s) the address space is freed using dlclose(3X). Any termination sections within the shared object(s) being freed are called at this time. For example:

```
    if (dlclose (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(3X).

The services of the runtime linker are defined in the header file `dlfcn.h` and are made available to an application via 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(3X)` 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 Chapter 3 of *Linker and Libraries Guide*. See `dladdr`(3X), `dlclose`(3X), `dlerror`(3X), `dlopen`(3X), and `dlsym`(3X) for use details.

Process Scheduling

The 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 may 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 offers an alternate real-time scheduling policy as well. 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 SunOS 5.x system environments, 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)` of the *man Pages(2): System Calls*.

Error Handling

Functions that do not conclude successfully almost always return a value of –1 to your program. (For a few functions in Section 2 of the *man Pages(2): System Calls*, there are a few 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 returned –1. See error descriptions in `intro`(2) of the *man Pages(2): System Calls*.

You can use the C language function `perror`(3C) to print an error message on `stderr` based on the value of `errno`.

Signals

Overview

The system defines a set of signals that can be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is normally blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify the handler to which a signal is delivered or specify that the signal is to be blocked or ignored. A process can also specify that an action is to be taken when signals occur.

Some signals cause a process to exit when they are not caught. This can be accompanied by creation of a `core` image file, containing the current memory image of the process for use in postmortem debugging. A process can choose to have signals delivered on a particular stack, so that sophisticated software stack manipulations are possible.

Not all signals have the same priority. If multiple signals are simultaneously pending and deliverable, the signal with the smallest number will be delivered first. A signal routine usually executes concurrently with the signal that caused its invocation, but other signals can still occur. Mechanisms are provided so that critical sections of code can protect themselves against the occurrence of specified signals.

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>`.

The signal functions include:

Table 2-2 Signal Functions

Function Name	Purpose

`sigaction`
`sigset`
`sighold`
`sigrelse`
`sigignore`
	Manage signal (detailed)

`sigaltstack`	Set or get signal alternate stack context
`signal` `sigpause`	Manage signal (simplified)
`sigpending`	Examine signals that are blocked and pending
`sigprocmask`	Change or examine signal mask
`kill`	Send a signal to a process or group of processes
`sigsend` `sigsendset`	Send a signal to a process or group of processes
`sigqueue`	Send a signal with a value to a process
`sigwaitinfosigti` `sigtimedwait`	Receive a value and signal synchronously
`sigsuspend`	Install a signal mask and suspend process until signal

Hardware Signals

Hardware signals are derived from exceptional conditions that can occur during execution. Such signals include

- SIGFPE—representing floating point and other arithmetic exceptions
- SIGILL— for illegal instruction execution
- SIGSEGV—for addresses outside the currently assigned area of memory or for accesses that violate memory protection constraints
- SIGBUS—for accesses that result in hardware-related errors

Other, more CPU-specific hardware signals exist, such as SIGIOT, SIGEMT, and SIGTRAP.

Software Signals

Software signals reflect interrupts generated by user request:

- SIGINT—the normal interrupt signal
- SIGQUIT—this more powerful quit signal usually causes a core image to be generated
- SIGHUP and SIGTERM—these signals provide graceful process termination, either because a user has "hung up" or through a user or program request
- SIGKILL—a more powerful termination signal that a process cannot catch or ignore
- SIGUSR1 and SIGUSR2—allow programs to define their own asynchronous events
- SIGRTMIN through SIGRTMAX—a range of signals which allow programs to define their own events

Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

Notification Signals

A process can request notification with a SIGPOLL signal when input or output is possible on a descriptor, or when an operation finishes.

A process can request to receive a SIGURG signal when an urgent condition arises on a communication channel.

Process Control Signals

A process can be notified by a signal sent to it or to the members of its process group.

- SIGSTOP—stops the process; this powerful signal cannot be caught
- SIGTSTP—indicates that a user request stopped the process
- SIGTTIN—indicates that an input request stopped the process
- SIGTTOU—indicates that an output request stopped the process
- SIGCONT—indicates that a process continued from a stopped state
- SIGCHLD—notifies a process that a child process has changed state, either by stopping or by terminating

Resource Limit Signals

Exceeding resource limits can generate signals.

- SIGXCPU occurs when a process nears its CPU time limit
- SIGXFSZ warns that the limit on file-size creation has been reached

Signal Handlers

A process has a handler associated with each signal. The handler controls the way the signal is delivered.

Each handler specifies an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) takes place if the signal occurs. The constants SIG_IGN and SIG_DFL, used as values for sa_handler, cause ignoring or defaulting of a condition.

Note – To reset a signal handler from within a signal handler, reset the signal handler routine that catches the signal (signal(*n*, SIG_DFL);) and unblock the blocked signal with sigprocmask.

Signal Set Operations

The sa_mask field specifies the set of signals to be masked when the handler is invoked; it implicitly includes the signal that invoked the handler.

Five operations are permitted on signal sets.

- sigemptyset—empties the signal set
- sigfillset—fills the signal set with every signal currently supported
- sigaddset—adds specific signals to the set
- sigdelset—deletes specific signals from the set
- sigismember—tests set membership

Initialize signal sets with a call to sigemptyset or sigfillset.

Unique Signal Properties

The sa_flags field specifies unique properties of the signal. Such properties include:

- whether or not functions should be restarted if the signal handler returns
- whether the signal action should be reset to SIG_DFL when it is caught
- whether subsequent occurrences of a signal which is already pending should be queued
- whether the handler should operate on the normal runtime stack or on a particular signal stack.

If *osa* is nonzero, the previous signal action is returned.

Signal Generation

A process can send a signal to another process or group of processes with the calls:

```
#include <signal.h>

int
kill(pid_t pid, int sig);


#include <signal.h>

int
sigsend(idtype_t idtype, id_t id, int sig);
```

```
int
sigsendset(procset_t *psp, int sig);
```

or

```
#include <signal.h>

int
sigqueue (pid_t pid, int signo, const union sigval value);
```

Unless the process sending the signal is privileged, its real or effective user ID must be that of the receiving process's real or saved user ID.

Signals can also be sent from a terminal device to the process group or session leader associated with the terminal. See the `termio(7I)` manual page for more information.

Signal Delivery

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process then it will be delivered.

The process of signal delivery:

- Adds the signal to be delivered and those signals specified in the associated signal handler's `sa_mask` to a set of those masked for the process
- Saves the current process context
- Places the process in the context of the signal handling routine

The call is arranged so that if the signal handling routine exits normally the signal mask is restored and the process resumes execution in the original context.

Note – For the process to resume in a different context it must arrange to restore the signal mask itself.

The mask of blocked signals is independent of handlers for delays. It delays the delivery of signals in the same way that a raised hardware interrupt priority level delays hardware interrupts. Preventing an interrupt from occurring by changing the handler is like disabling a device from further interrupts.

The signal handling routine `sa_handler` is called by a C call of the form

```
#include <siginfo.h>
#include <ucontext.h>

(*sa_handler)(int signo, siginfo_t *infop, ucontext_t *ucp);
```

The `signo` field gives the number of the signal that occurred. The `infop` field is either equal to 0 or points to a structure that contains information detailing the reason the signal was generated. This information must be explicitly asked for when the signal action is specified. The `ucp` field is a pointer to a structure containing the process's context before delivery of the signal. It restores the process's context upon return from the signal handler.

To block a section of code against one or more signals, use a `sigprocmask` call to add a set of signals to the existing mask and to return the old mask:

```
#include <signal.h>

int
sigprocmask(int SIG_BLOCK,const sigset_t *mask,sigset_t *omask);
```

The old mask can then be restored later with `sigprocmask`

```
#include <signal.h>

int
sigprocmask(int SIG_UNBLOCK, const sigset_t *mask, sigset_t *omask);
```

Or, the old mask can be reset with

```
#include <signal.h>

int
sigprocmask(int SIG_SETMASK, const sigset_t *mask, sigset_t *omask);
```

The `sigprocmask` call can be used to read the current mask without changing it by specifying a null pointer as its `mask` argument.

You can check conditions with some signals blocked, and then pause to wait for a signal and restore the mask, by using

```
#include <signal.h>

int
sigsuspend(const sigset_t *mask);
```

Applications can receive signals synchronously by using

```
#include <signal.h>

int
sigwaitinfo(const sigset_t *mask, siginfo_t *siginfo);

int
sigtimedwait(const sigset_t *mask,siginfo_t *siginfo,
            const struct timespec *timeout);
```

Programs maintaining complex or fixed-size stacks can use the call

```
#include <signal.h>

int
sigaltstack(const stack_t *ss, stack_t *oss);
```

where the `stack_t` structure contains

```
int     *ss_sp
long    ss_size
int     ss_flags
```

This provides the system with a stack based at `ss_sp` of size `ss_size` for signal delivery. The system automatically adjusts for direction of stack growth. `ss_flags` indicates whether the process is currently on the signal stack and whether or not the signal stack is disabled.

When a signal is to be delivered and the process has requested that it be delivered on the alternate stack (see `sigaction` above), the system checks whether the process is on a signal stack. If it is not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

For a process to take a nonlocal exit from the signal routine, or to run code from the signal stack that uses a different stack, use a `sigaltstack` call to reset the signal stack.

Process Scheduler 3

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.

This chapter describes the process scheduler for the process model. See the *Multithreaded Programming Guide* for scheduler information under the multithreading model.This chapter is addressed to programmers who need more control over order of process execution than they get using default scheduler parameters.

The SunOS 5.x system gives users absolute 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 to provide good response time to interactive processes and good throughput to processes that use a lot of CPU time.

The SunOS 5.x system scheduler offers a real-time scheduling policy as well as a time-sharing policy. Real-time scheduling allows users to set fixed priorities on a per-process basis. 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 therefore specify the order in which processes run.

A program can also be written so that its real-time processes have a guaranteed response time from the system. See Chapter 7, "Realtime Programming and Administration" for detailed information.

For most UNIX environments, the default scheduler configuration works well and no real-time processes are needed. Administrators should not change configuration parameters and users should not change scheduler properties of their processes. However, when the requirements for a program include strict timing constraints, real-time processes sometimes provide the only way to satisfy those constraints.

Note – Real-time processes used carelessly can have a dramatically 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.

There are a few reference manual entries with information on scheduler administration:

- `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.

The rest of this chapter is organized as follows.

- The "*Overview of the Process Scheduler*" tells what the scheduler does and how it does it. It also introduces scheduler classes.
- The "*Commands and Functions*" section describes and gives examples of the `priocntl`(1) command and the `priocntl`(2) and `priocntlset`(2) functions, which are the user interfaces to scheduler services. The `priocntl` functions allow you to retrieve scheduler parameters for a process or for a set of processes.
- "*Interaction with Other Functions*" describes the interactions between the scheduler and related functions.
- The "*Performance*" section discusses scheduler latencies about which some programs must be aware.

Overview of the Process Scheduler

Figure 3-1 shows how the SunOS 5.x process scheduler works:

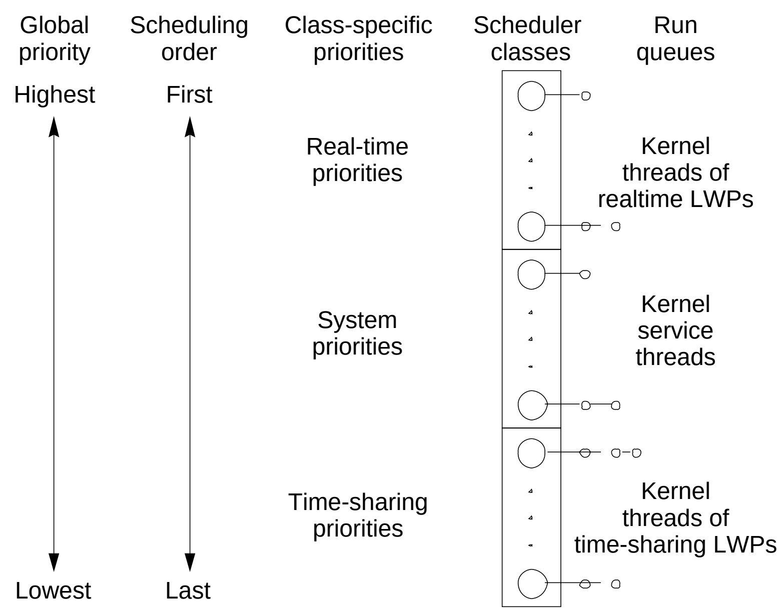


Figure 3-1 SunOS 5.x Process Scheduler

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

In the default configuration, the initialization process belongs to the time-sharing class. Because processes inherit their scheduler parameters, all user login shells begin as time-sharing processes in the default configuration.

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 uses up its time slice, sleeps, or is preempted by a higher-priority process. Processes with the same priority run round-robin.

Administrators specify default time slices in the configuration tables, but 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.

By default, 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, no kernel process and no time-sharing process run.

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 frequently enough to provide good response time, but not so frequently that it spends too much time doing the switching. Time slices are typically on the order of 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. On the other hand, 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.

The scheduler manages time-sharing processes using configurable parameters in the time-sharing parameter table `ts_dptbl`. 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 for use by 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 for those processes; 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 can run in predetermined order. Real-time priorities never change except when a user requests a change. Contrast this fixed-priority policy with the time-sharing policy, in which the system changes priorities to provide good interactive response time.

Privileged users can use the `priocntl` command or the `priocntl` function to assign real-time priorities.

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

Commands and Functions

Here is a programmer’s view of default process priorities.

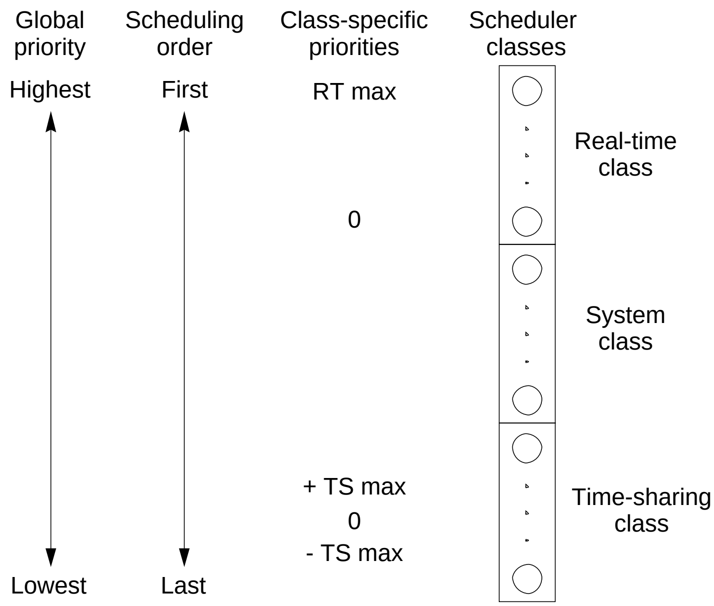


Figure 3-2 Process Priorities (Programmer's View)

From a user's or programmer's point of 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.

- Real-time priorities run from zero to a configuration-dependent maximum. The system maps them directly into global priorities. They never change except when a user changes them.
- System priorities are controlled entirely in the kernel. Users cannot affect them.
- Time-sharing priorities have a user-controlled component (the "user priority") and a component controlled by the system. The system does not change the user priority except as the result of a user request. The system changes the system-controlled component dynamically on a per-process

basis to provide good overall system performance; users cannot affect the system-controlled component. The scheduler combines these two components to get the process global priority.

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 of 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.

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 -cel` command reports global priorities for all active processes. The `priocntl` command reports the class-specific priorities that users and programmers use.

Note – Global process priorities and user-supplied priorities are in ascending order: numerically higher priorities run first.

The `priocntl`(1) command and the `priocntl`(2) and `priocntlset`(2) functions set or retrieve scheduler parameters for processes. The basic idea for setting priorities is 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.

You specify the target processes using an ID type and an ID. The ID type tells how to interpret the ID. [This concept of a set of processes applies to signals as well as to the scheduler; see `sigsend`(2).] The following table lists the valid ID types that you can specify.

Table 3-1 Valid `priocntl` ID Types

`priocntl` **ID types**
process ID
parent process ID
process group ID
session ID
class ID
effective user ID
effective group ID
all processes

These IDs are basic properties of UNIX processes. [See `intro`(2).] The class ID refers to the scheduler class of the process. `priocntl` works only for the time-sharing and the real-time classes, not for the system class. Processes in the system class have fixed priorities assigned when they are started by the kernel.

The `priocntl` *Command*

The `priocntl` command comes in four forms:

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

Here is the output of the `-l` option for the default configuration.

```
$ priocntl -l
CONFIGURED CLASSES
==================

SYS (System Class)

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

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

The `-d` option displays the scheduler parameters of a process or a set of processes. The syntax for this option is

`priocntl -d -i` *`idtype idlist`*

idtype tells what kind of IDs are in *idlist*. *idlist* is a list of IDs separated by white space. Here are the valid values for *idtype* and their corresponding ID types in *idlist*:

Table 3-2 Valid *idtype* Values

idtype	***idlist***
`pid`	process IDs
`ppid`	parent process IDs
`pgid`	process group IDs
`sid`	session IDs
`class`	class names (TS or RT)
`uid`	effective user IDs
`gid`	effective group IDs
`all`	

Here are some examples of the `-d` option of `priocntl`.

Display information on all processes

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

Display information on all time-sharing processes

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

Display information on all processes with user ID 103 or 6626

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

The `-s` option sets scheduler parameters for a process or a set of processes. The syntax for this option is

`priocntl -s -c` *`class class_options`* `-i` *`idtype idlist`*

idtype and *idlist* are the same as for the `-d` option described above.

class is `TS` for time-sharing or `RT` for real-time. You must be superuser to create a real-time process, to raise a time-sharing user priority above a per-process limit, or to raise the per-process limit above zero. Class options are class-specific:

Table 3-3 Class-Specific Options for `priocntl`

Class-specific options for `priocntl`			
class	-c *class*	Options	Meaning
real-time	RT	`-p` *pri*	priority
		`-t` *tslc*	time slice
		`-r` *res*	resolution
time-sharing	TS	`-p` *upri*	user priority
		`-m` *uprilim*	user priority limit

For a real-time process you can assign a priority and a time slice.

- The priority is a number from 0 to the real-time maximum as reported by `priocntl -l`; the default maximum value is 59.
- You specify the time slice as a number of clock intervals and the resolution of the interval. Resolution is specified in intervals per second. The time slice, therefore, is *tslc/res* seconds. To specify a time slice of one-tenth of a second, for example, you could specify a *tslc* of 1 and a *res* of 10. If you specify a time slice without specifying a resolution, millisecond resolution (a *res* of 1000) is assumed.

If you change a time-sharing process into a real-time process, it gets a default priority and time slice if you don't specify one. To change only the priority of a real-time process and leave its time slice unchanged, omit the `–t` option. To change only the time slice of a real-time process and leave its priority unchanged, omit the `–p` option.

For a time-sharing process you can assign a user priority and a user priority limit.

- The user priority is the user-controlled component of a time-sharing priority. The scheduler calculates the global priority of a time-sharing process by combining this user priority with a system-controlled component that depends on process behavior. The user priority has the same effect as a value set by `nice` (except that `nice` uses higher numbers for lower priority).
- The user priority limit is the maximum user priority a process can set for itself without being superuser. By default, the user priority limit is 0; you must be superuser to set a user priority limit above 0.

Both the user priority and the user priority limit must be within the user priority range reported by the `priocntl -l` command. The default range is -20 to +20.

You can lower and raise a process user priority as often as you like, as long as the value is below the process user priority limit. It is a courtesy to other users to lower your user priority for big chunks of low-priority work. On the other hand, if you lower your user priority limit, you must be superuser to raise it. A typical use of the user priority limit is to reduce permanently the priority of child processes or of some other set of low-priority processes.

The user priority can never be greater than the user priority limit. If you set the user priority limit below the user priority, the user priority is lowered to the new user priority limit. If you attempt to set the user priority above the user priority limit, the user priority is set to the user priority limit.

Here are some examples of the `–s` option of `priocntl`:

Make the process with ID 24668 a real-time process with default parameters

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

Make 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
```

Change 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
```

The –e option sets scheduler parameters for a specified command and executes the command. The syntax for this option is

`priocntl -e -c` *class class_options command* [*command arguments*]

The class and class options are the same as for the `–s` option described above.

Start a real-time shell with default real-time priority

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

Run make with a time-sharing user priority of -10.

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

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

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

The `priocntl` *Function*

```
#include    <sys/types.h>
#include    <sys/procset.h>
#include    <sys/priocntl.h>
#include    <sys/rtpriocntl.h>
#include    <sys/tspriocntl.h>
```

```
long priocntl(idtype_t idtype, id_t id, int cmd,
    cmd_struct  arg);
```

The `priocntl` function gets or sets the scheduler parameters of a set of processes. The input arguments follow.

- `idtype` is the type of ID you are specifying.
- `id` is the ID.
- `cmd` specifies which `priocntl` function to perform. The functions are listed in Table 3-4.
- `arg` is a pointer to a structure that depends on `cmd`.

Here are the valid values for *idtype*, which are defined in `priocntl.h`, and their corresponding ID types in *id*:

Table 3-4 Valid `priocntl.h` *idtypes*

idtype	Interpretation of id
P_PID	process ID (of a single process)
P_PPID	parent process ID
P_PGID	process group ID
P_SID	session ID
P_CID	class ID
P_UID	effective user ID
P_GID	effective group ID
P_ALL	all processes

Here are the valid values for `cmd`, their meanings, and the type of `arg`:

Table 3-5 Valid `cmd` Values

	priocntl Commands	
cmd	**arg Type**	**Function**
PC_GETCID	pcinfo_t	get class ID and attributes

Table 3-5 Valid cmd Values

	priocntl Commands	
cmd	**arg Type**	**Function**
PC_GETCLINFO	pcinfo_t	get class name and attributes
PC_SETPARMS	pcparms_t	set class and scheduling parameters
PC_GETPARMS	pcparms_t	get class and scheduling parameters

Here are the values priocntl returns on success:

- The GETCID and GETCLINFO commands return the number of configured scheduler classes.
- PC_SETPARMS returns 0.
- PC_GETPARMS returns the process ID of the process whose scheduler properties it is returning.

On failure, priocntl returns –1 and sets errno to indicate the reason for the failure. See priocntl(2) for the complete list of error conditions.

PC_GETCID, PC_GETCLINFO

The PC_GETCID and PC_GETCLINFO commands retrieve scheduler parameters for a class based on the class ID or class name. Both commands use the pcinfo structure to send arguments and receive return values:

```
typedef struct pcinfo {
    id_t    pc_cid;/* class id */
    char    pc_clname[PC_CLNMSZ];/* class name */
    long    pc_clinfo[PC_CLINFOSZ];/* class information */
} pcinfo_t;
```

The PC_GETCID command gets scheduler class ID and parameters given the class name. The class ID is used in some of the other priocntl commands to specify a scheduler class. The valid class names are TS for time-sharing and RT for real-time.

For the real-time class, pc_clinfo contains an rtinfo structure, which holds rt_maxpri, the maximum valid real-time priority. In the default configuration, this is the highest priority any process can have. The minimum valid real-time priority is zero. rt_maxpri is a configurable value

```
typedef struct rtinfo {
    short rt_maxpri;/* maximum real-time priority */
} rtinfo_t;
```

For the time-sharing class, `pc_clinfo` contains a `tsinfo` structure, which holds `ts_maxupri`, the maximum time-sharing user priority. The minimum time-sharing user priority is `–ts_maxupri`. `ts_maxupri` is also a configurable value.

```
typedef struct tsinfo {
    short ts_maxupri;/* limits of user priority range */
} tsinfo_t;
```

The following program is a substitute for `priocntl -l`; it gets and prints the range of valid priorities for the time-sharing and real-time scheduler classes

```
/*
 * 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);
```

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 function is useful in the examples below. Given a class name, it uses `PC_GETCID` to return the class ID and maximum priority in the class.

Note – The following examples omit the lines that include header files. The examples compile with the same header files used in the previous code example.

```
/*
 * 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);
}
```

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.

The following program uses `PC_GETCLINFO` to get the class name of a process based on the process ID. This program assumes the existence of a function `getclassID`, which retrieves the class ID of a process given the process ID; this function is given in the following section

```
/* Get scheduler class name given process ID. */

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);
}
```

PC_GETPARMS, PC_SETPARMS

The `PC_GETPARMS` command gets and the `PC_SETPARMS` command sets scheduler parameters for processes. Both commands use the `pcparms` structure to send arguments or receive return values:

```
typedef struct pcparms {
    id_t pc_cid;/* process class */
    long pc_clparms[PC_CLPARMSZ];/* class specific */
} pcparms_t;
```

Ignoring class-specific information for the moment, here is a simple function for returning the scheduler class ID of a process, as promised in the previous section.

```
/*
 * 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);
}
```

For the real-time class, pc_clparms contains an rtparms structure. rtparms holds scheduler parameters specific to the real-time class.

```
typedef struct rtparms {
    short  rt_pri;    /* realtime priority */
    ulong  rt_tqsecs; /* seconds in time quantum */
    long   rt_tqnsecs;/* additional nsecs in quantum */
} rtparms_t;
```

`rt_pri` is the real-time priority; `rt_tqsecs` is the number of seconds and `rt_tqnsecs` is the number of additional nanoseconds in a time slice. That is, `rt_tqsecs` seconds plus `rt_tqnsecs` nanoseconds is the interval a process can use the CPU without sleeping before the scheduler gives another process a chance at the CPU.

For the time-sharing class, `pc_clparms` contains a `tsparms` structure. `tsparms` holds the scheduler parameter specific to the time-sharing class.

```
typedef struct tsparms {
    short ts_uprilim;      /* user priority limit */
    short ts_upri;         /* user priority */
} tsparms_t;
```

ts_upri is the user priority, the user-controlled component of a time-sharing priority. ts_uprilim is the user priority limit, the maximum user priority a process can set for itself without being superuser. These values are described above in the discussion of the -s option of the priocntl command. Both the user priority and the user priority limit must be within the range reported by the priocntl -l command; this range is also reported by the PC_GETCID and PC_GETCLINFO commands to the priocntl function.

The PC_GETPARMS command gets the scheduler class and parameters of a single process. The return value of the priocntl is the process ID of the process whose parameters are returned in the pcparms structure. The process chosen depends on the idtype and id arguments to priocntl and on the value of pcparms.pc_cid, which contains PC_CLNULL or a class ID returned by PC_GETCID:

Table 3-6 What PC_GETPARMS Returns

Number of Processes Selected by idtype and id	pc_cid		
	RT class ID	**TS class ID**	**PC_CLNULL**
1	RT parameters of process selected	TS parameters of process selected	class and parameters of process selected
More than 1	RT parameters of highest-priority RT process	TS parameters of process with highest user priority	(error)

If idtype and id select a single process and pc_cid does not conflict with the class of that process, priocntl returns the scheduler parameters of the process. If they select more than one process of a single scheduler class, priocntl returns parameters using class-specific criteria as shown in the table. priocntl returns an error in the following cases:

- idtype and id select one or more processes and none is in the class specified by pc_cid.
- idtype and id select more than one process and pc_cid is PC_CLNULL.
- idtype and id select no processes.

The following program takes a process ID as its input and prints the scheduler class and class-specific parameters of that process.

```
/*
 * Get scheduler class and parameters of
 * process whose pid is input argument.
 */

main (argc, argv)
        int argc;
        char *argv[];
{
        pcparms_t  pcparms;
        rtparms_t  *rtparmsp;
        tsparms_t  *tsparmsp;
        id_t       pid, rtID, tsID;
        id_t       schedinfo();
        short      priority, tsmaxpri, rtmaxpri;
        ulong      secs;
        long       nsecs;

        pcparms.pc_cid = PC_CLNULL;
        rtparmsp = (rtparms_t *) pcparms.pc_clparms;
        tsparmsp = (tsparms_t *) pcparms.pc_clparms;
        if ((pid = atoi(argv[1])) <= 0) {
            perror ("bad pid");
            exit (1);
        }
/* get scheduler properties for this pid */
...
}
```

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.

The `pcparms` structure contains the new parameters: `pc_cid` contains the ID of the scheduler class to which the processes are to be assigned, as returned by `PC_GETCID`; `pc_clparms` contains the class-specific parameters:

- If `pc_cid` is the real-time class ID, `pc_clparms` contains an `rtparms` structure in which `rt_pri` contains the real-time priority and `rt_tqsecs` plus `rt_tqnsecs` contains the time slice to be assigned to the processes.
- If `pc_cid` is the time-sharing class ID, `pc_clparms` contains a `tsparms` structure in which `ts_uprilim` contains the user priority limit and `ts_upri` contains the user priority to be assigned to the processes.

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 listed above to get the real-time class ID and maximum priority.

```
/*
 * Input arg is proc ID. Make process a realtime
 * process with highest priority minus 1.
 */

main (argc, argv)
    int argc;
    char *argv[];
{
    pcparms_tpcparms;
    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);
    }
}
```

The following table lists the special values rt_tqnsecs can take when PC_SETPARMS is used on real-time processes. When any of these is used, rt_tqsecs is ignored. These values are defined in the header file rtpriocntl.h.

Table 3-7 Special Values for rt_tqnsecs

rt_tqnsecs	**Time Slice**
RT_TQINF	infinite
RT_TQDEF	default
RT_NOCHANGE	unchanged

RT_TQINF specifies an infinite time slice. RT_TQDEF specifies the default time slice configured for the real-time priority being set with the SETPARMS call. RT_NOCHANGE specifies no change from the current time slice; this value is useful, for example, when you change process priority but do not want to change the time slice. (You can also use RT_NOCHANGE in the rt_pri field to change a time slice without changing the priority.)

The priocntlset *Function*

```
#include<sys/types.h>
#include<sys/signal.h>
#include<sys/procset.h>
#include<sys/priocntl.h>
#include<sys/rtpriocntl.h>
#include<sys/tspriocntl.h>

long priocntlset(procset_t *psp, int cmd, cmd_struct arg);
```

The pri ocntlset function changes scheduler parameters of a set of processes, just like priocntl. priocntlset has the same command set as priocntl; the cmd and arg input arguments are the same. But while priocntl applies to a set of processes specified by a single idtype/id pair, priocntlset applies to a set of processes that results from a logical combination of two idtype/id pairs.

The input argument psp points to a procset structure that specifies the two idtype/id pairs and the logical operation to perform. This structure is defined in procset.h.

```
typedef struct procset {
        idop_t      p_op        /* operator connecting */
                                /* left and right sets */

    /* left set: */
        idtype_t    p_lidtype;  /* left ID type */
        id_t        p_lid;      /* left ID */

    /* right set: */
        idtype_t    p_ridtype;  /* right ID type */
        id_t    p_rid;          /* right ID */
} procset_t;
```

`p_lidtype` and `p_lid` specify the ID type and ID of one (“left”) set of processes; `p_ridtype` and `p_rid` specify the ID type and ID of a second (“right”) set of processes. `p_op` specifies the operation to perform on the two sets of processes to get the set of processes to operate on.

The valid values for `p_op` and the processes they specify are:

- `POP_DIFF`: set difference—processes in left set and not in right set
- `POP_AND`: set intersection—processes in both left and right sets
- `POP_OR`: set union—processes in either left or right sets or both
- `POP_XOR`: set exclusive-or—processes in left or right set but not in both

The following macro, also defined in `procset.h`, offers a convenient way to initialize a `procset` structure.

```
#define setprocset(psp, op, ltype, lid, rtype, rid) \
    (psp)->p_op= (op); \
    (psp)->p_lidtype= (ltype); \
    (psp)->p_lid= (lid); \
    (psp)->p_ridtype= (rtype); \
    (psp)->p_rid= (rid);
```

Here is a situation where `priocntlset` would be 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 follows. (This example uses the function `schedinfo`, which is defined above in the section on `PC_GETCID`.

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

```
 */

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);
    }

...
}
```

`priocntl` offers a simple scheduler interface that is adequate for many applications. When a process needs a more powerful way to specify sets, use `priocntlset`.

Interaction with Other Functions

Kernel Processes

The kernel assigns its daemon and housekeeping processes 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.

If the work load on a machine contains real-time processes that use too much CPU, they can lock out system processes, which can lead to trouble. Real-time applications must ensure that they leave some CPU time for system and other processes.

`fork` *and* `exec`

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

`nice`

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

`init`

The `init` process is treated as a special case by the scheduler. To change the scheduler properties of `init`, `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 processes are time-sharing processes. A process changes class only as a result of one of the `priocntl` functions.

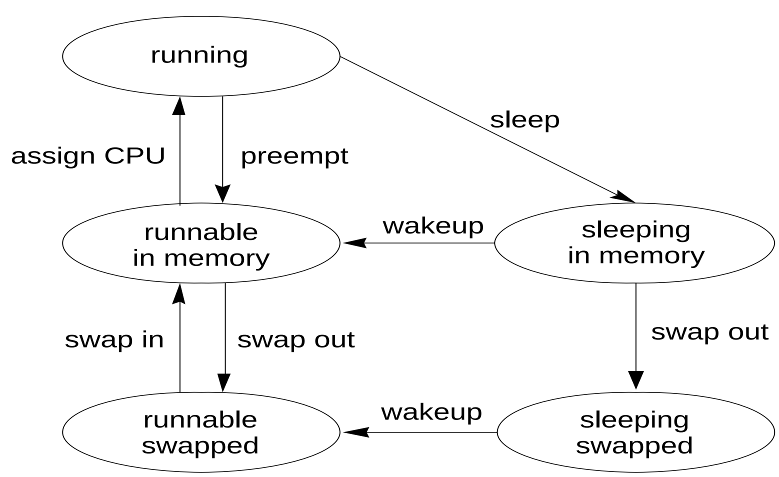
In the default configuration, all real-time process priorities are above any time-sharing process priority. This implies that as long as any real-time process is runnable, no time-sharing process or system process ever runs. So if a real-time application is not written carefully, it can completely lock out 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. In particular, the `sar`(1) command has options for reporting on all the factors discussed in this section.

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, and especially swapping, introduce 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). Of course, how much 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 112 for information about latencies in real-time applications.

Input/Output Interfaces 4

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 encoding binary data, 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: an nonterminated sequence of bytes.

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

- One form is the traditional style of file I/O described in "Basic File I/O" on page 52.
- The second form is the "standard file I/O". The buffering provided by standard I/O provides an easier interface and improved efficiency to an application that is run on a system without virtual memory. In an application running in a virtual memory environment, such as the Solaris 2.x system, 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 94. Mapping files is the most efficient and powerful form of file I/O for most applications run in the Solaris 2.x environment.

Note that it is not necessary to use traditional file I/O to obtain locking of file elements. The lighter weight synchronization mechanisms described in *Multithreaded Programming Guide* can be used more effectively with mapped files.

Basic File I/O

These functions perform basic operations on files:

Table 4-1 Basic File I/O Functions

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

Code Example 4-1, below, 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`(), by a return value of zero.

Code Example 4-1 Basic file I/O

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

main()
{
    int     fd, n;
    char    array[MAXSIZR]

    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 a file when you are done reading or writing it.

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

Code Example 4-2 Seek code

```
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, (char *)&rec, sizeof (rec));

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

Advanced File I/O

These functions create and remove directories and files, create links to existing files, and obtain or modify file status information:

Table 4-2 Advanced File I/O Functions

Function Name	**Purpose**
`link`	Link to a file
`access`	Determine accessibility of a file
`mknod`	Make a special or ordinary file
`chmod`	Change mode of file

`chown`
`lchown`
`fchown`
	Change owner and group of a file

Table 4-2 Advanced File I/O Functions (Continued)

Function Name	Purpose
`utime`	Set file access and modification times

`stat`
`lstat`
`fstat`
	Get file status

`fcntl`	Perform file control functions
`ioctl`	Control device
`fpathconf` `pathconf`	Get configurable path name variables

`opendir`
`readdir`
`closedir`
	Perform directory operations

`mkdir`	Make a directory
`readlink`	Read the value of a symbolic link
`rename`	Change the name of a file
`rmdir`	Remove a directory
`symlink`	Make a symbolic link to a file

File System Control

These functions allow you to control various aspects of the file system:

Table 4-3 File System Control Functions

Function Name	Purpose
`ustat`	Get file system statistics
`sync`	Update super block
`mount`	Mount a file system
`unmount`	Unmount a file system
`statvfs` `fstatvfs`	Get file system information
`sysfs`	Get file system type information

File and Record Locking

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 implies that the whole file is affected, while record locking means that only a specified portion of the file is locked. (In the SunOS 5.x system, file structure is undefined: 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 files.
- `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 have both mandatory and advisory file locking on the same file at the same time. The mode of the file at the time it is opened determines whether the existing locks on the 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`() is specified in Posix 1003.1 standard. `lockf`() is provided to be compatible with older applications.

Terminology

Some important definitions:

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 desired record structure.
cooperating processes	Processes use some technique to access a shared resource, often a synchronization mechanism.
read lock	Used to control access to segments of files. A read lock lets other processes also lock the same segment to read, but lets no other process have a write lock on an overlapping segment of the file.
write lock	Used to control access to segments of files. No other process can read or write the record.
advisory lock	The form of record lock that does not suspend execution of other processes. Advisory locking is not enforced on `creat`(2), `open`(2), `read`(2), or `write`(2) operations. Advisory locking lets calling processes poll the state of a lock and do other work if a lock is not available.
mandatory lock	A record lock that suspends 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.

Setting a File Lock

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

The first 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);
    }
    ...
```

The second example uses lockf(3C):

```
#include <unistd.h>

...

    /* make sure the file pointer is at the beginning of the file. */
    lseek(fd, (off_t)0, 0);
    if (lockf(fd, F_TLOCK, 0L) < 0) {
        if (errno == EAGAIN || errno == EACCES) {
            (void) fprintf(stderr,"File busy try again later!\n");
            return;
        }
        perror("lockf");
        exit(2);
    }
...
```

Note that the lockf(3C) example is simpler, but the fcntl(2) example shows more flexibility. Using fcntl(2), you can set the type and start of the lock request by setting a few structure variables. The lockf method sets only write (exclusive) locks; an additional function, lseek, is required to specify the start of the lock.

Opening a File for Record 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 the 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);
    }
    ...
```

Note – Mapped files cannot be locked with `flock()`. See `mmap(2)`.

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 are not set to zero.

Plan a failure strategy for when you cannot obtain all the required locks. Contention for records 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()`:

```
{
    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` 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 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 the `fcntl` call. The next example finds and prints indentifying data on all the locked segments of a file:

```
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 %8d %8d\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);
```

The `fcntl` function 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.

The `lockf` function with the `F_TEST` command can also be used to test if a process is blocking a lock. This function does not return information about where the lock is and which process owns the lock.

```
(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 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. Get mandatory enforcement 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
```

(Note that this is letter “l” and not the number “1”.) This command sets two permission bits in the file mode, which indicates mandatory locking on the file. The two bits in the mode are .1./.../..0/... An individual file cannot simultaneously be enabled for mandatory locking and have the set-group-ID on execution bit set. Nor can an individual file be enabled for mandatory locking and for group 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, as well as 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

These functions deal with a general terminal interface for controlling asynchronous communications ports:

Table 4-4 Terminal I/O Functions

Function Name	Purpose
tcgetattr tcsetattr	Get and set terminal attributes
tcsendbreak tcdrain tcflush tcflow	Perform line control functions
cfgetospeed cfgetispeed cfsetispeed cfsetospeed	Get and set baud rate
tcgetpgrp tcsetpgrp	Get and set terminal foreground process group ID
tcgetsid	Get terminal session ID

System V IPC 5

The SunOS 5.x system provides several mechanisms that allow processes to exchange data and synchronize execution. The simpler of these mechanisms are pipes, named pipes, and signals. These are limited, however, in what they can do:

- Pipes do not allow unrelated processes to communicate.
- Named pipes allow unrelated processes to communicate, but do not provide private channels for pairs of communicating processes; that is, any process with appropriate permission can read from or write to a named pipe.
- Sending signals with the `kill` function allows arbitrary processes to communicate, but the message consists only of the signal number.

The SunOS 5.x system provides an InterProcess Communication (IPC) package that supports three more versatile types of interprocess communication:

- 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.

When implemented as a unit, these three mechanisms share common properties:

- Each mechanism contains a “get” function to create a new entry or retrieve an existing one.
- Each mechanism contains a “control” function to query the status of an entry, to set status information, and to remove the entry from the system.

- Each mechanism contains one or more “operations” functions to perform various operations on an entry.

This chapter describes the functions for each of these three forms of IPC.

This information is for programmers who write multiprocess applications. These programmers should have a general understanding of what semaphores are and how they are used.

See the following manual pages as listed in Code Example 5-1 for more information about IPC.

Table 5-1 IPC Reference Manual Pages

`ipcrm`(1)	`ipcs`(1)	`intro`(2)
`msgget`(2)	`msgctl`(2)	`msgop`(2)
`semget`(2)	`semctl`(2)	`semop`(2)
`shmget`(2)	`shmctl`(2)	`shmop`(2)
`stdipc`(3C)		

Included in this chapter are several example programs showing the use of these IPC functions. You can accomplish the same task in many ways, so keep in mind that the example programs were written for clarity and not for program efficiency. Usually, functions are embedded within a larger user-written program that uses a particular function provided by the calls.

Permissions

Permissions for messages, semaphores, and shared memory can be extended to users other than the one for which the facility was created. The creating process identifies the default owner. Unlike files, however, the creator can assign ownership of the facility to another user; it can also revoke an ownership assignment. The current owner process, in turn, can grant read or write access to still other users.

The definition of the IPC permissions data structure `ipc_perm` is given in `<sys/ipc.h>`:

Code Example 5-1 IPC Permissions Data Structure

```
struct ipc_perm
{
    uid_t   uid;    /* owner’s user id */
```

```
    gid_t   gid;    /* owner's group id */
    uid_t   cuid;   /* creator's user id */
    gid_t   cgid;   /* creator's group id */
    mode_t  mode;   /* access modes */
    ulong   seq;    /* slot usage sequence number */
    key_t   key;    /* key */
    long    pad[4]; /*reserve area */
};
```

This structure is common to messages, semaphores, and shared memory. Permissions for an IPC facility are initialized by the creating process and can be modified by any process with permission to perform control operations on that facility.

Permissions are specified as octal values in the flags argument of the appropriate IPC creation or control function:

Table 5-2 Octal Permission Values

Access Permissions	Octal Value
Write by Owner	0200
Read by Owner	0400
R/W by Owner	0600
Write by Group	0020
Read by Group	0040
R/W by Group	0060
Write by Others	0002
Read by Others	0004
R/W by Others	0006

For instance, to get read access by the owner and read and write access by others, the permissions value is `0406`.

IPC Functions, Key Arguments, and Creation Flags

Processes requesting access to a common IPC facility must have a way to determine the identity of the facility. To do this, functions that initialize or provide access to an IPC facility use a *key* argument (of type `key_t`).

This key is a value known to all the programs, or one that can be derived from a common seed at run time. A common way to derive the key is with `ftok` (see `stdipc`(3C)). This converts a filename to a key value that is virtually unique within the system. The key value can be used by all programs (processes) attempting to access the facility.

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` (defined to be zero), 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 attempts 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 simply 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’` in the following figure) based on the string ("`/tmp`" in the following figure). The second argument evaluates to the combined permissions and control flags.

Messages

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

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()` for control operations.

Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively (see `msgop`(2)). When a message is sent, its text is copied to the message queue.

The `msgsnd()` and `msgrcv()` 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.

Structure of a Message Queue

A message queue contains a control structure with a unique ID, a linked list of message headers, and a buffer for the message text. The identifier for the queue is the `msqid`.

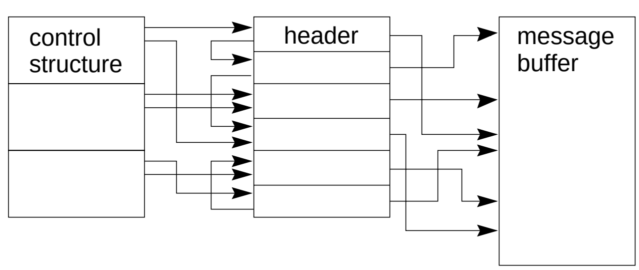


Figure 5-1 Structure of a Message Queue

The control structure for the message queue contains the following information:

- A permissions structure.
- A pointer to the first message on the queue.
- A pointer to the last message on the queue.
- The number of bytes in the queue.
- The number of messages in the queue.
- The maximum number of bytes allowed in the queue.
- The process ID (PID) of the last message sender.
- The PID of the last message receiver.
- The time the last message was sent.
- The time the last message was received.
- The time of the last change to the structure.

Each message header contains the following information:

- A pointer to the next message on the queue.
- The message type.
- The message text size.
- The message text address.

The message queue control structure is defined in `<sys/msg.h>`

Code Example 5-2 Message Queue Control Structure

```
struct msqid_ds
{
    struct ipc_permmsg_perm;      /* operation permission struct */
    struct msg     *msg_first;    /* ptr to first message on q */
    struct msg     *msg_last;     /* ptr to last message on q */
    ulong          msg_cbytes;    /* current # bytes on q */
    ulong          msg_qnum;      /* # of messages on q */
    ulong          msg_qbytes;    /* max # of bytes on q */
    pid_t          msg_lspid;     /* pid of last msgsnd */
    pid_t          msg_lrpid;     /* pid of last msgrcv */
    time_t         msg_stime;     /* last msgsnd time */
    long           msg_pad1;      /* reserved to expand time_t */
    time_t         msg_rtime;     /* last msgrcv time */
    long           msg_pad2;      /* time_t expansion */
    time_t         msg_ctime;     /* last change time */
    long           msg_pad3;      /* time expansion */
    long           msg_pad4[4];   /* reserve area*/
};
```

The definition of the message-header data structure is the following

Code Example 5-3 Message Header Structure

```
struct msg
{
    struct msg *msg_next;         /* ptr to next message on q */
    long        msg_type;         /* message type */
    short       msg_ts;           /* message text size */
    short       msg_spot;         /* message text map address */
};
```

Initializing a Message Queue with `msgget()`

The `msgget()` function initializes a new message queue. It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. When the call fails, it returns –1 and sets the external variable `errno` to the appropriate error code. The `msgget()` synopsis is:

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

int msgget (key_t key, int msgflg);
```

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. The `msgget()` function fails when this limit is exceeded.

The following example is a simple program that illustrates the `msgget()` function. The program prompts for a key, an octal permissions code, and for your choice of control flags. It allows all possible combinations. When `msgget` succeeds, it displays the message queue ID that the call returned. When `msgget()` fails, the program indicates that there was an error and displays the reason for the failure:

Code Example 5-4 Example of `msgget()` call

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

main()
{
    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");
        exit(0);
    }
```

Controlling Message Queues with msgctl()

The `msgctl()` function alters the permissions and other characteristics of a message queue. Its synopsis is

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

int msgctl(int msqid, int cmd, .../* struct msqid_ds *buf */);
```

Upon successful completion, the call returns zero. Upon failure, it returns –1 and sets `errno` appropriately.

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 sample illustrates the `msgctl`(2) function with all its various flags:

Code Example 5-5 Example `msgctl()` calls

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

    ...
    do_msgctl(msqid, IPC_STAT, &buf);
    ...
    do_msgctl(msqid, IPC_SET, &buf);
    ...


int
do_msgctl(msqid, cmd, buf)
struct msqid_ds*buf;  /* pointer to queue descriptor buffer */
int            cmd,    /* command code */
               msqid; /* queue ID */
{
    int rtrn;  /* hold area for return value from msgctl() */

    if (rtrn = msgctl(msqid, cmd, buf) == -1) {
        perror("msgctl: msgctl failed");
        exit(1);
    } else {
        return (rtrn);
    }
}
```

Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions (see the `msgop(2)` manual page) send and receive messages, respectively. Their synopses are:

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

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
        int msgflg);
```

On successful completion, each of these functions returns zero. When unsuccessful, each call returns –1 and sets the external variable errno to the appropriate error code.

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. Combine flags within the argument using the logical OR operator. When IPC_NOWAIT is set, a send or receive operation that cannot finish fails. For instance, a non-blocking msgrcv() operation fails when there is no message to receive. If MSG_NOERROR is set, then a message longer than the size specified by msgsz is truncated to that size. The trailing portion of the truncated message is lost. Without the MSG_NOERROR flag, attempting to receive a message that is longer than expected results in failure.

The msgtyp argument to msgrcv() indicates the type of message to receive. When msgtyp() equals zero, the call receives the first message on the queue. When it is greater than zero, the call receives the first message of the indicated type.

When msgtyp is less than zero, the call receives the first message on the queue with lowest type value, up to and including the absolute value of the argument. For instance, when msgtyp has a value of –3, the call retrieves the first message of type 1, if any, or the first message of type 2, if any, or the first message of type 3. It does not receive a message of type 4. This allows you to prioritize message processing according to type.

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

Code Example 5-6 Example msgsnd() and msgrcv() calls

```
#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 */
    int             msgsz;  /* message size */
    long            msgtyp; /* desired message type */
    int             msqid   /* message queue ID to be used */

    ...
```

```
msgp = (struct msgbuf *)malloc((unsigned)(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);

    ...
    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");
    ...
```

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.

A semaphore set consists of a control structure and an array of individual semaphores. By default, a set of semaphores can contain up to 25 elements. Your system administrator can alter this limit through the `SEMMSL` system configuration option.

Before a process can use a semaphore, the semaphore set must be initialized using `semget`(2). The semaphore owner or creator can change its ownership or permissions using `semctl`(2). Also, any process with permission to do so can use `semctl()` to perform control operations.

Semaphore operations are performed by the `semop`(2) function. This call accepts a pointer to an array of semaphore operation structures. Each structure in the operations array contains information about an operation to perform on a semaphore. The operations array is described in detail in the *Semaphore Operations* section.

Any process with read permission can test to see whether or not a semaphore has a zero value by supplying a 0 in the `sem_op` field of the operation structure. Operations to increment or decrement a semaphore require alter permission (write permission).

When an attempt to perform any of the requested operations fails, none of the semaphores is altered. The process blocks (unless the `IPC_NOWAIT` flag is set), and remains blocked until one of the following occurs:

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

When a semaphore operation fails, the call returns –1 and sets `errno` appropriately.

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()` call, the updates are made atomically. That is, no updates are done until all operations in the array can finish in order successfully.

When a process performs an operation on a semaphore, the system does not usually keep track of whether or not that operation has been undone. If a process with exclusive use of a semaphore terminates abnormally and neglects to undo the operation or free the semaphore, the semaphore remains locked in memory.

To prevent this, `semop()` accepts the `SEM_UNDO` control flag. When this flag is in effect, `semop()` allocates an *undo* structure for each semaphore operation. That structure contains the operation needed to return the semaphore to its previous state.

When the process dies, the system applies the operations in the undo structures. That way an aborted process does not leave 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 would perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value.

This insures that, unless the process is aborted, the values applied to the undo structure will eventually cancel out 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.

Structure of a Semaphore Set

A semaphore set is a control structure with a unique ID and an array of semaphores. The identifier for the semaphore or array is called the semid:

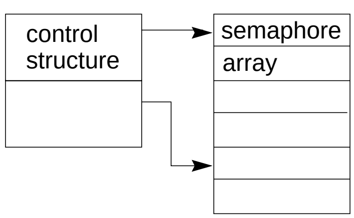


Figure 5-2 Structure of a Semaphore

The control structure for the semaphore contains the following information:

- The permissions structure
- A pointer to first semaphore in the array
- The number of semaphores in the array
- The time of the last operation on any semaphore the array

- The time of the last update to any semaphore in the array

Each semaphore structure in the array contains the following information:

- The semaphore value
- *The PID of the process performing the last successful operation
- The number of processes waiting for the semaphore to increase
- The number of processes waiting for the semaphore to reach zero

The control structure is defined in `<sys/sem.h>`

```
struct semid_ds
{
    struct ipc_perm    sem_perm;  /* operation permission struct */
    struct sem         *sem_base; /* ptr to first semaphore in set */
    ushort             sem_nsems; /* # of semaphores in set */
    time_t             sem_otime; /* last semop time */
    long               sem_pad1;  /* reserved for time_t expansion */
    time_t             sem_ctime; /* last change time */
    long               sem_pad2; /* time_t expansion */
    long               sem_pad3[4]; /* reserve area */
};
```

The `sem_perm` member of this structure uses `ipc_perm` (defined in `<sys/ipc.h>`) as a template.

The semaphore structure is defined in the same header file

```
struct sem
{
    ushort  semval;     /* semaphore text map address */
    pid_t   sempid;     /* pid of last operation */
    ushort  semncnt;    /* # awaiting semval > cval */
    ushort  semzcnt;    /* # awaiting semval = 0 */
};
```

Initializing a Semaphore Set with `semget()`

The `semget()` function initializes or gains access to a semaphore. When the call succeeds, it returns the semaphore ID (`semid`). When the call fails, it returns –1 and sets the external variable `errno` to the appropriate error code. The `semget()` function has the following synopsis

Code Example 5-7 Synopsis of `semget()`

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

int semget(key_t key, int nsems, int semflg);
```

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 assures 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. The `semget()` call fails when one of these limits is exceeded. Because of fragmentation between semaphore sets, it might not be possible to allocate all available semaphores.

The following program illustrates the `semget()` function. It begins by prompting for a hexadecimal key, an octal permissions code, and control command combinations selected from a menu. All possible combinations are allowed.

It then asks the number of semaphores in the array and issues the function to initialize the array. If the call succeeds, the program displays the returned semaphore ID. Otherwise, it displays an error message:

Code Example 5-8 Example `semget()` call

```
#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 with `semctl()`

The `semctl()` function allows a process to alter permissions and other characteristics of a semaphore set. Its synopsis is as follows

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

union semun {
    int val;
    struct semid_ds *buf;
    ushort * array;
};

int     semctl(int semid, int semnum, int cmd, union semun arg)
```

The `semid` value is 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. What you supply for `arg` depends upon the control flag given in `cmd`:

`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 program illustrates `semctl()`

Code Example 5-9 Example `semctl()` call

```
#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);
    }
```

Performing Semaphore Operations with `semop()`

The `semop()` function performs operations on a semaphore set. Its synopsis is as follows

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

int semop(int semid, struct sembuf *sops, size_t nsops);
```

The `semid` argument is the semaphore ID returned by a previous `semget()` 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>`

```
struct sembuf {
    ushort  sem_num;    /* semaphore # */
    short   sem_op;     /* semaphore operation */
    short   sem_flg;    /* operation flags */
};
```

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()` call, and is set to 10 by default.

The operation to be performed is determined as follows:

- A positive integer increments the semaphore value by that amount.
- A negative integer decrements the semaphore value by that amount. However, a semaphore can never take on a negative value. An attempt to set a semaphore to a value below zero either fails or blocks, depending on whether or not `IPC_NOWAIT` is in effect.
- A value of zero means to wait for the semaphore value to reach zero.

You can use the following control flags with `semop()`:

`IPC_NOWAIT`
This operation command can be set for any operations in the array. The function returns unsuccessfully without changing any semaphore values if any operation for which `IPC_NOWAIT` is set cannot be performed successfully. The function will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.

`SEM_UNDO`
This command allows individual operations in the array to be undone when the process exits.

The following program illustrates the `semop()` function

Code Example 5-10 Example `semop()` call

```
#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 Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on native virtual memory management and the `mmap`(2) function. However, Solaris 2.x also supports System V shared memory.

System V shared memory lets more than one process at a time attach a segment of physical memory to its virtual address space. 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 the `shmget`(2) function. This call can also be used to obtain 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 the `shmctl`(2) function; it can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`.

Once created, a shared segment can be attached to a process address space using the `shmat`() function; it can be detached using `shmdt`(). (See `shmop`(2) for details.)

The attaching process must have the appropriate permissions for `shmat()` to succeed. 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.

If the above-mentioned function fails, it returns –1 and sets the external variable `errno` to the appropriate value.

Structure of a System V Shared Memory Segment

A shared memory segment is composed of a control structure with a unique ID that points to an area of physical memory. The identifier for the segment is referred to as the `shmid`.

Figure 5-3 Structure of a Shared Memory Segment

The data structure includes the following information about the memory segment:

- Access permissions.
- Segment size.
- The PID of the process performing last operation.

- The PID of the creator process.
- The current number of processes to which the segment is attached.
- The time of the last attachment.
- The time of the last detachment.
- The time of the last change to the segment.
- Memory map segment descriptor pointer.

The structure definition for the shared memory segment control structure can be found in <sys/shm.h>. This structure definition is shown below

```
struct shmid_ds {
    struct ipc_permshm_perm;  /* operation permission struct */
    int            shm_segsz; /* size of segment in bytes */
    struct anon_map*shm_amp;  /* segment anon_map pointer */
    ushort         shm_lkcnt; /* number of times it is locked */
    pid_t          shm_lpid;  /* pid of last shmop */
    pid_t          shm_cpid;  /* pid of creator */
    ulong          shm_nattch;/* used only for shminfo */
    ulong          shm_cnattch;/* used only for shminfo */
    time_t         shm_atime; /* last shmat time */
    long           shm_pad1;  /* reserved for time_t expansion */
    time_t         shm_dtime; /* last shmdt time */
    long           shm_pad2;  /* reserved for time_t expansion */
    time_t         shm_ctime; /* last change time */
    long           shm_pad3;  /* reserved for time_t expansion */
    long           shm_pad4[4];/* reserve area */
};
```

Note that the `shm_perm` member of this structure uses `ipc_perm` as a template, as defined in <sys/ipc.h>.

Using `shmget()` *to Access a Shared Memory Segment*

The `shmget()` function is used to obtain access to a shared memory segment. When the call succeeds, it returns the shared memory segment ID (`shmid`). When it fails, it returns –1 and sets `errno` to the appropriate error code. The `shmget()` function has the following synopsis

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

int shmget(key_t key, int size, int shmflg);
```

The value passed as the `shmflg` argument must be an integer, which incorporates settings for the segment's permissions and control flags, as described under "Permissions" on page 66.

The `SHMMNI` system configuration option determines the maximum number of shared memory segments that are allowed, 100 by default.

The function fails if the `size` value is less than `SHMMIN` or greater than `SHMMAX`, the configuration options for the minimum and maximum segment sizes. By default, `SHMIN` is 1, `SHMAX` is 131072.

The following sample program illustrates the `shmget()` function:

Code Example 5-11 Sample Program to Illustrate `shmget()`

```
#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() */

    ...
    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 with `shmctl()`

The `shmctl()` function is used to alter the permissions and other characteristics of a shared memory segment. It synopsis is as follows

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

int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Figure 5-4 Synopsis of `shmctl()`

The `shmid` argument is the ID of the shared memory segment as returned by `shmget()`.

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 program illustrates the `shmctl()` function

Code Example 5-12 Sample `shmctl()` call

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

```
{
    int     cmd;    /* command code for shmctl() */
    int     shmid;  /* segment ID */
    struct shmid_dsshmid_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 with `shmat()` *and* `shmdt()`

The `shmat()` and `shmdt()` functions are used to attach and detach shared memory segments. Their synopses are as follows

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

void *shmat(int shmid, void *shmaddr, int shmflg);

int shmdt (void *shmaddr);
```

Upon successful completion, the `shmat()` function returns a pointer to the head of the shared segment; when unsuccessful, it returns `(void *) -1` and sets the external variable `errno` to the appropriate error code.

The `shmid` argument is the ID of an existing shared memory segment. The `shmaddr` argument is the address at which to attach the segment. If supplied as zero, the system provides a suitable address. For portability, it is usually better to allow the system to determine the address.

The `shmflg` argument is a control flag used to pass the `SHM_RND` and `SHM_RDONLY` flags to the `shmat()` function.

The `shmdt()` function detaches the shared memory segment located at the address indicated by `shmaddr`. Upon successful completion, `schmdt()` returns zero; when unsuccessful, it returns –1 and sets the external variable `errno` to the appropriate error code.

The following sample code illustrates calls to shmat() and shmdt():

Code Example 5-13 Sample `shmat()` and `shmdt()` calls

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

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. */

{
    register int   action;     /* action to be performed */
    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("shmop: shmat failed");
        nap--;
    } else
         (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
                    p–>shmaddr);
    ...

    ...
    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];
    }
}
...
```

Memory Management 6

Overview of the Virtual Memory System

The UNIX system provides a complete set of memory management mechanisms, providing applications complete control over the construction of their address space and permitting a wide variety of operations on both process address spaces and the variety of memory objects in the system.

Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents the user with mappings that simulate the traditional UNIX process memory environment, but other views of memory are useful as well.

The UNIX memory-management facilities do the following.

- Unify system operations on memory
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support
- Maintain consistency with the existing environment, in particular using the UNIX file system as the name space for named virtual-memory objects

Virtual Memory, Address Spaces, and Mapping

The system virtual memory (VM) consists of all available physical memory resources. Examples include local and remote file systems, processor primary memory, swap space, and other random-access devices. Named objects in the virtual memory are referenced though the UNIX file system. However, not all file system objects are in the virtual memory; devices that cannot be treated as storage, such as terminal and network device files, are not in the virtual memory. Some virtual memory objects, such as private process memory and shared memory segments, do not have names.

A process address space is defined by mappings onto objects in the system virtual memory (usually files). Each mapping is constrained to be sized and aligned with the page boundaries of the system on which the process is executing. Each page may be mapped (or not) independently. Only process addresses that are mapped to some system object are valid, for there is no memory associated with processes themselves—all memory is represented by objects in the system virtual memory.

Each object in the virtual memory has an object address space defined by some physical storage. A reference to an object address accesses the physical storage that implements the address within the object. The physical storage associated with virtual memory is thus accessed by transforming process addresses to object addresses, and then to the physical store.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made is not affected by the existence of the mapping. Thus, it cannot, in general, be expected that an object has an “awareness” of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a “page” is not a property of the object. Establishing a mapping to an object simply provides the potential for a process to access or change the object’s contents.

The establishment of mappings provides an access method that renders an object directly addressable by a process. Applications may find it advantageous to 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.

Networking, Heterogeneity, and Coherence

The VM system is designed to fit well with the larger UNIX heterogeneous environment. This environment extensively uses networking to access file systems—file systems that are now part of the system virtual memory.

Networks are not constrained to consist of similar hardware or to be based upon a common operating system; in fact, the opposite is encouraged, for such constraints create serious barriers to accommodating heterogeneity.

Although a given set of processes might *apply* a set of mechanisms to establish and maintain the properties of various system objects—properties such as page sizes and the ability of objects to synchronize their own use—a given operating system should not *impose* such mechanisms on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running the UNIX memory management system and also to be accessed by systems for which virtual memory and storage management techniques such as paging are totally foreign, such as PC-DOS. Such systems can continue to share access to the object, each using and providing its programs with the access method appropriate to that system.

Another consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In both of these cases, the object is being shared, and the applications must use some synchronization mechanism to guarantee the coherence of their transactions with it. The scope and nature of the synchronization mechanism is best left to the application to decide.

For example, file access on systems that do not support virtual memory access methods must be indirect, by way of `read` and `write`. Applications sharing files on such systems must coordinate their access using semaphores, file locking, or some application-specific protocols.

What is required in an environment where mapping replaces `read` and `write` as the access method is an operation, such as `fsync`, that supports atomic update operations.

The nature and scope of synchronization over shared objects is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing.

By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, the needs of applications are met without erecting barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs.

Memory Management Interfaces

The applications programmer gains access to the facilities of the virtual memory system through several sets of functions. This section summarizes these calls and provides examples of their use. For details, see the *man Pages(2): System Calls*.

Creating and Using Mappings

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

`mmap` establishes a mapping between a process address space and an object in the system virtual memory. It is the system's most fundamental function for defining the contents of an address space—all other system functions that contribute to the definition of an address space are built from `mmap`. The format of an `mmap` call is:

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

`mmap` establishes a mapping from the process address space at an address *paddr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. The value returned by `mmap` is an implementation-dependent function of the parameter *addr* and the setting of the `MAP_FIXED` bit of *flags*, as described below. A successful call to `mmap` returns *paddr* as its result. The address range *[paddr, paddr + len)*[1] must be valid for the address space of the process and the range *[off, off + len)* must be valid for the virtual memory object.

Note – The mapping established by `mmap` replaces any previous mappings for the process pages in the range [*paddr*, *paddr* + *len*].

---

1. Read the notation *[lower, lower + upper)* as "from and including the lower boundary up to, but not including, the upper boundary."

The parameter *prot* determines whether read, execute, write, or some combination of accesses are permitted to the pages being mapped. To deny all access, set *prot* to `PROT_NONE`. Otherwise, specify permissions by an OR of `PROT_READ`, `PROT_EXECUTE`, and `PROT_WRITE` (note that `PROT_EXECUTE` is specific to the SPARC architecture). A write access will fail if `PROT_WRITE` has not been set, though the behavior of the write can be influenced by setting `MAP_PRIVATE` in the *flags* parameter, as described below.

The *flags* parameter provides other information about the handling of mapped pages.

- `MAP_SHARED` and `MAP_PRIVATE` specify the mapping type, and one of them must be specified. The mapping type describes the disposition of store operations made by *this* process into the address range defined by the mapping operation.

  If `MAP_SHARED` is specified, write references will modify the mapped object. No further operations on the object are necessary to effect a change—the act of storing into a `MAP_SHARED` mapping is equivalent to doing a `write` function.

  On the other hand, if `MAP_PRIVATE` is specified, an initial write reference to a page in the mapped area will create a copy of that page and redirect the initial and successive write references to that copy. This operation is sometimes referred to as *copy-on-write* and occurs invisibly to the process causing the store. Only pages actually modified have copies made in this manner.

  The mapping type is retained across a `fork`.

Note – The private copy is not created until the first write; until then, other users who have the object mapped `MAP_SHARED` can change the object. That is, if one user has an object mapped `MAP_PRIVATE` and another user has the same object mapped `MAP_SHARED`, and the `MAP_SHARED` user changes the object before the `MAP_PRIVATE` user does the first write, then the changes appear in the `MAP_PRIVATE` user's copy that the system makes on the first write. If an application needs isolation from changes made by other processes, it should use `read` to make a copy of the data it is isolating.

MAP_PRIVATE mappings are used by system functions such as exec(2) when mapping files containing programs for execution. This permits operations by programs such as debuggers to modify the “text” (code) of the program without affecting the file from which the program is obtained.

- MAP_FIXED informs the system that the value returned by mmap must be exactly *addr*. The use of MAP_FIXED is discouraged, as it can prevent an implementation from making the most effective use of system resources.

  When MAP_FIXED is not set, the system uses *addr* as a hint to arrive at *paddr*. The *paddr* so chosen is an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. An *addr* value of zero grants the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken as a suggestion of a process address near which the mapping should be placed.

  When the system selects a value for *paddr*, it never places a mapping at address 0, nor replaces any extant mapping, nor maps into areas considered part of the potential data or stack “segments.” The system strives to choose alignments for mappings that maximize the performance of the hardware resources.

- MAP_NORESERVE specifies that no swap space is to be reserved in advance for a mapping. Without this flag, a MAP_PRIVATE mapping has swap space reserved for it when the mapping is first created; this swap space is later used to back the private pages that are created by copy-on-write operations.

  Without this advance reservation, swap space might not be available in the system when a copy-on-write is attempted; the system then fails the write access to the page and sends a SIGBUS signal to the process. However, a process can prevent swap space from being reserved in advance by setting the MAP_NORESERVE flag if that process is willing to handle the case in which swap space is not available.

  The advantage of using this flag is that a process can, for example, create and access a huge data segment on a machine that has a relatively small amount of swap space, as long as the process also provides for the case where writes into the segment might fail. Without MAP_NORESERVE it would be impossible to create this segment.

The file descriptor used in a `mmap` call need not be kept open after the mapping is established. If it is closed, the mapping will remain until such time as it is replaced by another call to `mmap` that explicitly specifies the addresses occupied by this mapping or until the mapping is removed either by process termination or a call to `munmap`.

Although the mapping endures independent of the existence of a file descriptor, changes to the file can influence accesses to the mapped area, even if they do not affect the mapping itself.

For instance, should a file be shortened by a call to `truncate`, such that the mapping now "overhangs" the end of the file, then accesses to that area of the file that no longer exists, SIGBUS signals will result.

It is possible to create the mapping in the first place such that it "overhangs" the end of the file—the only requirement when creating a mapping is that the addresses, lengths, and offsets specified in the operation be *possible* (such as, within the range permitted for the object in question), not that they exist at the time the mapping is created (or subsequently.)

Similarly, if a program accesses an address in a manner inconsistent with how it has been mapped (for instance, by attempting a store operation into a mapping that was established with only `PROT_READ` access), then a `SIGSEGV` signal will result. `SIGSEGV` signals will also result on any attempt to reference an address not defined by any mapping.

In general, if a program references an address that is inconsistent with the mapping (or lack of a mapping) established at that address, the system will respond with a `SIGSEGV` violation.

However, if a program references an address consistent with how the address is mapped, but that address does not evaluate *at* the time of the access to allocated storage in the object being mapped, then the system will respond with a `SIGBUS` violation.

In this manner a program (or user) can distinguish between whether it is the mapping or the object that is inconsistent with the access, and take appropriate remedial action.

Using `mmap` to access system memory objects can simplify programs in a variety of ways. Keeping in mind that `mmap` can really be viewed as just a means to access memory objects, it is possible to program using `mmap` in many cases where you might program with `read` or `write`.

However, it is important to realize that mmap can only be used to gain access to *memory* objects—those objects that can be thought of as randomly accessible storage. Thus, terminals and network connections cannot be accessed with mmap because they are not "memory." Magnetic tapes, even though they are memory devices, cannot be accessed with mmap because storage locations on the tape can only be addressed sequentially.

Some examples of situations that *can* be thought of as candidates for use of mmap over more traditional methods of file access include:

- Random access operations—either map the entire file into memory or, if the address space cannot accommodate the file or if the file size is variable, create "windows" of mappings to the object.
- Efficiency—even in situations where access is sequential, if the object being accessed can be accessed via mmap, an efficiency gain may be obtained by avoiding the copying operations inherent in accesses via read or write.
- Structured storage—if the storage being accessed is collected as tables or data structures, algorithms can be more conveniently written if access to the file is treated just as though the tables were in memory.

  Previously, programs could not simply make storage or table alterations in memory and save them for access in subsequent runs; however, when the addresses of the table are defined by mappings to a file, then changes to the storage *are* changes to the file, and are thus automatically recorded in it.
- Scattered storage—if a program requires scattered regions of storage, such as multiple heaps or stack areas, such areas can be defined by mapping operations during program operation.

The remainder of this section illustrates some other concepts surrounding mapping creation and use.

Mapping /dev/zero gives the calling program a block of zero-filled virtual memory of the size specified in the call to mmap. /dev/zero is a special device, that responds to read as an infinite source of bytes with the value 0, but when mapped creates an unnamed object to back the mapped region of memory.

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.

```
/*
 * Function to allocate a block of zeroed storage. Parameter is the
 * number of bytes desired. The storage is mapped as MAP_SHARED, so
 * that if a fork occurs, the child process will be able to access
 * and modify the storage. If we wished to cause the child's
 * modifications (as well as those by the parent) to be invisible to
 * the ancestry of processes, we would use MAP_PRIVATE.
 */
caddr_t
get_zero_storage(int len);
{
    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);
    return (result);
}
```

As written, this function permits a hierarchy of processes to use the area of allocated storage as a region of communication (for *implicit* interprocess communication purposes).

In some cases, devices or files are useful only if accessed via 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.

Finally, it is important to remember that mappings can be operated upon at the granularity of a single page. Even though a mapping operation may define multiple pages of an address space, there is *absolutely* no restriction that subsequent operations on those addresses must operate on the same number of pages.

For instance, an `mmap` operation defining ten pages of an address space may be followed by subsequent `munmap` (see below) operations that remove every other page from the address space, leaving five mapped pages each followed by an unmapped page.

Those unmapped pages may subsequently be mapped to different locations in the same or different objects, or the whole range of pages (or any partition, superset, or subset of the pages) used in other `mmap` or other memory management operations.

Further, any mapping operation that operates on more than a single page can partially succeed in that some parts of the address range can be affected even though the call returns an overall failure.

Thus, an `mmap` operation that replaces another mapping, if it fails, might have deleted the previous mapping and failed to replace it. Similarly, other operations (unless specifically stated otherwise) might process some pages in the range successfully before operating on a page where the operation fails.

Not all device drivers support memory mapping. `mmap` fails if you try to map a device that does not support mapping.

Removing Mappings

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

`munmap` removes all mappings for pages in the range *[addr, addr + len)* from the address space of the calling process.

It is not an error to remove mappings from addresses that do not have them, and *any* mapping, no matter how it was established, can be removed with `munmap`. `munmap` does not in any way affect the objects that were mapped at those addresses.

Cache Control

The UNIX memory management system can be thought of as a form of “cache management,” in which processor primary memory is used as a cache for pages from objects from the system virtual memory. Thus, there are a number of operations that control or interrogate the status of this cache, as described in this section

```
int
mincore(caddr_t addr, size_t len, char *vec);
```

`mincore` determines the residency of the memory pages in the address space covered by mappings in the range *[addr, addr + len)*.

Using the cache concept described earlier, this function can be viewed as an operation that interrogates the status of the cache, and returns an indication of what is currently resident in the cache. The status is returned as a char-per-page in the character array referenced by **vec* (which the system assumes to be large enough to encompass all the pages in the address range).

The low order bit of each character contains either a 1 (indicating that the page is resident in the system's primary storage), or a 0 (indicating that the page is not resident in primary storage). Other bits in the character are reserved for possible future expansion—therefore, programs testing residency should test only the least significant bit of each character.

Because the status of a page can change after `mincore` checks it, but before `mincore` returns the information, returned information might be outdated. Only locked pages are guaranteed to remain in memory

```
int
mlock(caddr_t addr, size_t len);

int
munlock(caddr_t addr, size_t len);
```

`mlock` causes the pages referenced by the mapping in the range *[addr, addr + len)* to be locked in physical memory. References to those pages (through mappings in this or other processes) will not result in page faults that require an I/O operation to obtain the data needed to satisfy the reference.

Because this operation ties up physical system resources and has the potential to disrupt normal system operation, use of this facility is restricted to the superuser. The system will not permit more than a configuration-dependent limit of pages to be locked in memory simultaneously. The call to `mlock` fails if this limit is exceeded.

`munlock` releases the locks on physical pages. Note that if multiple `mlock` calls are made through the same mapping, only a single `munlock` call is required to release the locks (in other words, locks on a given mapping do not nest).

However, if different mappings to the same pages are processed with `mlock`, then the pages will not be 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` operation or removed explicitly with `munmap`.

A lock will be 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 will be retained transparently along with the copy-on-write redirection (see mmap above for a discussion of this redirection)

```
int
mlockall(int flags);

int
munlockall(void);
```

mlockall and munlockall are similar in purpose and restriction to mlock and munlock, except that they operate on entire address spaces. mlockall accepts a *flags* argument built as a bit-field of values from the set:

MCL_CURRENT	Current mappings
MCL_FUTURE	Future mappings

If *flags* is MCL_CURRENT, the lock is to affect everything currently in the address space. If *flags* is MCL_FUTURE, the lock is to affect everything added in the future. If *flags* is (MCL_CURRENT | MCL_FUTURE), the lock is to affect both current and future mappings.

munlockall removes all locks on all pages in the address space, whether established by mlock or mlockall

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

msync supports applications that require assertions about the integrity of data in the storage backing their mapping, either for correctness or for coherent communications in a distributed environment.

msync causes all modified copies of pages over the range *[addr, addr + len)* to be flushed to the objects mapped by those addresses. In the cache analogy discussed previously, msync is the cache “write-back,” or flush, operation. It is similar in purpose to the fsync operation for files.

msync optionally invalidates each such cache entry so that the first subsequent reference to the page causes the system to obtain it from its permanent storage location.

The *flags* argument provides a bit field of values that influences the behavior of msync. The bit names and their interpretations are:

`MS_SYNC`	synchronized write
`MS_ASYNC`	return immediately
`MS_INVALIDATE`	invalidate caches

`MS_SYNC` causes `msync` to return only after all I/O operations are complete. `MS_ASYNC` causes `msync` to return immediately once all I/O operations are scheduled. `MS_INVALIDATE` causes all cached copies of data from mapped objects to be invalidated, requiring them to be obtained again from object storage upon the next reference.

Other Mapping Functions

```
long
sysconf(_SC_PAGESIZE);
```

`sysconf` returns the system-dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page, and instead should make use of `sysconf` to obtain that information.

Note that it is not unusual for page sizes to vary even among implementations of the same instruction set, increasing the importance of using this function for portability.

```
int
mprotect(caddr_t addr, size_t len, int prot);
```

`mprotect` has the effect of assigning protection *prot* to all pages in the range of [*addr, addr* + *len*). The protection assigned cannot exceed the permissions allowed on the underlying object.

For instance, a read-only mapping to a file that was opened for read-only access cannot be set to be writable with `mprotect` (unless the mapping is of the `MAP_PRIVATE` type, in which case the write access is permitted since the writes will modify copies of pages from the object, and not the object itself).

Address Space Layout

Traditionally, the address space of a UNIX process has consisted of exactly three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process stack. Text is read-only and shared, while the data and stack segments are private to the process.

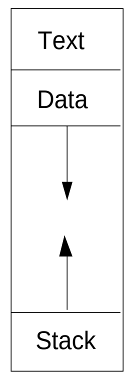


Figure 6-1 Traditional UNIX System Address-Space Layout

In the SunOS 5.x system, a process's address space is simply a vector of pages, and the division between different address-space segments is not so clear-cut. Process text and data spaces are simply groups of pages.[1]

There are often multiple text and data *segments*, some belonging to specific programs and some belonging to code running in shared libraries. The following figure illustrates one possible address space layout.

1. For compatibility, the system maintains address ranges that *should* belong to such segments to support operations such as extending or contracting the data segment's *break*. These are initialized when a program is initiated with `execve`().

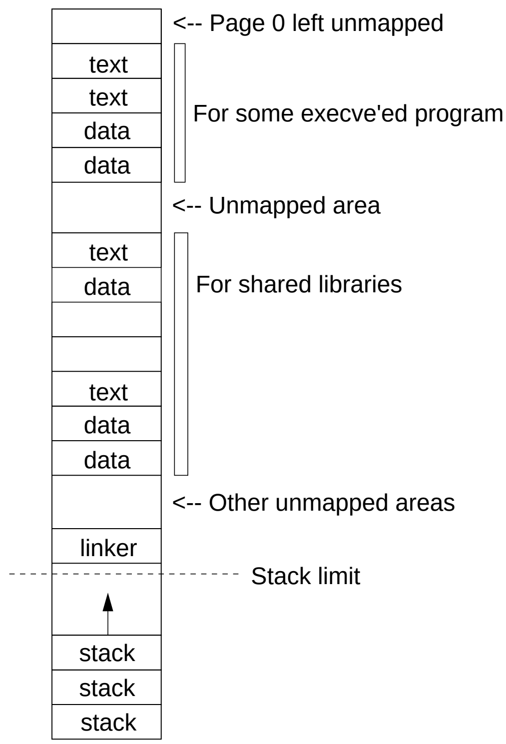


Figure 6-2 Address-Space Layout

Although the system still uses text, data, and stack segments, these should be thought of as constructs provided by the programming environment rather than by the operating system.

As such, it is possible to construct processes that have multiple segments of each *type*, or of types of arbitrary semantic value—programs no longer need to be built only from objects the system can represent directly.

For instance, a process address space may contain multiple text and data segments, some belonging to specific programs and some shared among multiple programs. Text segments from shared libraries, for example, typically appear in the address spaces of many processes.

A process address space is simply a vector of pages, and there is no necessary division between different address space segments. Process text and data spaces are simply groups of pages mapped in ways appropriate to the function they provide the program.

A process address space is usually sparsely populated, with data and text pages intermingled. The precise mechanics of the management of stack space is machine-dependent.

By convention, page 0 is not used. Process address spaces are often constructed through dynamic linking when a program is `exec`'d. Operations such as `exec` and dynamic linking build upon the mapping operations described previously.

Although the system can have multiple areas that can be considered "data" segments, for programming convenience the system maintains operations to operate on an area of storage associated with a process initial "heap storage area."

A process can manipulate this area by calling `brk` and `sbrk`:

```
caddr_t
brk(caddr_t addr);

caddr_t
sbrk(int incr);
```

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

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

Realtime Programming and Administration 7

This chapter describes writing and porting realtime applications to run under SunOS 5.x. This chapter is written for programmers experienced in writing realtime applications and administrators familiar with realtime processing and the SunOS system.

Basic Rules of Realtime Applications

Realtime 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 (too much to do).

A SunOS realtime process:

- runs in the RT scheduling class, as described in "Scheduling" on page 112
- locks down all the memory in its process address space, as described in "Memory Locking" on page 127
- 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 109

Realtime 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 realtime 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 realtime process, it:

- acquires the processor within the guaranteed dispatch latency period of becoming runnable (see"Dispatch Latency" on page 112)
- continues to run for as long as it remains the highest priority runnable process

A realtime 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.

Realtime 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.

Realtime processing can also significantly impact the operation of aspects of other applications active on a system running a realtime application. Since realtime 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.x 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 non-realtime 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 realtime 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.

Realtime 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 realtime process by acquiring a resource that is required by a realtime 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 7-1 where 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.

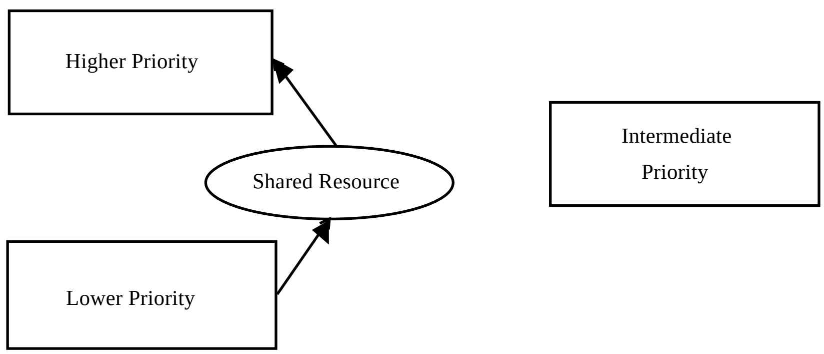


Figure 7-1 Unbounded Priority Inversion

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 Realtime Processes

Runaway realtime 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, try typing (Stop-A). You might have to repeat this procedure many times. If this doesn't work, disconnect the keyboard.

When a high priority realtime process will 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 will not respond to the control-C kill sequence.

Caution – Attempts to use a shell set at a higher priority than a runaway process will not succeed. The STREAMS processes that govern `tty` management are running at system priority, and so will not get scheduled. Therefore, keyboard input is not received by the shell, even when the shell is running at a higher priority.

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`(3), 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.

Realtime Files

SunOS 5.x provides no facilities to assure that files will be allocated as physically contiguous.

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

Scheduling

Realtime scheduling constraints are necessary to manage data acquisition or process control hardware. The realtime 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.x realtime scheduler, its priority queue, and how to use system calls and utilities that control scheduling. For more information about the functions described in this section, see the *man Pages(3): Library Routines*.

Dispatch Latency

The most significant element in scheduling behavior for realtime applications is the provision of a real-time scheduling class. The standard time-sharing scheduling class is not suitable for realtime applications because this scheduling class treats every process equally and has a limited notion of priority. Realtime 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, realtime applications can be developed with a bounded dispatch latency.

Figure 7-2 illustrates the amount of time it takes an application to respond to a request from an external event.

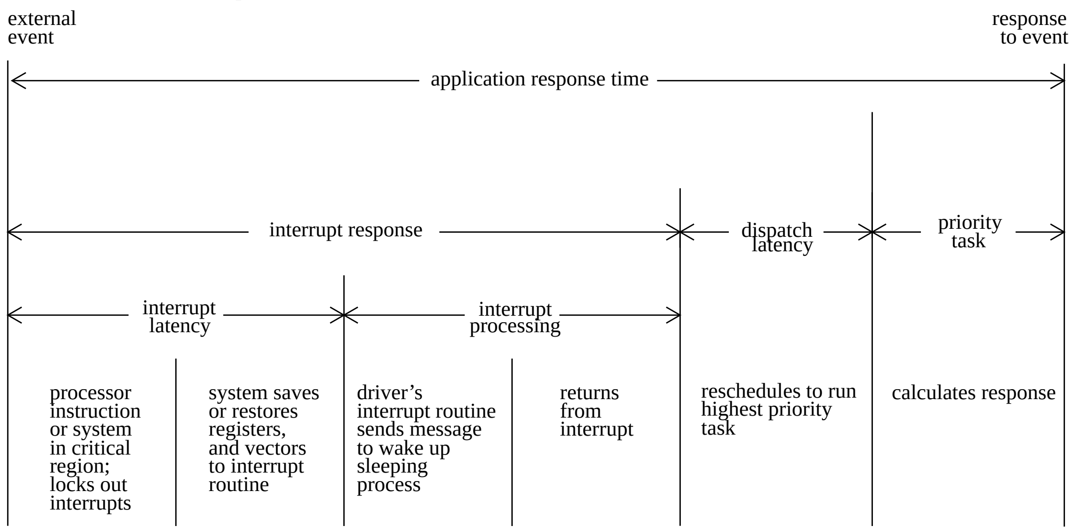


Figure 7-2 Application Response Time

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.x 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 7-3 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.

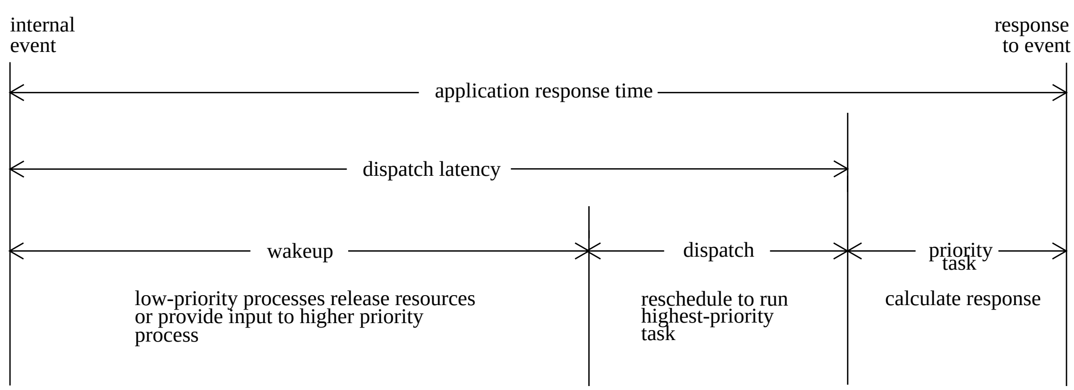


Figure 7-3 Internal Dispatch Latency

With the new scheduling techniques provided with realtime SunOS 5.x, the system dispatch latency time is within specified bounds.

As you can see in Table 7-1, dispatch latency improves with a bounded number of processes.

Table 7-1 Realtime System Dispatch Latency with SunOS 5.x

Workstation	Dispatch Latency: Bounded Number of Processes	Dispatch Latency: Arbitrary Number of Processes
SPARCstation 1	< 2.0 milliseconds in a system with fewer than 8 active processes	4.5 milliseconds
SPARCstation 1+	< 2.0 milliseconds in a system with fewer than 8 active processes	4.0 milliseconds
SPARCstation IPX	< 1.0 milliseconds in a system with fewer than 8 active processes	2.2 milliseconds
SPARCstation 2	< 1.0 milliseconds in a system with fewer than 16 active processes	2.0 milliseconds

Tests for dispatch latency and experience with such critical environments as manufacturing and data acquisition have proven that the Sun workstation is an effective platform for the development of realtime applications. (We apologize that the examples are not of current products.)

Scheduling Classes

The SunOS 5.x kernel dispatches processes by priority. The scheduler (or dispatcher) supports the concept of scheduling classes. Classes are defined as Realtime (`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, realtime processes have precedence over `sys` and `TS` processes, but administrators can configure systems so that `TS` and `RT` processes have overlapping priorities.

Figure 7-4 illustrates the concept of classes as viewed by the SunOS 5.x kernel.

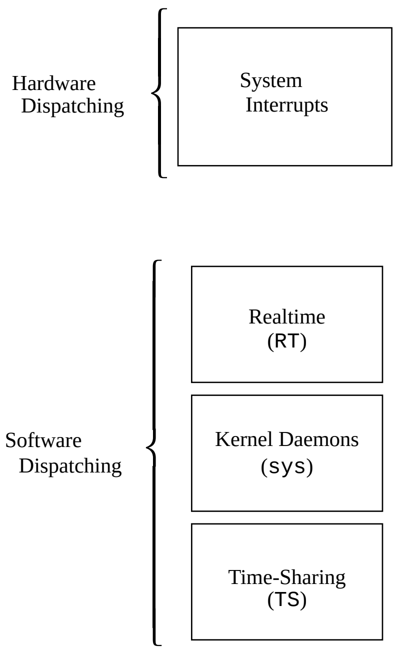


Figure 7-4 Dispatch Priorities for Scheduling Classes

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.

Realtime 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 realtime 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 7-5 illustrates, each class algorithm has its own method of nominating the highest priority process to place on the global run queue.

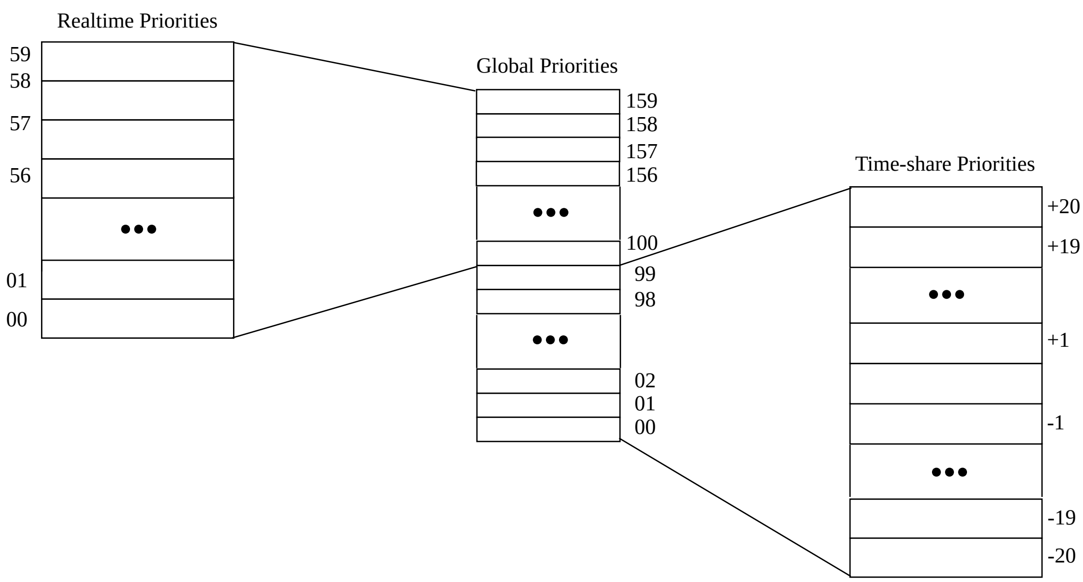


Figure 7-5 The Kernel Dispatch Queue

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 realtime (`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 realtime 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.x kernel can lead to priority inversion.

The term *blocking* describes the situation in which a process must wait for one or more processes to relinquish resources. If this blocking continues, it can lead to deadlines being missed, even for low levels of utilization.

The problem of priority inversion has been addressed for mutual-exclusion locks for the SunOS 5.x 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

There is no mechanism by which processes synchronizing with other processes will automatically inherit the priority of waiting processes. An application can bound its priority inversion by using priority ceiling emulation.

Under this model, the application associates a priority with each synchronization object, which is typically the highest priority of any process that can block on that object.

Each process then uses the following sequence when manipulating the shared resources

```
/*
 * raise process priority to maximum of current level
 * and synchronization object level
 */
    ...

/*
 * acquire synchronization object
 */
    ...
```

```
/*
 * execute the critical section
 */
    ...

/*
 * release synchronized object
 */
    ...

/*
 * return to previous process priority level
 */
    ...
```

System Calls That Control Scheduling

System calls implemented for realtime scheduling include the library calls and functions listed in this section. For more detail about using these, see the *man Pages(3): Library Routines.*

Using `priocntl`(2)

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

The `priocntl`(2) function provides an interface for specifying a realtime process, a set of processes, or a class to which the system call will apply. The `priocntlset`(2) system call also provides the more general interface for specifying an entire set of processes to which the system call is to apply.

The *idtype* and *id* arguments are used together to specify the set of processes on the queue. Depending upon the value of *idtype*, *id* can have values for a single process ID, a parent process ID, a process group ID, a session ID, a class ID, a user ID, a group ID, or a lightweight process ID.

The command arguments of `priocntl` 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 process or 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 realtime 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.

Utilities that Control Scheduling

The administrative utilities that control process scheduling are `dispadmin`(1M) and `priocntl`(1). Both these utilities support the `priocntl`(2) system call with compatible options and loadable modules. Using these utilities provides system administration functions that control realtime process scheduling during runtime. For more details about using these utilities, see the *man Pages(1): User Commands* and the *System Administration Guide, Volume II* guide.

Using `priocntl`**(1)**

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

Using `dispadmin`*(1M)*

The `dispadmin`(1M) 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 realtime class.

The following options are also available:

Table 7-2 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	when using –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 which loads the `rt_dptbl`.
- A `rt_dptbl` module that provides the dispatch parameters and a routine to return pointers to `config_rt_dptbl`.

- The `dispadmin` executable.

Then load the class specific module with the following command, where <module_name> is the class specific module.

```
modload /kernel/sched/<module_name>
```

Then 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 each scheduling class is a parameter table, `config_rt_dptbl` (RT), and `config_ts_dptbl` (TS). These tables are configurable by using a loadable module at boot time, or with `dispadmin`(1M) during runtime.

The Dispatcher Parameter Table

The in-core table for realtime establishes the properties for RT scheduling. The `config_rt_dptbl` structure consists of an array of parameters, `struct rt_dpent`, one for each of the `n` priority levels. The properties of a given priority level `i` are specified by the `i`th parameter structure in the array, `config_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`(1M).

`rt_quantum`
: The length of the time quantum allocated to processes at this level in ticks (see "Timestamp Functions" on page 145). 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 realtime administrator can change the behavior of the realtime portion of the scheduler by reconfiguring the config_rt_dptbl at any time. Two methods are described here.

The first method is to reconfigure the config_rt_dptbl parameter table with a loadable module which contains a new dispatch table loaded at boot time. The module containing the dispatch table is a separate module. This is the only method that can be used to change the number of realtime priority levels or the set of global scheduling priorities used by the realtime class. Note that changing the config_rt_dptbl affects the realtime processes that you set after the table gets updated.

A second method for examining or modifying the realtime parameter table on a running system is through using the dispadmin(1M) command. Invoking dispadmin for the realtime 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 the manual page for config_rt_dptbl found in the *man Pages(1M): System Administration Commands.*

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

Memory Locking

Locking memory is one of the most important issues for realtime applications. In a realtime 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 realtime applications in SunOS 5.x. For more details about using memory management functions and calls, see the *man Pages(3): Library Routines* for pertinent manual pages.

Overview

Under SunOS 5.x, 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 realtime process. In a realtime environment, a process must be able to guarantee memory residence for all or part of itself to reduce its memory access and dispatch latency.

For realtime in SunOS 5.x, 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`(3) 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`(3). 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`(3). 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. With SunOS 5.x, several functions and calls are available within the libraries supplied to perform fast, asynchronous I/O operations. For robustness, SunOS provides file synchronization operations and modes to prevent information loss and data inconsistency. See the *man Pages(3): Library Routines* for more detailed information.

Asynchronous I/O

Standard `UNIX` I/O is generally synchronous to the application programmer. An application that calls `read`(2) or `write`(2) is not usually allowed to proceed until that system call has finished, successfully or otherwise.

Realtime applications need *asynchronous* bounded I/O behavior. A process that issues an asynchronous I/O call does not wait until the I/O operation has been completed before it is allowed to proceed. Instead, the caller is notified that the I/O operation has finished at a later time while the process is doing something else.

Asynchronous I/O applies to any SunOS file. Files are opened in the synchronous way and no special flagging is required. An asynchronous I/O transfer is composed of 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 itself.

Asynchronous and standard I/O requests can be intermingled on any file descriptor. Note, however, that the system does not necessarily maintain any particular sequence of read and write requests. The system can and does arbitrarily resequence any and all pending read and write requests. If a specific sequence is required for the application, the application must assure the completion of prior operations before issuing the dependent requests.

Notification (SIGIO)

When an asynchronous I/O call returns successfully, the I/O operation has only been placed on the queue, waiting to be done. The actual operation also has a return value and a potential error identifier. These are 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 {
    int 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 person with two or more asynchronous I/O operations pending has no certain way to determine which request or even whether either request is the cause of the SIGIO signal. A process receiving a SIGIO should check all its conditions which could be generating the SIGIO signal.

Using aioread(3)

The aioread(3) function is the asynchronous version of read(2). In addition to the normal read arguments, aioread takes the arguments specifying a file position and the address of an aio_result_t structure at which the system is to store the result information about the operation.

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

Using aiowrite(3)

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

The file position specifies a seek to be performed within the file before the operation. If the aiowrite 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.

Using aiocancel(3)

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

Using aiowait(3)

A call to the `aiowait`(3) function 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. The `aiowait` function returns a pointer to the `aio_result_t` structure for the completed operation.

Using poll(2)

When you prefer to poll devices rather than to depend on a `SIGIO` interrupt, use the `poll`(2) system call. You can also poll to determine the origin of an `SIGIO` interrupt.

Using close(2)

Files are closed by a call to `close`(2). The call to `close` cancels any outstanding asynchronous I/O request that can be cancelled. The `close` function waits on an operation that cannot be cancelled. When a call to `close` returns, there is no asynchronous I/O pending for the file descriptor.

Only asynchronous I/O requests that are queued to the specified file descriptor are cancelled when a file is closed. Any I/O requests that are pending for other file descriptors are not cancelled.

Synchronized I/O

Applications may 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.x, data is successfully transferred for a write operation to a regular file when the system ensures that all data written is readable on any

subsequent open of the file (even one that follows a system or power failure) in the absence of a failure of the physical storage medium. Data is successfully

transferred for a read operation 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 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 if 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 successfully 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(3R)` functions explicitly synchronize a file to secondary storage:

```
int fsync (int fildes);
int fdatasync (int fildes);
```

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

Applications can arrange that each I/O operation is synchronized before the operation completes. Setting the `O_DSYNC` flag on the file description via `open(2)` or `fcntl(2)` ensures that all I/O writes `(write(2), aiowrite(3))` 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 I/O file 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), aioread(3))` 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.x as they relate to realtime 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 three, “Interprocess Communication,” and the *man Pages(3): Library Routines*.

Overview

Realtime 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 can be 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 may be used to send a small amount of information between processes. The sender can use the sigqueue(3R) function to send a signal together with a small amount of information to a target process:

```
int sigqueue(pid_t pid, int signo, const union sigval value);
union sigval {
    int     sival_int;      /* integer value */
    void    *sival_ptr;     /* pointer value */
};
```

The target process must have the SA_SIGINFO bit set for the given signal number (see sigaction(2)), in order that occurrences of the signal occurring when that signal is already pending will be queued.

The target process can receive the signals either synchronously or asynchronously. By leaving that signal blocked (sigprocmask(2)) and calling either sigwaitinfo(3R) or sigtimedwait(3R), the signal will be received synchronously, with the value sent by the caller of sigqueue() being stored in the si_value member of the siginfo_t argument. By leaving the signal unblocked, the arrival will be delivered to the signal handler specified by sigaction(), 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(). Thereafter, a call to sigqueue() either successfully enqueues at the target process or fails within a bounded amount of time.

Pipes

Pipes provide one-way communication between processes. Pipes are created by a process using the pipe(2) system call. The pipe(2) system call returns two file descriptors, the first for reading and the second for writing. Once the pipe is created, the process must create other processes with the `fork`(2) system call, which allows the processes to communicate among themselves. 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. Data is sent into the pipe by calls to `write`(2V) using the writing file descriptor.

Data is received from the pipe by calls to `read`(2V) using the reading file descriptor. The `read` call is usually a blocking function: it does not return to the caller until some data can be returned. To get a non-blocking `read`, the pipe can be set so that it doesn't block by using the `ioctl`(2) or `fcntl`(2) functions.

A `read` on an empty, non-blocking pipe returns with an indication that no data is available.

Named Pipes

SunOS 5.x provides *named pipes* or *FIFO*s. 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. A FIFO can be created with `mknod`(2).

A process connects to a FIFO through a call to `open`(2V). A process that opens a FIFO for a read is blocked until that FIFO has been opened by a process for writing. The decision about whether or not reads block is made in the `open` call or by using a subsequent call to `fcntl`.

As with pipes, data in a FIFO is treated as a byte stream. Input is obtained from a FIFO with calls to `read` and output is sent with calls to `write`. A process ends use of a FIFO through a call to `close`(2).

IPC Message Queues

IPC message queues provide a powerful means of communicating between processes by allowing any number of processes to send and receive from the same message queue. Messages are passed as blocks of arbitrary size, not as byte streams. Each message includes an integer type, which can be used by application convention as a message priority, or as message categories. The latter usage provides multiple flows of messages with a single message queue. This can be simpler than opening an arbitrary number of pipes or FIFOs when a large number are required. Note that IPC insertion is strictly FIFO.

IPC message queue structures are initiated by a call to `msgget`(2). A message is sent by a call to `msgsnd`(2), and `msgrcv`(2) is called to extract a message from the queue structure. The `msgctl`(2) system call controls various functions on a message queue structure, including removal.

IPC Semaphores

The IPC semaphore is a mechanism that synchronizes access to shared resources. IPC semaphores are created in arrays, each element of which can be used to control the execution of processes that call for operations on the array elements.

Create an array of IPC semaphores with a call to `semget`(2). Query or set individual semaphores or the complete array of semaphores with calls to `semctl`(2). Acquire and release a semaphore or the array of semaphores with calls to `semop`(2). Look in `intro`(2) for more information about information structures and the operation of IPC semaphores.

Note that using IPC 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 processes wishing to communicate. Applications use stores to send data and fetches to receive communicated data. SunOS 5.x provides two mechanisms for shared memory: memory mapped files and IPC 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 138 for more information.

Memory Mapped Files

The system call mmap(2) connects a shared memory segment to the caller's memory. 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.

The mmap(2) system call can also be used to map a file or a segment of a file to a process's memory. While this technique is very convenient in some applications, it is easy to forget that any access to the mapped file segment might result in implicit I/O. This can make an otherwise bounded process have unpredictable response times. The function msync(3) forces immediate or eventual copies of the specified memory segment to its permanent storage location(s).

The process can later change the access protection of the segment by the system call mprotect(2). The segment is specified by address and length.

The system call munmap(2) disconnects a mapped memory segment. The segment is specified by address and length.

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.

IPC Shared Memory

A shmget(2) call can be used either to create and obtain a shared memory segment or to obtain an existing shared memory segment. The call specifies an identifying key, the size of the segment, and a flag parameter. The flags contain the usual access permission bits and can contain a flag to create a new segment. The shmget function returns an identifier that is analogous to a file identifier.

The shared memory segment is made accessible to the process by a call to shmat(2). The shared memory segment becomes a virtual segment of the process memory space and can be freely written to and read from depending on creating permissions. The shared memory segment is detached from a process's memory space by a call to `shmdt`(2). The `shmctl` system call can be used to control a variety of functions on an IPC shared memory object, including removal.

Shared Memory Synchronization

In sharing memory, a portion of memory can be mapped into the address space of one or more processes. This allows shared access to that portion of memory by the attached processes. No method of coordinating access is automatically provided, so nothing prevents two processes from writing to the shared memory at the same time. For this reason, it is typically used with semaphores, which are used to synchronize processes.

Choice of IPC Mechanism

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.x 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 discusses the techniques of asynchronous network communication using Transport-Level Interface (TLI) for realtime applications. SunOS provides support for asynchronous network processing of TLI events using a combination of STREAMS asynchronous features and the non-blocking mode of the TLI library routines.

For more information on the Transport-Level Interface, see the *Transport Interfaces Programming Guide* and the*man Pages(3): Library Routines*.

Modes of Networking

The Transport-Level Interface provides two modes of service: *connection-mode* and *connectionless-mode*.

Connection-Mode Service

The *connection-mode* 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 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.

Networking Programming Models

Like file and device I/O, network transfers can be done synchronously or asynchronously with process service requests.

Synchronous Networking

Synchronous networking proceeds similarly to synchronous file and device I/O. Like the `write`(2) function, the request to send returns after buffering the message, but might suspend the calling process if buffer space is not immediately available. Like the `read`(2) function, a request to receive suspends execution of the calling process until data arrives to satisfy the request. Because SunOS 5.x provides no guaranteed bounds for transport services, synchronous networking is inappropriate for processes that must have realtime behavior with respect to other devices.

Asynchronous Networking

Asynchronous networking is provided by non-blocking service requests. Additionally, applications can request asynchronous notification when a connection might be established, when data might be sent, or when data might be received.

Asynchronous Connectionless-Mode Service

Asynchronous connectionless mode networking is conducted by configuring the endpoint for non-blocking service, and either polling for or receiving asynchronous notification when data might be transferred. If asynchronous notification is used, the actual receipt of data typically takes place within a signal handler.

Making the Endpoint Asynchronous

After the endpoint has been established using `t_open`(3), and its identity established using `t_bind`(3), the endpoint can be configured for asynchronous service. This is done by using the `fcntl`(2) function to set the `O_NONBLOCK` flag on the endpoint. Thereafter, calls to `t_sndudata`(3) for which no buffer space is immediately available return `-1` with `t_errno` set to `TFLOW`. Likewise, calls to `t_rcvudata`(3) for which no data are available return `-1` with `t_errno` set to `TNODATA`.

Asynchronous Network Transfers

Although an application can use the `poll`(2) function to wait for the receipt of data on an endpoint, it might be necessary to receive asynchronous notification when data has arrived. This can be done by using the `ioctl`(2) function with the `I_SETSIG` command to request that a `SIGPOLL` signal be sent to the process upon receipt of data at the endpoint. Applications should check for the possibility of multiple messages causing a single signal.

In the following example, `protocol` is the name of the application-chosen transport protocol.

Code Example 7-1 Asynchornous Network Transfers

```
#include <sys/types.h>
#include <tiuser.h>
#include <signal.h>
```

```
#include <stropts.h>

int             fd;
struct t_bind   *bind;
void            sigpoll(int);

    fd = t_open(protocol, O_RDWR, (struct t_info *) NULL);

    bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
    ...     /* set up binding address */
    t_bind(fd, bind, bin

    /* make endpoint non-blocking */
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

    /* establish signal handler for SIGPOLL */
    signal(SIGPOLL, sigpoll);

    /* request SIGPOLL signal when receive data is available */
    ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

    ...

void sigpoll(int sig)
{
    int                 flags;
    struct t_unitdata  ud;

    for (;;) {
        ... /* initialize ud */
        if (t_rcvudata(fd, &ud, &flags) < 0) {
            if (t_errno == TNODATA)
                break;  /* no more messages */
            ... /* process other error conditions */
    }
    ... /* process message in ud */
}
```

Asynchronous Connection-Mode Service

For connection-mode service, an application can arrange for not only the data transfer, but for the establishment of the connection itself to be done asynchronously. The sequence of operations depends on whether the process is attempting to connect to another process or is awaiting connection attempts.

Asynchronously Establishing a Connection

A process can attempt a connection and asynchronously complete the connection. The process first creates the connecting endpoint, and, using `fcntl`(), configures the endpoint for non-blocking operation. As with connectionless data transfers, the endpoint can also be configured for asynchronous notification upon completion of the connection and subsequent data transfers. The connecting process then uses the `t_connect`(3) function to initiate setting up the transfer. Then the `t_rcvconnect`(3) function is used to confirm the establishment of the connection.

Asynchronous Use of a Connection

To asynchronously await connections, a process first establishes a non-blocking endpoint bound to a service address. When either the result of `poll`() or an asynchronous notification indicates that a connection request has arrived, the process can get the connection request by using the `t_listen`(3) function. To accept the connection, the process uses the `t_accept`(3) function. The responding endpoint must be separately configured for asynchronous data transfers.

The following example illustrates how to request a connection asynchronously

Code Example 7-2 Asynchronous Connection Request

```
#include <tiuser.h>

int             fd;
struct t_call   *call;

    fd = .../* establish a non-blocking endpoint */

    call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR);
    .../* initialize call structure */
    t_connect(fd, call, call);

    /* connection request is now proceeding asynchronously */

    .../* receive indication that connection has been accepted */
    t_rcvconnect(fd, &call);
```

The following example illustrates listening for connections asynchronously

Code Example 7-3 Asynchronous LIstening

```
#include <tiuser.h>

int             fd, res_fd;
struct t_call   call;

    fd = ... /* establish non-blocking endpoint */

    .../*receive indication that connection request has arrived */
    call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
    t_listen(fd, &call);

    .../* determine whether or not to accept connection */
    res_fd = ... /* establish non-blocking endpoint for response */
    t_accept(fd, res_fd, call);
```

Asynchronous Open

Occasionally, an application might be required to dynamically open a regular file in a file system mounted from a remote host, or on a device whose initialization might be prolonged. However, while such an open is in progress, the application would be unable to achieve realtime response to other events. Fortunately, SunOS 5.x provides a means of solving this problem by having a second process perform the actual open and then pass the file descriptor to the realtime process.

Transferring a File Descriptor

The STREAMS interface under SunOS 5.x provides a mechanism for passing an open file descriptor from one process to another. The process with the open file descriptor uses the `ioctl`(2) function with a command argument of `I_SENDFD`. The second process obtains the file descriptor by calling the `ioctl`() function with a command argument of `I_RECVFD`.

In this example, the parent process prints out information about the test file, and creates a pipe. Next, the parent creates a child process, which opens the test file, and passes the open file descriptor back to the parent through the pipe. The parent process then displays the status information on the new file descriptor.

Code Example 7-4 File Descriptor Transfer

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include <stdio.h>

#define TESTFILE "/dev/null"
main(int argc, char * argv)
{
    int fd;
    int pipefd[2];
    struct stat statbuf;

    stat(TESTFILE, &statbuf);
    statout(TESTFILE, &statbuf);
    pipe(pipefd);
    if (fork() == 0) {
        close(pipefd[0]);
        sendfd(pipefd[1]);
    } else {
        close(pipefd[1])
        recvfd(pipefd[0]);
    }
}

sendfd(int p)
{
    int tfd;

    tfd = open(TESTFILE, O_RDWR);
    ioctl(p, I_SENDFD, tfd);
}

recvfd(int p)
{
    struct strrecvfd rfdbuf;
    struct stat statbuf;
    char        fdbuf[32];

    ioctl(p, I_RECVFD, &rfdbuf);
    fstat(rfdbuf.fd, &statbuf);
    sprintf(fdbuf, "recvfd=%d", rfdbuf.fd);
    statout(fdbuf, &statbuf);
}
```

```
statout(char *f, struct stat *s)
{
    printf("stat: from=%s mode=0%o, ino=%d, dev=%d, rdev=%d\n",
        f, s->st_mode, s->st_ino, s->st_dev, s->st_rdev);
    fflush(stdout);
}
```

Timers

This section describes the timing facilities available for realtime applications under SunOS 5.x. Realtime applications that want to take advantage of these mechanisms will require detailed information from the manual pages of the routines listed in this section. These can be found in the *man Pages(3): Library Routines.*

The timing functions of SunOS 5.x 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`(2) 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`().

SunOS 5.x 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.x to provide timestamps accurate to one microsecond.

Interval Timer Functions

Realtime applications often schedule their activities through the use of interval timers. Interval timers can be either of two types: a "*one-shot*" type or a "*periodic*" type.

The one-shot is an armed timer that is set with an initial expiration time relative either to current time or to an absolute time. This timer expires once and is then disarmed. Such a timer might be useful for clearing buffers after the data has been transferred to storage, or to time-out an operation that should have finished.

The periodic timer is armed with the 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 the timer is automatically rearmed. This timer might be 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 periodic timer interval (typically 10 ms).

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.

There are twos set of timers interfaces in SunOS 5.x. The `setitimer(2)` and `getitimer(2)` interfaces provide access to fixed set timers, called the BSD timers, using the timeval structure to specify time intervals. The POSIX timers are specifically related to POSIX clocks; the only POSIX clock currently supported is CLOCK_REALTIME. POSIX timer operations are expressed in terms of the timespec structure.

The functions `getitimer(2)` and `setitimer(2)` respectively retrieve and establish the value of the specified BSD interval timer. There are three BSD interval timers available to a process, including a realtime 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.

The `timer_create(3R)` function can create up to {TIMER_MAX} POSIX timers. At the time of creation, the caller can specify what signal and what associated value will be sent to the process upon timer expiration. The `timer_gettime(3R)` and `timer_settime(3R)` functions respectively retrieve and establish the value of the specified POSIX interval timer.

Expirations of POSIX timers while the required signal is pending delivery are counted, and the function `timer_getoverrun(3R)` retrieves the count of such expirations. The function `timer_delete(3R)` deallocates a POSIX timer.

Code Example 7-5 illustrates how to use the `setitimer` interface to generate a periodic interrupt, and how to control the arrival of timer interrupts.

Code Example 7-5 Controlling Timer Interrupts

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

#define TIMERCNT 8

voidtimerhandler();
int timercnt;
structtimeval alarmtimes[TIMERCNT];

main()
{
    struct itimerval times;
    sigset_tsigset;
    int     i, ret;
    struct sigaction act;

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

    /* set up handler for SIGALRM */
    act.sa_handler = 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, 0, NULL, NULL);
    while (timercnt < TIMERCNT) {
        ret = sigsuspend(&sigset);
    }
    printtimes();
}

void timerhandler(sig, siginfo, context)
    int     sig;
    siginfo_tsiginfo;
    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("%d.%06d\n", alarmtimes[i].tv_sec,
                alarmtimes[i].tv_usec);
    }
}
```

Full Code Examples

A

Code Example A-1 Sample Program to Illustrate msgget()

```
/*
 * 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("%li", &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-2 Sample Program to Illustrate `msgctl()`

```
/*
 * 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_dsbuf;    /* 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 intrtrn;/* 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-3 Sample Program to Illustrate `msgsnd()` and `msgrcv()`

```
/*
 * 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 intask();
extern voidexit();
extern char*malloc();
extern voidperror();

charfirst_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:
     *      charmtext[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((unsigned)(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-4 Sample Program to Illustrate `semget()`

```
/*
 * 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-5 Sample Program to Illustrate `semctl()`

```
/*
 * 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-6 Sample Program to Illustrate semop()

```
/*
 * 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_dssemid_ds;      /* status of semaphore set */

static char    error_mesg1[] = "semop: Can't allocate space for %d\
                               semaphore values. Giving up.\n";
static charerror_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-7 Sample Program to Illustrate `shmget()`

```
/*
 * 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-8 Sample Program to Illustrate `shmctl()`

```
/*
 * 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-9 Sample Program to Illustrate `shmat()` and `shmdt()`

```
/*
 * 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 voidcatcher();
extern voidexit();
static     good_addr();
extern voidperror();
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_bufsegvbuf; /* 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);
}
```

Code Example A-10 Example of Record Locking With Lock Promotion

The next example demonstrates inserting 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 has a read lock on it already. 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` 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` 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.

```
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-11 Record Write Locks With `lockf()`

```
/* 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);
}
```

Index

O

P

R

S

T

U

V

W

Z

Adobe PostScript