DEC OSF/1

Guide to Realtime Programming

Order Number: AA–PJUVA–TE
January 1992

Product Version: DEC OSF/1 Version 1.0 or higher

This guide describes how to use POSIX 1003.4 Draft 10 (P1003.4/D10) functions and interprocess communication functions to write realtime applications that run on DEC OSF/1 systems. This guide is intended for experienced application programmers.

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, January 1992

The Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

ML–S1690

This document is available on CDROM

This document was prepared with VAX DOCUMENT, Version 1.2.

Contents

2 Process Scheduling and Priorities

3 Clocks and Timers

7 Messages

8 Shared Memory

9 Semaphores

10 Pipes

11 Signals

A DEC OSF/1 Realtime Functional Summary

Index

Examples

Figures

Tables

About This Guide

This guide is designed for programmers who are using DEC OSF/1 compilers. Users may be writing new realtime applications or they may be porting existing realtime applications from other systems.

Purpose of this Guide

This guide explains how to use POSIX 1003.4 Draft 10 (P1003.4/D10) functions in combination with other system and library functions to write realtime applications. This manual does not attempt to teach programmers how to write applications.

The audience for this manual is the application programmer or system engineer who is already familiar with the C programming language. The audience using realtime features is expected to have experience with UNIX operating systems. They also should have experience with UNIX program development tools.

This manual does not present function syntax or reference information. The online reference pages present syntax and explanations of these functions.

Structure of this Guide

This manual consists of eleven chapters and one appendix, organized as follows:

- Chapter 1, Introduction to Realtime Programming, describes the realtime functionality supported by the DEC OSF/1 operating system.
- Chapter 2, Process Scheduling and Priorities, describes use of the P1003.4/D10 functions to determine and set priority for processes in your application. This chapter also describes the priority scheduling policies provided by the DEC OSF/1 operating system.
- Chapter 3, Clocks and Timers, describes use of the P1003.4/D10 functions for constructing and using high-resolution clocks and timers.

- Chapter 4, Memory Locking, describes the use of P1003.4/D10 functions for locking and unlocking memory.
- Chapter 5, Asynchronous Input and Output, describes the use of P1003.4 /D10 functions for asynchronous input and output.
- Chapter 6, Interprocess Communication Overview, provides a general introduction to Interprocess Communication (IPC) and in particular, an overview of System V IPC.
- Chapter 7, Messages, describes the creation and use of message queues for interprocess communication and synchronization in realtime applications.
- Chapter 8, Shared Memory, describes the creation and use of shared memory areas for interprocess communication.
- Chapter 9, Semaphores, describes the creation and use of semaphores for interprocess synchronization. An example illustrates how to use semaphores and shared memory in combination.
- Chapter 10, Pipes, describes the creation and use of pipes and named pipes for interprocess communication.
- Chapter 11, Signals, describes the creation and use of POSIX 1003.1 signals for interprocess communication. This chapter also discusses asynchronous signals used for some P1003.4/D10 functions.
- Appendix A, DEC OSF/1 Realtime Functional Summary, provides a table of commands and functions useful for realtime application development.

Related Documents

The following documents are relevant to writing realtime applications:

- *OSF/1 Application Programmer's Guide*
- *POSIX Conformance Document*
- *The C Programming Language* by Kernighan and Ritchie
- *Guide to Developing International Software*
- Online *Reference Pages*

To view online reference pages for the P1003.4/D10 functions, use the `man` or `whatis` commands, described in Section 4.

Using the man Command

System commands and library functions (including P1003.4/D10 functions) have no printed reference material. Instead, the information is shipped on the system software kit and can be accessed through the `man` command. The `man` command provides online displays of the reference pages. You can use options to direct the `man` command to display online summaries of specific reference pages, to use special formatting when preparing the reference page for viewing or printing, and to search alternate reference page directories for specified reference pages.

Use the `man` command to access the online reference pages for the P1003.4/D10 functions discussed in this manual. If you need help on using the `man` command, use the following command:

```
# man man
```

If you do not specify an option, the `man` command formats and displays one or more specified reference pages. If multiple reference pages match a specified name, only the first matching reference page is displayed. If there are multiple matches in one section for a specified name, the matching page in the first alphabetically occurring subsection is displayed.

Conventions

The following conventions are used in this manual:

Convention	Meaning
%	The default user prompt is the user's system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign is the default `superuser` prompt.
>> CPU*nn*>>	The console subsystem prompt is two right angle brackets. On a system with more than one central processing unit (CPU) the prompt displays two numbers: the number of the CPU, and the number of the processor slot containing the board for that CPU.
user input	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	In text, this typeface indicates the exact name of a command, option, partition, pathname, directory, or file. This typeface is used in interactive examples to indicate system output. It is also used in code examples and other screen displays.

Convention	Meaning
variable	This typeface indicates variable information, such as user-supplied information in commands, syntax, or example text.
...	Horizontal ellipsis indicates that the preceding item can be repeated one or more times. It is used in syntax descriptions and function definitions.
. . .	Vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
UPPERCASE lowercase	The system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
`cat`(1)	Cross-references to the online reference pages include the appropriate section number in parentheses. For example, a reference to `cat`(1) indicates that you can find the material on the `cat` command in Section 1 of the online reference pages.

1

Introduction to Realtime Programming

A realtime application is one in which the system's timely and predictable response to external events is critical. To accomplish this, a realtime system provides features for efficient interprocess communication and synchronization, a fast interrupt response, fast input and output (I/O), and an efficient memory-locking scheme.

This chapter includes the following sections:

- Realtime Overview, Section 1.1
- Realtime System Capabilities, Section 1.2
- Realtime Process Synchronization, Section 1.3
- Standards, Section 1.4

1.1 Realtime Overview

A realtime system recognizes and responds to asynchronous external events within a predictable amount of time, and may process and store large amounts of data. Realtime applications provide an action or an answer to an external event in a timely and predictable manner. Failure to provide the action or answer within the predicted amount of time can lead to catastrophic consequences. An unpredictable realtime application can result in loss of data, loss of deadlines, or loss of plant production. Examples of realtime applications include process control, factory automation robotics, vehicle simulation, scientific data acquisition, image processing, built-in test equipment, music or voice synthesis, and analysis of high energy physics.

While typical realtime applications require high speed, they cover a wide range of time dependencies. A "timely and predictable manner" has a different definition in each application. What may be fast in one application may be slow or late in another. For example, an experimenter in high energy physics needs to collect data in microseconds while a meterologist monitoring the environment might need to collect data in intervals of several minutes. However, both applications need a predictable and reliable response time.

Realtime application designers code to handle the requirements of response time and data throughput appropriate for the realtime application. Not all realtime applications require a fast response time, nor do all process large quantities of data. The term "realtime" does not necessarily imply high speed. An environmental monitoring application is a realtime application that may not imply high speed. It might require that readings of wind speed and direction and environmental pollutants be taken at 10-minute intervals. Though this is not a demanding realtime requirement, if a reading were missed, it would be impossible to recover lost data.

Realtime applications are classified as either hard realtime or soft realtime. Hard realtime applications require very fast response within a predetermined amount of time for the application to function properly. If response time fails to meet the specified deadline, the application fails. An example of a hard realtime application is a missile guidance control system where a late response to a needed correction might lead to disaster.

Soft realtime applications also require a very fast response time, but the application does not fail if a deadline is missed. Soft realtime applications sometimes process large amounts of data. An example of a soft realtime application is an airline reservation system where an occasional delay is tolerable.

Often, realtime applications must respond within a specified time to events generated by equipment. For example, in a car's automatic braking system, a realtime application must sense that one or more wheels are locked and, within a very small window of time, signal for the appropriate wheel brakes to release momentarily and to correct the condition that caused them to lock.

Most realtime applications require high I/O throughput and fast response time to asynchronous external events. The ability to process and store large amounts of data is the key metric for data collection applications. Realtime applications that require high I/O throughput rely on continuous processing of large amounts of data. The primary requirement of such an application is to acquire a number of data points equally spaced in time.

High data throughput requirements are typically found in signal-processing applications such as:

- Sonar and radar analysis
- Telemetry
- Vibration analysis
- Speech analysis

- Music synthesis

Likewise, a continuous stream of data points must be acquired for many of the qualitative and quantitative methods used in the following types of applications:

- Gas and liquid chromatography
- Mass spectrometry
- Automatic titration
- Colorimetry

For some applications, the throughput requirements on any single channel are modest. However, an application may need to handle multiple data channels simultaneously, resulting in a high aggregate throughput. Realtime applications, such as medical diagnosis systems, need a response time of around 1 second while simultaneously handling data from, perhaps, ten external sources.

Although high I/O throughput may be important for realtime control systems, another key metric is the speed at which the application responds to asynchronous external events and its ability to schedule and provide communication between multiple tasks. Realtime applications must capture input parameters, perform decision-making operations, and compute updated output parameters within a given time frame.

Some realtime applications, such as flight simulation programs, require a response time of microseconds while simultaneously handling data from a large number of external sources. The application might acquire several hundred input parameters from the cockpit controls; compute updated position, orientation, and speed parameters; and then send several hundred output parameters to the cockpit console and a visual display subsystem.

The key to realtime is the application's ability to react to interrupts within a short period of time. All specified conditions and data must be handled correctly within a predetermined amount of time. This means that the application must respond to an interrupt regardless of what the lower-priority task may be doing at the time of the interrupt. Although response time requirements may vary, realtime features provide efficient interrupt handling and high data throughput to meet the requirements of time-critical applications.

1.2 DEC OSF/1 Realtime System Capabilities

The DEC OSF/1 operating system supports facilities to enhance the performance of realtime applications. DEC OSF/1 realtime facilities make it possible for the operating system to guarantee that the realtime application has access to resources whenever it needs them and for as long as it needs them. That is, the realtime applications running on the DEC OSF/1 operating system can respond to external events regardless of the impact on other executing tasks or processes.

The realtime applications written to run on the DEC OSF/1 operating system make use of and rely on the following system capabilities:

- A preemptive kernel
- Fixed-priority scheduling policies
- Realtime clocks and timers
- Memory locking
- Asynchronous I/O
- Process communication facilities

All of these realtime facilities work together to form the DEC OSF/1 realtime environment. In addition, realtime applications make full use of process synchronization techniques and facilties, as summarized in Section 1.3.

1.2.1 The Value of a Preemptive Kernel

A realtime environment must be able to respond to an event within a bounded (generally quite short) period of time. A preemptive kernel guarantees that a higher-priority process can quickly interrupt a lower-priority process, regardless of whether the low-priority process is in user or kernel mode. Whenever a higher-priority process becomes runnable, a preemption is requested, which causes the higher-priority process to displace the running, lower-priority process. A preemptive kernel guarantees a deterministic response to realtime events by providing the ability to respond to realtime requests.

Every realtime application interacts with the operating system in two modes: user mode and kernel mode. User mode processes allow the application user to interact with the application. User mode processes call utilities, library functions, and other user applications.

In kernel mode, the application accesses and interacts with the operating system. During execution, a user process often calls system functions, switching the context from user to kernel mode.

The amount of time it takes for a higher-priority process to displace a lower-priority process is referred to as Process Preemption Latency. In a realtime environment, the primary concern of application designers is the Maximum Process Preemption Latency that can occur at runtime. Designers must understand system timing contraints before designing time-critical applications.

1.2.1.1 Nonpreemptive Kernel

By definition a nonpreemptive kernel, such as the standard UNIX kernel, does not allow a user process to preempt a process executing in kernel mode. Once a running process issues a system call and enters kernel mode, preemptive context switches are disabled until the system call is completed. Although there are voluntary context switches, a system call may take an arbitrarily long time to execute without voluntarily giving up the processor. During that time, the process that has made the system call may be holding up the execution of a higher-priority, runnable, realtime process.

The Maximum Process Preemption Latency for a nonpreemptive kernel is the maximum amount of time it can take for the running process to switch out of kernel mode back into user mode and then be preempted by a higher-priority process. Under these conditions it is not unusual for worst-case preemption to take seconds, which is clearly unacceptable for many realtime applications.

1.2.1.2 Preemptive Kernel

A preemptive kernel, such as the DEC OSF/1 realtime kernel, allows the operating system to respond as quickly as possible to a process preemption request. The DEC OSF/1 kernel can break out of kernel mode to honor the preemption request.

A preemptive kernel supports the concept of process synchronization, while maintaining data integrity, with the ability to respond quickly to interrupts. The kernel employs mechanisms to protect the integrity of kernel data structures and defines the restrictions on where the kernel cannot preempt execution.

The Maximum Process Preemption Latency for a preemptive kernel is exactly the amount of time required to preserve system and data integrity and preempt the running process. Under these conditions it is not unusual for worst-case preemption to take milliseconds.

1.2.1.3 Comparing Latency

Figure 1–1 and Figure 1–2 illustrate the Process Preemption Latency that can be expected from a nonpreemptive kernel and a preemptive kernel. In both figures, a higher-priority, realtime process makes a preemption request, but the amount of elapsed time until the request is honored depends on the kernel. Latency is represented as the shaded area.

Figure 1–1 Nonpreemptive Kernel

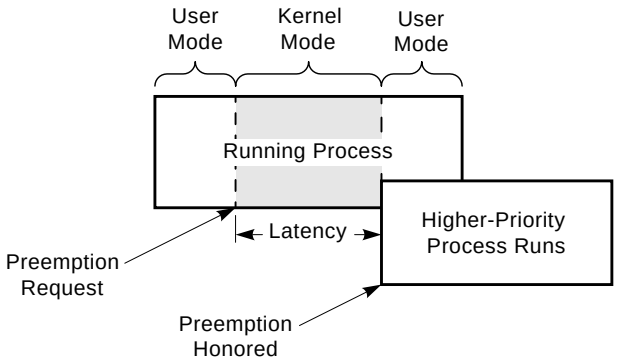
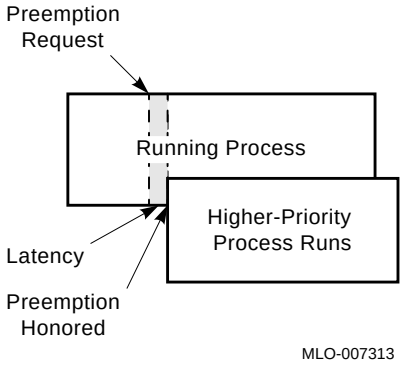


Figure 1–1 shows the expected latency of a nonpreemptive kernel. In this situation, the currently running process moves back and forth between user and kernel mode as it executes. The realtime process advances to the beginning of the priority process list, but cannot preempt the running process while it runs in kernel mode. The realtime process must wait until the running process either finishes executing or changes back to user mode before the realtime process is allowed to preempt the running process. The latency in this situation could be as long as seconds.

Figure 1–2 shows the expected latency of a preemptive kernel. In this situation the running process is quickly preempted and the realtime process takes its place on the run queue. With a preemptive kernel, latency is minimized and can be measured in terms of milliseconds.

Figure 1–2 Preemptive Kernel



1.2.2 Fixed-Priority Scheduling Policies

The scheduler determines how CPU resources are allocated to executing processes. Each process has a priority that associates the process with a run queue. Each process starts out with an initial priority that can change as the application executes depending on the algorithm used by the scheduler or application requirements.

The algorithm or set of rules that governs how the scheduler selects runnable processes, how processes are queued, and how much time each process is given to run is called a scheduling policy. Scheduling policies work in conjunction with priority levels. Generally speaking, the higher a process's priority, the more frequently the process is allowed to execute. But the scheduling policy may determine how long the process executes. The realtime application designer balances the nature of the work performed by the process with process's priority and scheduling policy to control use of system resources.

If realtime is installed on your system, the DEC OSF/1 operating system supports two distinctly different scheduling interfaces: the `nice` interface and the realtime interface. The `nice` interface provides functions for managing nonrealtime applications running at nonrealtime priority level. The `nice` interface uses the timesharing scheduling policy, which allows the scheduler to dynamically adjust priority levels of a process.

The DEC OSF/1 realtime interface supports a nonrealtime, timesharing scheduling policy and two fixed-priority, preemptive scheduling policies for realtime applications. Under the timesharing scheduling policy, process priorities are automatically adjusted by the scheduler. Under the fixed-priority scheduling policies, the scheduler will never automatically change the

priority of a process. Instead, the application designer determines when it is appropriate for a process to change priorities.

The realtime interface provides a number of functions to allow the realtime application designer to control process execution. In addition, realtime scheduling policies are attached to individual processes, giving the application designer control over individual processes.

POSIX scheduling policies have overlapping priority ranges: The highest priority range is reserved for realtime applications, the middle priority range is used by the operating system, and the lowest priority range is used for nonprivileged user processes. Realtime priority ranges loosely map to the `nice` priority range, but provide a wider range of priorities for a realtime process.

Not all realtime processes need to run in the realtime priority range. When using the realtime interface, each process starts execution under the timesharing scheduling policy with an associated timesharing priority. The application designer determines which processes are time-critical and under what circumstances processes should run at an elevated priority level. The application designer calls the P1003.4/D10 functions to set the appropriate priority and scheduling policy.

Under the first-in first-out (SCHED_FIFO) scheduling policy, a running process continues to execute if there are no other higher-priority processes. Under the control of the user, a running process can raise its priority to avoid being preempted by another process. Therefore, a high-priority, realtime process running under the first-in first-out scheduling policy can use system resources as long as necessary to finish realtime tasks.

Under the round-robin (SCHED_RR) scheduling policy, the highest-priority process runs until either its alloted time (quantum) is complete or the process is preempted by another, higher-priority process. A process continues to execute as long as the waiting processes are at a lower-priority. Therefore, high priority processes running under the round-robin scheduling policy can share the processor with other time-critical processes.

When a process raises its priority and preempts a running process, the scheduler saves the runtime context of the preempted process so that context can be restored once the process is allowed to run again. The preempted process remains in a runnable state even though it was preempted.

For information on using the priority and scheduling policy functions, refer to Chapter 2.

1.2.3 Realtime Clocks and Timers

The DEC OSF/1 systemwide clock, CLOCK_REALTIME[1], provides the timing base for per-process timers and is the primary source for timer synchronization. This clock maintains user and system time as well as the current time and date. The resolution of the CLOCK_REALTIME clock is such that it provides the basic mechanism to support realtime per-process timers and high resolution sleep.

Clock and timer functions allow you to retrieve and set the systemwide clock, retreive and correct for clock drift rate, suspend execution for a period of time, provide high resolution timers, and use asynchronous signal notification.

Timers in realtime applications must be able to respond quickly to asynchronous external events. Timers often schedule tasks and events in time increments considerably smaller than the traditional one-second timeframe. Because the CLOCK_REALTIME clock and realtime timers use seconds and nanoseconds as the basis for time intervals, the resolution for the system clock, realtime timers, and the `nanosleep` function has a fine granularity. For example, in a robotic data acquisition application, information retrieval and recalculation operations may need to be completed within a 4 milliseconds timeframe. Timers are created to fire every 4 milliseconds to trigger the collection of another round of data. On expiration, a timer sends a signal to the calling process.

Realtime timers must be flexible enough to allow the application to set timers based on either absolute or relative time. Furthermore, timers must be able to fire as a one-shot or periodic timer. The application creates timers in advance, but specifies timer characteristics when the timer is set.

Realtime applications must be able to establish and manipulate timers based on the needs of the application. Some applications may require only one or two timers; others may require multiple timers within a single process. The P1003.4/D10 timing facilities support multiple per-process timers up to a system-defined limit. Each timer is created and armed independently, which means that the application designer controls the action of each and every timer.

For information on using the clock and timer functions, refer to Chapter 3.

[1] CLOCK_REALTIME is the TIME-OF-DAY clock for DEC OSF/1

1.2.4 Memory Locking

A realtime application cannot afford long latencies in the execution of critical code. In a virtual memory system, a process may have part of its address space paged in and out of memory in response to system demands for critical space. A realtime application needs a mechanism to guarantee that time-critical processes are locked into memory and not subjected to memory management appropriate only for timesharing applications.

Memory locking is one of the primary tools available to the DEC OSF/1 realtime application designer to reduce latency. Without locking time-critical processes into memory, the latency introduced by paging would introduce involuntary and unpredictable time delays at runtime.

The P1003.4/D10 memory-locking functions allow the application designer to lock process address space into memory. The application can lock in not only the current address space, but also any future address space the process may use during execution.

For information on using the memory-locking functions, refer to Chapter 4.

1.2.5 Asynchronous I/O

DEC OSF/1 asynchronous I/O allows the calling process to immediately regain control of execution once an I/O operation is queued. This capability is desirable in many different applications ranging from graphics and file servers to dedicated realtime data acquisition and control systems. Without asynchronous I/O the process waits while I/O completes before continuing execution. With asynchronous I/O once an I/O request is queued, control is immediately returned to the calling process. The process immediately continues execution, thus overlapping tasks.

Many realtime applications need this ability to overlap application processing and I/O operations. Often, one process simultaneously performs multiple I/O functions while other processes continue execution. For example, a process can queue data for output without blocking (waiting for I/O completion). Applications need to gather large quantities of data from multiple channels within a short, bounded period of time. In such a situation, blocking I/O may work at cross purposes with application timing constraints. Asynchronous I/O performs nonblocking I/O, allowing simultaneous reads and writes, which frees processes for additional processing.

Notification of asynchronous I/O completion is optional. If you choose to use signal notification, the signal is specified in the `aiocb` structure, thereby eliminating the need to call the `signal` function and providing faster interprocess communication.

For information on using the asynchronous I/O functions, refer to Chapter 5.

1.2.6 Interprocess Communication

DEC OSF/1 interprocess communication facilities allow the application designer to synchonize independently executing processes by passing data within an application. Processes can pursue their own tasks until they must synchronize with other processes at some predetermined point. When they reach that point, they wait for some form of communication to occur. Interprocess communication can take any of the following forms:

- Messages, Chapter 7

  Messages consist of user-defined structures that specify the length and type of message as well as carry the message text.

- Shared memory, Chapter 8

  Shared memory is the fastest form of interprocess communication. As soon as one process writes data to the shared memory area, it is available by other processes using the same shared memory.

- Semaphores, Chapter 9

  Semaphores are most commonly used to control access to system resources, such as shared memory regions. Semaphores can be either binary or counting semaphores.

- Pipes, Chapter 10

  Pipes are used to transfer small amounts of data among related processes.

- Named pipes, Chapter 10

  Named pipes are like pipes, except that named pipes use file descriptors. Named pipes can communicate with only small numbers of unrelated processes.

- Signals, Chapter 11

  Signals provide a means to communicate to a large number of processes, but communication is limited to a signal number. Signals for timer expiration and asynchronous I/O use a data structure, making signal delivery asynchronous, fast, and reliable.

Some forms of interprocess communication are traditionally supplied by the operating system and some are specifically modified for use in realtime functions. All allow a user- or kernel-level process to communicate with a user-level process. Interprocess communication facilities are used to notify processes that an event has occurred or to trigger the process to respond to

an application-defined occurrence. Such occurrences can be asynchronous I/O completion, timer expiration, data arrival, or some other user-defined event.

In a realtime environment it is often necessary to reduce the time interval required for process communication. It is not always sufficient to simply verify that communication has taken place. A delay in interprocess communication can affect overall performance of the realtime application. To provide rapid signal communication on timer expiration and asynchronous I/O completion, these functions send signals via a `sigevent` structure rather than through the traditional nonrealtime signal mechanism. The application designer controls which signals are sent and establishes signal handlers appropriate for the event.

For information on using asynchronous signals for interprocess communication, refer to Chapter 11.

1.2.7 Realtime Needs and System Features

Table 1–1 summarizes the common realtime needs and the features or capabilities available through the P1003.4/D10 functions and the DEC OSF/1 operating system. The realtime needs, in the left column of the table, are ordered according to their requirement for fast system performance.

Table 1–1 Realtime Needs Summary

Realtime Need	Realtime Feature
Change the availability of a process for scheduling	Use the scheduler functions to set the scheduling policy and priority of the process
Keep critical code or data highly accessible	Use the memory locking functions to lock the process address space into memory
Perform an operation while another operation is in progress	Create a child process or separate thread
	Use asynchronous I/O
Perform I/O quickly or for special purposes	Use asynchronous I/O
Share code or data between processes	Use shared libraries
	Use shared memory
Synchronize access to resources shared between cooperating processes	Use binary or counting semaphores

(continued on next page)

Table 1–1 (Cont.) Realtime Needs Summary

Realtime Need	Realtime Feature
Communicate between processes	Use messages, semaphores, shared memory, signals, pipes, and names pipes
Synchronize a process with a time schedule	Set and arm per-process timers
Synchronize a process with an external event or program	Use signals
	Use semaphores
	Cause the process to sleep and to awaken when needed

1.3 Process Synchronization

Use of synchronization techniques and restricting access to resources can ensure that critical and noncritical tasks execute at appropriate times with the necessary resources available. Concurrently executing processes require special mechanisms to coordinate their interactions with other processes and their access to shared resources. In addition, processes may need to execute at specified intervals.

Realtime applications synchronize process execution through the following techniques:

- Waiting for a specified time
- Waiting for semaphores
- Waiting for communication
- Waiting for other processes

The basic mechanism of process synchronization is waiting. A process must synchronize its actions with the arrival of an absolute or relative time, or until a set of conditions is satisfied. Waiting is necessary when one process requires another process to complete a certain action, such as releasing a shared system resource, or allowing a specified amount of time to elapse, before processing can continue.

The point at which the continued execution of a process depends on the state of certain conditions is called the "synchronization point." Synchronization points represent intersections in the execution paths of otherwise independent processes, where the actions of one process depend on the actions of another process.

The application designer identifies synchronization points between processes and selects the functions best suited to implement the required synchronization.

The application designer identifies resources such as message queues and shared memory that the application will use. Failure to control access to critical resources can result in performance bottlenecks or inconsistent data. For example, the transaction processing application of a national ticket agency must be prepared to simultaneously process purchases from sites around the country. Ticket sales are transactions recorded in a central database. Each transaction must be completed as either rejected or confirmed before the application performs further updates to the database. The application performs the following synchronization operations:

- Restricts access to the database
- Provides a reasonable response time
- Ensures against overbookings

Processes compete for access to the database. In doing so, some processes must wait for either a confirmation or rejection of a transaction.

1.3.1 Waiting for a Specified Time

A process can postpone execution for a specified period of time or until a specified time and date. This synchronization technique allows processes to work periodically and to carry out tasks on a regular basis. To postpone execution for a specified period of time, use one of the following two methods:

- The sleep functions
- Per-process timers

The `sleep` function has a granularity of seconds while the `nanosleep` function uses nanoseconds. The granularity of the `nanosleep` function may make it more suitable for realtime applications. For example, a vehicle simulator application may rely on retrieval and recalculation operations that are completed every 5 milliseconds. The application requires a number of per-process timers armed with repetition intervals that allow the application to retrieve and process information within the 5 millisecond deadline.

Realtime clocks and timers allow an application to synchronize and coordinate activities according to a predefined schedule. Such a schedule might require repeated execution of one or more processes at specific time intervals or only once. A timer is set (armed) by specifying an initial start time value and an interval time value. Realtime timing facilties provide applications with the

ability to use relative or absolute time and to schedule events on a one-shot or periodic basis.

1.3.2 Waiting for Semaphores

The semaphore gives a process a way to synchronize its access to a resource shared with other processes, most commonly, shared memory. A semaphore is a kernel data structure, shared by two or more processes, that enforces either exclusive or metered access to the shared resource. Exclusive access means that only one process can access the resource at a time; metered access means that up to a specified number of processes can access the resource simultaneously. Exclusive access is achieved through the use of binary semaphores.

The semaphore takes its name from the signaling system railroads developed to prevent more than one train from using the same length of track, a technique that enforces exclusive access to the shared resource of the railroad track. A train waiting to enter the protected section of track waits until the semaphore shows that the track is clear, at which time the train enters the track and sets the semaphore to show the track is in use. Another train approaching the protected track while the first train is using it waits for the signal to show that the track is clear. When the first train leaves the shared section of track, it resets the semaphore to show that the track is clear.

The sempahore protection scheme works only if all the trains using the shared resource cooperate by waiting for the semaphore when the track is busy and resetting the semaphore when they have finished using the track. If a train enters a track marked busy without waiting for the signal that it is clear, a collision can occur. Conversely, if a train exiting the track fails to signal that the track is clear, other trains will think the track is in use and refrain from using it.

The same is true for processes synchronizing their actions through the use of semaphores and shared memory. To gain access to the resource protected by the semaphore, cooperating processes must lock and unlock the semaphore. A calling process must check the state of the semaphore before performing a task. If the semaphore is locked, the process is blocked and waits for the semaphore to become unlocked. Binary semaphores restrict access to a shared resource by allowing access to only one process at a time.

An application can protect the following resources with semaphores:

- Global variables, such as file variables, pointers, counters, and data structures. Synchronizing access to these variables means preventing simultaneous access, which also prevents one process from reading information while another process is writing it.

- Hardware resources, such as disk and tape drives. Hardware resources require controlled access for the same reasons as global variables; that is, simultaneous access could result in corrupted data.
- The kernel. A binary semaphore can allow processes to alternate execution by limiting access to the kernel on an alternating basis.

For information on using shared memory and semaphores, refer to Chapter 8 and Chapter 9.

1.3.3 Waiting for Communication

Typically, communication between processes is used to trigger process execution so the flow of execution follows the logical flow of the application design. As the application designer maps out the program algorithm, dependencies are identified for each step in the program. Information concerning the status of each dependency is communicated to the relevant processes so that appropriate action can be taken. Processes synchronize their execution by waiting for something to happen; that is, by waiting for communication that an event occurred or a task was completed. The meaning and purpose of the communication are established by the application designer.

Interprocess communication facilitates application control over the following:

- When and how a process executes
- The sequence of execution of processes
- How resources are allocated to service the requests from the processes

Section 1.2.6 introduces the forms of interprocess communication available to the realtime application designer. For further information on using interprocess communication facilities, refer to Chapters 6 through 10.

1.3.4 Waiting for Other Processes

Waiting for other processes means waiting until the process has terminated. To postpone execution while waiting for other processes, a parent process can wait for a child process or thread to terminate. A child process or a thread is created by the waiting process. Often, the child process needs to complete some task before the waiting parent process can continue. In such a situation, the actions of the parent and child processes are sometimes synchronized in the following way:

- The parent process creates the child process.
- The parent process synchronizes with the child.
- The child process executes until it terminates.

- The termination of the child process signals the parent process.
- The parent process resumes execution.

When a process calls the `fork` function, it creates a child process and executes an exact copy of the parent process. The child process begins execution at the instruction after the call to the `fork` function. Typically, the child process immediately issues a call to one of the `exec` functions, which in turn executes a new process image. The child process exits when it terminates normally and the parent process continues executing at the point where it left off. The child process can load an image from disk, or some other source, and schedule it to run.

The parent process can continue execution in parallel with the child process. However, if child processes are used as a form of process synchronization, the parent process can use other synchronization mechanisms such as semaphores and signals while the child process executes.

For information on using semaphores and signals, refer to Chapter 9 and Chapter 11.

1.4 Standards

A number of standards apply to application designers working in the UNIX environment. Primary among these are:

- ANSI
- ISO
- POSIX
- SVID

The purpose of standards is to enhance the portability of programs and applications; that is, to create code that is independent of the hardware or even the operating system on which the application runs. Standards allow users to move between systems without major retraining. In addition, standards introduce internationalization concepts as part of application portability.

ANSI standards apply to programming languages, networks and communication protocols, character coding, and database systems.

ISO, POSIX, and SVID standards apply to the operating system. For the most part, these standards apply to applications coded in the C language. These standards are not mutually exclusive; the DEC OSF/1 realtime environment uses a complement of these standards.

Additional information on POSIX standards is contained in the *IEEE Standard Portable Operating System Interface for Computer Environments* manuals, published by the Institute of Electrical and Electronics Engineers, Inc.

1.4.1 Including Common Definition Files

Common definition files, sometimes called include or header files, let you share common information between source files in an application. These files usually define known constants, declare routine and data types, define data structures, or declare function prototypes such as library functions or system services. Common definition files typically have a `.h` suffix.

To specify an include file in your source code, use the `#include` directive in column 1. For example, to include the header file for the P1003.4/D10 memory-locking functions, you must include the `mlock.h` header file as follows:

```
#include <mlock.h>
```

The C macro preprocessor searches for the file only in the default directory, `/usr/include`. You can specify a pathname for the file in the `#include` directive. The following example specifies that the `sem.h` header file be included from the `/usr/include/sys` directory.

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

You can also use the `-I`*dir* option on the compile command to specify additional pathnames to be searched by the C preprocessor. Note that if you specify multiple `-I` directives on the compile command, the files are processed in the order in which they appear on the compile command.

When the include file name is in quotes, the C preprocessor searches first in the directory where the source file resides, followed by the specified directory pathname *dir*, then the standard directory `/usr/include`. When the include file name is in angle brackets, the C preprocessor searches first in the specified pathname and then the `/usr/include` directory.

1.4.2 Compiling with the Realtime Library

You must explicitly load the required realtime runtime libraries when you compile realtime applications. Specify the `-lrt` option on the command line. The *rt* specification is an abbreviation of the realtime library name.

To find the realtime library, the `ld` linker expands the command specification by replacing the `-l` with `lib` and adding the specified `rt` characters and the `.a` suffix. Since the linker searches default directories in an attempt to locate the realtime archive library, you must specify the pathname.

To specify the realtime archive library, include the pathname of the library as part of the command syntax. The linker searches libraries in the order that you specify. The following example specifies that librt.a is to be included from the /usr/ccs/lib directory.

```
# cc myprogram.c /usr/ccs/lib/librt.a
```

The -L switch forces the linker to search in the directory specified on the -L switch first; then the other directories associated with the driver command are used. You could specify the full library pathname as follows:

```
# cc myprogram.c -L/usr/ccs/lib -lrt
```

By default all global symbols are exported from the archive library.

Most drivers allow you to view the passes of the driver program and the libraries being searched by specifying the -v option on the compile command.

To link an application against the realtime library, you must specify the library path. The -L switch specifies the pathname as an additional search directory for unresolved global references. The realtime library is kept in /usr/ccs/lib.

The realtime library uses the libc.a library. When you compile an application, the libc.a library is automatically pulled into the compilation.

On the link command, however, you must include the libc.a library. Since files are processed in the order in which they appear on the link command line, libc.a must appear on after librt.a. For example, you would link an application with the realtime library, librt.a, as follows:

```
# ld myprogram.o -L/usr/ccs/lib -lrt -lc
```

The -L switch ensures that the linker will find the archive version of the library first. The -l switch directs the linker to search the path for archive libraries.

1.4.3 Compiling with the Asynchronous I/O Library

When you compile an application that uses asynchronous I/O, include the necessary libraries on the command line. The following example shows the specification required if your application uses asynchronous I/O.

```
# cc -non_shared -D_POSIX_4SOURCE myprogram.c -laio -lpthreads -lbsd -lmach -lc_r
```

1.4.4 Defining the POSIX Environment

When you write applications that use POSIX functions, you must compile your application in the POSIX programming environment by using one of two methods. One method is to set a preprocessor symbol before you issue the compile command. The other method is to set the preprocessor symbol on the compiler command line.

The steps below set the preprocessor symbol in the source code:

1. Define the _POSIX_4SOURCE preprocessor symbol by either creating a local header file that defines the symbol or by including the symbol directly in your source file. The purpose of the definition is to control the visibility of named constants and identifiers. When you define the _POSIX_4SOURCE symbol, the _POSIX_SOURCE symbol is automatically defined.

   Define the _POSIX_4SOURCE symbol if you are writing conforming realtime POSIX applications. The following example shows a local header file that defines the _POSIX_4SOURCE preprocessor symbol:

   ```
   #define _POSIX_4SOURCE 1
   ```

2. Include the local header file in your source application as the first `#include` directive in your source code. Be sure to include the local header file in each source file for your application.

3. Compile you application.

To use realtime functionality, you must define the _POSIX_4SOURCE symbol. When the _POSIX_4SOURCE symbol is defined, the preprocessor includes the conforming header information for both POSIX 1003.1 and P1003.4/D10 in your program object.

Another method is to set the preprocessor symbol on the compile command line, using the `-D` option. This switch triggers nested `ifdefs` in the `include` files located in `/usr/include` and enables definitions that pertain to the POSIX standard. Using this switch ensures that the required definitions will be present. Note that you must include the `<standards.h>` header file in your source code. The following example uses `-D` option to define _POSIX_4SOURCE symbol.

```
# cc -D_POSIX_4SOURCE myprogram.c
```

In this example, the `-D` option defines the _POSIX_4SOURCE symbol to the value of 1. This definition is in effect only during the execution of the `cc` command. For more information on the `-D` option, see the reference page for the `cc` command.

If your application fails to compile, you may need to check your programming environment to make sure that the realtime kit is installed on your system. The lack of the realtime kit and its function library will cause your program to fail.

2

Process Scheduling and Priorities

The ability to control scheduling is an important requirement for realtime application designers. Control over scheduling takes two forms: controlling how the scheduler selects processes to run, and controlling the priority of a process.

The scheduling policy determines how the scheduler selects runnable processes, how processes are queued for execution, and how much time each process is given to run.

Scheduling policies work in conjunction with priority levels. A global priority range applies to all scheduling policies, but each policy has an associated priority range. The nature of the work performed by the processes helps determine the scheduling policy and priority best suited for the application's needs.

Realtime applications must be able to control process priorities in order to service external events in a timely and predictable manner. DEC OSF/1 P1003.4/D10 realtime facilities provide for a higher priority range as well as a choice of scheduling policies for greater control over application execution. Realtime functions allow processes to change both scheduling policies and priorities depending on application needs. At runtime, the combination of these realtime features gives the user control over system resources.

This chapter includes the following sections:

- Process Scheduling, Section 2.1
- Scheduling Policies, Section 2.2
- Process Priorities, Section 2.3
- Scheduling Functions, Section 2.4
- Priority and Policy Example, Section 2.5

2.1 Process Scheduling

Applications are often divided into a number of programs. Each program might run concurrently with one or more others; each program might run conditionally; or one of the programs might execute noncritical code while the others run critical code.

Each program is in turn composed of processes and threads. These processes can be detached or subprocesses, depending on application needs. Each process has a priority: That is, each process table entry contains a priority field used in process scheduling.

2.1.1 Process States

At runtime processes exist in various states: running, runnable, or waiting. When a process is created, it is immediately ready to run (runnable). The movement of a process from the runnable to the running state is controlled by the scheduler. The scheduler maintains a list of runnable processes at each priority level. When a process in the runnable state gains control of the processor and begins to execute, it is in the running state. Depending on the scheduling policy and priority of the running process, the process may return to the runnable state, be preempted, or wait. Table 2–1 describes these three process states.

Table 2–1 Process States

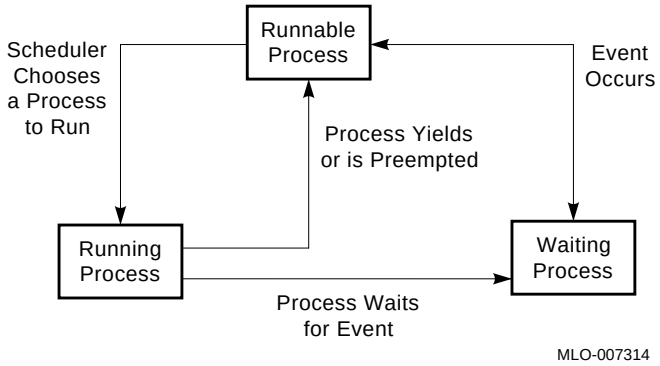
State	Description
Running	The process has control of the processor and is executing code.
Runnable	The process is in memory, eligible to run, but is not running. A runnable process waits in the queue with other runnable processes until the running process gives up control of the processor. At that time, the highest-priority runnable process will enter the running state.
Waiting	The process has given up eligibility to run until a condition or set of conditions is satisfied. A process may be waiting for a signal from another process, a wakeup call, a timer expiration, I/O completion, or any number of other events to occur.

During program execution, a process or thread may undergo many transitions from one state to another. All processes that compete with other processes to run on a single processor will move at least between the runnable and running states. To enter the running state, a process must first be in memory and in the runnable state. When it leaves the running state, a process may enter into either of the two other states, runnable or waiting.

Unless processes are locked into memory, they may be paged out to make room for another process. To guard against unwanted paging, realtime applications should use the P1003.4/D10 memory-locking functions, as described in Chapter 4.

Figure 2–1 displays the possible states of processes and represents with arrows the various state changes.

Figure 2–1 Process States



The process in the running state is designated as the current process. If a process is running, it has control of the kernel and is executing. However, if the process is in the runnable or waiting state, the process could be preempted before it runs.

A runnable process is one that is eligible to be selected to run. Runnable processes reside on the process list.

A waiting process awaits satisfaction of one or more wait conditions, such as a timeout, sleep, or the completion of some action.

2.1.2 The Scheduler

The primary function of the scheduler is to make scheduling decisions for the kernel. The scheduler makes certain that the highest-priority runnable process executes. The scheduler also maintains the kernel's scheduling database, representing the state of the system, in a consistent and accurate state. For example, the scheduler keeps process lists, which are priority-ordered queues of runnable processes in correct order. Whether selecting a runnable process to

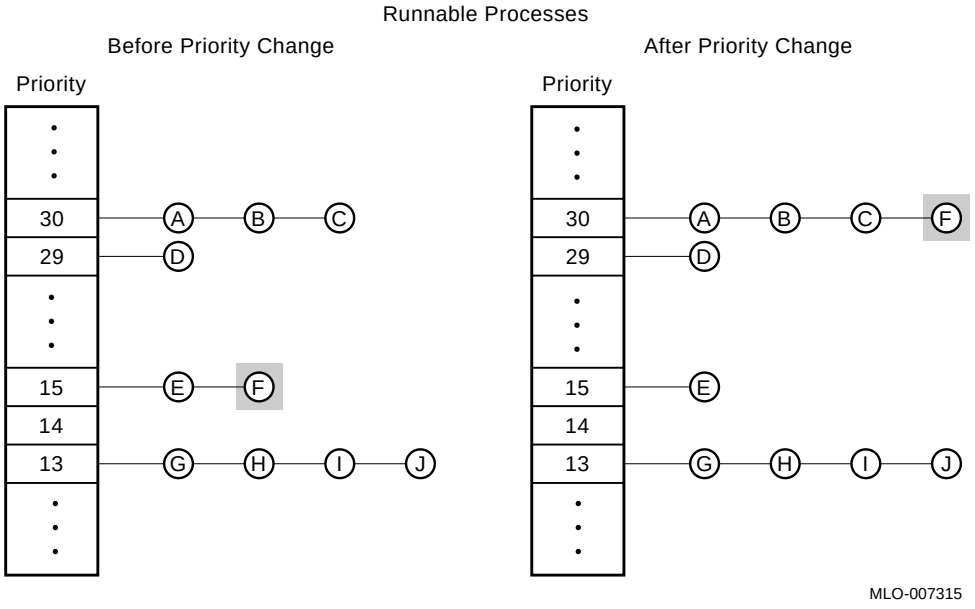
run or removing a process from the run queue, the scheduler applies a common set of selection criteria.

The scheduler determines which of a number of runnable processes is executed at any particular moment. The scheduler keeps track of the set of runnable processes and selects the highest-priority process to run.

Runnable processes are organized into process lists, or queues. The scheduler imposes order on the execution of the process list by placing the process that should run next at the beginning of the list, while the process that should wait the longest to run is placed at the end of the list. Generally speaking, the order of execution is on a first-in first-out (FIFO) basis. When a process becomes runnable, it takes it's position at the end of the process list for its priority.

Figure 2–2 illustrates the general principles of process scheduling.

Figure 2–2 Order of Execution



Processes A, B, and C are in the process list for the highest priority used in this illustration. Process A is at the beginning of the process list for priority 30. That means that Process A executes first, then processes B and C, respectively. When there are no more processes remaining in the process list for priority 30, the scheduler looks to the next lowest priority, finds process D at the beginning of the process list, and executes process D.

When a process changes priority, it goes to the end of the process list for its new priority. Figure 2–2 shows process F changing priority from 15 to 30. At priority 15 process F is at the end of the process list. When process F changes to priority 30, the process goes to the end of the process list for priority 30. At priority 30 process F is queued to execute after process C, but before process D.

The scheduling policy determines the length of execution for a process. The priority of a process, combined with the scheduling policy, determines how the process is scheduled. In a timesharing environment the scheduler recalculates the priority of a process after a process executes and periodically readjusts the priority of every eligible process. With a fixed-priority scheduling policy, the priority is not modified by the scheduler.

Processes are rescheduled when one of the following events occurs:

- The running process enters the runnable or waiting state.
- A higher-priority process becomes runnable.
- A process changes scheduling policy.

When one of these events occurs, the scheduler reexamines the current scheduling scheme to determine which other process is promoted to the running state. The scheduler considers only processes in the runnable state and makes its choice depending on the priority and scheduling policy specified for the runnable processes. When a process whose priority is higher than that of the currently running process becomes runnable, the scheduler preempts the lower-priority process, returning it to the runnable state. Then the scheduler promotes the higher-priority process to the running state. This method is called "preemptive priority scheduling" and gives the user an effective way to schedule time-critical processes. Between processes of equal priority, the scheduler chooses on the basis of the specified scheduling policy.

Figure 2–3 illustrates how processes can change from the running state to the runnable state within the queue for a single priority. In this illustration, processes move in and out of the running state as they are preempted by a higher-priority process, a process finishes its quantum, or a process changees priority.

As processes are selected to run or move from the end to the beginning of the process list, the scheduler continually updates the kernel database and the process list for each priority.

Figure 2–3 Process Events

Event	Reaction	The Running Process Is:	The Runnable Processes Are:
G reaches beginning of the queue and starts its quantum	G moves to running		
		G	H I
A is a higher priority, becomes runnable, and preempts G	G preempted - goes to the beginning of the queue		
		A	G H I
A yields or enters waiting state	G runs again to finish its quantum		
		G	H I
G finishes its quantum	G goes to the end of the queue H moves to running		
		H	I G
A is a higher priority, becomes runnable, and preempts H	H preempted - goes to the beginning of the queue		
		A	H I G
A raises priority of K	K changes priority K goes to the end of the queue		
		A	H I G K

MLO-007316

2.1.3 Scheduling Interfaces

The DEC OSF/1 operating system provides two separate, but related interfaces to scheduling policies: one that supports the default, timesharing scheduling policy (the `nice` scheduling interface) and one that supports the scheduling

policies defined by the P1003.4/D10 standard (the realtime scheduling interface). These interfaces use different priority ranges and are managed through different function calls. The `nice` interface allows you to set process priority while the realtime interface allows you to set both the process priority and the scheduling policy.

The default scheduling interface is the `nice` interface, which has the following characteristics:

- Supports only the timesharing scheduling policy
- Supports priorities in the 20 through –20 range
- Uses a default priority of 0
- Uses lower priority numbers to represent higher priority
- Provides relative priorities that can be changed by the scheduler
- Supports relative priority changes by the user through a call to the `nice`, `renice`, or `setpriority` functions

The realtime interface provides support for multiple scheduling policies, including the timesharing scheduling policy. You can change the scheduling policy and priority of a process running under any P1003.4/D10 scheduling policy. The realtime interface has the following characteristics:

- Supports the timesharing, first-in first-out (FIFO) and round-robin scheduling policies
- Supports priorities in the 0 through 63 range
- Uses a default priority of 19
- Supports absolute, fixed priorities
- Uses a higher priority number to represent a higher priority
- Supports absolute priority changes by the user through a call to one of the P1003.4/D10 functions, `sched_set_sched_param` or `sched_setscheduler`
- Supports scheduling policy changes by the user through a call to the `sched_setscheduler` function

Priorities are changed by the scheduler only if you select the timesharing scheduling policy. Note that you can use only the `nice`, `renice`, or `setpriority` functions to change the priority of a process if the process is running under the timesharing scheduling policy. If the realtime interface is used to change the scheduling policy of a process to first-in first-out or round-robin, the process is no longer affected by the `nice`, `renice`, or `setpriority` functions.

The nice interface logically divides priorities into two ranges, nonprivileged user and system. While these ranges reflect the nature of the work commonly associated with the priorities within a range, there is no clear distinction between the ranges. For example, system processing can be done in the nonprivileged user priority range.

The realtime interface divides the priority range in a similar way, but also provides absolute control over scheduling. The application designer can determine the priorities of other processes and precisely set the priority of each realtime process, to better determine when processes will run relative to one another. This way, the scheduler can guarantee that a critical process will run whenever it is needed, for as long as it is needed. Time-critical realtime processes must be able to run at a very high priority, but must also be able to yield execution to other realtime processes in a deterministic manner.

The realtime interface allows you to alter the scheduling policy, which gives you more control over when processes execute by more precisely defining how individual processes are scheduled to run relative to one another. P1003.4/D10 scheduling policies include two fixed-priority scheduling policies and the standard timesharing policy. You can use the timesharing policy for nonrealtime applications but will want to use either of the fixed-priority policies for realtime applications.

Regardless of the scheduling interface, the scheduler uses the same method to determine which process runs next: the process at the beginning of the highest priority process list.

2.2 Scheduling Policies

Whether or not a timesharing process runs is often determined not by the needs of the application, but by the scheduler's algorithm. The scheduler determines the order in which processes execute and sometimes forces resource-intensive processes to yield to other processes.

Other users' activities on the system at that time affect scheduling. Whether or not a realtime process yields to another process can be based on a quantum or the scheduling policy.

2.2.1 The Nature of the Work

Scheduling policies are designed to give you flexibility and control in determining how work is performed so that you can balance the nature of the work with the behavior of the process. Essentially, there are three broad categories of work:

- Timesharing Processing

Used for interactive and noninteractive applications with no critical time limits but a need for reasonable response time and high throughput.

- System Processing

  Performs work on behalf of the system, such as paging, networking, and accessing files. The responsiveness of system processing impacts the responsiveness of the whole system.

- Realtime Processing

  Used for critical work that must be completed within a certain time period, such as data collection or device control. The nature of realtime processing often means that missing a deadline makes the data invalid or causes damage.

To control scheduling policies you must use the P1003.4/D10 realtime scheduling functions and select an appropriate scheduling policy for your process. DEC OSF/1 P1003.4/D10 scheduling policies are set only through a call to the `sched_setscheduler` function. The `sched_setscheduler` function recognizes the scheduling policies by keywords beginning with SCHED_ as follows:

- SCHED_OTHER, Timesharing scheduling
- SCHED_FIFO, First-in first-out scheduling
- SCHED_RR, Round-robin scheduling

All three scheduling policies have overlapping priority ranges to allow for maximum flexibility in scheduling. When selecting a priority and scheduling policy for a realtime process, consider the nature of the work performed by the process. Regardless of the scheduling policy, the scheduler selects the process at the beginning of the highest-priority, nonempty process list to become a running process.

2.2.2 Timesharing Scheduling

The P1003.4/D10 timesharing scheduling policy, SCHED_OTHER, allows realtime applications to return to a nonrealtime scheduling policy. In timesharing scheduling, a process starts with an initial priority that either the user or the scheduler can change. Timesharing processes run until the scheduler recalculates process priority, based on the system load, the length of time the process has been running, or the value of `nice`. Section 2.3.1 describes timesharing priorities changes in more detail.

Under the timesharing scheduling policy, the scheduler enforces a quantum. Processes are allowed to run until they are preempted, yield to another process, or finish their quantum. If no higher-priority processes are waiting to run, the executing process is allowed to continue. However, while a process is running, the scheduler changes the process's priority. Over time, it is likely that a higher-priority process will exist because the scheduler adjusts priority. If a process is preempted or yields to another process, it goes to the end of the process list for the new priority.

2.2.3 Fixed-Priority Scheduling

With a fixed-priority scheduling policy, the scheduler does not adjust process priorities. If the application designer sets a process at priority 30, it will always be queued to the priority 30 process list, unless the application or the user explicitly changes the priority.

As with all scheduling policies, fixed-priority scheduling is based on the priorities of all runnable processes. If a process waiting on the process list has a higher priority than the running process, then the running process is preempted for the higher-priority process. However, the two fixed-priority scheduling policies (SCHED_FIFO and SCHED_RR) allow greater control over the length of time a process waits to run.

Fixed-priority scheduling relies on the application designer or user to manage the efficiency of process priorities relative to system workloads. For example, you may have a process that must be allowed to finish executing, regardless of other activities. In this case, you may elect to increase the priority of your process and use the first-in first-out scheduling policy, which guarantees that a process will never be placed at the end of the process list if it is preempted. In addition, the process's priority will never be adjusted and it will never be moved to another process list. With fixed-priority scheduling policies, you must explicitly set priorities by calling either the `sched_set_sched_param` or `sched_setscheduler` function. Thus, realtime processes using fixed-priority scheduling policies are free to yield execution resources to each other in an application-dependent manner.

If you are using a fixed-priority scheduling policy and you call the `nice` or `renice` function to adjust priorities, the function returns without changing the priorities.

2.2.3.1 First-in First-out Scheduling

The first-in first-out scheduling policy, SCHED_FIFO gives maximum control to the application. This scheduling policy does not enforce a quantum. Rather, each process runs to completion or until it blocks or voluntarily yields to another process at the same priority.

Processes scheduled under the first-in first-out scheduling policy are chosen from a process priority list that is ordered according to the amount of time its processes have been on the list without being executed. Under this scheduling policy, the process at the beginning of the highest-priority, nonempty process list is executed first. The next process moves to the beginning of the list and is executed next. Thus execution continues until that priority list is empty. Then the process at the beginning of the next highest-priority, nonempty process list is selected and execution continues. A process runs until execution finishes or the process is preempted by a higher-priority process.

The process at the beginning of a process list has waited at that priority the longest amount of time, while the process at the end of the list has waited the shortest amount of time. Whenever a process becomes runnable, it is placed on the end of a process list and waits until the processes in front of it have executed. When a process is placed in an empty high-priority process list, the process will preempt a lower-priority running process.

If an application changes the priority of a process, the process is removed from its list and placed at the end of the new priority process list.

The following rules determine how runnable processes are queued for execution using the first-in first-out scheduling policy:

- When a process is preempted, it goes to the beginning of the process list for its priority.
- When a blocked process becomes runnable, it goes to the end of the process list for its priority.
- When a running process changes the priority or scheduling policy of another process, the changed process goes to the end of the new priority process list.
- When a process voluntarily yields to another process, it goes to the end of the process list for its priority.

The first-in first-out scheduling policy is well suited for the realtime environment because it is deterministic. That is, processes with the highest priority always run, and among processes with equal priorities, the process that has been runnable for the longest period of time is executed first. You can achieve complex scheduling by altering process priorities.

2.2.3.2 Round-Robin Scheduling

The round-robin scheduling policy, SCHED_RR, is a logical extension of the first-in first-out scheduling policy. A process running under the round-robin scheduling policy is subject to the same rules as a process running under the fixed-priority scheduling policy, but a quantum is imposed on the running process. When a process finishes its quantum, it goes to the end of the process list for its priority.

Processes under the round-robin scheduling policy may be preempted by a higher-priority process before the quantum has expired. A preempted process goes to the beginning of its priority process list and completes the previously unexpired portion of its quantum when the process resumes execution. This ensures that a preempted process regains control as soon as possible.

Figure 2–4 shows process scheduling using a quantum. One portion of the figure shows the running process, the other portion of the figure shows what happens to running processes over time. Process G is removed from the beginning of the process list, placed in the run queue, and begins execution. Process B, a higher priority process, enters the runnable state while process G is running. The scheduler preempts process G to execute process B. Since process G had more time left in its quantum, the scheduler returns process G to the beginning of the process list, keeps track of the amount of time left in process G's quantum, and executes process B. When process B finishes, process G is again moved into the run queue and finishes its quantum. Process H, next in the process list, executes last.

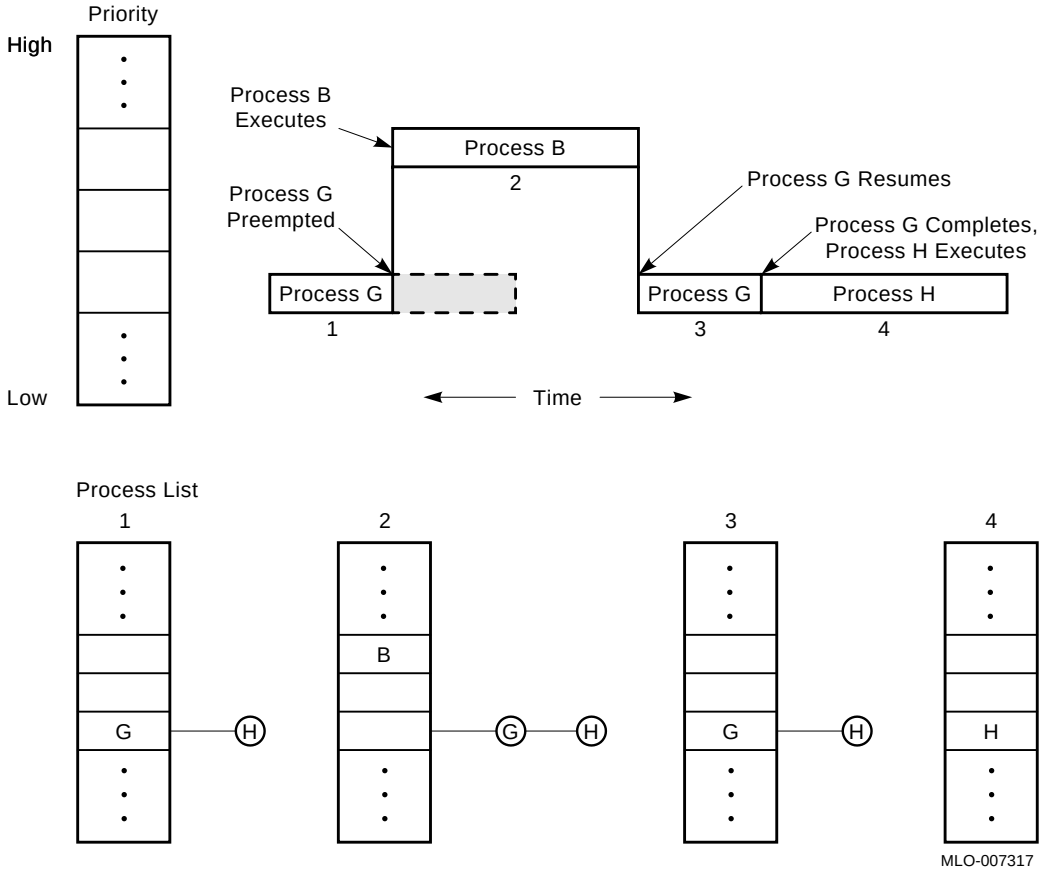
Round-robin scheduling is designed to provide a facility for implementing time-slice algorithms. You can use the concept of a quantum in combination with process priorities to facilitate time-slicing. You can use the `sched_get_rr_interval` function to retrieve information concerning the quantum used in round-robin scheduling.

2.3 Process Priorities

All applications are given an initial priority, either implicitly by the operating system or explicitly by the user. If you fail to specify a priority for a process, the kernel assigns the process an initial priority.

You can specify and manage a process's priority using either `nice` or P1003.4/D10 functions. The `nice` functions are useful for managing priorities for nonrealtime, timesharing applications. However, realtime priorities are higher than the `nice` priorities and make use of the P1003.4/D10 scheduling

Figure 2–4 Preemption—Finishing a Quantum



policies. Realtime priorities can be managed only by using the associated P1003.4/D10 functions.

In general, process scheduling is based on the concept that tasks can be prioritized, either by the user or by the scheduler. Each process table entry contains a priority field used in process scheduling. Conceptually, each priority level consists of a process list. The process list is ordered with the process that should run first at the beginning of the list and the process that should run last at the end of the list. Since a single processor can execute only one process at a time, the scheduler selects the first process at the beginning of the highest priority, nonempty process list for execution.

Priority levels are organized in ranges. The nonprivileged user application runs in the same range as most applications using the timesharing scheduling policy. Most users need not concern themselves with priority ranges above this range. Privileged applications (system or realtime) use higher priorities than nonprivileged user applications. In some instances, realtime and system processes can share priorities, but most realtime applications will run in a priority range that is higher than the system range.

2.3.1 Priorities for the nice Interface

The nice interface priorities are divided into two ranges: the higher range is reserved for the operating system, and the lower range is for nonprivileged user processes. With the nice interface, priorities range from 20 through –20, where 20 is the lowest priority. Nonprivileged user processes typically run in the 20 through 0 range. Many system processes run in the range 0 through –20 (or higher). Table 2–2 shows the ranges nice interface priority ranges.

Table 2–2 Priority Ranges for the nice Interface

Range	Priority Level
Nonprivileged user	20 through 0
System	0 through –20

A numerically low value implies a high priority level. For example, a process with a priority of 5 has a lower priority than a process with a priority of 0. Similarly, a system process with a priority of –5 has a lower priority than a process with a priority of –15. System processes can run at nonprivileged user priorities, but a user process can only increase its priority into the system range if the owner of the user process has superuser privileges.

Processes start at the default base priority for a nonprivileged user process (0). Since the only scheduling policy supported by the nice interface is timesharing, the priority of a process changes during execution. That is, the nice parameter represents the highest priority possible for a process. As the process runs, the scheduler adds offsets to the initial priority, adjusting the process's priority downward from or upward toward the initial priority. However, the priority will not exceed (be numerically lower than) the nice value.

The nice interface supports relative priority changes by the user through a call to the nice, renice, or set_priority functions. Interactive users can specify a base priority at the start of application execution using the nice command. The renice command allows users to interactively change the priority of a running process. An application can read a process's priority by calling the getpriority function. Then the application can change a process's priority by

calling the setpriority function. These functions are useful for nonrealtime applications but do not affect processes running under one of the P1003.4/D10 fixed-priority scheduling policies, described in Section 2.2.

Refer to the reference pages for more information on the getpriority, setpriority, nice, and renice functions.

2.3.2 Priorities for the Realtime Interface

Realtime interface priorities are divided into three ranges: with the highest range reserved for realtime, the middle range used by the operating system, and the low range used for nonprivileged user processes. DEC OSF/1 realtime priorities loosely map to the nice priority range, but provide a wider range of priorities. Processes using the P1003.4/D10 scheduling policies must also use the DEC OSF/1 realtime interface priority scheme. Table 2–3 shows the DEC OSF/1 realtime priority ranges.

Table 2–3 Priority Ranges for the DEC OSF/1 Realtime Interface

Range	Priority Level
Nonprivileged user	SCHED_PRIO_USER_MIN through SCHED_PRIO_USER_MAX
System	SCHED_PRIO_SYSTEM_MIN through SCHED_PRIO_SYSTEM_MAX
Realtime	SCHED_PRIO_RT_MIN through SCHED_PRIO_RT_MAX

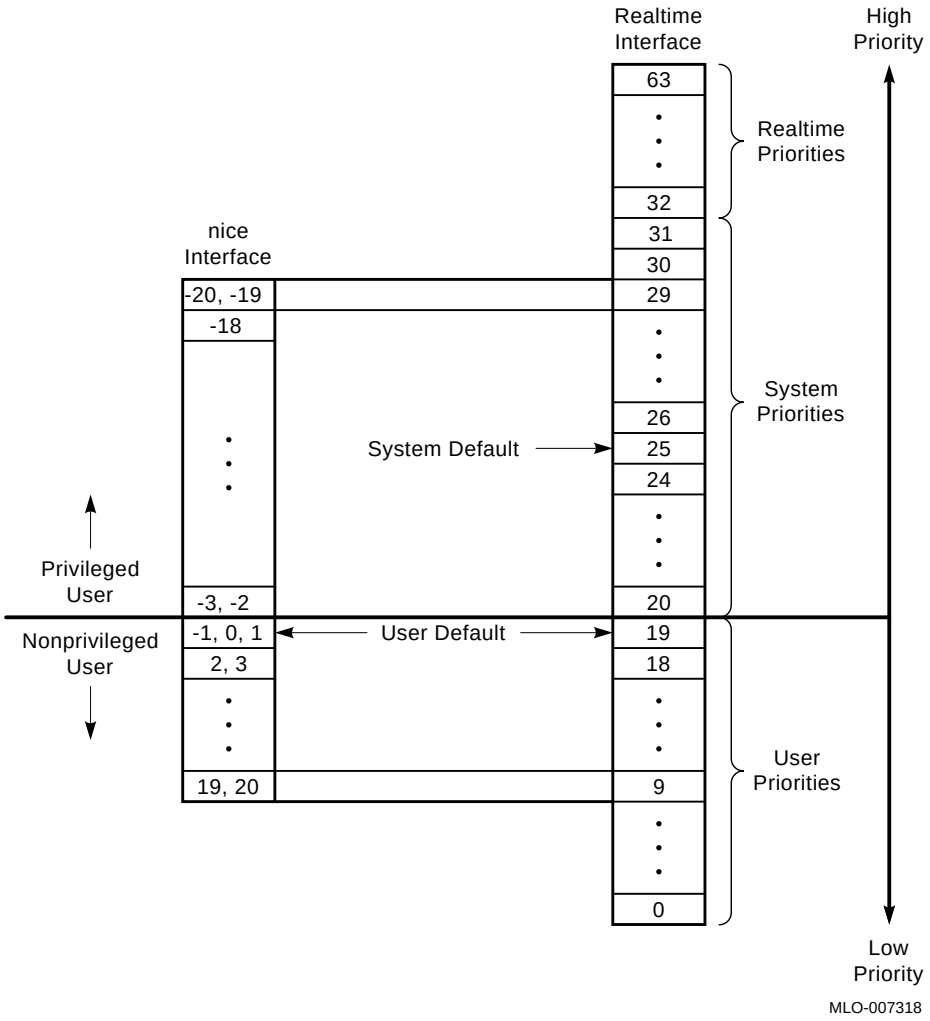
Realtime interface priority levels are the inverse of the nice priority levels; a numerically high value implies a high priority level. A realtime process with a priority of 32 has a higher priority than system processes, but a lower priority than another realtime process with a priority of 45. Realtime and system processes can run at nonprivileged user priorities, but a nonprivileged user process cannot increase its priority into the system or realtime range without superuser privileges.

The default initial priority for processes using realtime priorities is 19. The default scheduling policy is timesharing.

Figure 2–5 illustrates the relationship between these two priority interfaces.

Note that hardware interrupts have a higher priority than realtime processes and therefore preempt realtime processes.

Figure 2–5 Priority Ranges for the nice and Realtime Interfaces



2.3.3 Configuring Realtime Priorities

You should assign realtime priorities according to the critical nature of the work the processes perform. Some applications may not need to have all processes running in the realtime priority range. Applications that run in a realtime range for long periods of time may prevent the system from performing necessary services, which could cause network and device timeouts

or data overruns. Some processes perform adequately if they run under a fixed-priority scheduling policy at priority 19. Only critical processes running under a fixed-priority scheduling policy should run with priorities in the realtime range, 32 through 63.

Although P1003.4/D10 functions let you change the scheduling policy while your application is running, it is better to select a scheduling policy during application initialization than to change the scheduling policy while the application executes. However, you may find it necessary to adjust priorities within a scheduling policy as the application executes.

It is recommended that all realtime applications provide a way to configure priorities at runtime. You can configure priorities using the following methods:

1. Providing a default priority within the realtime priority range by calling the `sched_get_priority_max` and `sched_get_priority_min` functions
2. Using an `.rc` initialization file, which override the default priority, or using environment variables, which override the default priority
3. Adjusting priority during initialization by calling the `sched_set_sched_param` function

Each process should have a default base priority appropriate for the kind of work it performs and each process should provide a configuration mechanism for changing that base priority. To simplify system management, make the hardcoded default equal to the highest priority used by the application. At initialization, the application should set its process priorities by subtracting from the base priority. Use the constants given in the `<sched.h>` header file as a guide for establishing your default priorities.

The `<sched.h>` header file provides the following constants that may be useful in determining the optimum default priority:

```
SCHED_PRIO_USER_MIN
SCHED_PRIO_USER_MAX
SCHED_PRIO_SYSTEM_MIN
SCHED_PRIO_SYSTEM_MAX
SCHED_PRIO_RT_MIN
SCHED_PRIO_RT_MAX
```

These values are the current values for default priorities. When coding your application, use the constants rather than numerical values. The resulting application will be easier to maintain should default values change in the future.

Debug your application in the nonprivileged user priority range before running the application in the realtime range. If a realtime process is running at a level higher than kernel processes and the realtime process goes into an infinite loop, you must reboot the system to stop process execution.

Although priority levels for DEC OSF/1 system priorities can be adjusted using the `nice` or `renice` functions, these functions have a ceiling that is below the realtime priority range. To adjust realtime priorities, use the `sched_get_sched_param` and `sched_set_sched_param` P1003.4/D10 functions, discussed in Section 2.4.3. You should only adjust process priorities for your own application. Adjusting system process priorities could cause unexpected side effects.

2.4 Scheduling Functions

Realtime processes must be able to select dynamically the most appropriate priority level and scheduling policy. A realtime application often modifies the scheduling policy and priority of a process, performs some function, and returns to its previous priority. Realtime processes must also be able to yield system resources to each other in response to specified conditions. Eight P1003.4/D10 functions, as summarized in Table 2–4 satisfy these realtime requirements. Refer to the reference pages for a complete description of these functions.

Table 2–4 P1003.4/D10 Process Scheduling Functions

Function	Description
`sched_getscheduler`	Returns the scheduling policy of a specified process
`sched_get_sched_param`	Returns the scheduling priority of a specified process
`sched_get_priority_max`	Returns the maximum priority allowed for a scheduling policy
`sched_get_priority_min`	Returns the minimum priority allowed for a scheduling policy
`sched_get_rr_interval`	Returns the quantum allowed for the round-robin scheduling policy
`sched_setscheduler`	Sets the scheduling policy and priority of a specified process
`sched_set_sched_param`	Sets the scheduling priority of a specified process
`sched_yield`	Yields execution to another process

All the preceding functions, with the exception of the `sched_yield` function, require a process ID, parameter, (*pid*). In all the P1003.4/D10 priority and

scheduling functions, a *pid* value of zero indicates that the function call refers to the calling process. Use zero in these calls to eliminate using the `getpid` or `getppid` functions.

The priority and scheduling policy of a process are inherited across a `fork` or `exec` system call.

You must have `superuser` privileges to change the priority or scheduling policy of a process. Changing the priority or scheduling policy of a process causes the process to be queued to the end of the process list for its new priority.

2.4.1 Determining Limits

Three functions allow you to determine scheduling policy parameter limits. The `sched_get_priority_max` and `sched_get_priority_min` functions return the appropriate maximum or minimum priority permitted by the scheduling policy. These functions can be used with any of the P1003.4/D10 scheduling policies, first-in first-out, round-robin, or timesharing. You must specify one of the following keywords when using these functions:

- SCHED_FIFO
- SCHED_RR
- SCHED_OTHER

The `sched_get_rr_interval` function returns the current quantum for process execution under the round-robin scheduling policy.

2.4.2 Retrieving the Priority and Scheduling Policy

Two functions return the priority and scheduling policy for realtime processes, `sched_get_sched_param` and `sched_getscheduler`, respectively. You do not need special privileges to use these functions, but you need `superuser` privileges to set both priority and scheduling policy.

If the *pid* is zero for either functions, the value returned is the priority or scheduling policy for the calling process. The values returned by a call to the `sched_getscheduler` function indicate whether the scheduling policy is SCHED_FIFO, SCHED_RR, or SCHED_OTHER.

2.4.3 Setting the Priority and Scheduling Policy

Use the sched_get_sched_param function to determine the initial priority of a process; use the sched_set_sched_param function to establish a new priority. Adjusting priority levels in response to predicted system loads and other external factors allows the system administrator or application user greater control over system resources. When used in conjunction with the first-in first-out scheduling policy, the sched_set_sched_param function allows a critical process to run as soon as it is runnable, for as long as it needs to run. This occurs because the process preempts other lower-priority processes. This can be important in situations where scheduling a process must be as precise as possible.

The sched_set_sched_param function takes two parameters. The *pid* parameter specifies the process to change. If the *pid* parameter is zero, priority is set for the calling process. The *param* parameter specifies the new priority level. The specified priority level must be within the inclusive range for the minimum and maximum values for the scheduling policy selected for the process.

The sched_setscheduler function sets both the scheduling policy and priority of a process. Three parameters are required for the sched_setscheduler function: *pid*, *policy*, and *param*. If the *pid* parameter is zero, the scheduling policy and priority will be set for the calling process. The *policy* parameter identifies whether the scheduling policy is to be set to SCHED_FIFO, SCHED_RR, or SCHED_OTHER. The *param* parameter indicates the priority level to be set and must be within the range for the the indicated scheduling policy.

Notification of a completed priority change may be delayed if the calling process has been preempted. The calling process is notified when it is again scheduled to run.

If you are designing portable applications (POSIX strictly conforming applications), be careful not to assume that the *priority* field is the only field in the sched_param structure. All the fields in a sched_param structure should be initialized before the structure is passed as the *param* argument to the sched_set_sched_param or sched_setscheduler. Example 2–1 shows how a process can initialize the fields using only constructs provided by the P1003.4/D10 standard.

Example 2–1 Initializing Priority and Scheduling Policy Fields

```
/* Change to the SCHED_FIFO policy and the highest priority, then  */
/* lowest priority, then back to the original policy and priority. */

#include <sched.h>

#define CHECK(sts,msg)  \
  if (sts == -1) {      \
    perror(msg);        \
    exit(-1);           \
  }

main ()
{
  struct sched_param param;
  int my_pid = 0;
  int old_policy, old_priority;
  int sts;
  int low_priority, high_priority;

       /* Get parameters to use later.  Do this now  */
       /* avoid overhead during time-critical phases.*/

  high_priority = sched_get_priority_max(SCHED_FIFO);
  CHECK(high_priority,"sched_get_priority_max");
  low_priority = sched_get_priority_min(SCHED_FIFO);
  CHECK(low_priority,"sched_get_priority_min");

       /* Save the old policy for when it's restored. */

  old_policy = sched_getscheduler(my_pid);
  CHECK(old_policy,"sched_getscheduler");

       /* Get all fields of the param structure.  This is where */
       /* fields other than priority get filled in.             */

  sts = sched_get_sched_param(my_pid, &param);
  CHECK(sts,"sched_get_sched_param");

       /* Keep track of the old priority. */

  old_priority = param.priority;

       /* Change to SCHED_FIFO, highest priority.  The param   */
       /* fields other than priority get used here.            */

  param.priority = high_priority;
  sts = sched_setscheduler(my_pid, SCHED_FIFO, &param);
  CHECK(sts,"sched_setscheduler");

       /* Change to SCHED_FIFO, lowest priority.  The param */
       /* fields other than priority get used here, too.    */
```

(continued on next page)

Example 2–1 (Cont.) Initializing Priority and Scheduling Policy Fields

```
param.priority = low_priority;
sts = sched_set_sched_param(my_pid, &param);
CHECK(sts,"sched_set_sched_param");

     /* Restore original policy, parameters.  Again, other  */
     /* param fields are used here.                         */

param.priority = old_priority;
sts = sched_setscheduler(my_pid, old_policy, &param);
CHECK(sts,"sched_setscheduler 2");

exit(0);
```

A process is allowed to change the priority of another process only if the target process runs on the same node as the calling process and at least one of the following conditions is true:

- The calling process is a privileged process with a real or effective UID of zero.
- Either the real user UID or the effective user UID of the calling process is equal to either the real user UID or the saved-set user UID of the target process.
- Either the real group GID or the effective group GID of the calling process is equal to either the real group ID or the saved-set group GID of the target process, and the calling process has group privilege.

Before changing the priority of another process, determine which UID is running the application. Use the `getuid` system call to determine the real UID associated with a process.

2.4.4 Yielding to Another Process

Use the `sched_yield` function to control how processes at the same priority access kernel resources. Sometimes, in the interest of cooperation, it is important that a running process give up the kernel to another process at the same priority level. You can force processes to cooperate by resetting priorities, but this requires multiple function calls.

The `sched_yield` function causes the scheduler to look for another process to run and forces the caller to return to the runnable state. The process that calls the `sched_yield` function resumes execution after all runnable processes of equal priority have been scheduled to run. If there are no other runnable processes at that priority, the caller continues to run. The `sched_yield` function causes the process to yield for one cycle through the process list. That is, after a call to `sched_yield`, the target process goes to the end of its priority

process list. If another process of equal priority is created after the call to sched_yield, the new process is queued up after the yielding process.

The sched_yield function is most useful when used with the first-in first-out scheduling policy. Since the round-robin scheduling policy imposes a quantum on the amount of time a process runs, there is less need to use sched_yield. The round-robin quantum regulates the use of system resources through time-slicing. The sched_yield function is also useful when a process does not have permission to set its priority but still needs to yield execution.

2.5 Priority and Policy Example

Example 2–2 determines the amount of time in a round-robin quantum, saves the current scheduling parameters, and sets a realtime priority. Using the round-robin scheduling policy, the example loops through a test until a call to the sched_yield function causes the process to yield.

Example 2–2 Using Priority and Scheduling Functions

```
#include <timers.h>
#include <time.h>
#include <sched.h>
#define LOOP_MAX 10000000
#define CHECK_STAT(stat, msg)  \
     if (stat == -1)           \
     { perror(msg);            \
       exit(-1);               \
     }

main()
{
     struct sched_param my_param;
     int      my_pid = 0;
     int      old_priority, old_policy;
     int      stat;

     struct timespec rr_interval;
     int      try_cnt, loop_cnt, tmp_nbr = 0;

      /* Determine the round-robin quantum */

stat = sched_get_rr_interval (&rr_interval);
CHECK_STAT(stat, "sched_get_rr_interval");
printf("Round-robin quantum is %lu seconds, %ld nanoseconds\n",
     rr_interval.tv_sec, rr_interval.tv_nsec);

      /* Save the current scheduling parameters */
```

(continued on next page)

Example 2–2 (Cont.) Using Priority and Scheduling Functions

```
old_policy = sched_getscheduler(my_pid);
stat = sched_get_sched_param(my_pid, &my_param);
CHECK_STAT(stat, "sched_get_sched_param - save old priority");
old_priority = my_param.priority;

      /* Set a realtime priority and round-robin */
      /* scheduling policy */

my_param.priority = SCHED_PRIO_RT_MIN;
stat = sched_setscheduler(my_pid, SCHED_RR, &my_param);
CHECK_STAT(stat, "sched_setscheduler - set rr priority");

      /* Try the test */

for (try_cnt = 0; try_cnt < 10; try_cnt++)

      /* Perform some CPU-intensive operations */

     {for(loop_cnt = 0; loop_cnt > LOOP_MAX; loop_cnt++)
           {
            tmp_nbr+=loop_cnt;
            tmp_nbr-=loop_cnt;
            }

       printf("Completed test %d\n",try_cnt);
       sched_yield();
       }

      /* Lower priority and restore policy */

my_param.priority = old_priority;
stat = sched_setscheduler(my_pid, old_policy, &my_param);
CHECK_STAT(stat, "sched_setscheduler - to old priority");
}
```

3

Clocks and Timers

Realtime applications must be able to operate on data within strict timing constraints in order to schedule application or system events. Timing requirements can be in response to either the need for high system throughput or fast response time. Applications requiring high throughput may process large amounts of data and use a continuous stream of data points equally spaced in time. For example, electrocardiogram research uses a continuous stream of data for qualitative and quantitative analysis.

Applications requiring a fast response to asynchronous external events must capture data as it comes in and perform decision-making operations or generate new output data within a given time frame. For example, flight simulator applications may acquire several hundred input parameters from the cockpit controls and visual display subsystem with calculations completed within a 5 millisecond time frame.

DEC OSF/1 P1003.4/D10 timing facilities allow applications to use relative or absolute time and to schedule events on a one-shot or periodic basis. Applications can create multiple timers for each process, up to a system-defined maximum number of timers.

This chapter includes the following sections:

- The Systemwide Clock, Section 3.1
- Types of Timers, Section 3.2
- Data Structures Associated with Timing Facilities, Section 3.3
- Resolution of the System Clock and Timers, Section 3.4
- Signals and Timers, Section 3.5
- Timer Functions, Section 3.6
- High-Resolution Sleep, Section 3.7
- Clocks and Timers Example, Section 3.8

The correctness of realtime applications often depends on satisfying timing constraints. A systemwide clock is the primary source for synchronization and high-resolution timers to support realtime requirements for scheduling events. The P1003.4/D10 timing functions perform the following tasks:

- Set a systemwide clock, obtain the current value of the clock, and fine tune the accuracy of the system time.
- Set per-process timers and use asynchronous signals on timer expiration
- Retrieve the resolution and maximum values for the systemwide clock and per-process timers
- Permit the calling thread or process to suspend execution for a period of time or until a signal is delivered

Timing facilities are most useful when combined with other synchronization techniques. This chapter concentrates primarily on those functions associated with setting system time and creating high-resolution timers.

Although non-POSIX functions are available for creating timers, application programmers striving for standards conformance, portability, multiple per-process timers, and flexibility in using timers should use the P1003.4/D10 timing facilties described in this chapter.

3.1 Clock Functions

The supported time-of-day clock is the CLOCK_REALTIME clock, defined in the `<timers.h>` header file. The CLOCK_REALTIME clock is a systemwide clock, visible to all processes running on the system. If all processes could read the clock at the same time, each process would see the same value.

The CLOCK_REALTIME clock measures the amount of time elapsed since 00:00:00:00 January 1, 1970 Greenwich Mean Time (GMT), otherwise known as the Epoch. The CLOCK_REALTIME clock uses nanoseconds as its lowest level of granularity.

Table 3–1 lists P1003.4/D10 timing functions for the specified clock.

Table 3–1 Clock Functions

Function	Description
clock_getres	Returns the resolution and maximum value of the specified clock
clock_gettime	Returns the current value for the specified clock
clock_gettimedrift	Returns the value of the clock drift rate as set by the most recent call to clock_settimedrift
clock_settime	Sets the specified clock to the specified value
clock_settimedrift	Sets the drift rate for the specified clock, in parts per billion (nanoseconds), to the specified value

Currently the only clock type supported is the CLOCK_REALTIME clock. Therefore, you use the name CLOCK_REALTIME as the *clock_id* argument in all P1003.4/D10 clock functions.

The values returned by the clock_gettime function can be used to determine values for the creation of realtime timers. Setting the clock or changing the drift rate for one application will not affect the expiration interval of armed timers.

The clock_getres function returns the maximum possible value that can be used in the clock_settime function to set the current system time. However, you cannot set the resolution of the specified clock.

3.1.1 Retrieving System Time

Both the time and clock_gettime functions return the value of the systemwide clock as the number of elapsed seconds since the Epoch. The timespec data structure (used for the clock_gettime function) also contains a member to hold the value of the number of elapsed nanoseconds not comprising a full second.

Example 3–1 shows the difference between the time as returned by the time and clock_gettime functions.

Example 3–1 Returning Time

```
#include <timers.h>
#include <time.h>

main()
{
struct timespec tp;

      /* Call time */

printf("time returns %d seconds\n", time((long *) 0));

       /* Call clock_gettime */

clock_gettime(CLOCK_REALTIME, &tp);
printf("clock_gettime returns:\n");
printf("%d seconds and %d nanoseconds\n", tp.tv_sec, tp.tv_nsec);
 }
```

Example 3–1 returns 876764530 seconds from the `time` function and returns 876764530 seconds and 0000674633 nanoseconds from the `clock_gettime` function.

The `time` function returns a long integer containing the number of seconds that have elapsed since the Epoch. The `clock_gettime` function passes a pointer to the `timespec` structure and assigns the values contained in the *tv_sec* and *tv_nsec* members to the *tp.tv_sec* and *tp.tv_nsec* variables.

If you plan to write the current time to a device or file, you may want to change the time format returned by the `clock_gettime` function. See Section 3.1.4 for information on converting the time format.

3.1.2 Setting the Clock

The `clock_settime` function lets you set the time for the specified clock. If you have an application that monitors time over the network use the `clock_settime` function to synchronize with other systems. However, under normal circumstances you would not need to call the `clock_settime` function.

Note that armed timers (pending execution) associated with the clock may be affected by resetting the systemwide clock. If timers are pending execution, use the `clock_settimedrift` function to adjust the clock slowly. Armed timers are not affected by the `clock_settimedrift` function.

You must have `superuser` privileges to use the `clock_settime` and `clock_settimedrift` functions.

3.1.3 Managing Clock Drift

As a form of interval timer, the CLOCK_REALTIME clock responds to interrupts with every clock tick. A counter associated with clock interrupts causes the clock time to deviate slightly over time. This deviation, or variance, can be either positive or negative. Applications using high-resolution timers may want to use the P1003.4/D10 clock drift functions to maintain the integrity of the clock.

The initial drift rate for any clock is zero, but over time, it may vary by intervals centered around zero. The clock gains time if the clock drift rate is positive; the clock loses time if the clock drift rate is negative. The P1003.4/D10 clock drift functions provide a way to slow down or speed up a clock while ensuring that time, as measured by the specified clock, is a monotonically increasing quantity. For example, if you need to set a clock back in time, use a negative drift to slow down the clock gradually until it converges to the correct value. Applications and timers that rely on the clock will suffer minimal impact from changes to the time base.

Use the `clock_settimedrift` function to fine tune the clock slowly by setting the clock's drift rate. Pending timeouts are not effected by applied drifts to the system clock. Nor is the drift reflected in the resolution of the system clock. Note that you need `superuser` privileges to use the `clock_settimedrift` function. The `clock_gettimedrift` function returns the value of the clock drift rate as set by the most recent call to the `clock_settimedrift` function.

The following example calls the `clock_settimedrift` function to set the clock drift rate to 1000 nanoseconds and prints out the previous drift rate for the clock as returned by the `clock_settimedrift` function. A call to the `clock_gettimedrift` function checks the new drift rate.

```
#include <timers.h>

#define SUCCESS 0

main(){
int clock_id = CLOCK_REALTIME;
int ppb = 1000;
int oppb;


if(clock_settimedrift(clock_id, ppb, &oppb) == SUCCESS)
   printf("previous ppb is %d\n", oppb);

if(clock_gettimedrift(clock_id, &oppb) == SUCCESS)
   printf("current ppb is %d\n",oppb);
}
```

Note that the clock drift functions should not be used in conjunction with Distributed Time Services (DTS).

3.1.4 Converting Time Values

Clock and timer functions use the number of seconds and nanoseconds since the Epoch. Although this method is precise and suitable for the machine, it is not meaningful for application users. If your application outputs or receives time information from users, you will want to convert time data into a more readable format.

If you use the `time` or `clock_gettime` function to retrieve system time, the input and return values are expressed in elapsed seconds since the Epoch. Your application should define the format for both user input and output and then convert these time values for use by the program. Applications can store the converted time values for future use.

The C language provides a number of functions to convert and store time in both a `tm` structure and an ASCII format. When you pass the time in seconds to these functions, some functions return a pointer to a `tm` structure, while others return an ascii string. Some functions also correct for time zones and daylight saving time. To select the most appropriate time conversion function for your application, refer to the reference pages for each of these functions.

Note that these C routines use seconds as the smallest unit of time. Table 3–2 lists the date and time conversion functions.

Table 3–2 Date and Time Conversion Functions

C Function	Description
`asctime`	Converts a broken-down time into a 26-character string
`ctime`	Converts a time in seconds since the Epoch to an ASCII string in the form generated by `asctime`
`difftime`	Computes the difference between two calendar times (time1–time0) and returns the difference expressed in seconds
`gmtime`	Converts a calendar time into a broken-down time, expressed as GMT
`localtime`	Converts a time in seconds since the Epoch into a broken-down time
`mktime`	Converts the broken-down local time in the `tm` structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by `time`
`tzset`	Sets the external variable *tzname*, which contains current timezone names

The converted time values for the functions listed in Table 3–2 are placed in a time structure (`tm`) defined in the `<time.h>` header file, as follows:

```
struct tm  {
       int tm_sec,                    /* Time in seconds (0-59)          */
           tm_min,                    /* Time in minutes (0-59)          */
           tm_hour,                   /* Time in hours  (0-23)           */
           tm_mday,                   /* Day of the month (1 to 31)      */
           tm_mon,                    /* Month (0 to 11)                 */
           tm_year,                   /* Year (last 2 digits)            */
           tm_wday,                   /* Day of the week (Sunday=0)      */
           tm_yday,                   /* Day of the year (0 to 365)      */
           tm_isdst;                  /* Daylight savings time (always 0) */
      long tm_gmtoff;                 /* Offset from GMT in seconds      */

     };
```

3.2 Types of Timers

Two types of timers are provided to support realtime timing facilities: one-shot and periodic timers. Timers can be set up to expire only once (one-shot) or on a repetitive (periodic) schedule. A one-shot timer is armed with an initial expiration time, expires only once, and then is disarmed. A periodic timer expires, and then reloads the repetition interval, rearming the timer.

Both types of timers are armed with an initial expiration value, but whether or not the timer expires only once or periodically depends on the value specified for the timer interval. If the interval is zero, the timer expires only once. If the interval is specified as a value other than zero, then the timer expires repetitively. After the initial expiration time, the repetition interval is loaded again and the timer continues.

The initial expiration value can be relative to the current time or an absolute time value. A relative timer has an initial expiration time based on the amount of time elapsed, such as 30 seconds from the start of the application or 0.5 seconds from the last timer expiration. An absolute timer has a specific initial expiration value.

Absolute and relative timers are often used in combination. You can use an absolute timer to determine when the timer first expires, and set subsequent timer expiration relative to the first expiration. For example, an application may need to collect data between midnight and 3:00 A.M. Data collection during this three hour period may be staged in 12 minute intervals. In this case, absolute timers are used to start and stop the data collection processes at midnight and 3:00 A.M. Relative timers are used to initiate data collection in 12 second intervals.

The values specified in the arguments to the `timer_settime` function determine whether the timer is a one-shot or periodic and absolute or relative timer. Refer to Section 3.6.2 for more information on the `timer_settime` function.

3.3 Data Structures Associated with Timing Facilities

The <time.h> header file contains structures for manipulating clock and timer constructs. The timespec and itimerspec data structures in <timers.h> are used in many of the P1003.4/D10 realtime clock and timer functions. The timespec data structure contains members for both second and nanosecond values. This data structure sets up a single time value and is used by many P1003.4/D10 functions that accept or return time value specifications. The itimerspec data structure contains two timespec data structures. This data structure sets up an initial timer and repetition value used by P1003.4/D10 per-process timer functions.

The <signal.h> header file contains a sigevent structure for specifying the signal number to be sent on timer expiration.

3.3.1 Using the timespec Data Structure

The timespec data structure consists of two members, *tv_sec* and *tv_nsec* and takes the following form:

```
typdef struct timespec{
     time_t tv_sec;                /* Seconds      */
     long tv_nsec;                 /* Nanoseconds  */
     }timespec;
```

The *tv_nsec* member is valid only if its value is greater than zero and less than the number of nanoseconds in a second. The total possible time interval described by the timespec structure is $tv\_sec * 10^9 + tv\_nsecs—1$ nanoseconds. (The total possible time interval is limited by the maximum resolution of the specified clock.)

The timespec structure is used in P1003.4/D10 functions to return and set the specified clock and to return the resolution of clocks, timers, and nanosleep.

3.3.2 Using the itimerspec Data Structure

The itimerspec data structure consists of two timespec structures and takes the following form:

```
typdef struct itimerspec{
       struct timespec it_interval;   /* Timer period     */
       struct timespec it_value;      /* Timer expiration */
       }itimerspec;
```

The two timespec structures specify an interval value and an initial expiration value, both of which are used in all timer functions related to setting up timers. The values specified for the member structures identify the timer as one-shot or periodic. Table 3–3 summarizes the ways that values for the two members of the itimerspec structure are used to characterize timers.

Table 3–3 Values Used in Setting Timers

Member	Value	Description	Function
it_value	Nonzero	Expiration value	Sets up an absolute or relative timer
it_value	Zero	No expiration value	Disarms a timer
it_interval	Nonzero	Interval reload value	Sets up a periodic timer
it_interval	Zero	No reload value	Sets up a one-shot timer

The *it_value* specifies the initial amount of time before the timer expires. A nonzero value for the *it_value* member indicates the amount of time until the first time the timer expires. A zero value for the *it_value* member disarms the timer.

Once the timer is called and has expired for the first time, the *it_interval* member specifies the interval after which the timer will expire again. That is, the value of the *it_interval* member is reloaded when the timer expires and timing continues. A nonzero value for the *it_interval* member specifies a periodic timer. A zero value for the *it_interval* member causes the timer to expire only once. Afterward the *it_value* member is set to zero and the timer is disarmed.

For example, to arm a timer to execute only once, 5.25 seconds from now, specify the following values for the members of the `itimerspec` structure:

```
mytimer.it_value.tv_sec = 5;
mytimer.it_value.tv_nsec = 250000000;
mytimer.it_interval.tv_sec = 0;
mytimer.it_interval.tv_nsec = 0;
```

The following example arms a timer to execute 15 seconds from now and then at 0.5 second intervals.

```
mytimer.it_value.tv_sec = 15;
mytimer.it_value.tv_nsec = 0;
mytimer.it_interval.tv_sec = 0;
mytimer.it_interval.tv_nsec = 500000000;
```

In the preceding examples, the timer is armed relative to the current time. To set up a timer with an absolute initial expiration time, such as 10:00 A.M., convert the absolute initial expiration value (in seconds and nanoseconds) to the correct offset from the current time.

Because the value of the *tv_nsec* member is expressed in nanoseconds, it may be somewhat cumbersome. To simplify specifying values for the *tv_nsec* member, define a symbolic constant.

```
#define NSECS_PER_SEC 1000000000;

mytimer.it_value.tv_nsec = NSECS_PER_SEC/4;
```

Or, use an assignment statement, such as this:

```
mytimer.it_value.tv_nsec = 1000000000/4;
```

See Section 3.6 for more information on relative and absolute timers.

3.3.3 Using the sigevent Data Structure

The sigevent structure delivers the signal on timer expiration. The *evp* argument of the timer_create function points to a *sigevent* structure, which contains the signal number of the signal to be sent upon expiration of each timer.

The *sigevent* structure is defined in the <signal.h> header file and contains the following members:

```
void       *sevt_value;   /* Not currently supported - specify as NULL */
signal_t    sevt_signo;   /* Signal sent on timer expiration           */
```

The *sevt_value* member is an application-defined value to be passed to the signal-catching function at the time of signal delivery. This member is used in P1003.4/D10 Realtime Signals, which are not currently supported. Specify a value of NULL for this member.

The *sevt_signo* member specifies the signal number to be sent on expiration of the timer.

3.4 Resolution of the System Clock and Timers

DEC OSF/1 provides three P1003.4/D10 functions that return the resolution values for the system clock, user-specified timers, and the nanosleep function. These resolution functions return the minimum number of seconds and nanoseconds supported by the specified clock.

In addition, the resolution functions return the maximum values allowed for setting the clock, timers, or the nanosleep function. Table 3–4 lists the resolution functions for P1003.4/D10 timing facilities.

Table 3–4 Resolution Functions for Timing Facilities

Function	Description
clock_getres	Returns the resolution and maximum value of the specified clock
nanosleep_getres	Returns the resolution and maximum value supported by nanosleep
timer_getres	Returns the resolution and maximum value of an absolute or relative timer

The following example calls the clock_getres function to determine clock resolution and the maximum value for the clock.

```
#include <malloc.h>
#include <timers.h>

main()
{
struct timespec     clock_resolution;
struct timespec     clock_max_value;
int stat;

stat = clock_getres(CLOCK_REALTIME, &clock_resolution, &clock_max_value);

printf("Clock resolution is %d seconds, %d nanoseconds\n",
     clock_resolution.tv_sec, clock_resolution.tv_nsec);

printf("The maximum time value is %d seconds, %d nanoseconds\n",
     clock_max_value.tv_sec, clock_max_value.tv_nsec);
}
```

3.5 Timers and Signals

You create a timer with the timer_create function, which is associated with a sigevent structure. When using timers, you set the timers with an initial expiration value and an interval value for when you want the timer to expire. When the timer expires, the kernel sends the specified signal to the process that called the timer. Therefore, you should set up a signal handler to catch the signal once it is sent to the calling process.

To use signals with timers, include the following steps into your application:

1. Create and declare a signal handler.
2. Set the sigevent structure to specify the signal you want sent on timer expiration.
3. Establish a signal handler.

4. Create the timer.

If identical signals are delivered from multiple timers, the signals are compressed into a single signal. Therefore, if you have multiple timers, you may want to specify a different signal for each timer. Refer to Chapter 11 for more information on signals and signal handling.

3.6 Timer Functions

Clocks and timers allow an application to synchronize and coordinate activities according to a user-defined schedule. DEC OSF/1 P1003.4/D10 timers have the ability to issue periodic timer requests initiated by a single call from the application. You can have more than one outstanding timer interval on the same timer type in order to trigger different functions.

Table 3–5 lists the P1003.4/D10 timing functions available for realtime applications.

Table 3–5 Timer Functions

Function	Definition
`timer_create`	Returns a unique timer ID used in subsequent calls to identify a timer based on the systemwide clock
`timer_delete`	Removes a previously allocated, specified timer
`timer_getres`	Returns the resolution and maximum value of the specified clock
`timer_gettime`	Returns the amount of time before the specified timer is due to expire and the repetition value
`timer_settime`	Sets the value of the specified timer either to an offset from the current clock setting or to an absolute value

The value returned by the `timer_getres` function depends on whether the timer is an absolute or relative timer and may not be the same as the value returned by the `clock_getres` function.

Note that timers do not have global IDs, which means that they are not inherited by a child process after a call to the `fork` or `exec` system calls. You cannot arm a timer, call the `exec` system call, and have the `exec` image receive the signal.

3.6.1 Creating Timers

The timer_create function allocates a timer, returns a timer ID, and points to a sigevent structure. The timer ID is unique within the calling process and exists for the life of that timer. The timer ID does not exist until it is returned by the timer_create function. The timer is not armed until you make a call to the timer_settime function, which sets the values for the specified timer.

The timer functions perform a series of tasks necessary for setting up timers. To create a timer, you must set up appropriate data structures, call the timers, and set up a signal handler to catch the signal when the timer expires. To use timers in a realtime application, follow these steps:

1. Include <timers.h> and <signal.h> in the application source file.
2. Declare the variable names for your itimerspec data structure to specify interval and expiration values.
3. Establish a sigevent structure containing the signal to be passed to the process on timer expiration. (P1003.4/D10 Realtime Signals are not supported in this release, so specify NULL as the value of the *data* member of the sigevent structure.)
4. Set up a signal handler in the calling process to catch the signal when the timer expires.
5. Call the timer_create function to create a timer and associate it with the specified clock. Specify a signal to be delivered when the timer expires.
6. Initialize the itimerspec data structure with the required values.
7. Call the timer_settime function to initialize and activate the timer as either an absolute or relative timer.
8. Call the timer_delete function when you want to remove the timer.

The maximum number of per-process timers (TIMER_MAX) is defined in the <limits.h> header file.

The timer_create function also takes an *evp* argument, which is a pointer to a sigevent structure. This structure defines the signal to be sent to the calling process when the timer expires. You can either use the default signal, SIGALRM, or you can specify a particular signal.

3.6.2 Setting Timer Values

The `timer_settime` function determines whether the timer is an absolute or relative timer. This function sets the initial expiration value for the timer as well as the interval time used to reload the timer once it has reached the initial expiration value. The arguments for the `timer_settime` function perform the following functions:

1. The *timerid* argument identifies the timer.
2. The *abstime* argument determines whether the timer behaves as an absolute or relative timer.

   The *abstime* argument determines whether a timer will function as an absolute or relative timer. If the *abstime* argument is a nonzero value, the timer is treated as an absolute timer; if the *abstime* argument is zero, the timer is treated as a relative timer.
3. The *value* argument determines the initial expiration value and repetition value for the timer.

   - The *it_value* member of the *value* argument establishes the initial expiration time.

     If it is an absolute timer, the `timer_settime` function interprets the next expiration value as equal to the difference between the absolute time specified by the *it_value* member of the *value* argument and the current value of the specified clock. The timer then expires when the clock reaches the value specified by the *it_value* member of the *value* argument.

     If it is a relative timer, the `timer_settime` function interprets the next expiration value as equal to the interval specified by the *it_value* member of the *value* argument. The timer will expire in *it_value* seconds and nanoseconds from when the call was made. After a timer is started as an absolute or relative timer, its behavior is driven by whether it is a one-shot or periodic timer.
   - The *it_value* member of the *value* argument can disable a timer.

     To disable a periodic timer, call the timer and specify the value of zero for the *it_value* member.
   - The *it_interval* member of the *value* argument establishes the repetition value.

     The timer interval is specified as the value of the *it_interval* member of the `itimerspec` structure in the *value* argument. This value determines whether the timer functions as a one-shot or periodic timer.

After a one-shot timer expires, the expiration value (*it_value* member) is set to zero. This indicates no next expiration value is specified, which disarms the timer.

A periodic timer is armed with an initial expiration value and a repetition interval. When the initial expiration time is encountered, it is then loaded again with the repetition interval and the timer starts again. This continues until the application exits. To arm a periodic timer, set the *it_value* member of the *value* argument to the desired expiration value and set the *it_interval* member of the *value* argument to the desired repetition interval.

4. The *ovalue* argument stores the timer expiration values returned by the `timer_gettime` function. If the timer is not armed, the *ovalue* is equal to zero. If you displace an active timer, the *ovalue* will contain the amount of time remaining in the interval.

Note that if you call the `timer_settime` function and pass a nonzero value to the *it_interval* member of the *value* argument, you can effectively change the timer from a one-shot to a periodic timer. You cannot call the `timer_settime` function to reset or disarm a timer that is pending execution.

3.6.3 Retrieving Timer Values

The `timer_gettime` function returns two values: the amount of time before the timer expires and the repetition value set by the last call to the `timer_settime` function. If the timer is disarmed, a call to the timer with the `timer_gettime` function returns a zero for the value of the *it_value* member. To arm the timer again, call the `timer_settime` function for that timer ID and specify a new expiration value for the timer. When the timer is called again, it is rearmed.

The `timer_getres` function returns the resolution and the maximum value of the specified timer. The resolution of absolute and relative timers may be different. Also, timer resolution may be different from clock resolution.

3.6.4 Disabling Timers

Once a one-shot timer expires, the timer is disarmed, but the timer ID is still valid. The timer ID is still current and can be rearmed with a call to the `timer_settime` function. To remove the timer ID and disable the timer, use the `timer_delete` function. Note that if you delete a timer that is still armed no signal will be sent.

3.7 High-Resolution Sleep

To suspend process execution temporarily using the P1003.4/D10 timer interface, call the `nanosleep` function. The `nanosleep` function suspends execution for a specified number of nanoseconds, providing a high-resolution sleep. A call to the `nanosleep` function suspends execution until either the specified time interval expires or a signal is delivered to the calling process.

Only the calling thread sleeps with a call to the `nanosleep` function. In a threaded environment, other threads within the process continue to execute.

The `nanosleep` function has no effect on the delivery or blockage of signals. The action of the signal must be to invoke a signal-catching function or to terminate the process. When a process is awakened prematurely, the *rmtp* argument contains the amount of time remaining in the interval minus the amount of time the process actually slept.

The `nanosleep_getres` function returns the `nanosleep` resolution as well as the maximum value supported by the `nanosleep` function.

3.8 Clocks and Timers Example

Example 3–2 demonstrates the use of P1003.4/D10 realtime timers. The program creates both absolute and relative timers. The example demonstrates concepts using multiple signal types to distinguish between timer expirations. The program loops continuously until the program is terminated by a Ctrl/C from the user.

Example 3–2 Using Timers

```
 /* The following program demonstrates the use P1003.4/D10    */
 /* Realtime Timers in conjunction with POSIX 1003.1 Signals. */
 /* The program creates a set of timers and then blocks       */
 /* waiting for either timer expiration or program termination*/
 /* via SIGINT. */

#include <sys/types.h>
#include <sys/limits.h>
#include <sys/time.h>
#include <sys/signal.h>
#include <timers.h>
#include <sys/errno.h>

      /* Constants and Macros */

#define NULL 0
#define SUCCESS  0
#define FAILURE -1
#define TRUE  1
#define FALSE 0
#define ABS 1
#define REL 0
#define TIMERS 3

#define MIN(x,y) (((x) < (y)) ? (x) : (y))

sig_handler();
void timeaddval();
struct sigaction sig_act;

      /* Control structure for timer examples */

struct timer_definitions {
       int type;                      /* Absolute or Relative Timer  */
       struct sigevent evp;           /* Event structure             */
       struct itimerspec timeout;     /* Timer interval              */
       };


/* Initialize timer_definitions array for use in example as follows: */
/*   { type, { sevt_value, sevt_signo }, { it_iteration, it_value } }*/

struct timer_definitions timer_values[TIMERS] = {
      { ABS, {NULL,SIGALRM}, {0,0, 10,5} },
      { ABS, {NULL,SIGUSR1}, {0,1000000, 5,30000} },
      { REL, {NULL,SIGUSR2}, {0,0, 0,500000} }
      };

timer_t timerid[TIMERS];
int timers_available;                  /* number of timers available */

      /* This program demonstrates the use of P1003.4/D10 timers */
```

(continued on next page)

Example 3–2 (Cont.) Using Timers

```
void timer_example()
{
      int status, i;
      int clock_id = CLOCK_REALTIME;
      struct timespec current_time;
      struct timeval time;

      /*  Initialize the sigaction structure for the handler. */

      sig_act.sa_handler = (void *)sig_handler;
      sig_act.sa_flags = 0;
      sigemptyset(&sig_act.sa_mask);

      /*  Determine if it's possible to create TIMERS timers. */
      /*  If not, create TIMER_MAX timers.                    */

      timers_available = MIN(TIMER_MAX,TIMERS);

      /* Create "timer_available" timers, using a unique signal    */
      /* type to denote the timers expiration. Then initialize     */
      /* a signal handler to handle timer expiration for the timer.*/

      for (i = 0; i < timers_available; i++) {
           timerid[i] = timer_create(clock_id, &timer_values[i].evp);
                if (timerid[i] == FAILURE) {
                     perror("Failed to create_timer: ");
                     exit(FAILURE);
                     }
           sigaction(timer_values[i].evp.sevt_signo, &sig_act, 0);
           }

       /* Establish a handler to catch crtl-c and use it for exiting. */

      sigaction(SIGINT, &sig_act, 0);         /* Catch Ctrl-C */

      /* Queue the following Timers: (see timer_values structure    */
      /* for details)                                               */
      /* 1.  An absolute one-shot timer  (Notification via SIGALRM).*/
      /* 2.  An absolute periodic timer. (Notification via SIGUSR1).*/
      /* 3.  A relative one shot timer.  (Notification via SIGUSR2).*/

      /* The number of TIMERS queued depends on timers_available    */
```

(continued on next page)

Example 3–2 (Cont.) Using Timers

```
      for (i = 0; i < timers_available; i++) {
           if (timer_values[i].type == ABS) {
                status = clock_gettime(CLOCK_REALTIME,&current_time);
                timeaddval(&timer_values[i].timeout.it_value,
                &current_time);
                }
           status = timer_settime(timerid[i], timer_values[i].type,
                &timer_values[i].timeout, NULL);
           if (status == FAILURE) {
                perror("timer_settime failed: ");
                exit(FAILURE);
                }
           }

      /* Loop forever.  The application will exit in the signal */
      /* handler when a SIGINT is issued (Ctrl/C will do this). */

      for(;;) pause();
}

      /*  Handle timer expiration or program termination. */

sig_handler(signo)
int signo;
{
     int i, status;

     switch (signo) {
           case SIGALRM: /* SIGARLM notifications, do required processing*/
                break;
           case SIGUSR1: /* SIGUSR1 notifications, do required processing*/
                break;
           case SIGUSR2: /* SIGUSR2 notifications, do required processing*/
                break;
           case SIGINT:
                for (i = 0; i < timers_available; i++)  /* Delete timers    */
                     status = timer_delete(timerid[i]);
                exit(1);  /* Exit if Ctrl/C is issued */
        }
     return;
}

      /* Add two timevalues: t1 = t1 + t2 */
```

(continued on next page)

Example 3–2 (Cont.) Using Timers

```
void timeaddval(t1, t2)
struct timespec *t1, *t2;
     {
        t1->tv_sec += t2->tv_sec;
        t1->tv_nsec += t2->tv_nsec;
        if (t1->tv_nsec < 0) {
                t1->tv_sec--;
                t1->tv_nsec += 10000000;
                }
        if (t1->tv_nsec >= 1000000) {
                t1->tv_sec++;
                t1->tv_nsec -= 10000000;
                }
      }

main()
{
        timer_example();
}
```

4

Memory Locking

Memory management facilities ensure that processes have effective and equitable access to memory resources. The operating system maps and controls the relationship between physical memory and the virtual address space of a process. These activities are, for the most part, transparent to you and controlled by the operating system. However, for many realtime applications you may need to make more efficient use of system resources by explicitly controlling virtual memory usage.

Memory locking is one way to ensure that a process stays in primary memory and is exempt from paging. In a realtime environment, a system must be able to guarantee that it will lock a process in memory to reduce latency for data access, instruction fetches, buffer passing between processes, and so forth. Locking a process's address space in memory helps ensure that the application's response time satisfies realtime requirements. As a general rule, time-critical processes should be locked into memory.

This chapter covers the following sections:

- Memory Management, Section 4.1
- Memory Areas, Section 4.2
- Memory-Locking and Unlocking Functions, Section 4.3
- Memory-Locking Example, Section 4.4

Note that memory locking controls access to system resources, while shared memory and semaphores are used to synchronize the order in which multiple processes use resources. Refer to Chapter 8 and Chapter 9 for information on using shared memory and semaphores.

4.1 Memory Management

In a multiprogramming environment, it is essential for the operating system to share available memory effectively among the processes. Memory management policies are directly related to the amount of memory required to execute those processes. Memory management algorithms are designed to optimize the number of runnable processes in primary memory while avoiding conflicts that adversely affect system performance. If a process is to remain in memory, the kernel must allocate adequate units of memory. If only part of a process needs to be in primary memory at any given time, then memory management can work together with the process scheduler to make optimal use of resources.

Virtual address space is divided into fixed-sized units, called pages. Each process is usually composed of a number of pages, which are independently moved in and out of primary memory as a process executes. Normally, a subset of a process's pages resides in primary memory when the process is executing.

Since the amount of primary memory available is finite, paging is often done at the expense of some pages; to move pages in, others must be moved out. If the page that is going to be replaced is modified during execution, that page is written into a paging file on disk. This page is brought back into primary memory as needed, again replacing a current page. Execution is delayed while the kernel retrieves the needed page.

Paging is generally transparent to the process. However, if the process is very large or if pages are frequently being paged in and out, the system overhead required for paging may decrease process efficiency. The amount of paging can be decreased by increasing the size of physical memory or by locking the pages into memory.

For realtime applications, having adequate memory is more important than for nonrealtime applications. Realtime applications must ensure that processes are locked into memory and that there is an adequate amount of memory available for both realtime processes and the system. Latency due to paging is often unacceptable for critical realtime tasks.

4.2 Allocating Memory

In a timesharing environment, the goal is to share all resources equally while at the same time not using more memory than is absolutely necessary. In a realtime environment the goal is to reduce latency and emphasize process performance, even if it is at the expense of less critical processes. As a result, realtime applications usually lock realtime processes into memory.

Memory-locking functions provide explicit control over memory allocation for realtime processes. These functions give the user complete control over paging to help guarantee response time.

The application may explicitly extend its virtual memory on request by calling one of the memory management functions. The malloc, calloc, and alloca functions provide a way to allocate additional memory. The malloc function maintains a list of free blocks, according to size, and then calls the sbrk function to get more memory from the system when there is no suitable space free. The calloc function allocates space for an array with a specified number of elements and initializes the space to zeros. The alloca function allocates the specified number of bytes in the caller's stack frame. This stack space is automatically freed when the function that called the alloca function returns to the caller.

Once some space has been allocated, you can call the mallinfo function to determine the optimum parameter settings for these functions.

The sbrk function adds bytes to the data segment and returns a pointer to the new area. If the maximum size of the data segment is exceeded or there is a temporary lack of space, an error is returned. To reduce the number of errors, use the sysconf function first to determine limits defined for the data segment.

The sbrk function can also be used to determine the amount of data or text space used by a process during execution or to change the size of the space. Call sbrk(0) at the beginning of execution and then again at the end of execution. The difference between the two returned values is equal to the amount of space used by the process. The following example returns the initial value of *init_pointer*:

```
init_pointer = sbrk(0);
```

Also use the sbrk call to extend the amount of memory associated with a specific process. The following example changes the size of the memory area by *incr* number of bytes and returns the new end of the area:

```
addr = sbrk(incr);
end_result caddr_t addr;
int incr;
```

If you use sbrk to change the space allocated for your process, first use the sysconf function to determine the correct argument value to the sbrk function.

The getrlimit function may be used to determine the maximum possible size of the data segment. You can use the setrlimit function to set resources. Using the setrlimit function, it is not possible to set the break beyond the value returned from a call to the getrlimit function. However, DEC OSF/1 provides the mmap function, which allows the use of discontiguous memory

areas. The mmap function creates a new mapped file or shared memory region. Using the mmap function, references to areas above the break may be legitimate memory references which will not produce memory violations.

Since the sbrk function is not a POSIX function, you may want to refer to the online reference pages for the calloc, alloca, and mmap functions to determine which method is best for your application.

Example 4–1 uses the getrlimit, malloc, sbrk, and sysconf functions to allocate additional memory.

Example 4–1 Allocating Additional Memory

```
#include <sys/resource.h>  /* for getrlimit function */
#include <time.h>          /* for getrlimit function */
#include <malloc.h>        /* for malloc function    */
#include <unistd.h>        /* for sysconf function   */

#define NULL 0
#define MEM_CHUNK_PAGES 8192
#define CHECK_STAT(stat, msg)   \
     if (stat == -1) {          \
          perror(msg);          \
          exit(-1);             \
     }

long page_size;
char *additional_space;
struct rlimit data_limit;
long data_pages;

void show_data_limits()
{
int stat;
int data_free;
int cur_data_break;

      /* Find the maximum allowable size of the data segment */

stat = getrlimit(RLIMIT, &data_limit);
CHECK_STAT(stat, "Get limit for data size");
data_free = data_limit.rlim_max - data_limit.rlim_cur;
printf ("\n Data Size limit from getrlimit \
         \n     maximum limit is ..... 0x%8.8x bytes \
         \n     current limit is ..... 0x%8.8x bytes \
         \n     can still use ........ 0x%8.8x bytes \n",
         data_limit.rlim_max, data_limit.rlim_cur, data_free);

      /* Find out how many pages are allowed in the data segment */
```

(continued on next page)

Example 4–1 (Cont.) Allocating Additional Memory

```
data_pages = data_limit.rlim.cur / page_size;
printf ("Pages in data segment ..... 0x%8.8x \n",data pages);

      /* Return the current data segment break */

cur_data_break = sbrk(0);
printf ("Data breaks at address ..... 0x%8.8x \n", cur_data_break);
}

void allocate_memory()
{
int mem_chunk_bytes;

      /* Allocate additional data space */

mem_chunk_bytes = MEM_CHUNK_PAGES * page_size;
printf ("\n Allocate 0x%8.8x pages of memory", MEM_CHUNK_PAGES);
printf ("\n Allocate 0x%8.8x bytes of memory \n", mem_chunk_bytes);
additional_space = (char *) malloc(mem_chunk_bytes);
printf ("Address of memory is ..... 0x%8.8x \n", additional_space);
}

void grow_data_segment()
}
int stat;
int more_pages;

printf ("Not enough space in data segment \n");
printf ("Increase using setrlimit \n");

more_pages = MEM_CHUNK_PAGES - data_pages;
data_pages = data_pages + more_pages + 16;
data_limir.rlim_cur = data_pages * page_size;
stat = setrlimit (RLIMIT_DATA, &data_limit);
CHECK_STAT (stat, "Set new limit for data size");
}

main ()
{

      /* Use sysconf to get page size */

page_size = sysconf(_SC_PAGE_SIZE);
printf ("Page size is ..... 0x%4.4x bytes \n", page_size);

show_data_limits();
allocate_memory();
```

(continued on next page)

Example 4–1 (Cont.) Allocating Additional Memory

```
if additional_space == NULL) {
     grow_data_segment();
     show_data_limits();
     allocate_memory();
     }
}
```

4.3 P1003.4/D10 Memory-Locking and Unlocking Functions

Realtime application developers should consider memory locking as a required part of program initialization. Many realtime applications remain locked for the duration of execution, but some may want to lock and unlock memory as the application runs. DEC OSF/1 P1003.4/D10 memory-locking functions let you lock the entire process at the time of the function call and throughout the life of the application, or to selectively lock and unlock as needed.

Two P1003.4/D10 functions allow you to lock memory, which makes the process memory or segments of the process immune to paging. The `mlock` function lets you lock an address range into memory, and the `mlockall` function lets you lock all of a process's memory (both current and future). You can remove memory locks with corresponding calls to the `munlock` and `munlockall` functions.

DEC OSF/1 P1003.4/D10 memory-locking functions offer the following advantages:

- Portability
- Ability to preallocate and lock memory space
- Ability to lock memory with fewer function calls

Realtime applications often need to lock the entire process for the life of the application. Memory-locking functions globally track which regions are locked and which are not. If the data or text segment of a process is shared, then locked data or text is locked for all sharing processes.

Table 4–1 lists the P1003.4/D10 memory functions.

Table 4–1 Memory-Locking Functions

Function	Description
mlockall	Locks a process's address space
munlockall	Unlocks a process's address space
mlock	Locks a specified region of a process's address space
munlock	Unlocks a specified region of a process's address space

The P1003.4/D10 memory functions support locking the entire address space or a selected range of one or more pages. The address must be on a page boundary and all pages mapped by the specified range are locked. Therefore, you must determine how far the return address is from a page boundary and align it before making a call to the mlock function.

Use the sysconf(_SC_PAGE_SIZE) function to determine the page size. The size of a page can vary from system to system. To ensure portability, call the sysconf function as part of your application or profile when writing applications that use the memory-locking functions. The <mlock.h> header file defines the maximum amount of memory that can be locked. Use the getrlimit function to determine the amount of total memory.

Example 4–2 allocates data space, determines page size, determines if the new memory is aligned, then locks and unlocks the new segment into memory.

Example 4–2 Aligning and Locking a Memory Segment

```
#include <unistd.h>    /* for sysconf function         */
#include <malloc.h>    /* for malloc function          */
#include <mlock.h>     /* for memory locking functions */

#define NEW_SEGMENT_SIZE 65536
#define CHECK_STAT(stat,msg)   \
     if (stat == -1{           \
          perror(msg);         \
          exit(-1);            \
     }

char  *additional_space;

lock_memory(mem_addr, mem_size)
     char *mem_addr, int mem_size;

{
     int stat;
     long page_size;
     int page_offset;
     long mem_lock_addr, mem_lock_size;

page_size = sysconf(_SC_PAGE_SIZE);

      /* Determine if memory is on page boundary */

page_offset = (long)mem_addr % page_size;
printf ("Distance from page boundary is ..... 0x%4.4x bytes \n",
     page_offset);

mem_lock_addr = (long)mem_addr - page_offset;
mem_lock_size = mem_size + page_offset;
printf ("Start memory lock at address ..... 0x%8.8x \n",
     mem_lock_addr);

      /* Lock new segment into memory */

stat = mlock(mem_lock_addr, mem_lock_size);
CHECK_STAT (stat, "mlock part of memory");

      /* Unlock segment from  memory  */

stat = munlock(mem_lock_addr, mem_lock_size);
CHECK_STAT (stat, "munlock part of memory");
}

main ()
{
      /* Allocate bytes of data */

additional_space = (char *) malloc(NEW_SEGMENT_SIZE);
printf ("Address of data to lock is ..... 0x%8.8x \n",additional_space);
```

(continued on next page)

Example 4–2 (Cont.) Aligning and Locking a Memory Segment

```
lock_memory(additional_space, NEW_SEGMENT_SIZE);
}
```

Exercise caution when you lock memory; if your processes require a large amount of memory and your application locks memory as it executes, your application may take resources away from other processes. In addition, you could attempt to lock more virtual pages than can be contained in physical memory.

Note that the memory-locking functions should not be used in conjunction with Distributed Time Services (DTS). You need `superuser` privileges to use the P1003.4/D10 memory-locking functions.

4.3.1 Locking Memory

Memory locking applies to a process's address space. Only the pages mapped into a process's address space can be locked into memory. When the process exits, pages are removed from the process's address space and the locks are removed. Shared memory may be mapped into more than one address space, and multiple locks may be set on pages within these address mappings. Memory locking applies to any shared libraries that the application is linked against. Therefore, all such locks must be removed before the shared pages are unlocked.

Two P1003.4/D10 functions, `mlock` and `mlockall`, are used to lock memory. The `mlock` function allows the calling process to lock a selected region of address space used by a process. The `mlockall` function causes all of a process's address space to be locked. Locked memory remains locked until either the process exits or the application calls the corresponding `munlock` or `munlockall` function.

Memory locks are not inherited across a `fork` and all memory locks associated with a process are unlocked on a call to the `exec` function or when the process terminates.

For most realtime applications the following control flow minimizes program complexity and achieves greater determinism by locking the entire address into memory.

1. Perform non-realtime tasks, such as open files or allocate memory
2. Lock the address space of the process calling `mlockall` function
3. Perform realtime tasks

4. Release resources and exit

4.3.1.1 Locking a Specified Region

The `mlock` function locks a preallocated specified region. The address and size arguments of the `mlock` function determine the boundaries of the preallocated region. On a successful call to the `mlock` function, the specified region becomes locked. Memory is locked by the system according to system-defined pages. If the address and size arguments specify an area smaller than a page, the kernel rounds up the amount of locked memory to the next page. The `mlock` function locks all pages containing any part of the requested range, which can result in locked addresses below and above the requested range.

Repeated calls to `mlock` could request more physical memory than is available, subsequent processes must wait for locked memory to become available. Preallocating and locking regions is recommended for realtime applications. Realtime applications often cannot tolerate the latency introduced when a process must wait for lockable space to become available.

If the process requests more locked memory than will ever be available in the system, an error is returned.

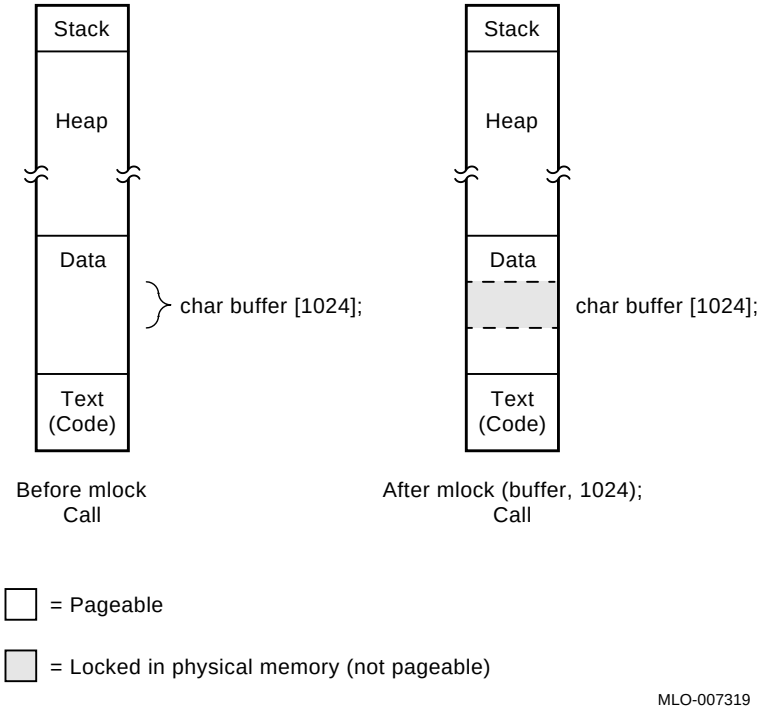
Figure 4–1 illustrates memory allocation before and after a call to the `mlock` function. Prior to the call to the `mlock` function, buffer space in the data area is not locked and is therefore subject to paging. After the call to the `mlock` function the buffer space cannot be paged out of memory.

4.3.1.2 Locking an Entire Process Space

The `mlockall` function locks all of the pages mapped by a process's address space. On a successful call to `mlockall`, the specified process becomes locked and memory resident. The `mlockall` function takes two flags, MCL_CURRENT and MCL_FUTURE, which determine whether the pages mapped are those currently used by the process or if any pages mapped in the future are to be locked. You must specify at least one flag for the `mlockall` function to lock pages. If you specify both flags, the address space to be locked is constructed from the logical OR of the two flags.

If you specify MCL_CURRENT only, all currently mapped pages of the process's address space are memory resident and locked. Subsequent growth in any of the specified region is not locked into memory. If you specify the MCL_ FUTURE flag, all future pages are locked in memory. If you specify both MCL_ CURRENT and MCL_FUTURE, then the current pages are locked and the size of the locked process or segment grows as the process's address space grows

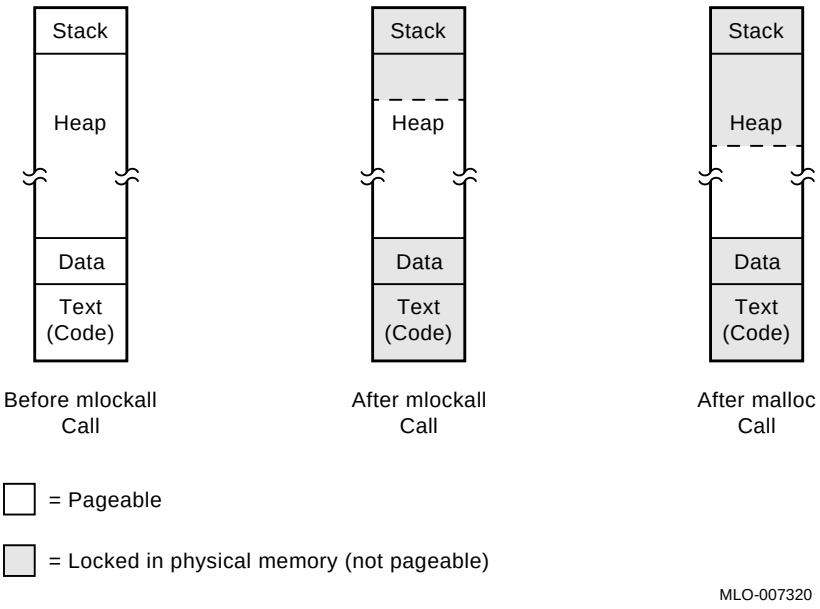
Figure 4–1 Memory Allocation with mlock



when the process executes. Subsequent growth is automatically locked into memory.

Figure 4–2 shows memory allocation before and after a call to the `mlockall` function. Prior to the call to the `mlockall` function, space is not locked and is therefore subject to paging. After a call to the `mlockall` function, which specifies the MCL_CURRENT and MCL_FUTURE flags, all memory used by the process both currently and in the future is locked into memory. The call to the `malloc` function increases the amount of memory locked for the process.

Figure 4–2 Memory Allocation with mlockall



4.3.2 Unlocking Memory

Locked space is automatically unlocked when the application exits, but you can also explicitly unlock space. The `munlock` function unlocks the specified address range regardless of the number of times the `mlock` function was called. In other words, you can lock address ranges over multiple calls to the `mlock` function but remove the locks with a single call to the `munlock` function. Space locked with a call to the `mlock` function must be unlocked with a corresponding call to the `munlock` function.

The `munlockall` function unlocks all pages mapped by a call to the `mlockall` function, even if the MCL_FUTURE flag was specified on the call to the `mlockall` function. After the call to the `munlockall` function, the MCL_FUTURE flag is canceled. Additional locking can be accomplished only with another call to a memory-locking function, such as `mlockall` with the MCL_CURRENT flag.

4.4 Memory-Locking Example

Example 4–3 demonstrates how to use all options allowed in the memory-locking functions. The example locks and unlocks memory, allocates additional memory, and uses the current and future capabilities of the `mlockall` function.

Example 4–3 Using the Memory-Locking Functions

```
/* This program demonstrates how to use the P1003.4/D10 */
/* memory-locking functions.*/

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <mlock.h> /* memory locking definitions */

main()
{
     char *allocated_buffer;
     char buffer[32768];
     unsigned int size;
     int  pages, status;

      /* Lock the array buffer[] into memory */

     if (mlock(buffer, sizeof(buffer)))
          printf ("mlock error %d \n", errno);

      /* Unlock the array buffer[] */

     if (munlock(buffer, sizeof(buffer)))
          printf ("munlock error %d \n", errno);

      /* Lock the process's CURRENT address space */

     if (mlockall(MCL_CURRENT))
          printf ("mlockall CURRENT error %d \n", errno);

      /* Allocate some memory -- This memory is pageable */
      /* unless it is explicilty locked. Free the memory */

     allocated_buffer = (char *)malloc (sizeof(buffer));
     free (allocated_buffer);

      /* Unlock the entire address space of the process */

     if (munlockall())
          printf ("munlockall error %d \n", errno);

      /* Lock the process's CURRENT address space and        */
      /* automatically lock any new memory that is allocated */
```

(continued on next page)

Example 4–3 (Cont.) Using the Memory-Locking Functions

```
     if (mlockall(MCL_FUTURE))
          printf ("mlockall FUTURE error %d \n", errno);

      /* Allocate some memory -- This memory is NOT pageable  */
      /* unless it is explicilty unlocked. Free the memory    */

     allocated_buffer = (char *)malloc (sizeof(buffer));
     free (allocated_buffer);

      /* Unlock the entire process address space */

     if (munlockall())
          printf ("munlockall error %d \n", errno);
}
```

5

Asynchronous Input and Output

Your program can handle input/output (I/O) operations on a file in one of two ways: synchronously or asynchronously. When operating synchronously, the process calling the I/O request is blocked until the I/O operation is complete and regains control of execution only when the request is completely satisfied. When operating asynchronously, the process calling the I/O request immediately regains control of execution once the I/O operation is queued to the device. When the I/O operation is completed (either successfully or unsuccessfully), the calling process can be notified of the event by a signal.

Using asynchronous I/O can increase the throughput of a process because the process continues execution while the system completes the final steps of the I/O operation. Asynchronous I/O allows you to overlap process execution and I/O operations. This overlapping allows one process to perform several separate simultaneous I/O operations while the calling process continues application processing.

This chapter includes the following sections:

- Data Structures Associated with Asynchronous I/O, Section 5.1
- Asynchronous I/O Functions, Section 5.2
- Asynchronous I/O Example, Section 5.3

Asynchronous I/O is most commonly used in realtime applications requiring high-speed or high-volume data collection or low priority journaling functions. Compute-intensive processes can use asynchronous I/O instead of polling for completion or blocking. For example, an application may collect intermittent data from multiple channels. Because the data arrives asynchronously, that is, when it is available rather than according to a set schedule, the receiving process must queue up the data to be read and immediately be free to receive the next data transmission. Another application may require such a high volume of reads, writes, and computations that it becomes practical to queue up a list of I/O operations and continue processing while the I/O requests are

being serviced. Applications can perform multiple I/O operations to multiple devices while making a minimum number of function calls. The P1003.4/D10 asynchronous I/O functions are designed to help meet these realtime needs.

You can perform asynchronous I/O operations using any open file descriptor, including named pipes and sockets.

5.1 Data Structures Associated with Asynchronous I/O

Many of the asynchronous I/O functions use the asynchronous I/O control block (aiocb). This control block contains asynchronous operation information, such as the initial point for the read operation, the number of bytes to be read, the priority of the I/O operation, the number of the signal sent on completion, and status information. The control block contains information similar to that required for a read or write function, but additionally contains members specific to asynchronous I/O operations. The aiocb structure contains the following members:

```
int             aio_whence;   /* Initial position of file pointer */
off_t           aio_offset;   /* Number of bytes in the offset    */
volatile char   *aio_buf;     /* Character pointer to information */
size_t          aio_nbytes;   /* Number of bytes                  */
int             aio_reqprio;  /* Priority of I/O operation        */
struct sigevent aio_event;    /* Number of the signal to be sent  */
int             aio_flag;     /* I/O completion status flag       */
aiohandle_t     aio_handle;   /* Identifies the I/O operation     */
```

Note that you cannot reuse the aiocb structure while an asynchronous I/O request is pending. To determine if the aiocb is in use, use the aio_return and aio_error functions.

5.1.1 Identifying the Location

The *aio_whence*, *aio_offset*, and *aio_nbytes* members provide information about the starting point and length of the data to be read or written. The *aio_buf* and *aio_nbytes* members provide information about where the information is located in memory.

The *aio_whence* and *aio_offset* members of the aiocb structure provide the asynchronous I/O functions with the same information that would be supplied to the lseek function. That is, asynchronous I/O functions perform an implied call to the lseek function. You do not have to call the lseek function to reset the offset for an open file.

Note, however, that the behavior of the implied lseek function is different from an explicit call. A subsequent unsuccessful call to the read or write function does not advance the pointer; the file offset remains unchanged. An unsuccessful call to an aio_read or aio_write function can change the pointer.

5.1.2 Setting the Priority

The *aio_reqprio* member specifies a priority for the asynchronous I/O operation. The priority assigned to each I/O request is an indication of the preferred order of execution relative to other I/O requests made by a single process. Numerically higher values indicate higher priority requests.

Specifying priority on an asynchronous I/O request is a way to order the execution of a series of I/O requests issued from a single process. This prioritizing is independent of process scheduling and has no effect on the priority of the calling process.

The value of the *aio_reqprio* member must be within the range AIO_PRIO_MIN and AIO_PRIO_MAX (the minimum and maximum value for the *aio_reqprio* member). You can specify a default priority by using the value AIO_PRIO_DFL.

Higher priority requests are submitted before lower priority requests. Asynchronous I/O requests issued at the same priority are executed on a first-in/first-out basis.

5.1.3 Specifying a Signal

You can send a signal on completion of every read and write operation, regardless of whether the operation is issued from a call to the `aio_read`, `aio_write`, or `lio_listio` function. In addition, you can send a signal on completion of the `lio_listio` function. See Chapter 11 for more information on signals and signal handling.

The *aio_event* member refers to a `sigevent` structure, which contains the signal number of the signal to be sent upon completion of the asynchronous I/O request. The `sigevent` structure is defined in the `signal.h` header file and contains the following members:

```
void      *sevt_value;     /* Not currently supported - specify as NULL */
signal_t   sevt_signo;     /* Signal sent on I/O completion             */
```

The *sevt_value* member is an application-defined value to be passed to the signal-catching function at the time of signal delivery. This member is used in P1003.4/D10 Realtime Signals, which are not currently supported. Specify a value of NULL for this member.

The *sevt_signo* member specifies the signal number to be sent on completion of the operation. The *aio_flag* member indicates whether or not a signal will be sent to the calling process when the operation completes. You can enable signals by setting the AIO_EVENT bit in the *aio_flag* member to 1. In this case, no signal is sent, but the error status and return status for the operation

are set appropriately and can be retreived using the aio_error and aio_return functions.

Note that the sigevent structure is used for both asynchronous I/O and per-process timers.

Instead of specifying a signal, you can poll for I/O completion when you expect the I/O operation to be complete.

5.1.4 Establishing a Handle

All I/O has a return value and an error status associated with each operation. Synchronous I/O return values are returned only when the I/O completes and the error status is posted to errno. Asynchronous I/O returns a value when the operation is successfully queued for servicing, then the asynchronous I/O function itself has an associated return value and error status.

Return values for the I/O request are posted to errno, but the return values for completion are posted to the aio_error and aio_return functions. To retrieve these statuses, asynchronous I/O functions have a handle associated with each I/O request. The application can use the handle to poll for completion during and after the I/O operation. Use the aio_error and aio_return functions to retreive status and return values for I/O completion.

The *aio_handle* member of the aiocb structure is set when the I/O operation is first submitted. The handle is a unique identifier for the submitted I/O operation and is used for retrieving the error and return status of the asynchronous I/O operation.

The *aio_handle* member of the aiocb structure is used to track whether the I/O operation was queued and the status of the operation. The AIO_EVENT bit in the *aio_flag* member indicates whether or not a signal will be sent to the calling process when the operation completes. If the AIO_EVENT bit is zero, no signal is posted when the I/O operation completes, but error and return status are still set.

5.2 Asynchronous I/O Functions

The asynchronous I/O functions combine a number of tasks normally performed by the user during synchronous I/O operations. Normally, the application calls the lseek function, performs the I/O operation, and then either polls for status or calls the signal function to notify the calling process.

Asynchronous I/O functions provide the following capabilities:

- Both regular and special files can handle I/O requests.
- One file descriptor can handle multiple read and write operations.

- I/O operations can be prioritized.
- Multiple read and write operations can be issued to multiple open file descriptors.
- Both sequential and random access devices can handle I/O requests.
- Queued I/O requests can be canceled.
- The porocess can be suspended to wait for I/O completion.
- I/O requests can be tracked: when the request is queued, in progress, and completed.

Table 5–1 lists the functions for performing and managing asynchronous I/O operations. Refer to the online reference pages for a complete description of these functions.

Table 5–1 Asynchronous I/O Functions

Function	Description
`aio_cancel`	Cancels one or more requests pending against the file descriptor
`aio_error`	Returns the error status of a specified operation
`aio_read`	Queues a read request on the specified file descriptor
`aio_return`	Returns the status of an operation
`aio_suspend`	Suspends the calling process until at least one of the specified requests has completed
`aio_write`	Queues a write request to the specified file descriptor
`lio_listio`	Initiates a list of requests

5.2.1 Reading and Writing

Asynchronous and synchronous I/O operations are logically parallel operations. The asynchronous function calls `aio_read` and `aio_write` perform the same I/O operations as the `read` and `write` functions. However, the `aio_read` or `aio_write` function returns control to the calling process once the the I/O is queued rather than waiting for the I/O operation to complete. For example, when reading data from a file synchronously, the application regains control only after all the data is read. Execution of the calling process is delayed until the `read` operation is complete. Note that interactions between asynchronous reads and writes with synchronous reads and writes can yield unpredictable results.

On the other hand, when reading data from a file asynchronously, the calling process regains control right after the call is issued, before the read-and-return cycle is complete. The aio_read function returns once the read request is initiated or queued for delivery, even if delivery is delayed. The calling process can use the time normally required to transfer data to execute some other task.

A typical application using asynchronous I/O includes the following steps:

1. Create and fill the asynchronous I/O control block (aiocb).
2. Call the open function to open a specified file and get a file descriptor for that file. After a call to the open function, the file pointer is set to the beginning of the file. Select flags as appropriate. [1]
3. If you use signals, establish a signal handler to catch the signal returned on completion of the asynchronous I/O operation.
4. Call the aio_read or aio_write function to request asynchronous I/O operations.
5. Call aio_suspend if your application needs to wait for the I/O operations to complete.
6. Call the aio_error and aio_return functions to retrieve status information.
7. Call the close function to close the file. The close function waits for all asynchronous I/O to complete before closing the file.

On a call to either the _exit or fork function, the status of outstanding asynchronous I/O operations is undefined. If you plan to use asynchronous I/O operations in a child process, call the exec function before you call the I/O functions.

5.2.2 Using List-Directed Input/Output

The lio_listio function takes as an argument an array of pointers to I/O control block structures, which allows the calling process to initiate a list of I/O requests. Therefore, you can submit multiple operations as a single function call.

You can control whether the lio_listio function returns once the list of operations has been queued or after the operations have been completed. The *mode* argument controls when the lio_listio function returns and can have any one of the following three values:

- LIO_ASYNC, queues the operation, returns, and signals when the operation is complete.

[1] Do not use the select system call with asynchronous I/O, the results are undefined.

- LIO_NOWAIT, queues the operation, returns, but does not signal when the operation is complete.
- LIO_WAIT, queues the operation, suspends the calling process until the operation is complete, and does not signal when the operation is complete.

The *list* argument is a pointer to a list-directed I/O control block (liocb) structure. The liocb structure is defined in the <aio.h> header file and contains the following members:

```
int            lio_opcode;
int            lio_fildes;
struct aiocb   lio_aiocb;
```

The *lio_opcode* member defines the I/O operation to be performed, the *lio_fildes* member identifies the file descriptor, and the aiocb structure is contained in an lio_aiocb structure. The combination of these members makes it possible to specify individual read and write operations as if they had been submitted individually. The values contained in the lio_aiocb structure are the same as those found in aiocb structures used for aio_read and aio_write function calls. Each read or write operation in list-directed asynchronous I/O has its own status, return value, signal, and associated handle.

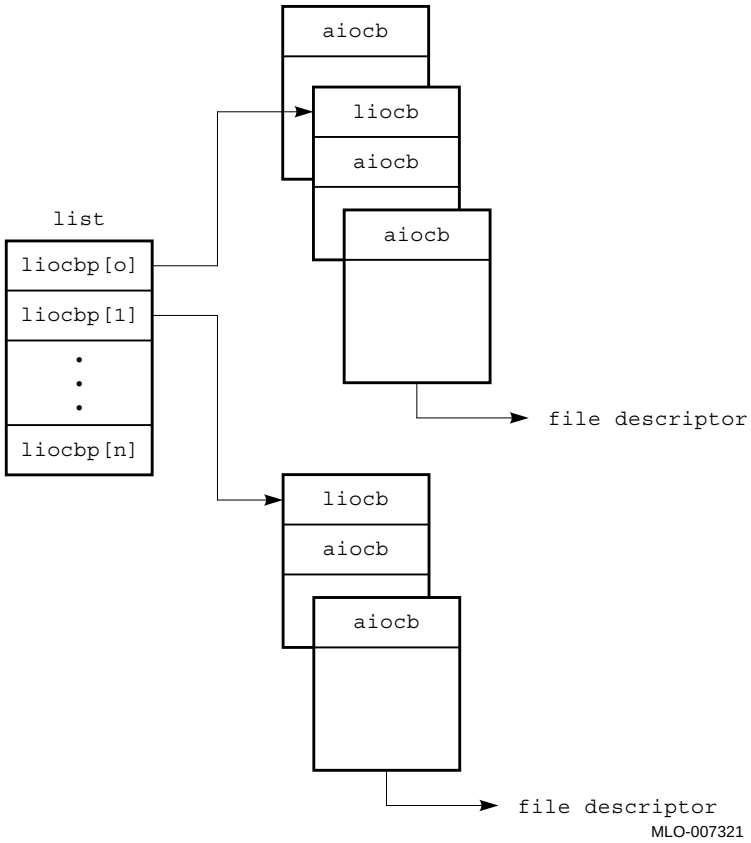
Figure 5–1 illustrates how the asynchronous I/O data structures relate to each other.

You can prioritize execution of the elements in list-directed asynchronous I/O. Refer to Section 5.1.2 for more information on prioritizing asynchronous I/O operations initiated by a single process.

The *lio_opcode* member specifies the operation to be performed. The three operations (LIO_READ, LIO_WRITE, and LIO_NOP) work very much like the aio_read, aio_write, and aio_cancel functions. The LIO_READ and LIO_WRITE operations submit a read and write request as if it had been submitted with a call to an aio_read or aio_write function with the *fildes* argument equal to the *lio_fildes* argument. The LIO_NOP operation causes the aiocb structure to be ignored.

To submit list-directed asynchronous read or write operations, use the lio_listio function. As with other asynchronous I/O functions, you must first establish the aiocb control block structures for the individual read and write operations. The information contained in this structure is used during the operations and is then updated when the operation completes. To use list-directed asynchronous I/O in your application, you may use the following steps:

Figure 5–1 Representation of Asynchronous I/O Data Structures



1. Create and fill the list-directed control blocks, `lio_aiocb` and `liocb`.
2. Call the `open` function to open the specified files and get file descriptors for the files. After a call to the `open` function, the file pointer is set to the beginning of the file. Select flags as appropriate.
3. If you use signals, establish signal handlers to catch the signals returned on completion of individual operations and upon completion of the `lio_listio` function.
4. Call the `lio_listio` function. If your application uses signals, specify LIO_ASYNC on the `lio_listio` function.
5. Call the `close` function to close the files. The `close` function waits for all I/O to complete before closing the file.

As with other asynchronous I/O operations, any open function that returns a file descriptor is appropriate. On a call to either the _exit or fork function, the status of outstanding asynchronous I/O operations is undefined.

5.2.3 Determining Status

Asynchronous I/O requires status values when the operation is successfully queued for servicing. In addition, asynchronous I/O requires return and status values when the operation is complete. The status requirements for asynchronous I/O are more complex than the functionality provided by the errno function, so status retrieval for asynchronous I/O is accomplished through the aio_error and aio_return functions.

Each I/O operation has a handle associated with it to allow the calling process to determine the status of each queued operation. The handle is used when asynchronous I/O status functions are called to return the error status or the return status associated with a specific asynchronous I/O request. The calling process can use the handle to poll for completion of a specific operation. For example, you cannot reuse the aiocb structure while an asynchronous I/O request is pending. However, you can use the aio_return and aio_error functions to determine if the aiocb is in use.

The aio_error and aio_return functions take only the *handle* argument. These functions use the handle to retrieve the error status or return values for the operation identified by the *handle* argument. If the operation completes successfully, the aio_error function returns a 0 and the aio_return function returns the number of bytes that were transferred.

If the operation has not yet completed when the call is made to the aio_error and aio_return functions, the value returned is EINPROG, to indicate that the operation is in progress.

If you use list-directed asynchronous I/O, each asynchronous I/O operation in the list has a unique handle.

5.2.4 Canceling and Suspending I/O

Sometimes there is a need to suspend or cancel an asynchronous I/O operation once it has been issued. For example, there may be outstanding requests when a process exits, particularly if the application uses slow devices, such as terminals.

The aio_cancel function cancels one or more outstanding I/O requests against a specified file descriptor. The *aiocbp* argument points to an aiocb control block for a specified file descriptor. If the operation is successfully canceled, the error status indicates success, but there is no event notification to the calling

process. If, for some reason, the operation cannot be canceled, then normal completion and notification takes place.

The aio_cancel function can return any one of the following values:

- AIO_ALLDONE indicates that none of the requested operations could be canceled because they had already completed when the call to the aio_cancel function was made.
- AIO_CANCELED indicates that all requested operations were canceled.
- AIO_NOTCANCELED indicates that some of the requested operations could not be canceled because they were in progress when the call to the aio_cancel function was made.

In the event that the asychronous I/O operation is terminated with an error or the operation is canceled by a call to the aio_cancel function, the pointer is advanced.

If the value of AIO_NOTCANCELED is returned, call the aio_error function and check the status of the individual operations to determine which ones were canceled and which ones could not be canceled.

The aio_suspend function lets you suspend the calling process until at least one of the asynchronous I/O operations referenced by the *aiocbp* argument has completed. You can also use the aio_suspend function to suspend the calling process until a signal interrupts the function. If the operation has completed when the call to the aio_suspend function was made, the function returns without suspending the calling process. Your application must already have initiated an I/O request with a call to a aio_read, aio_write, or lio_listio function prior to an attempt to suspend the asynchronous I/O request with a call to the aio_suspend function.

5.3 Asynchronous I/O Example

Example 5–1 establishes reads and writes data from buffers. First, the example defines input and output files, one with readonly permission and one with writeonly permission. Then, the program opens the buffers for reading and writing and initializes the aiocb structure for asynchronous write operations. Asynchronous I/O is used to write to the buffer, but a regular read operation reads from the buffer. The aio_error and aio_return functions check the status of the write operations.

Example 5–1 Using Asynchronous I/O

```
#include <sys/types.h>
#include <sys/file.h>
#include <errno.h>
#include <unistd.h>
#include <malloc.h>
#include <aio.h>

#define BUF_CNT 2      /* Number of buffers */

main(int argc, char **argv)
{
     int f1,f2,rec_cnt=0;
     typedef char *buf_p;
     buf_p buf[BUF_CNT];
     struct aiocb a_write;
     size_t xfer_size;
     int buf_index,total=0;

      /* Check the arguments */

     if( argc < 3 )
     {
          printf("\nusage: input-file output-file buffer-size(kb)");
          exit(0);
     }
     if( (f1=open(argv[1],O_RDONLY)) == -1)
     {
          perror(argv[1]);
          exit(errno);
     }
     if( (f2=open(argv[2],O_WRONLY|O_CREAT,0777)) == -1 )
     {
          perror(argv[2]);
          exit(errno);
     }

      /* Convert buffer size to kB */

     xfer_size = atol(argv[3])*1024;

      /* Allocate the buffers */

     for( buf_index=0; buf_index < BUF_CNT; buf_index++ )
          buf[buf_index] = (buf_p)malloc(xfer_size);
     buf_index=0;

      /* Initialize AIOCB for write */

     a_write.aio_offset=0;          /* Always write from */
     a_write.aio_whence=SEEK_CUR;   /* Current position  */
```

(continued on next page)

Example 5–1 (Cont.) Using Asynchronous I/O

```
     /* Copy the file */

    while( f1 != -1 )
    {
         int buf_len;

     /* Read the next buffer */

    buf_len = read( f1, buf[buf_index], xfer_size);
    if( rec_cnt )          /* For all but the first write */
         {

     /* Wait for completion of the previous write */

              if( aio_error(a_write.aio_handle)== EINPROG )
              {
                   struct aiocb *wait_list;
                   wait_list= &a_write;
                   aio_suspend( 1, &wait_list);
              }

     /* Update the total */

          total += aio_return(a_write.aio_handle);
          }

     /* Check for end of file */

    if( buf_len <= 0 )
        break;

     /* Set the buffer information and issue the write */

    a_write.aio_nbytes = buf_len;
    a_write.aio_buf = buf[buf_index];
    aio_write( f2, &a_write);

     /* Update record count and position to the next buffer */

    rec_cnt++;
    buf_index ^= 1;
    }

     /* Close the files */

    close(f1);
    close(f2);
    printf("Copied: %d records\n",rec_cnt);
}
```

6

Interprocess Communication Overview

Interprocess communication (IPC) is the exchange of information between two or more processes. In single-process programming, modules within a single process communicate using global variables and function calls, with data passing between the functions and the callers. When programming using separate processes, with images in separate address spaces, you need to use additional communication mechanisms. A number of facilities are provided for interprocess communication, including messages, shared memory, semaphores, pipes, and signals.

This chapter includes the following sections:

- IPC and Process Synchronization, Section 6.1
- System V IPC Overview, Section 6.2
- System V IPC Permission Structure, Section 6.3
- The ftok Function, Section 6.4

6.1 IPC and Process Synchronization

Interprocess communication is a way of transfering data between cooperating processes within an application. Processes can pursue their own tasks until they must synchronize with other processes at some predetermined point in their execution. When they reach that point, they wait for some form of interprocess communication. IPC can take any of the following forms:

- Messages
- Shared Memory
- Semaphores
- Pipes
- Named Pipes
- Signals

Some forms of IPC are traditionally supplied by the operating system and some are specifically modified for use in realtime functions. All allow a user- or kernel-level process to communicate with a user-level process. IPC facilities are used to notify processes that an event has occurred or to trigger a process's response to an application-defined occurrence, such as asynchronous I/O completion, timer expiration, data arrival, or some other user-defined event.

Messages, shared memory, and semaphores are collectively known as System V IPC. They are high-speed, reliable data transfer facilities commonly used to synchronize process execution.

Use of synchronization techniques and restricting access to resources can ensure that critical and noncritical tasks execute at appropriate times with the necessary resources available. Concurrently executing processes require special mechanisms to coordinate their interactions with other processes and their access to shared resources. In addition, processes may need to execute at specified intervals.

Realtime applications synchronize process execution through the following techniques:

- Waiting for a specified time
- Waiting for semaphores
- Waiting for communication
- Waiting for other processes

The basic mechanism of process synchronization is waiting. A process must synchronize its actions with the arrival of an absolute or relative time, or until a set of conditions is satisfied. Waiting is necessary when one process requires another process to complete a certain action, such as releasing a shared system resource, or allowing a specified amount of time to elapse, before processing can continue.

The point at which the continued execution of a process depends on the state of certain conditions is called the "synchronization point." Synchronization points represent intersections in the execution paths of otherwise independent processes, where the actions of one process depend on the actions of another process. The application designer identifies synchronization points between processes and selects the functions best suited to implement the required synchronization.

In a realtime environment it is often necessary to reduce the time interval required for process communication. It is not always sufficient to simply verify that communication has taken place. A delay in interprocess communication can affect the overall performance of the realtime application. To provide rapid signal communication on timer expiration and asynchronous I/O completion, these functions send signals via a `sigevent` structure rather than through the traditional signal mechanism. The application designer identifies resources such as message queues and shared memory that the application will use. Failure to control access to critical resources can result in performance bottlenecks or inconsistent data.

6.2 System V IPC Overview

The System V IPC facilities were designed to work together, share common properties, and overcome many of the limitations of other forms of interprocess communication.

The three IPC facilities work together but offer distinctly different forms of interprocess communication. The following list summarizes the characteristics of each IPC facility:

- Message operations allow the process to send, receive, and control a message queue.

  Message queues work by exchanging data in buffers, which means any number of other processes, whether or not they are related, can communicate. If a process has all the access rights, it can send or receive messages through the queue. The receiving process can select incoming messages of a specified type.

- Shared memory operations allow the process to attach, detach, and lock shared memory segments into physical memory.

  Shared memory allows processes to share parts of their virtual address space, which means that processes communicate quickly, without having to copy information. Processes having the right ID and permission can access the same memory. Shared memory can also be locked into physical memory, which provides optimal communication conditions for realtime processes.

- Semaphore operations allow the process to get, release, increment, and decrement both binary and counting semaphores.

  Semaphores offer processes a way to synchronize their operations, most commonly to synchronize access to shared memory.

Table 6–1 lists the functions used to create, open, and manipulate IPC facilities in an application.

Table 6–1 IPC Functions

Messages	Shared Memory	Semaphores	Description
<sys/msg.h>	<sys/shm.h>	<sys/sem.h>	Header files
msgget	shmget	semget	Open or create IPC channel
msgctl	shmctl	semctl	IPC control operations
msgsnd	shmat	semop	Specific IPC operations
msgrcv	shmdt		Specific IPC operations

System V IPC facilities are systemwide to allow different processes to communicate, whether or not they are contained within the same application. For this reason, access to IPC channels is controlled in much the same way as access to files, through IPC identifiers. Every process that shares an appropriate identifier (IPC ID) and has the right permission can use the facilities provided for messages, shared memory, and semaphores.

The general approach to using any of the System V IPC facilties is as follows:

1. Begin code execution at a nonrealtime priority level.
2. Call the ftok function to get a unique *key* identifier, based on the *path* and *id* argiments passed to the ftok function. Use the *key* in subsequent calls to IPC get functions.
3. Include code to remove the IPC facilities and associated data structures if the process is aborted.
4. Call the IPC get function to create an IPC channel and associate it with the data structures. (If an IPC object already exists, it is opened.) The get function returns a unique identifier. Use this identifier in subsequent calls to IPC functions.

   The IPC get functions initialize IPC data structures with access and permission information.
5. Optionally, if using shared memory, allocate the semaphore that will be used to control access to the shared memory region.

6. If necessary, use the ctl functions to control access or to unlink the IPC channels.
7. For shared memory, lock memory regions as needed.
8. Set realtime priorities and scheduling policies as needed.
9. Execute realtime work.
10. Remove the IPC channel when the process exits or it is no longer needed. If you do not remove the IPC channel with the ctl function, use the ipcrm function.

You can use the ipcs command to retrieve information about the status of IPC facilities. The ipcs command displays information on currently active message queues, shared memory, and semaphores. Since the kernel keeps track of IPC facilities dynamically, the information returned from the ipcs command changes as IPC facilities are used.

Options on the ipcs command give you access to all IPC information contained in the IPC data structures. You can request all information or you can specify selective information, such as the process ID (PID) of the last process to send a message, the maximum number of bytes allowed for messages in the message queue, the number of processes attached to a shared memory segment, or the time the last semaphore operation was completed on the set.

See the online reference pages for syntax information about the ipcs command and each of the System V IPC functions. Note that the msgsnd and msgrcv functions are included in the reference pages under the more general name, msgop. The shmat and shmdt functions are included under the more general name, shmop.

6.3 System V IPC Permission Structure

Each IPC facility uses a basic IPC structure (ipc_perm) to define access permissions. The owner process issues the call to the get function to allocate the IPC entities associated with the data structure. All other processes that use the IPC entity must access it by using the IPC ID, so that no additional structures are initialized. The ipc_perm structure is similar to that maintained for files; it contains members for the effective user ID (UID), effective group ID (GID), access groups, and a key to manage the creation of IPC channels.

Messages, shared memory, and semaphores all use an ipc_perm data structure. The ipc_perm structure is defined in the <sys/ipc.h> header file and has the following form:

```
struct ipc_perm {
      ushort uid;          /* Owner's ID              */
      ushort gid;          /* Owner's group ID        */
      ushort cuid;         /* Creator's UID           */
      ushort cgid;         /* Creator's group ID      */
      ushort mode;         /* Access mode             */
      ushort seq;          /* Sequence number         */
      key_t key;           /* Key                     */
};
```

The ipc_perm structure, used in combination with individual IPC permission structures, provides IPC facilities with the following common properties:

- They all maintain a table describing the instances of the IPC facility usage.
- Each entry into the table is identified by a user-specified key.
- They all use a call to a get function to create new table entries or open existing ones.
- They all use a permission structure based on IDs.
- The parameters to IPC calls require a key and flags.
- They all use the same algorithm to find the index into the table from the descriptor.
- They all maintain status information including the time, date, and process ID (PID) of the last access.

6.3.1 Creating IPC Channels

Before you can use an IPC channel, you must create the channel or open an existing one. The get functions for each of the IPC facilities perform this task. The three IPC get functions (msgget, shmget, and semget) require at least the *key* and *flag* parameters. Both shared memory and semaphore get function calls use additional parameters to establish the size of the shared memory or the number of semaphores.

The value of the *key* argument on the get function call determines how the IPC channel is established. If the *key* is not already in use, the function returns a new IPC ID. If the *key* is in use, the existing IPC ID is returned. If the *key* is specified as 0 (to indicate IPC_PRIVATE), a unique, exclusive IPC ID is returned. Specifying IPC_PRIVATE for the *key* parameter is usually done when you plan to share an IPC ID among related processes.

The *key* and the IPC ID are directly related — processes sharing the same *key* will also share the same IPC ID. You can use the ftok function to create a *key*. This method allows processes to share an IPC channel in a client and server relationship. See Section 6.4 for more information on using the ftok function.

The *flag* parameter accepts combinations of flags and a permissions number. The combinations of these elements determine whether a calling process has permission and the nature of that permission.

The kernel checks the *mode* bits to determine if the caller has permission for the requested operation. If IPC_PRIVATE is specified, only the owner and related processes can access the IPC channel. If the calling process is not the owner and is not in the group, then the *mode* bits must be set for world access before permission is granted. In addition, the appropriate access bits must be set before an operation is performed. That is, to perform a read operation the read bit must be set.

Table 6–2 shows the flags used by the IPC `get` functions.

Table 6–2 Flags Used in IPC get Functions

Flag	Description
IPC_CREAT	Creates a new entry for the specified *key* or returns the entry if it already exists.
IPC_EXCL	Creates a new entry for the specified *key* if one does not exist. The call fails if the specified *key* already exists. This flag must be used with the IPC_CREAT flag.
IPC_NOWAIT	Returns an error if the request must wait.

Whenever you create a new IPC channel with one of the `get` functions, the `ipc_perm` data structure is initialized. Fields are initialized for owner and creator IDs (UID and CUID), user and group IDs (UID and GID), and mode. The IPC_CREAT flag returns a unique IPC ID or returns the existing entry, if one already exists. The IPC_EXCL flag must be used in conjunction with the IPC_CREAT flag. This combination guarantees that a new channel is created, but does not guarantee exclusive access to the IPC channel. When you use the IPC_PRIVATE key, it is not necessary to use either the IPC_CREAT or IPC_EXCL flags.

For example, if you want to create a shared memory segment of 1024 bytes and make certain that only the owner has read and write permission, use the following function call:

```
shm_id = shmget (IPC_PRIVATE, 1024, 0600);
```

The unique IPC ID created by the call to the `shmget` function is stored in the variable *shm_id*.

Child processes inherit the IPC ID if the call to `fork` is done after the call to IPC `get`.

When processes are unrelated, you must call an IPC get function from each process with which you want to communicate. In addition, each process must use the same key parameter. Since the IPC_PRIVATE key returns a unique IPC ID, you may not want to use it in this situation. Instead, you may find it easier to use the IPC_CREAT and, possibly, IPC_EXCL flags.

If the process has superuser privileges, it is always allowed access. Any process can specify a *flag* argument of zero to bypass permission problems with the *mode* bits, as long as an access channel exists.

The IPC_CREAT flag either creates a new IPC ID or returns the existing IPC ID. If you want the call to fail if an IPC ID already exists, use the IPC_EXCL flag. See the intro reference page and reference pages for the individual IPC functions for more information.

6.3.2 Controlling IPC Channels

The IPC ctl functions control access to existing IPC channels and unlink the IPC channels. The flags used in the ctl functions can be used to alter permissions and remove IPC channels along with all of the associated data structures.

The parameters for IPC ctl functions include the *cmd* parameter. The *cmd* parameter specifies a control operation that can be performed on the data contained in a data structure that describes the facility created with a call to one of the IPC get functions. Calls to IPC ctl functions can specify changes to the owner IDs (UID).

Table 6–3 shows the flags used in all IPC ctl functions to retrieve and manipulate data structure members.

Table 6–3 Flags Used in IPC ctl Functions

Flag	Description
IPC_RMID	Removes an existing IPC channel and deletes the associated data structure
IPC_SET	Copies a user-specified structure to the IPC data structure
IPC_STAT	Copies the IPC data structure to a user-specified structure

The kernel maintains data structures for the IPC facilities but the user can modify some members of these data structures. The owner of an IPC facility can change the user ID (UID), group ID (GID), and the mode. Other members can be modified only by a process with superuser privileges or by the owner.

To modify a member of a data structure, use one of the IPC ctl functions with the IPC_STAT flag to get the current values. Because IPC data structures are kept in the kernel, you need to specify a local buffer as the location to where the structure is copied. Access the data structure in this buffer and set the members you want to change. Then use the IPC ctl functions with the IPC_SET flag to change the original data structure. The IPC_SET flag overwrites the original data structure with the new information.

Note that for all IPC facilities, you must either be the owner or have superuser privileges to use the IPC_SET or IPC_RMID flag.

6.3.3 Removing IPC Channels

The last step in interprocess communication is to remove an IPC facility. IPC facilities are not automatically removed once an application exits; they continue to exist unless explicitly removed by the application. The fact that an IPC channel remains accessible can be helpful when processes exit at different times, but when the entire application is done, you should call one of the ctl functions and pass the IPC_RMID flag to it. For example, to remove a shared memory segment, make the following function call:

```
shmctl (shm_id, IPC_RMID, 0);
```

This call removes both the IPC ID and the associated data structure.

If you do not remove the IPC facilities while your application executes, you can remove the IPC ID with the ipcrm command. You can choose to remove the identifier as returned by a call to one of the IPC get functions, or you can choose to remove the *key*. The ipcrm command removes the identifier and deletes the data structure and queue associated with the IPC facility.

6.4 The ftok Function

All IPC facilities require the user to supply a *key* to be used by the msgget, shmget, and semget functions. One method for forming a *key* is to use the ftok function. This function returns a *key* based on two parameters, *path* and *id*. The returned *key* is subsequently used by the IPC get functions.

Client and server processes must first agree on a single *path* to be used as an interprocess communication channel between them. This *path* could be the name of a common data file or the *path* of the server daemon, as long as the pathname is for an existing file and is accessible to the process. You must have read permission on the file and execute permission on the directories of the entire pathname before you can use the ftok function. The application then calls the ftok function to convert the *path* into an interprocess communication *key*.

The *id* argument is a character that uniquely identifies the project. The following example returns a *key* that can be used in other IPC functions.

```
keyofmine = ftok ("/usr/users/example1", 'X');
```

If unrelated processes create the IPC facility or if multiple IPC channels are required, then these processes should use the same *path* and *id* arguments to the ftok function. You can also use the ftok function to help your application overcome synchronization problems. If you are not certain which process will create the IPC ID, or if you think the creating process may execute earlier than the other communicating processes, call the ftok function. The following example calls the semget function to create a set of three semaphores that can be read only by the owner:

```
sem_id = semget (ftok ("/usr/users/example2", 'X'), 3, IPC_CREAT|0600);
```

Using the following function call, other processes in the group could communicate using the same ID.

```
sem_id = semget (ftok ("/usr/users/example2", 'X'), 3, 0660);
```

On the other hand, if the other processes will be using the IPC channel and the first process fails to create the IPC ID, then subsequent calls to that channel will fail. If the IPC ID does not exist at the time of the call and the IPC_CREAT flag is not specified, then there is no corresponding IPC ID. The following function call overcomes this potential problem by using the IPC_EXCL flag in addition to the IPC_CREAT flag:

```
sem_id = semget (ftok ("/usr/users/example2", 'X'), 3, IPC_CREAT|IPC_EXCL|0600);
```

Using these flags together ensures that an IPC key is created in the event that none previously existed.

7
Messages

Message queues are user-defined data structures that specify the length and type of message and carry the message text. Essentially, message queues are linked lists that are accessed by sending and receiving processes, allowing flexibility and control over interprocess communication. Message queues use data structures to store multiple messages that can be accessed by multiple processes, read in any order, prioritized according to application needs, and periodically polled for specific content.

Applications that contain multiple processes sharing message queues use named spaces as a way to connect communication between cooperating processes.

This chapter includes the following sections:

- Data Structures Associated with Messages, Section 7.1
- The Message Interface, Section 7.2
- Message Queue Example, Section 7.3

7.1 Data Structures Associated with Messages

The `ipc_perm` structure is the basic permission structure for all System V IPC. Messages use the `ipc_perm` structure as well as other structures tailored to message queues. Message structures are defined in the `<sys/msg.h>` header file.

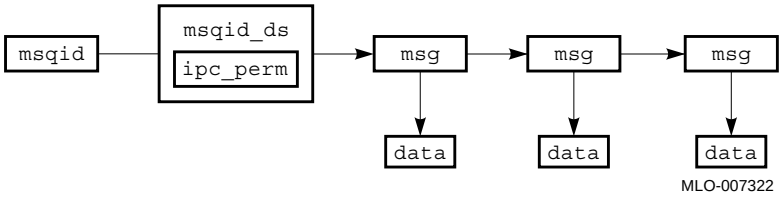
A call to the `msgget` function creates the message queue identifier, *msqid*. Each message queue identifier has an associated message queue and a data structure. This data structure is called `msqid_ds` and takes the following form:

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

struct msqid_ds {
     struct ipc_perm msg_perm; /* Operation permission structure          */
     struct msg  *msg_first;   /* Pointer to first message on queue       */
     struct msg  *msg_last;    /* Pointer to last message on queue        */
     ushort msg_cbytes;        /* Current number of bytes on queue        */
     ushort msg_qnum;          /* Number of messages on queue             */
     ushort msg_qbytes;        /* Maximum number of bytes on queue        */
     ushort msg_lspid;         /* Pid of last msgsnd operation            */
     ushort msg_lrpid;         /* Pid of last msgrcv operation            */
     time_t msg_stime;         /* Last msgsnd time - seconds since Epoch  */
     time_t msg_rtime;         /* Last msgrcv time - seconds since Epoch  */
     time_t msg_ctime;         /* Last change time - seconds since Epoch  */
};
```

The msqid_ds data structure holds information such as the number of messages on the queue, the number of bytes on the queue, the PID of the process that sent or received the last message, and timestamps for activities.

The msg_perm structure is contained in an ipc_perm structure. Figure 7–1 shows how message structures relate to each other.

Figure 7–1 Representation of Message Data Structures



The <msg.h> header file contains information concerning message size, the system-wide maximum number of queued messages, and other limits pertaining to message queues.

7.1.1 Establishing Message Permissions

Processes can use message queues to read or write messages as long as the processes have permission. The IPC message facility uses a msg_perm structure to determine permission. The msg_perm structure is an ipc_perm structure, but uses only information specific to messages. The msg_perm structure contains the following members:

```
ushort cuid;            /* Creator user ID                    */
ushort cgid;            /* Creator group ID                   */
ushort uid;             /* Owner's user ID                    */
ushort gid;             /* Owner's group ID                   */
ushort mode             /* Read/write (or alter) permission   */
u_short seq;            /* Slot usage sequence number         */
key_t key;              /* Key                                */
```

In the msgop and msgctl functions check permission needed to use message queues. Permission is interpreted as follows:

```
00400     Read by user
00200     Write (or alter) by user
00060     Read, Write (or alter) by group
00006     Read, Write (or alter) by others
```

Read and write (or alter) permissions are granted to a process if the ID for the calling process matches one or more combinations of permissions or if the effective user ID (UID) of the process is superuser. Access permissions are similar to those used for files. If you do not specify access for IDs other than the owner process, only the owner and superusers will be able to access the structure.

7.1.2 Establishing Message Structure

A call to msgsnd or msgrcv sends or receives a message from the associated queue. The *msgp* parameter for these functions points to a structure containing the message. The kernel does not interpret the content of messages. You can customize messages by defining your own structure. This structure takes the following form:

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

struct msgbuf {
     long mtype;         /* Message type    */
     char mtext [];      /* Message data    */
     };
```

The *mtype* member for a send operation can only be zero or a positive integer. The *mtype* member for a receive operation can only be zero or negative. This member can be used by the receiving process for message selection. The *mtext* member is any form of data (text or binary).

7.2 The Message Interface

The message interface is a set of structures and data that allows you to send messages to a message queue. The message queue is a linked list that serves as a holding place for messages being sent to and received by processes sharing the message queue. You can specify a message type or prioritize messages based on the message type.

This section discusses the functions used to create, control, and remove the message queue and messages in the queue. Table 7–1 lists the functions that allow you to create and control messages.

Table 7–1 Message Functions

Function	Description
`msgget`	Creates or returns a message queue identifier for use in other message functions
`msgctl`	Provides control for message operations and has options to return and set message descriptor parameters and remove the descriptor
`msgsnd`	Sends a message to the queue associated with the message queue identifier
`msgrcv`	Reads a message from the queue associated with the message queue identifier and places it in a user-defined structure

7.2.1 Creating and Opening a Message Queue

To set up a message queue, first create a new message queue or open an existing queue using the `msgget` function. To determine which course of action is taken, the kernel searches the array of message queues to determine if a message queue indentifier already exists with the specified *key*. If there is no entry, the kernel allocates a new message queue structure, initializes it, and returns the identifier. If one already exists, then the `msgget` function checks permissions.

If your application consists of related processes, you may want to use the IPC_PRIVATE key on the function as follows:

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

msq_id = msgget (IPC_PRIVATE, 0660);
```

This call creates a new message queue with read and write permission for the owner and group. If you call the msgget function before calling fork to create child processes, the IPC channel and permissions are inherited by the child process, thus enabling all related processes to communicate with one another. Read and write permissions for messages allow the process to receive (read) messages and send (write) messages.

If your application consists of unrelated processes, you may need to use the ftok function to establish a key based on other criteria. Other unrelated processes that will communicate through the message queue will have to call the msgget function to get the message queue identifier.

```
msq_id = msgget (ftok ("/usr/users/example3", 'X'), IPC_CREAT|0600);
```

This call creates a new message queue, but the ftok call returns a key based on the */usr/users/example3* pathname and the character *X*. As other processes call the msgget function, *msq_id* is returned for them to use in other message calls.

7.2.2 Sending and Receiving Messages

Once a message queue is open, you can send messages to another process with the msgsnd function. The msgsnd function takes four parameters, including: the message queue identifier, a pointer to a message structure, the size of the data, and action to take if the kernel runs out of buffer space. The kernel checks operation permissions, length of the message, the status of the message queue, and the message flag. If all kernel checks are successful, the message is added to the list of message headers on the message queue.

The message flag parameter is either 0 or IPC_NOWAIT. If you specify *msgflg* of 0, then the sending process sleeps if the message cannot be sent to the specified queue. If the queue is full, the msgsnd function will sleep until other messages have been removed from the queue and space is available. If the process is specified as IPC_NOWAIT, the msgsnd function returns immediately with an error status.

The following example attempts to write a message, but if the message queue is full it returns an error status without blocking the process.

```
msgsnd (msq_id, mymessage, messagesize, IPC_NOWAIT);
```

Once a message has been placed on a queue, you can retrieve the message with a call to the `msgrcv` function. The `msgrcv` function takes five parameters, including: the message queue identifier, a pointer to a message structure, the size of the data, the type of message the user wants to read, and action to take if the kernel runs out of buffer space.

As with the `msgsnd` function, the kernel checks operation permissions. If the requested message type is 0, the first message on the linked list is read. Then the kernel checks for processes waiting to send messages and queues them as space becomes available. If the message size is greater than that allowed for a single message segment, the kernel truncates the message.

Specify the message flag parameter as either 0 or IPC_NOWAIT. If you specify a *msgflg* of 0, then the receiving process sleeps if there is no message of the specified type on the queue. If you specify the IPC_NOWAIT flag, the process returns immediately with an error status.

A process can control the type of messages it receives by setting the *msgtyp* parameter. This parameter allows the receiving process to prioritize messages on a specified queue or to conserve queue identifiers. The *msgtyp* parameter specifies the type of requested message as follows:

- If *msgtyp* is equal to 0, the first message in the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

If you assign lower message types to messages of higher importance, you can receive the most important messages first. If you assign higher message types to less important messages, you can delay delivery of the messages as more important events are executed.

The following example reads a message without blocking, that is, the process looks to the queue for messages to read and does not sleep:

```
count = msgrcv (msq_id, mymessage, messagesize, pid, IPC_NOWAIT);
```

The return value, *count*, is the number of bytes returned to the user. The *pid* is used as the message type.

Prioritizing messages also lets you multiplex messages or use a single message queue as if it were multiple message queues. If one process is the server for several client processes, the server can receive messages of one type while the clients receive messages of another type.

7.2.3 Controlling and Removing a Message Queue

The `msgctl` function allows you to query or set the status of the message queue identifier, *msqid*. This function also removes a message queue.

The *cmd* argument can take one of three command control flags, which determine what action is taken by the `msgctl` function. Table 7–2 describes the command control flags.

Table 7–2 Message Command Control Flags

Command	Description
IPC_RMID	Removes a message queue identifier and the associated message queue and data structure
IPC_SET	Sets the user and group IDs (UID and GID), operation mode values, and the size of the message queue
IPC_STAT	Returns the status information in the associated data structure for a specified message queue identifier and copies it into a user-specified buffer

These control flags allow you to control messages by performing the following functions:

- Return all message structure member values and status in user memory
- Change operation permissions
- Remove messages, message queues, and their associated data structures

Note that to use the IPC_SET and IPC_RMID flags you must either be the owner or have `superuser` privileges.

The IPC_RMID flag removes the message queue identifier and its associated data structures. When you specify the IPC_RMID control flag, you need only supply the message queue identifier and the IPC_RMID control flag on the function call. You can leave the *buf* argument as NULL. When you have finished using a message queue, you should remove it either in this manner before the application exits or with the `ipcrm` command.

The IPC_RMID flag removes a message queue from the system as follows:

```
msgctl (msq_id, IPC_RMID, 0);
```

The last process in your application to use the message queue should remove it before exiting. If your application uses messages and signals in combination, you can set up a signal handler to remove the message queue.

The IPC_SET control flag allows you to modify the user ID (UID), group ID (GID), or mode values associated with the specified message queue identifier.

The IPC_STAT control flag copies status information into a user-specified buffer where it can be inspected or monitored. If you determine that the status information is no longer valid or you wish to make changes to the status information, use the IPC_SET control flag.

7.3 Message Queue Example

Example 7–1 reads message types and text from a terminal and places the messages into a message queue. Then the ftok function is used to generate a message queue key and the key is kept in a file. The message queue is created using IPC_CREAT.

This example is a partial example, but gives the framework essential for using message queues for interprocess communication. This program needs to be started before the reader process as it creates the shared memory and semaphores used by both processes.

Example 7–1 Using Message Queues

```
#include <errno.h>
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>

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

#define READ_WRITE 0660
#define MTEXT_SIZE 80
#define test_error(a) if((a)== -1){perror("msg");exit(errno);}
#define loop() while(1)

      /* A simple message structure */

struct msg_buf {
     long type;
     char mtext[MTEXT_SIZE];
     };

static jmp_buf sjb;           /* Setjmp environment buffer */
static void end_child(){      /* Signal handler when child dies */
     longjmp(sjb, 1);
     }
```

(continued on next page)

Example 7–1 (Cont.) Using Message Queues

```
main()
{
     int msg_id,                  /* Message queue identifier */
          child;                  /* ID of child process      */
     struct msg_buf m;            /* Message buffer           */
     size_t status;               /* Receive status/length    */

     signal(SIGCHLD,end_child);    /* Called when child dies  */
     m.type=1;                     /* Message type must be >0 */
     msg_id = msgget( IPC_PRIVATE, READ_WRITE); /* Create message queue */
     test_error(msg_id);

      /* fork the child process.  The child sends and the parent */
      /* receives. */

     if( (child=fork()) == 0 ) loop() {

      /* The child process reads from the terminal and generates */
      /* messages until EOF.  On EOF, it exits which raises a    */
      /* signal in the parent. */

     printf("Enter message text: ");
     if( gets(m.mtext) == NULL )
          break;
     status = msgsnd(msg_id, &m, strlen(m.mtext)+1, 0);
     test_error(status);
     sleep(1);
     }

      /* The parent process calls setjmp.  When the signal comes */
      /* in after the child dies, the signal handler function,   */
      /* end_child(), does a longjmp forcing 1 to be returned.   */

     else if( setjmp(sjb) == 0) loop() {

      /* The parent reads the messages and writes them to the */
      /* terminal. */

     status = msgrcv(msg_id, &m, sizeof(m.mtext), 0, 0);
     test_error(status);
     printf("message received: %s\n",m.mtext);
     }

     printf("%s exiting\n",child?"Reciever":"Sender");
 }
```

8

Shared Memory

The fastest method for interprocess communication is shared memory because processes communicate without the overhead of system calls into the kernel. Using shared memory, processes communicate directly by sharing portions of their virtual address space. When one process writes to a location in the shared area, the data is immediately available to other processes sharing the same memory area.

This chapter includes the following sections:

- Data Structures Associated with Shared Memory, Section 8.1
- The Shared Memory Interface, Section 8.2
- Shared Memory Example, Section 8.3

8.1 Data Structures Associated with Shared Memory

The kernel maintains a structure for every shared memory segment. To create shared memory, call the `shmget` function. This function initializes the specified shared memory data structure with user and group IDs (UID and GID), *mode*, and so forth. An existing memory segment is accessed if one already exists.

A shared memory identifier, (*shmid*), is a unique positive integer returned by a call to the `shmget` function. Each shared memory identifier has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. This data structure is referred to as `shmid_ds` and takes the following form:

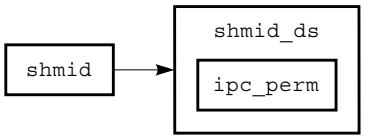
```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
struct shmid_ds {
     struct ipc_perm shm_perm; /* Operation permission structure         */
     int shm_segsz;            /* Size of segment                        */
     ushort shm_cpid;          /* Creator pid                            */
     ushort shm_lpid;          /* Pid of last operation                  */
     short shm_nattach;        /* Number of current attaches             */
     time_t shm_atime;         /* Last shmat  time - seconds since Epoch  */
     time_t shm_dtime;         /* Last shmdt  time - seconds since Epoch  */
     time_t shm_ctime;         /* Last change time - seconds since Epoch  */
     };
```

On a call to the shmget function, all time stamps are set to 0, except for *shm_ctime*, which is set to the current time. As processes access the shared memory segment, these members track the most recent events.

Figure 8–1 illustrates how shared memory structures relate to each other.

Figure 8–1 Representation of Shared Memory Data Structures



8.1.1 Establishing Shared Memory Permissions

Processes access a shared memory segment as long as the calling process has permission. Shared memory uses the shm_perm structure to determine permission. This structure is identical for each type of interprocess communication but the structure name differs. This shm_perm structure contains the following members:

```
     ushort cuid;            /* Creator user ID                   */
     ushort cgid;            /* Creator group ID                  */
     ushort uid;             /* User ID                           */
     ushort gid;             /* Group ID                          */
     ushort mode             /* Read/write (or alter) permission  */
     u_short seq;            /* Slot usage sequence number        */
     key_t key;              /* Key                               */
```

The shmop and shmctl functions check permissions needed to use a shared memory segment. Permission is interpreted as follows:

```
00400     Read by user
00200     Write (or alter) by user
00060     Read, Write (or alter) by group
00006     Read, Write (or alter) by others
```

Read and write (or alter) permissions are granted to a process if the ID for the calling process matches one or more of these permissions or if the effective user ID (UID) of the process is superuser.

8.1.2 Controlling Shared Memory

The shmctl function takes flags to control shared memory. The shmctl function allows you to retrieve status information and to lock and unlock shared memory. Table 8–1 shows the flags used on the shmctl function.

Table 8–1 Shared Memory Command Control Flags

Flags	Description
IPC_RMID	Removes a shared memory ID and the associated data structure
IPC_SET	Sets the user and group IDs and operation mode values
IPC_STAT	Returns the status information in the associated data structure and copies it into a user-specified buffer

With the exception of the IPC_STAT flag, you must either be the owner or have superuser privileges to use these flags.

The control flags allow you to control shared memory by performing the following functions:

- Return all shared memory structure member values and status in user memory
- Change operation permissions
- Remove shared memory and associated data structures
- Lock a shared memory segment
- Unlock a shared memory segment

8.2 The Shared Memory Interface

The shared memory and semaphore functions let you control access to shared memory so that address space use is coordinated. Using semaphores in conjunction with shared memory, you can make one process wait while another process reads from shared memory. In addition, shared memory can be locked and unlocked, which prevents a shared memory segment from being paged out of memory.

Table 8–2 summarizes the functions used to create and control shared memory.

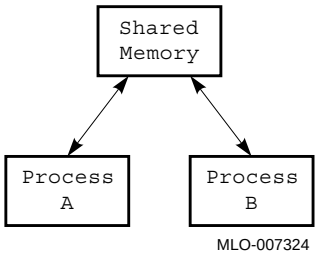
Table 8–2 Shared Memory Functions

Function	Description
`shmget`	Returns the shared memory identifier
`shmctl`	Provides shared memory control operations
`shmat`	Attaches the shared memory segment to the data segment of the calling process
`shmdt`	Detaches the shared memory from the data segment of the calling process

All processes using the shared memory segment must first call the `shmget` function to establish shared memory. The `shmget` function establishes a shared memory segment, but does not provide access. Instead, the calling process provides access by attaching to the shared memory segment through a call to the `shmat` function. All cooperating processes attach to the shared memory segment using the `shmat` function and detach from the segment using the `shmdt` function.

Figure 8–2 illustrates how two processes access the same shared memory segment. To guard against corruption of this segment, access to shared memory is controlled by using semaphores for synchronization, as discussed in Chapter 9.

Figure 8–2 Two Processes Using Shared Memory



8.2.1 Creating and Opening Shared Memory

A multiprocess application typically contains one or more processes. A controlling process can create shared memory regions early in the life of the application and then dynamically delete the shared memory.

A call to the `shmget` function creates a new region of shared memory or returns the identifier of an existing one. This region is identified by the descriptor *shmid*. On a call to the `shmget` function, the kernel searches the shared memory table for the specified *key*. If there is no entry and the user sets the IPC_CREAT or IPC_PRIVATE flag, the kernel allocates new shared memory and initializes the shared memory data structure. The identifier is then returned to the calling process. If the shared memory already exists, the `shmget` function saves permission modes, sets a pointer to the table entry, and sets a flag to indicate that no memory was allocated for the region.

If your application consists of related processes, you can set the IPC_PRIVATE flag as follows:

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

shm_id = shmget (IPC_PRIVATE, 3072, 0600);
```

This call creates a shared memory segment of 3072 bytes with read and write permission for the owner. By calling the `shmget` function before calling `fork` to create child processes, the child process inherits the IPC channel as well as the IPC permissions. This enables all related processes to read and write from the shared memory segment.

If your application consists of unrelated processes, you can use the `ftok` function to establish a *key* based on other criteria. Other unrelated processes that communicate through the shared memory must call the `shmget` function to get the *shmid*, as follows:

```
shm_id = shmget (ftok ("/usr/users/example3", 'X'), 3072, IPC_CREAT|0600);
```

This call creates a new shared memory segment, but the ftok call returns a *key* based on the */usr/users/example3* pathname and the character *X*. As other processes call the shmget function, *shm_id* is returned for them to use in other shared memory calls.

Example 8–1 creates a shared memory region, determines the virtual address, maps the shared region into virtual memory, and writes the shared memory address.

Example 8–1 Creating Shared Memory

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

char *start_addr;
FILE *keyfile;
int shm_key, shm_id;

keyfile = fopen ("/tmp/shared-mem", "w");
shm_key = ftok ("/tmp/shared-mem", 'a');

      /* Create shared region */

shm_id = shmget (shm_key, 4096, IPC_CREAT | 0760 );
if (shm_id == -1){
     perror ("Shared memory region creation");
     exit (1);
     }

      /* System determines the virtual address */

start_addr = 0;

      /* Map the region into virtual memory */

start_addr = (char *) shmat (shm_id, start_addr, SHM_RND);
if ((long) start_addr == -1){
     perror ("Shared memory attachment failure");
     exit (1);
     }

      /* Write the shared memory address */

fprintf (keyfile, "%8.8x\n",start_addr);
fclose (keyfile);
```

Example 8–1 creates a temporary file (named /tmp/shared-mem) which is used by the ftok function to generate the *key*. If you use this method, you may use any standard file as long as it exists for the life of the application. Once the virtual address of the shared memory is known, it is written into the temporary

file. Other processes attaching to the shared memory use the file as the *key*, and read the address from the file.

The call to the shmget function specifies the size and *key* of the region and returns the *id*. The example creates a 4096-byte region with protections of read, write, and delete for the user, read and write for the group, and no world access.

Example 8–1 determines where to place the shared memory region in the virtual address space. By default, the kernel adds to the current end of memory and locates the shared memory there, but you can choose the location, as shown in this example.

The decision about where to place the shared memory region can affect all processes that use the shared memory. For example, if you store absolute pointers to parts of the shared memory region, each process should map the region into the same virtual address space.

8.2.2 Attaching and Detaching Shared Memory

Use the shmat function to attach the shared memory region to your virtual address space. Once the shmget call creates a shared memory segment or returns an existing one, use the shmat function to attach a region to its address space. The kernel also sets a flag to indicate that the region is not freed until the last process attached to that shared memory calls the shmdt function and detaches the shared memory or exits, as in the following example:

```
vaddr = shmat (shm_id, 0, 0);
```

In this example, the call to the shmat function returns the starting address of the shared memory segment. The second argument, *shmaddr*, is 0, which specifies that the kernel selects the address space for the caller. You can specify a non-zero value for *shmaddr*, but it may make the application less portable.

Shared memory should not overlap other regions in the virtual address space. Chose a shared memory segment such that other regions cannot grow into the shared memory space. In particular, data and stack regions for processes can grow due to the activity of various functions, or they can grow dynamically as the process executes. To guard against overlap, keep shared memory segments clear of data areas and away from the top of the stack. Specifying a value of 0 for *shmaddr* helps to avoid these problems.

Note also, that shared memory must be page aligned.

The last argument, *shmflg*, is 0, which specifies that read and write permissions are granted as long as they were specified in the shmget call. The SHM_RDONLY flag is the only other valid value for *shmflg*.

Example 8–2 shows the steps involved in attaching shared memory.

Example 8–2 Attaching Shared Memory

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

char *start_addr;
FILE *keyfile;
int shm_key, shm_id;

      /* Fetch the key */

keyfile = fopen ("/tmp/shared-mem", "r");
if (keyfile == NULL){
     fprintf ("Shared memory not available\n");
     exit (1);
     }
shm_key = ftok ("/tmp/shared-mem", 'a');

      /* Access shared region */

shm_id = shmget (shm_key, 4096, 0);
if (shm_id == -1){
     perror ("Shared memory region access");
     exit (1);
     }

      /* Determine the virtual address */

fread (keyfile, "%x",&start_addr);
fclose (keyfile);

      /* Map the region into virtual memory */

start_addr = (char *) shmat (shm_id, start_addr, 0);
if ((long) start_addr == -1){
     perror ("Shared memory attachment failure");
     exit (1);
     }
```

Example 8–2 first determines the *key* to be used by all processes accessing the shared memory region. The address of the shared memory is stored in the temporary file /tmp/shared-mem and is read from the file. Access to the shared memory is obtained through a call to the shmget function.

The example determines where to locate the shared memory region and attaches the shared memory region through a call to the shmat function.

8.2.3 Locking Shared Memory

You can lock and unlock a shared memory segment into physical memory using the memlk and memunlk functions. You can lock shared memory to eliminate paging and to increase execution speed of the application.

8.2.4 Removing Shared Memory

When a process is finished using a shared memory segment, you can detach it from memory with a call to the shmdt function, as shown in the following example:

```
shmdt = (shm_id);
```

The shmdt function detaches the shared memory by disociating the identifier from the shared memory table entry, but it does not remove the table entry. You must use the shmctl function with the IPC_RMID flag to remove the shared memory segment from the system.

Once shared memory segment is detached, remove it from memory with a call to the shmctl function with the IPC_RMID flag, as follows:

```
shmctl (shm_id, 0, IPC_RMID, 0);
```

The last process to use the shared memory should remove it with an explicit call to the shmctl function. You can set up a signal handler to remove the shared memory as one of the tasks performed by the last process in your application.

If you used a temporary file such as /tmp/shared-mem to store the address of the shared memory, you may want to remove both the file and the shared memory from memory since there are limits to the number of IPC facilities that can be created. The example below determines the *id* of the shared memory, deletes the temporary file, then detaches both.

```
shm_key = ftok ("/tmp/shared-mem", 'a');
shm_id = shmget (shm_key, 4096, 0);

unlink ("/tmp/shared-mem");

shmdt (shmid);
i = shmctl (shm_id, IPC_RMID, &tmp);
    if (i == -1) perror ("Shared memory object deletion");
```

8.3 Shared Memory and Semaphores

When using shared memory, processes map the same area of memory into their address space. This allows for fast interprocess communication because the data is immediately available to any other process using the same shared memory. If your application has multiple processes contending for the same shared memory resource, you must coordinate access.

Binary semaphores can be used to regulate access to shared memory and to determine if a shared memory resource is available. Typically, an application will begin execution at a nonrealtime priority level, then perform the following tasks when an application uses shared memory and semaphores:

- Create a shared memory region
- Determine the virtual address and map the region into memory
- Use the same key that you used to establish shared memory to create and reserve a binary semaphore
- Adjust the process priority and scheduling policy as needed
- Before a read or write operation, lock (reserve) the semaphore
- After a read or write operation, unlock (release) the semaphore

Refer to Chapter 9 for information on binary semaphores and an example using semaphores and shared memory.

9

Semaphores

Semaphores do not transfer data; rather, semaphores are used by cooperating processes to synchronize access to resources (most commonly, shared memory). Semaphores can protect the following resources available to multiple processes from uncontrolled access:

- Global variables, such as file variables, pointers, counters, and data structures. Protecting these variables means preventing simultaneous access by more than one process, such as reading information as it is being written by another process.
- Hardware resources, such as disk and tape drives. Hardware resources require controlled access because simultaneous access can result in corrupted data.
- The kernel. The kernel, like a global variable, is a shared resource. A semaphore can allow processes to alternate execution by limiting access to the kernel on an alternating basis.

Semaphore protection works only if all the communicating processes using the shared resource cooperate by waiting for the semaphore when it is unavailable and resetting the semaphore count when relinquishing the resource. For cooperating tasks, semaphores are mutual exclusion flags (mutexes) that lock and unlock a resource.

This chapter includes the following sections:

- Data Structures Associated with Semaphores, Section 9.1
- Binary and Counting Semaphores, Section 9.2
- Semaphores as Event Flags, Section 9.3
- The Semaphore Interface, Section 9.4
- Semaphore Example, Section 9.5

Incrementing or decrementing the semaphore value manages the locks. System V semaphores are handled by the kernel such that the semaphore operations of testing the current value and decrementing (or incrementing) the value are done atomically. These two atomic operations, increment and decrement, are done by the kernel such that no other process can adjust the semaphore values until all operations are complete.

9.1 Data Structures Associated with Semaphores

A semaphore is a set of nonnegative integers, that can range from 1 to an implementation-defined maximum. Each value in the set can assume any value within the permitted range. The kernel contains data structures that define semaphore sets, specify access permission, keep count of the number of processes waiting for access to the resource, and time-stamp semaphore activities.

You create or access a semaphore with a call to the `semget` function. The `semget` function creates or returns the semaphore identifier (*semid*), a unique positive integer. Each semaphore identifier has a set of semaphores and a data structure associated with it. All processes using that semaphore set use the same identifier to access it.

A semaphore object is a `semid_ds` structure that points to an array of `sem` structures. When you call the `semget` function, the `semid_ds` structure is initialized and establishes the identity of the semaphore object. The `semid_ds` structure takes the following form:

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

struct semid_ds {
     struct ipc_perm sem_perm; /* Operation permission structure          */
     struct sem *sem_base;     /* Pointer to array of semaphores          */
     ushort sem_nsems;         /* Number of semaphores in set             */
     time_t sem_otime;         /* Last semop time -  seconds since Epoch  */
     time_t sem_ctime;         /* Last semctl time - seconds since Epoch  */
     };
```

The value of *sem_nsems* is equal to the number of semaphores, *nsems*, in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. The *sem_num* values run sequentially from 0 to the value of *sem_nsems*–1.

9.1.1 Establishing Semaphore Operation Permissions

Processes can use semaphores as long as the calling process has permission. Semaphores use a sem_perm structure to determine permission. The sem_perm structure is an ipc_perm structure, that uses information specific to semaphores.

```
ushort cuid;            /* Creator user ID                    */
ushort cgid;            /* Creator group ID                   */
ushort uid;             /* Owner's user ID                    */
ushort gid;             /* Owner's group ID                   */
ushort mode             /* Read/write (or alter) permission   */
u_short seq;            /* Slot usage sequence number         */
key_t key;              /* Key                                */
```

In the semop and semctl functions check permission needed to use semaphores. Permission is interpreted as follows:

```
00400     Read by user
00200     Write (or alter) by user
00060     Read, Write (or alter) by group
00006     Read, Write (or alter) by others
```

Read and write (or alter) permissions are granted to a process if the ID for the calling process matches one or more combinations of permissions or if the effective user ID (UID) of the process is superuser.

9.1.2 Tracking Semaphore Activity

A semaphore is a data structure as defined in the <sys/sem.h> header file. The sem structure takes the following form:

```
struct sem {
     ushort semval;         /* Semaphore value              */
     short sempid;          /* Pid of last operation        */
     ushort semncnt;        /* Number waiting semval > cval */
     ushort semzcnt;        /* Number waiting semval = 0    */
     };
```

The *semval* member is a nonnegative integer. The *sempid* member is equal to the process ID (PID) of the last process that performed a semaphore operation on this semaphore. The *semncnt* member is a count of the number of processes that are currently suspended awaiting this semaphore's *semval* to become greater than its current value. The *semzcnt* member is a count of the number of processes that are currently suspended awaiting this semaphore's *semval* to be 0.

In addition to keeping track of the set values for a semaphore, the kernel keeps other information for each value in the set. The ID (PID) of the process that performed the last operation on the value, a count of the number of processes waiting for the value to become zero or to increase is also kept.

The semaphore operation structure is also defined in the <sys/sem.h> header file and is of the type sembuf. This structure sets up the fields used in the semop function for semaphore reserve and release operations. The sembuf structure takes the following form:

```
struct sembuf {
     ushort sem_num;                /* Semaphore number     */
     short sem_op;                  /* Semaphore operation  */
     short sem_flag;                /* Operation flags      */
     };
```

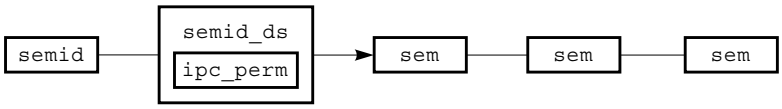
The value of the *sem_num* member identifies each individual semaphore in the semaphore set and is used to control and track individual resources. For example, if your application uses one semaphore structure to control access to each of 20 temperature measurement devices, then the value of the *sem_num* member would range from 0 to 19.

The value of the *sem_op* member is the value that you want to add to the current value of the semaphore. To track more than one resource, specify a value greater than 1 for the value of *sem_op*. Then you can increment or decrement as you would for any other counting construct.

The *sem_flag* member sets the operation flags IPC_NOWAIT or SEM_UNDO.

Figure 9–1 illustrates how the semaphore data structures relate to each other.

Figure 9–1 Representation of Semaphore Data Structures



MLO-007325

9.2 Binary and Counting Semaphores

You can use semaphores to keep track of a large number of resources, such as 20 analog measurement devices or numerous tape drives. You can also use semaphores to track the availability of each resource. For example, if your application takes temperature readings from 20 devices, one semaphore in your array of semaphores could count from 0 to 19, while the other semaphores control access so that only one reading is taken at a time. To accomplish this, you need to use both binary and counting semaphores.

Semaphores are essentially counting structures that are used to keep track of resource usage. A semaphore with a maximum count of 1 is called a binary semaphore because it has only two states, 0 and 1. Binary semaphores protect shared resources from uncontrolled multiple access. For example, if a process has access to a shared memory segment, the value of the semaphore is set to 0. If the process relinquishes access, the value of the semaphore is set to 1. In this case, the maximum count for the semaphore is 1. Such semaphores are often called mutual exclusion semaphores or mutexes.

Semaphore operations are accomplished with a call to the `semop` function. For binary semaphores, operations are essentially reserving the resource and releasing it after exclusive use is complete.

The reserve operation checks to see if the resource is available or is reserved by another process. If the resource is not already reserved, a reservation is made and the process continues. If the resource is reserved, the process making the second reservation waits (is blocked) until the first process releases the resource. Several processes may be blocked waiting for a resource to become available.

The release operation sets the semaphore value to indicate that the resource is not reserved. The waiting process, if there is one, is unblocked and it accesses the resource.

A counting semaphore acts as a meter that allows multiple processes to access a resource. Counting semaphores permit a predetermined number of processes to share a resource one at a time, while also permitting a single process to reserve the shared resource for exclusive access. All of the processes that share a resource must agree on which semaphore array controls the resource. With each operation, the semaphore counter is incremented or decremented. When the semaphore counter reaches 0, the process requesting the next access waits until some other process releases the resource and the count is incremented. Several processes can be blocked waiting for access.

Access to a resource is either granted or denied; however, the value of the semaphore operation is the result of arithmetic calculations of absolute values. Refer to the reference pages for a full explanation of how the semop function calculates these values.

Most applications use both binary and counting semaphores. You can to set up an array of semaphores with one counting semaphore and multiple binary semaphores. This array can track the availability of individual resources and resource allocation for the complete set of resources.

9.3 Semaphores as Event Flags

All processes that have access to a particular event flag can examine or change the value of the flag. Cooperating processes must know the key, and agreee on how the semaphores will be utilized.

The primitive operations are summarized as follows:

- Wait — Initializes the event flag to the clear value, which is 1
- Set — Sets the event flag to the value of 0
- Clear — Sets the event flag to a value of 1
- Read — Retrieves the current value of the event flag
- Wait — Blocks execution until some other process, or a signal handler within the process, sets the event flag

You set or clear the event flag using the semctl function and the SETVAL control flag as follows:

```
semctl(sem_id, sem_num, SETVAL, 1);      /* Initializes an event flag */
semctl(sem_id, sem_num, SETVAL, 1);      /* Clears an event flag      */
semctl(sem_id, sem_num, SETVAL, 0);      /* Sets an event flag        */
```

Use the semctl function and the GETVAL control flag to read the current value of the flag.

Use the semop function to wait for the event flag to be set. Since a set flag has been defined as 0, the process should call the semop with an operation code of 0. This causes the process to wait until the event flag is set.

Example 9–1 illustrates one way to use a semaphore as an event flag.

Example 9–1 Using Semaphores as Event Flags

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

int sem_id;             /* Semaphore array ID, from semget */
int sem_num;            /* Semaphore number in array       */
struct sembuf sops;     /* Semaphore operation structure   */
     .
     .
     .
sops.sem_num = sem_num; /* Set the semaphore number  */
sops.sem_op = 0;        /* Operation                 */
sops.sem_flg = 0;

for(;;){                /* Loop for signal deliver   */
     if(semop(sem_id,&sops, 1) == -1)
     {
          if(errno != EINTR){          /* Failed - signal ? */
          perror("semop failure");
          exit(1);                     /* Failed */
     }
continue;              /* Signal, check event flag again */
     }
     else
     break;            /* Event flag is set   */
     }
          .
          .
          .
```

Note that you cannot use semaphores to wait for logical combinations of event flags or to retrieve the previous event flag setting.

9.4 The Semaphore Interface

The functions relating to semaphores follow the same general logic as for other System V IPC facilities. The `semget` function allocates a semaphore set, but the members of the semaphore set data structure are not set until the process calls the `semctl` function.

Table 9–1 lists the functions that allow you to create and control semaphores.

Table 9–1 Semaphore Functions

Function	Description
semget	Returns the semaphore identifier
semctl	Provides semaphore control operations
semop	Performs an array of semaphore operations on the set of semaphores associated with the semaphore

The semget function allocates semaphores. Operations on semaphores are sometimes referred to as reserve operations. The phrase "reserve a semaphore" refers to operations that decrement a semaphore value. Semaphore values are decremented when a semaphore is requested, locked, set, or blocked.

The phrase "release a semaphore" refers to operations that increment a semaphore value. Semaphore values are incremented when a semaphore is released, unlocked, cleared, or awakened. Section 9.4.3 discusses how to use the semop function to get and release semaphores.

9.4.1 Creating and Opening Semaphores

A call to the semget function allocates a specified number of semaphores in one set of semaphores, which is identified by the descriptor *semid*. On a call to the semget function, (if there is no entry and the user set the IPC_CREAT or IPC_PRIVATE flag), the kernel allocates an entry that points to an array of semaphore structures (semid_ds structures) with a user-specified number of elements. The identifier is then returned to the calling process. The entry also specifies the number of semaphores in the array and the time of the last call to the semctl and semop functions.

If your application consists of related processes, you may want to set the IPC_ PRIVATE flag on the function:

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

sem_id = semget (IPC_PRIVATE, 4, 0600);
```

This call creates a set of four semaphores with read and write permission for the owner. Be sure to initialize the individual semaphores in the array immediately after you create the array.

When you call the semget function before calling fork to create child processes, the child process inherits the IPC channel as well as the IPC permissions. This enables all related processes to read to and write from the semaphore.

If your application consists of unrelated processes, you can use the ftok function to establish a key based on other criteria. Other unrelated processes that communicate through the shared memory must call the semget function to get the *semid*.

```
sem_id = semget (ftok ("/usr/users/example4", 'X'), 4, IPC_CREAT|0600);
```

This call creates a new set of semaphores, but the call to the ftok function returns a *key* based on the */usr/users/example4* pathname and the character *X*. As other processes call the semget function, *sem_id* is returned for them to use in other semaphore functions.

The following example creates a semaphore set containing 25 semaphores. In this example, the file /tmp/semaphore is used in the ftok call to generate the *key*.

```
int sem_key, sem_id;

sem_key = ftok ("/tmp/semaphore", 'a');
sem_id = semget (sem_key, 25, IPC_CREAT | 0777);
```

This example creates the maximum number of semaphores in an array (25), numbered from 0-24. To access a previously allocated semaphore array, a process needs only to determine the *sem_id* from the agreed-upon *key*.

```
int sem_key, sem_id;

sem_key = ftok ("/tmp/semaphore", 'a');
sme_id = semget (sem_key, 25, 0);
```

If you create shared memory and a semaphore at the same time, you can use the temporary file that holds the virtual address of the shared memory as the *key* for both. In this case, the call to the ftok function would specify the same file name and a different *id*, as follows:

```
sem_key = ftok ("/tmp/shared-mem", 'b');
```

9.4.2 Controlling Semaphores

The semctl function takes a number of flags to control the values of the members of semaphore (sem) data structures. The sem structure is used to initialize semaphores, to create binary and counting semaphores, or to initialize structure members to serve as event flags. The semctl function allows you to retrieve status. The semctl function takes more flags than other IPC ctl functions to allow you access to the status of individual members of the sem structure.

The `semctl` function uses the GETALL flag to copy the values contained in the `sem` structure into an array. You can then alter the values and pass them back to the `sem` structure with the SETALL flag. When this command is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared on exit.

If you use the SETVAL flag, the *semadj* value corresponding to the specified semaphore in all processes is cleared.

You must have read permission to use all of these flags, except SETALL and SETVAL. You must have alter permission to use SETALL or SETVAL. Table 9–2 shows the flags used on the `semctl` function.

Table 9–2 Semaphore Command Control Flags

Flags	Description
GETALL	Places all *semvals* into array pointed to by *arg.array*
GETNCNT	Returns the value of *semncnt*
GETPID	Returns the value of *sempid*
GETVAL	Returns the value of *semval*
GETZCNT	Returns the value of *semzcnt*
IPC_RMID	Removes a semaphore ID and the associated data structure
IPC_SET	Sets the user and group IDs and operation mode values
IPC_STAT	Returns the status information in the associated data structure and copies it into a user-specified buffer
SETALL	Sets all *semvals* according to the array pointed to by *arg.array*
SETVAL	Sets the value of *semval* to *arg.val*

These control flags allow you to control semaphores by performing the following functions:

- Return the value of the semaphore
- Set the value of a semaphore
- Return the ID (PID) of the last process that performed an operation on the semaphore set
- Return the number of processes waiting for a semaphore value to be equal to zero
- Return the number of processes waiting for a semaphore value to increment

- Return all semaphore values
- Set all semaphore values
- Return all semaphore structure member values and status
- Change operation permissions
- Remove semaphores, semaphore sets, and their associated data structures

Note that you must be either the owner or have superuser privileges to use the IPC_SET or IPC_RMID flags.

9.4.2.1 Using SETALL to Initialize Semaphores

You can use the SETALL flag to initialize all the semaphores at once. The SETALL flag does the initialization atomically and there is no need to use the SETALL flag more than once.

Example 9–2 illustrates how to initialize semaphores using the SETALL flag on the semctl function and how to retrieve the *semval* values using the GETALL flag.

Example 9–2 Initializing Semaphores with SETALL

```
/* This program first allocates semaphores with a call to semget */
/* then uses the SETALL flag with the semctl to initialize each  */
/* semaphore. */

/* The semctl function is made again with the GETALL flag to     */
/* verify initialization. The results are printed out.           */

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

#define numberofsems   2
#define firstsem       0
#define secondsem      1

int sem_id;
ushort semarray[numberofsems];

main()
{
     if ((sem_id = semget (IPC_PRIVATE, numberofsems, 0600)) == -1)
          { perror ("Call to semget failed");
            exit (1);
          }                          /* Exit on error. */

     else printf ("sem_id is %d\n", sem_id);
```

(continued on next page)

Example 9–2 (Cont.) Initializing Semaphores with SETALL

```
      /* Print the value of sem_id.  */

semarray[firstsem]  = 8;       /* Assign values to the semaphores */
semarray[secondsem] = 9;       /* using semctl SETALL.            */

     if (semctl (sem_id, 0, SETALL, semarray) == -1)
          {perror ("Call to semctl SETALL failed");
           exit (1);
         }                          /* Exit on error. */

      /* Verify initialization with a call to semctl GETALL */

     else if (semctl (sem_id, 0, GETALL, semarray) == -1)
          {perror ("Call to semctl GETALL failed");
          exit (1);
          }      /* Exit on error. */

     else {
          printf ("The semval value of firstsem = %d\n"' semarray[firstsem]);
          printf ("The semval value of secondsem = %d\n"' semarray[secondsem]);
          }
```

Example 9–2 illustrates how you can call the semctl function with the GETALL flag to verify the results.

9.4.2.2 Using SETVAL to Initialize Semaphores

To initialize the semaphores one at a time, use the SETVAL flag. Example 9–3 illustrates how to initialize semaphores using the SETVAL flag on the semctl function. Note that you cannot set a semaphore value to a negative number.

Example 9–3 Initializing Semaphores with SETVAL

```
/* This program first allocates semaphores with a call to */
/* semget, then uses the SETVAL flag with the semctl to   */
/* separately initialize each semaphore. */

#include <sys/stypes.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define numberofsems   2
#define firstsem       0
#define secondsem      1

int sem_id;
ushort semarray[numberofsems];

main()
{
     if ((sem_id = semget (IPC_PRIVATE, numberofsems, 0600)) == -1)
          {perror ("Call to semget failed");
           exit (1);
          }      /* Exit on error. */

     else printf ("sem_id is %d\n", sem_id);

      /* Print out the value of sem_id  */

semctl (sem_id, firstsem, SETVAL, 8);
semctl (sem_id, secondsem, SETVAL, 9);

      /* Initialize semaphores using semctl and SETVAL.*/
}
```

9.4.2.3 Initializing Binary and Counting Semaphores

You control whether the semaphore will be a binary or a counting semaphore by the value you assign to the *arg* argument in the `semctl` function. To use the semaphore as a binary semaphore, set the value of each argument in the array to the value of 1.

```
int sem_id;           /* ID of the semaphore array, from semget */
int sem_num;          /* semaphore number in semaphore array    */

semctl (sem_id, sem_num, SETVAL, 1);
                      /* Sets the value to 1                    */
```

To create a counting semaphore, set the value of the access count of each semaphore that will be used as a counting semaphore in the array of semaphores. The following example initializes the access count to ACOUNT for simultaneous shared access.

```
int sem_id;           /* ID of the semaphore array, from semget */
int sem_num;          /* semaphore number in semaphore array    */

semctl (sem_id, sem_num, SETVAL, ACOUNT);
                      /* Sets the value to ACOUNT               */
```

9.4.3 Using Semaphore Operations

After you have created your semaphore sets with a call to the semget function and initialized the semaphore set values with a call to the semctl function, use the semop function to perform operations on the semaphore sets.

The semop function is used to perform an array of semaphore operations atomically on the set of semaphores identified by the semaphore identifier. The syntax for the semop function includes a pointer (*sops*) to an array of semaphore operation structures (sembuf structures) as well as the number of structures in the array (*nsops*).

The semop function manipulates members of the sembuf structure to allow access to a resource. The semop function changes the value of the *sem_op* member to reflect whether you are reserving or releasing a semaphore.

The value of the *sem_num* member identifies each individual semaphore in the semaphore set and controls and tracks individual resources. For example, if your application used one semaphore to control access to each of 20 temperature measurement devices, then the value of the *sem_num* member would range from 0 to 19.

The *sem_op* member stores the value that you want to add to the current value of the semaphore. For example, to reserve a semaphore, make the value of *sem_op* negative. To release a semaphore, make the value of *sem_op* positive. If *sem_op* has a value of 0, neither a reserve nor release operation is performed. To track more than one resource, specify a value greater than 1 for the value of *sem_op* and increment or decrement as you would for any other counting construct.

The use of semaphores to share resources among processes will work only if processes release the resource immediately after they finish using it. As you code your application, take care not to do a release operation on a semaphore you have not reserved. If you do, the value of the semaphore will increase.

The *sem_flag* member sets the operation flags.

If the kernel cannot do all the operations, the process sleeps until the kernel can finish. To keep the process from sleeping, specify the IPC_NOWAIT flag.

The SEM_UNDO flag performs cleanup work when that the application exits. If the SEM_UNDO flag is set, semaphore reservations are automatically released when a process exits. It is good practice to use SEM_UNDO to prevent possible deadlocks in the event that the process terminates.

9.4.3.1 Reserving a Semaphore

To reserve a semaphore, call the semop function with an operation code of –1 on the appropriate semaphore. Example 9–4 reserves a binary semaphore.

Example 9–4 Reserving Binary Semaphores

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

int sem_id;                   /* Semaphore array ID, from semget */
int sem_num;                  /* Semaphore number in array       */
struct sembuf sops;           /* Semaphore operation structure   */
   .
   .
   .
sops.sem_num = sem_num;       /* Set the semaphore number        */
sops.sem_op = -1;             /* Operation -1                    */
sops.sem_flg = SEM_UNDO;      /* Undo on process exit            */

while (1) {                   /* Loop for signal delivery        */
    if (semop (sem_id, &sops, 1) == -1) {
        if (errno !=EINTR) {  /* Failed - signal?                */
            perror ("semop failure");               /* no        */
            exit (1);
            }
        }                      /* It was a signal                */
    }                          /* Retry semop                    */
   .
   .
   .
```

The following example shows the code for a reserve operation on semaphore number 19 in an array of 20 semaphores.

```
sops.sem_num = 19;       /* Set the semaphore number */
sops.sem_op  = -1;       /* Operation -1             */
sops.sem_flag   = 0;     /* No flags                 */
semop (sem_id, sops, 20);
```

This operation adds the *sem_op* value of –1 to the *semval* specified for semaphore number 19. If the value of *semval* is greater than or equal to 1, then the arithmetic is performed. If *semval* is less than 1, the calling process sleeps until some other process releases the semaphore, which increments the value of *semval*, or semaphore operations are interrupted by a signal, or the semaphore is removed with a call to semctl with the IPC_RMID flag.

To reserve a counting semaphore with a count of ACOUNT, as in the example, call the semop function with an operation code of –ACOUNT on the appropriate semaphore operation. The following example shows the operation to add the *sem_op* value of –ACOUNT to the *semval* specified for *sem_num*.

```
sops.sem_num = sem_num;       /* Set the semaphore number */
sops.sem_op  = -ACOUNT;       /* Operation -ACOUNT        */
sops.sem_flag   = 0;          /* No flags                 */
semop (sem_id, sops, 1);
```

9.4.3.2 Releasing a Semaphore

To release a semaphore, make the value of *sem_op* positive. The following example shows the code for a release operation on a binary semaphore:

```
sops[0].sem_num = sem_num;       /* Set the semaphore number */
sops[0].sem_op  = 1;             /* Operation +1             */
sops.sem_flag   = 0;             /* No flags                 */
semop (sem_id, sops, 1);
```

This operation adds the *sem_op* value of 1 to the *semval* specified for *sem_num*. A release operation does not cause the calling process to sleep, but the kernel will wake any process that is sleeping while waiting for the value of the *sem_op* to change.

To release a counting semaphore with a count of ACOUNT, call the semop function with an operation code of ACOUNT on the appropriate semaphore operation. The following example shows code for this operation:

```
sops.sem_num = sem_num;       /* Set the semaphore number */
sops.sem_op  = ACOUNT;        /* Operation +ACOUNT        */
sops.sem_flag   = 0;          /* No flags                 */
semop (sem_id, sops, 1);
```

To ensure that the calling process does not sleep while waiting for a semaphore to become available, specify the IPC_NOWAIT flag on the semop function.

You can also release a semaphore with the SEM_UNDO flag, which automatically releases any semaphore reservation when the process exits. Such an application might have code that looks like the following:

```
sops.sem_num = sem_num;       /* Set the semaphore number       */
sops.sem_op = -1;             /* Operation -1                   */
sops.sem_flg = SEM_UNDO;      /* Undo on process exit           */
```

9.4.4 Removing Semaphores

To remove semaphores and their associated data structures you can use the IPC_RMID flag on the semctl function, or the ipcrm command. You can use the semctl function to remove a semaphore while your application is running. Use the ipcrm command to remove the semaphore when the application is not running.

The last process in your application to use the semaphore set should remove it with a call to either the semctl or semop function. If a process is waiting for a semaphore and you delete the semaphore, the waiting process will get an EIDRM error. As with other IPC facilities, you can set up a signal handler to remove the semaphore set as one of the tasks performed by the last process in your application.

If you are the owner of the semaphore or have superuser privileges, you can remove semaphores and semaphore sets from within your application by using the IPC_RMID flag on the semctl function. When a process is finished doing operations on semaphores, remove the semaphores from memory as follows:

```
int sem_key, sem_id;

sem_key = ftok  ("/tmp/semaphore", 'a');
semctl (sem_id, 0, IPC_RMID, &tmp);
```

You can also remove semaphores from the system with the ipcrm command after the application is done running. Refer to the reference pages for more information on the ipcrm command.

9.5 Semaphore Example

It is important that two processes not use the same area of shared memory at the same time. Binary semaphores protect access to resources such as shared memory. Before reading or writing to a shared memory region, a process can lock the semaphore to prevent another process from accessing the region until the read or write operation is completed. When the process is finished accessing the shared memory region, the process unlocks the semaphore and frees the shared memory region for use by another process.

Example 9–5 uses semaphores as event flags to ensure that the writer and reader processes have exclusive, alternating access to the shared memory region. For example, the writer process writes a block of text, then the reader process reads, and then the writer process writes again.

Example 9–5 is a partial example, but it gives the framework essential for using shared memory and semaphores together for interprocess communication and resource access control. This program needs to be started before the reader process, as it creates the shared memory and semaphores used by both processes.

The `makefile` for this program follows the example.

Example 9–5 Using Semaphores and Shared Memory

```
/* This program reads data from standard input into a shared */
/* memory segment then it is copied by reader.*/

/* A child process is created as a reader and writes data to */
/* standard output from the shared memory segment. */

#include <errno.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>
#include <fcntl.h>

struct shm_seg *create_shm();
struct shm_seg{
     int nbytes;
     char buf[4096];
     };

#define WRITE_TURN 0
#define READ_TURN  1
#define SET        0
#define CLEAR      1
#define FILE_KEY   "/tmp/file_key"

#define syserr(s){   \
     perror(s);      \
     exit(1);        \
     }

#define set_flag(sid, semp){            \
     if(semctl(sid, semp, SETVAL, SET)) \
     syserr("semctl: SETVAL");          \
     }

#define clear_flag(sid, semp){            \
     if(semctl(sid, semp, SETVAL, CLEAR)) \
     syserr("semctl: SETVAL");            \
     }
```

(continued on next page)

Example 9–5 (Cont.) Using Semaphores and Shared Memory

```
#define SEG_SIZE sizeof(struct shm_seg)
#define MAX_CHARS SEG_SIZE - sizeof(int)

char child_msg[] = "READER:  write data to stdout from buffer...\n";
char parent_msg[] = "WRITER: read data from stdin into buffer...\n";

/******************  Main ***************************/

main(int argc, char *argv[])
{
int sem_id;
int shm_id;

struct shm_seg *shm_addr;
int  nb;

      /* Create key file for ftok, if it doesn't already exist */

if((open(FILE_KEY, O_RDWR|O_CREAT, 00666)) == -1)
     syserr("open:");

sem_id = create_sem(FILE_KEY, 's');
shm_addr = create_shm(FILE_KEY, 'm', &shm_id);

      /********* Child process:  the reader *********/

if(fork() == 0){
     for(;;){
          if(waitfor(sem_id, READ_TURN) == -1){
               perror("reader: semop");
               break;  /* Break if EOF */
               }

          if(shm_addr->nbytes == 0)
               break;
               write(1,child_msg, sizeof(child_msg));
               if((write(1, shm_addr->buf, shm_addr->nbytes))== -1){
                    perror("reader: write");
                    break;
                    }
      /* Give the writer a turn */

               clear_flag(sem_id, READ_TURN);
               set_flag(sem_id, WRITE_TURN);
                }
        set_flag(sem_id, READ_TURN);
        exit(0);
}                       /* Child process */

else{

      /********* Parent Process: the writer *********/
```

(continued on next page)

Example 9–5 (Cont.) Using Semaphores and Shared Memory

```
for(;;){      /* Wait for turn */

     if(waitfor(sem_id, WRITE_TURN) == -1){
          perror("writer: semop");
          break;
          }
      write(1, parent_msg, sizeof(parent_msg));

         /* Read data from stdin into shared memory segment */

      if((nb = read(0,shm_addr->buf, MAX_CHARS)) == -1){
           perror("writer: read");
           break;
           }

        shm_addr->nbytes = nb;

        /* Give the reader a turn */

        clear_flag(sem_id, WRITE_TURN);
        set_flag(sem_id, READ_TURN);
        if(nb==0)
        break;               /* EOF */
}

if(unlink(FILE_KEY) == -1)
     perror("unlink");

/* Wait until reader has a turn, and seen the EOF - we now know */
/* the child must have finished, so we can delete the semaphores*/
/* and shared memory. */

waitfor(sem_id, WRITE_TURN);

if(semctl(sem_id, 0, IPC_RMID, 0) == -1)
     perror("semctl: IPC_RMID");

shmdt(shm_addr);
exit(0);
}
}                /* main */

/***************** create_sem Function *****************/
/* This function creates a cluster of two semaphores.  */
/* One is used to signify that the writer process can  */
/* access the shared memory, the other to signify that */
/* the reader can have a turn. */

int create_sem(char *fn, char *key_char)
     {key_t sem_key;
      int   sem_id;
```

(continued on next page)

Example 9–5 (Cont.) Using Semaphores and Shared Memory

```
/* Generate a key used to create the semaphore */

      sem_key = ftok(fn, key_char);

/* Create a cluster containing 2 semaphores */

      if((sem_id = semget(sem_key, 2, IPC_CREAT | 00777)) == -1)
           syserr("create_sem: semget");

/* Initialize event flags - WRITER gets first turn to access shared
   memory */

set_flag(sem_id, WRITE_TURN);
clear_flag(sem_id, READ_TURN);

return(sem_id);
}                  /* create_sem */


/***************** create_shm Function *****************/
/* This function creates a shared memory segement large*/
/* enough to hold a shm_seg structure.  This structure */
/* (composed of a length and a text field) is used to  */
/* pass data in shared memory between the writer and   */
/* reader process. */

/* The shared memory id is returned in shm_id, and the */
/* attached address of the segment is returned as the  */
/* function value. */

struct shm_seg *create_shm(char *fn, char *key_char, int *shm_id)
     {struct shm_seg *start_addr;
      FILE *keyfile;
      key_t shm_key;

      /* Set up the key - ftok returns a key based on the */
      /* given path and single character id */

shm_key = ftok(fn, key_char);

      /* Create the shared region object - SEG_SIZE bytes, */
      /* protected rw-rw-rw-.  IPC_CREATE means create if  */
      /* it doesn't already exist */

if((*shm_id = shmget(shm_key, SEG_SIZE, IPC_CREAT | 0666)) == -1)
     syserr("Create_shm: shmget");

      /* Map the shared region into virtual memory - SHM_RND */
      /* means align on a page boundary */

start_addr = (void *)shmat(*shm_id, 0, SHM_RND);
```

(continued on next page)

Example 9–5 (Cont.) Using Semaphores and Shared Memory

```
if((long *) start_addr == (void *)-1)
     syserr("create_shm: shmat");
return(start_addr);
}                                          /* create_shm */

int waitfor(int sem_id, int sem_num)
     {struct sembuf sops;     /* Semaphore operation structure */
      sops.sem_num = sem_num; /* Set the semaphore number      */
      sops.sem_op = 0;
      sops.sem_flg = SEM_UNDO;
      return(semop(sem_id, &sops, 1));
      }

/*************************************************************/
/*************************************************************/

/* # makefile for semaphore program.  */

/* CFLAGS = -g3 -non_shared -O  \      */
/* -DLANGUAGE_C  -D_OSF_SOURCE         */

/* LDFLAGS = -L/usr/ccs/lib            */

/* sem_test: sem_test.o                */
/*  $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $? -lrt  */

/* sem_test.o: sem_test.c              */
```

10
Pipes

A pipe is a structure that facilitates interprocess communication that provides a flow of data between related processes. One process reads from an I/O channel while another process writes to the I/O channel. All pipes require a sending process (called a writer) and a receiving process (called a reader). Pipes are unidirectional and will not work properly unless both a reader and a writer are identified. However, only one reader and one writer can be associated with a pipe.

This chapter includes the following sections:

- Regular Pipes, Section 10.1
- Named Pipes, Section 10.2

There are two types of pipes; regular and named pipes. Regular pipes are invoked by the `pipe` system call and are known only to processes which are descendants of the process that invoked the `pipe` system call. Named pipes are identical to regular pipes except for the way that processes access the pipe. Named pipes use file descriptors and are accessed by a pathname.

Pipes, whether they are regular or named pipes, use the stream I/O model. Data is transferred without any interpretation by the system. Messages in pipes have no record boundaries.

10.1 Regular Pipes

Pipes can be used between parent and child processes or between child processes of the same parent. Data moves in a one-way flow with a single pipe or in a two-way flow if you create more than one pipe. Data transfer using pipes is subject to rules for reading and writing. If you open the pipe with both read and write access, then you have a two-way pipe. If you open the pipe with either read or write, then you have a one-way pipe.

Regular and named pipes use stream I/O to direct data to and from cooperating processes. Data is transferred without any interpretation by the system. Because of this, information sent to a pipe is read in the order in which it is written and there is no mechanism to determine the length of the data sent, stored, or received. If your application needs to interpret the data, the reader and the writer processes must take care of that task.

Pipes move data from one I/O channel to another, which means that a pipe is a memory buffer. Reads from a pipe remove the data from the buffer. Each pipe holds up to PIPE_MAX bytes of data as defined in the `<limits.h>` header file.

A process can read its own data from a pipe, so use the `close` function to control the flow of information. As long as you use sequential reads and writes, you can use a pipe anywhere you would normally use a file descriptor. If all write channels to a pipe are closed, the reader of that pipe will read an end-of-file (EOF) when the pipe is empty.

Writing to a full pipe (PIPE_MAX) blocks the process because the process waits until the pipe empties enough to take the data. Likewise, reading an empty pipe blocks the process because the process waits until there is something in the pipe to read. To avoid blocking, use the O_NONBLOCK flag on the `fcntl` function. If no data is available for the operation or the operation would block the calling process, –1 is returned and the error is EWOULDBLOCK.

10.1.1 Creating a Pipe

Pipes are created by a call to the `pipe` function and are accessed by the file descriptors contained in an integer array. The system uses file descriptors as handles for various objects: including disk files, special files, sockets, and pipes. By convention, always read and write to the file descriptors in both parent and child processes. Use the `sysconf` function to determine how many file descriptors are allowed per process.

The first three file descriptors in any process are:

- File descriptor 0 — Standard input (`stdin`)
- File descriptor 1 — Standard output (`stdout`)
- File descriptor 2 — Standard error (`stderr`)

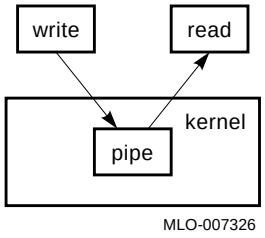
Subsequent file descriptors are allocated sequentially. A call to the `pipe` function, for example, returns two additional file descriptors as follows:

- File descriptor 3 — read
- File descriptor 4 — write

In a two-way pipe, both processes can read and write from the pipe and each process can read the data written by itself. Therefore, it is sometimes easiest to use pipes for for read-only or write-only communication by closing either the write or read end of the pipe in each process.

Figure 10–1 shows a one-way pipe. The parent process writes data to the pipe while the child process reads data from the pipe. Example 10–1 creates a pipe, creates a child process, and then reads a line from `stdin` and writes it to the pipe. The child reads a line from the pipe and writes it to `stdout`.

Figure 10–1 One-Way Pipe



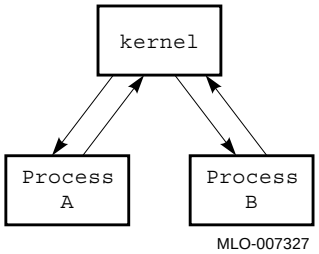
Example 10–1 Creating a Child Process and a Pipe

```
/* This program creates a pipe and a child process. The parent */
/* reads a line from stdin and writes it to the pipe. The child*/
/* reads a line from the pipe and writes it to stdout. */

#include stdio <stdio.h>

main()
{
int  pid,                    /* Process ID returned by fork()           */
     n,                      /* Number of bytes read from pipe by child */
     fd[2];                  /* Array that holds pipe file  descriptors */
char par_line [81],          /* Line buffer for parent   */
     chi_line [81];          /* Line buffer for child */

if  (pipe(fd) == -1)                        /* Create a pipe */
     perror ("pipe.c: pipe failed"), exit(1);

if  ((pid = fork()) == -1)                 /* Create a child */
     perror ("pipe.c: pipe failed"), exit(1);

if (pid == 0)                        /* Child process; execute code */
     {close (fd[1]);                    /* Close write end of pipe */
      n = read(fd[0], chi_line, 80);    /* Read from pipe */
      chi_line[n] = '\0';
      printf ("Child: your line was %s\n", chi_line);
      exit (0);                         /* Successful exit from child */
      }

else                        /* Parent process; execute parent's code */
     {close (fd[0]);                      /* Close read side of pipe   */
      printf ("Enter line: ");
      gets (par_line);                    /* Read line from stdin        */
      write (fd[1], par_line, strlen(par_line)); /* Write line to pipe  */
      wait(0);                            /* Wait for child to exit      */
      exit (0);                           /* Successful exit from parent */
      }
}
```

The system synchronizes read and write activities by blocking when there are not enough characters in the pipe to read or when the pipe is too full to receive a write.

In situations where blocking will significantly delay process execution, you may want to use the O_NONBLOCK flag on the `fcntl` function.

Figure 10–2 Two-Way Pipe



10.1.2 Redirecting stdin, stdout, stderr to Pipes

The information written to or read from a pipe can be redirected to different file descriptors, such as stdin, stdout, and stderr.

Figure 10–2 illustrates two-way pipe communication between two processes. Two-way pipes can be created and managed by using file descriptors.

The dup2 function allows you to duplicate file descriptors. The dup2 function can be used to redirect a process's stdin, stdout, or stderr to a pipe.

First, you create a pipe; then you close an existing file descriptor with a call to the close function. Next, you call the dup2 function, supplying the write channel to the pipe as the object to which the newly allocated descriptor points. Now any writes to stdout (which the system knows as file descriptor 1) are written to the pipe. The writer process writes to file descriptor 1 just as it had before, but now file descriptor 1 points to a pipe rather than stdout.

The following example fragment shows how to use the dup2 function to redirect stdout in a parent. The parent executes the following functions:

```
int nfd[2];
pipe(fd);
```

```
if (fork !=0) {         /* Parent creates two child processes */
     if (fork !=0) {
          dup2(fd[1], stdin);
          close (fd[0]);
               .
               .
               .
          }
else {               /* Child */
     dup2(fd[0], stdout);
     close (fd[1]);
          .
          .
          .
      }
}
```

10.1.3 Creating Pipes with popen

Use the popen function to create a child process that executes a Bourne shell (sh) command. A popen call also creates a one-way pipe between the parent and a child process. The popen function combines the pipe, fork, and exec functions and performs the following tasks:

- Creates a pipe
- Creates a child process
- Creates a Bourne shell in the child process that executes the shell command specified in the popen call
- Causes the shell command to read or write the pipe to communicate with the parent process
- Returns a standard I/O file pointer as the channel to the pipe for the parent to read or write

The value returned by the popen function is a standard I/O file pointer, used for either input or output, depending on the *type* specified in the command.

The pclose function closes the I/O stream created by a call to the popen function.

10.2 Named Pipes

Named pipes (FIFOs) are the same as regular pipes except that named pipes are special files in the files system. You open named pipes with the `open` system call and the *pathname* associated with the file. Data in named pipe special files has no record boundaries.

One shortcoming of regular pipes is that they can only be used between processes that share a common parent. Pipes are passed from one process to another by the `fork` function, and all open files are shared between the parent and child process after the `fork` call. Unrelated processes cannot use regular pipes for communication because they do not share open files. Some other form of interprocess communication must be used for communication between two unrelated processes. One form of communication can be the named pipe. [1]

A named pipe provides a one-way flow of data between unrelated processes. Named pipes are very similar to regular pipes and follow many of the same rules. For example, they both use buffers to store data and read and write to other processes. However, unlike regular pipes, named pipes can communicate with unrelated processes. That means that processes can use the same buffering and synchronization techniques offered by the system for use with regular pipes, even if the processes are not related.

Unlike regular pipes, named pipes have an identifier and exist in a file or directory. The file for the named pipe continues to exist until it is explicitly removed.

Named pipes are created by a call to the `mknod` function. The `mknod` command is most commonly used by system managers or users with `superuser` privileges to create new device entries, but it can also be used by a nonprivileged user to create a named pipe.

The `mknod` function takes three parameters: *pathname*, *mode*, and *dev*. The *pathname* parameter takes a character string specifying the pathname of the file to be created. The *mode* parameter specifies the file type and the access permissions for the file. Refer to the reference pages for an explanation of the `mknod` function and an explanation of the values used in the parameters.

Use the `open`, `fdopen`, or `fopen` functions to associate an open file descriptor with a standard stream I/O. Once this is done, you must decide how the application will handle the data. To reduce the possibility that your process will be blocked while waiting for a reader, writer, or appropriate data, you can use the O_NONBLOCK flag on the `fcntl` function.

---

[1] Named pipes are sometimes referred to as first-in, first-out pipes or FIFOs.

Named pipes must be opened by at least one reader and one writer. If only the reader or only the writer opens the pipe, a signal is generated, and the calling process is suspended until another process opens the pipe. If the process has not called the `signal` function to handle the signal, the default action is to terminate the process.

A regular pipe no longer exists when it is not in use or its application terminates. Because named pipes are files, you must take several steps to remove them. First, call the `close` function to close the open file descriptors, then remove a named file with the `rm` or `unlink` command.

11
Signals

The signal interface is a traditional form of interprocess communication and is generally used to notify processes that something has happened in one process that affects another process. Signals are often sent asynchronously; that is, the receiving process cannot predict when a signal will arrive. The application must contain code to take action once a signal is received. The action can be to ignore the signal, terminate the process, or catch the signal by executing a handler function.

Often, signals are referred to as "software interrupts" and are the software equivalent of a hardware interrupt. Signals are the kernel's mechanism for communicating events to processes. Signals are also sent by a user process to notify another process of an event such as the expiration of a timer.

This chapter includes the following sections:

- P1003.4/D10 Realtime Signals, Section 11.1
- The Signal Interface, Section 11.2

Signals do not pass data, do not identify the sending process, and are not prioritized or queued. Nevertheless, signals are used by timers and other events to trigger the start of a signal handler once the signal is received.

11.1 P1003.4/D10 Realtime Signals

The P1003.4/D10 standard extends signal generation and delivery for realtime functions requiring asynchronous notification. Currently, asynchronous I/O and timer functions generate signals as an explicit parameter to the asynchronous I/O and timer function calls. When using these functions, you do not have to call a separate function to deliver signals.

Signal delivery for the P1003.4/D10 realtime functions uses a `sigevent` structure. The `sigevent` structure is supplied as an argument (either directly or indirectly) to the function call. The `sigevent` structure is defined in the `<signal.h>` header file and contains the following members:

```
void     *sevt_value;      /* Not currently supported - specify as NULL */
int       sevt_signo;      /* Signal sent on I/O completion             */
                           /* or timer expiration                       */
```

The *sevt_value* member is an application-defined value to be passed to the signal catching function at the time of signal delivery. This member is used in P1003.4/D10 Realtime Signals, which are not fully implemented at this time. Specify a value of NULL for this member.

The *sevt_signo* member specifies the signal number to be sent on completion of the asynchronous I/O operation or on timer expiration. In both instances, you must set up a signal handler to execute once the signal is received. You can use `sigaction` or `signal` function to specify the action required. Refer to Chapter 3 and Chapter 5 for examples of using signals with these functions.

11.2 The Signal Interface

Signal use consists of two actions: sending and receiving. Either the sending process posts a signal to the receiving process, or the kernel can send a signal. Examples of events that send a signal include hardware faults, the `kill` function, or terminal input. The receiving process can respond by allowing the signal to terminate the process, or it can take action such as blocking the signal or invoking a routine to carry out an appropriate action.

Once a signal is sent, it is delivered, unless delivery is blocked. When blocked, the signal is marked pending. Pending signals are delivered immediately once they are unblocked. To determine whether a blocked signal is pending, use the `sigpending` function.

Applications use signals to inform processes of the occurrence of asynchronous events. Processes can send signals to each other using the `kill` functions or the kernel can send signals to processes. Available signals include:

- Signals that prescribe actions to be performed by the receiving process, such as SIGALRM
- Signals that indicate the occurrence of an event, such as SIGCLD
- Signals related to exceptions, such as SIGFPE

Many functions are associated with signals. For example, the `pause`, `wait`, and `waitpid` functions suspend the execution of a process until an appropriate signal arrives. Several functions deal with the signal set itself, such as `sigemptyset`, which creates an empty set of signals, and `sigpending`, which checks whether any blocked signals are currently pending.

For each type of signal, a process can use the sigaction function to declare an associated signal-catching function. Such a function is executed asynchronously when the signal is delivered to the process. The process may also choose to ignore the signal or take a default action when it receives the signal.

When two or more unblocked signals are pending, the kernel delivers the pending unblocked signal with the lowest numeric signal number.

Table 11–1 lists the signal control functions in two categories: those used to establish and manipulate sets of signals and those used to send signals or respond to them.

Table 11–1 Signal Control Functions

Function	Description
	Controlling a Signal Set
sigaction	Examines or specifies the action of a specific signal
sigaddset	Adds a signal to a set of signals
sigdelset	Deletes a signal from an existing set of signals
sigemptyset	Initializes a set of signals by excluding all signal definitions
sigfillset	Initializes a set of signals by including all signal definitions
sigismember	Tests whether a signal is a member of an existing set of signals
sigpending	Stores a set of pending signals in a specified signal set
sigprocmask	Examines or changes the signal mask of the calling process
sigsuspend	Replaces the signal mask of the calling process and then suspends the process

(continued on next page)

Table 11–1 (Cont.) Signal Control Functions

Function	Description
	Sending and Responding to Signals
alarm	Sends the calling process a SIGALRM signal after a specified number of seconds
kill	Sends a signal to a process or a group of processes
nanosleep	Suspends the current process either for a specified period or until a signal of a certain type is delivered
pause	Suspends the calling process until a signal of a certain type is delivered
sleep	Suspends the current process either for a specified period or until a signal of a certain type is delivered
wait	Lets a parent process get status information from a child that has stopped and delays the parent process until a signal arrives or one of its child processes terminates
waitpid	Lets a parent process get status information from a specific child that has stopped and delays the parent process until a signal arrives from that child or that child terminates

11.2.1 Sending Signals

Signals are sent by a user process, the kernel, or a driver program. There are basically four conditions that can generate signals:

- A user-level process sends a signal to another user-level process.

  For example, the call to kill can terminate the process or activate a signal handler to perform some other action.

- A kernel-level process sends a signal to a user-level process.

  For example, the kernel may send a signal to notify a process that hardware conditions prevent further execution.

- A driver program sends a signal to a user-level process.

  For example, the user initiates application control by pressing a Ctrl/C from a terminal.

- A user-level process sends a signal to itself.

  For example, a process needs to track software conditions, such as timer expiration or asynchronous I/O completion.

A process sends a signal to another process (or an entire process group) by using the `kill` system call. The first argument to the `kill` system call is the process ID of the receiving process. The second argument identifies the signal to be sent or indicates that a group of processes is to be signaled.

Signals can be sent from a keyboard. To see which signals are mapped to keys on your keyboard, issue the command `stty everything`. Signals sent from a keyboard are received by all processes in the process group associated with the terminal.

In Example 11–1, a parent process sends a signal to its child, which handles the signal and exits.

Example 11–1 Sending Signals Between Processes

```
/* The parent process sends SIGINT to a child process. */
/* The child process handles the signal and exits.     */

#include <signal.h>
#include <stdio.h>

main()
{
int pid;        /* The child's PID is returned by fork()   */
void SIGINT_handler          /* The signal handler routine */

if ((pid = fork()) == 0)    /* Child process; execute child's code */
     {signal (SIGINT, SIGINT_handler);   /* Make signal handler    */
      pause();                           /* Wait for a signal      */
     }

else                       /* Parent process: executes parent's code */
     {sleep (1);           /* Wait 1 second for child to be born    */
      kill(pid, SIGINT);   /* Send signal to child                  */
      wait (0);            /* Wait until child terminates           */
      exit (0);            /* Successful exit                       */
      }
}

void SIGINT_handler(signal_number)  /* Identify the signal received */
     int signal_number;             /* (SIGINT = 2) and exit       */
     {
     printf("Signal %d received from parent.\n", signal_number);
     exit(0);                       /* Successful exit             */
     }
```

11.2.2 Blocking Signals

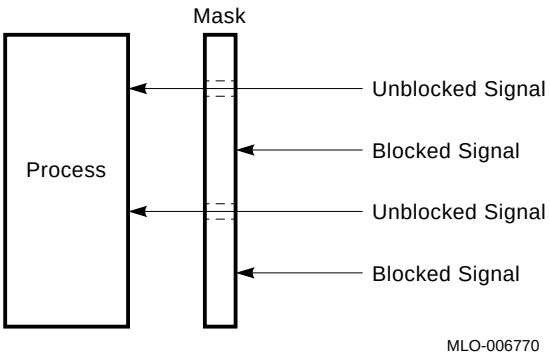
A signal can be blocked to protect certain sections of code from receiving signals when the work should not be interrupted. Unlike ignoring a signal, blocking a signal postpones the signal until the process is ready to handle it.

Each process has a signal mask, a set of bits, each one corresponding to a specific type of signal as defined in the `<signal.h>` header file. Signals are blocked or unblocked through this bit mask. Each bit represents one of the signal conditions — if the *n*th bit in the mask is set, then signal *n* is blocked.

The signal mask is initialized by the parent process and can be manipulated to control signal delivery. Signals are explicitly blocked and unblocked by manipulation of the signal mask. Initially, a process copies the signal mask of its parent. The process can use the first 9 functions listed in Table 11–1 to manipulate and examine its signal mask, thus regulating the signals to which it is sensitive.

Figure 11–1 represents a mask blocking two signals. In this illustration, two signal bits are set, blocking signal delivery for the specified signals.

Figure 11–1 Signal Mask that Blocks Two Signals



A blocked signal is marked as pending when it arrives and is handled as soon as the block is released. Multiple occurrences of the same signal are not saved; that is, if a signal is generated more than once while the signal is already pending, only one instance of the signal is delivered.

A user process can change the signal mask by calling the sigprocmask or sigsuspend functions. The sigprocmask function lets you replace or alter the signal mask of the calling process; the first argument to this function determines the action taken. If you specify the SIG_SETMASK flag as the first argument, you can replace the current signal mask with a new signal mask. The SIG_BLOCK and SIG_UNBLOCK flags allow you to increase or decrease the set of blocked signals.

Use the sigsuspend function to suspend the process until one of the signals is received. The argument to the sigsuspend function specifies the signals used by the signal mask specified in *sigmask* and then suspends the process. The process remains suspended until a signal is delivered that either executes a signal-handling function or terminates the process.

The sigprocmask function is useful when you want to set a mask but are uncertain as to which signals are still blocked. You can retrieve the current signal mask by calling sigprocmask (SIG_BLOCK, NULL, &oldmask). The sigpending function determines which signals are pending but are blocked from delivery. After the critical code is executed, use the sigprocmask or sigsuspend functions to release any blocked signals and restore the old mask, as in the following example:

```
     .
     .
     .
sigset_t newmask, oldmask;

sigemptyset(&newmask);
sigemptyset(&oldmask);
sigaddset(&newset, SIGSYS);
sigaddset(&newset, SIGTRAP);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

      /* Code protected from SYSSYS and SIGTRAP goes here */

      /* Release blocked signals and restore old mask */

sigprocmask(SET_SETMASK, &oldmask, NULL);
     .
     .
     .
```

The sigprocmask function restores the original signal mask and allows the blocked signal to be delivered if one or both signals became pending while the protected code was executing.

The sigaction, sigprocmask, and sigsuspend signal-handling functions take arguments that point to a *sigset_t* type. This type contains information about the signal data objects as they pertain to the application. For example, the signal set could contain either the set of signals blocked from delivery to a process or the set pending for a process. The sigsetops primitive functions let you manipulate the sets of signals defined in the sigset structure. The sigsetops primitive functions also let you initialize the signal set to include or exclude all signals, add or delete individual signals, and contents of the set. See Section 11.2.3.4 for more information on using the sigsetops primitive functions.

11.2.3 Managing Signals

Signals are managed by the sigaction or signal functions. Both functions can take one of three actions for each signal it receives:

- Ignore the signal — Discarded the signal as if it were never sent
- Take the default action — Allow the system to determine the signal action
- Catch the signal — Pass control to a user routine

When the signal is ignored, the process does not receive notification of the signal. Most applications catch the signal and set up user-written signal handlers to take care of the event that triggered the signal. The handler is executed and then passes control back to the process at the point where the signal was received, and execution continues. Handlers can also send error messages, save information about the status of the process when the signal was received, or transfer control to some other point in the application.

Refer to the reference pages for a complete description of the default actions associated with individual signals.

11.2.3.1 Using the sigaction Function

The sigaction function allows the calling process to examine and specify the action to be taken for a signal. If you set a signal-handling action with a call to the sigaction function, the user-specified action remains set until explicitly reset with another call to the sigaction function.

When a signal is caught by a routine established by the sigaction function, a new signal mask is created and used temporarily.

The sigaction function uses a sigaction structure to describe the action taken. This structure is in the <signal.h> header file and contains the following fields;

```
struct sigaction
     {void *sa_handler    /* SIG_DFL, SIG_IGN, or a pointer to a function */
      sigset_t sa_mask    /* Additional set of signals to be blocked      */
      int sa_flags        /* Flags to affect behavior of the signal       */
     };
```

If the action is not specified as NULL, it points to a sigaction structure specifying the action associated with the signal. If the action is specified as NULL, signal handling is unchanged by the call to the sigaction function, but you can use the call to inquire about the current handling of a specified signal. The *sa_handler* field of the sigaction structure identifies the action associated with a specific signal.

If the *sa_handler* field specifies a signal-catching function, the *sa_mask* field identifies the additional set of signals to be added to the process's signal mask before the signal-catching function is called. This signal mask is used for the duration of the process's signal handler or until modified by another call to sigaction, sigprocmask, or sigsuspend function. This new mask is formed by taking the union of the current signal mask and the value of the signal that triggered the call to the signal handler. If the user-specified signal handler is successful, the original mask is restored.

Example 11–2 shows a program that sets an alarm to go off after the number of seconds specified in the command line that invokes the program. The call to the sigaction function establishes the signal handler announce, making the signal handler responsive to the SIGALRM signal. After arming the alarm, the process pauses. When the SIGALRM signal arrives, the signal handler responds, waking the process and printing a message.

Example 11–2 Using the alarm Function

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

main(argc,argv)
int argc;
char **argv;
{
void announce();
struct sigaction action;

if (argc != 2)
     fprintf(stderr,"Usage: %s seconds\n",argv[0]), _exit(1);

     sigemptyset(&action.sa_mask);
     action.sa_flags = 0;

     action.sa_handler        = announce;
     sigaction(SIGALRM, &action, NULL);

     alarm((unsigned) atoi(argv[1]));
     pause();

     puts("main continues after signal handler");
     _exit(0);
}

void announce(signo)
     int signo;
     {
          printf("Received signal %d - Awake after alarm\n", signo);
     }
```

11.2.3.2 Using the signal Function

The signal function is a simple way to manage signals. The signal function takes two arguments. The *sig* argument identifies the signal, such as SIGALRM. The *func* argument specifies what to do with the signal. The *func* argument can be the address of a signal handler function, or the vaues SIG_DFL, SIG_IGN, which are defined in the <signal.h> header file. Calls to the signal function could look like any of these examples:

```
signal (SIGIO, SIG_IGN);          /* Ignore the signal            */
signal (SIGCHLD, SIG_DFL);        /* Accept signal default action */
signal (SIGALRM, myhandler);      /* Call a handler               */
```

If you specify the SIG_DFL flag, the signal's default action is taken. This can be to ignore the signal, stop the process, or to terminate the process.

11.2.3.3 Using Signal Handlers

A routine that is declared to be a signal handler is passed three arguments when the signal it handles is received by the process, but it need not declare or use any of them. The *signal_number* argument is the value of the signal. The *code* argument specifies additional information supplied with some signals. For example, if the signal is SIGFPE (floating point exception), *code* might be one of the following values:

- FLTOVF_FAULT — Specifies floating-point overflow
- FPE_FLTDIV_FAULT — Specifies floating-point divide by 0
- ILL_RESAD_FAULT — Indicates an attempt to access a reserved address space

The *scp* argument points to a sigcontext structure, defined in <signal.h>. This structure stores the process context as it was before the signal was sent in case the context needs to be restored after handling a signal.

Example 11–3 handles the SIGINT signal. First it cleans up the condition that generated the SIGINT signal; then it stops the program.

Example 11–3 Handling Signals

```
/* This program prompts for input in file 'tmp'.    */
/* If interrupted by Ctrl/C, remove 'tmp' and exit. */

#include <stdio.h>
#include <signal.h>

main()
{
FILE *fp;                  /* File pointer to 'tmp'          */
char c;                    /* Character read from terminal   */
void sigint_handler();     /* The SIGINT signal handler      */

if (signal (SIGINT, SIG_IGN) != SIG_IGN)

      /* If SIGINT is already being ignored,     */
      /* Don't declare a handler for it          */

     signal (SIGINT, sigint_handler);

      /* Make sigint_handler handle all SIGINT   */
      /* Signals. signal() blocks other SIGINTs  */
      /* While a SIGINT is being handled.        */
```

(continued on next page)

Example 11–3 (Cont.) Handling Signals

```
fp=fopen("tmp","w");           /* Open file 'tmp' for writing  */
printf("Enter text. \n");      /* Prompt for text              */
while ((c=getchar()) != EOF)   /* Get a char and write it to 'tmp'*/
     putc(c, fp);

puts("EOF typed before CTRL/C");
exit (0);                 /* Successful exit */
}

      /* Remove 'tmp' file, and kill this */
      /* Program. Do not return to main() */

void sigint_handler()
     {if ( unlink("tmp") != -1)
          puts("The tmp file has been removed.");
          exit(1);
     }
```

A signal sent from the keyboard, such as an interrupt (SIGINT), is sent to all processes associated with the terminal. However, the shell turns off interrupts sent to background processes. That is why Example 11–3 calls the `signal` function for SIGINT and tests its value before declaring a handler for SIGINT. If the program, `write_text.c`, declares all SIGINTs are to be handled by its handler, then the shell does not turn off interrupts when the process is running in the background. The `write_text.c` program tests the current state of interrupt handling and continues to ignore interrupts if they are currently being ignored.

Example 11–4 shows the code for a process that creates a child that in turn creates and uses a signal handler, `catchit`. The child process also calls `sigaction` to make `catchit` responsive to the signal SIGUSR1. Then the child process pauses until the signal handler catches the signal and exits.

The parent process sleeps for one second, allowing the child to run. Then the parent:

- Calls `kill` to send the SIGUSR1 signal to the child
- Waits for the child process to terminate

The `catchit` signal handler calls `_exit` to terminate the child process, sending a signal to the parent.

When the parent receives the process termination signal from the child, it prints a message and stops.

Example 11–4 Sending a Signal to Another Process

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

main()
{
pid_t pid;

if ((pid = fork()) == 0) {        /* Child */
struct sigaction action;
void catchit();

sigemptyset(&action.sa_mask);
action.sa_flags = 0;

action.sa_handler = catchit;
if (sigaction(SIGUSR1, &action, NULL) == -1)
     perror("sigusr: sigaction"), _exit(1);
     pause();     /* Never get here */
     }

else {                /* Parent */
     int stat;
     sleep(1);        /* Allow child to run */
     kill(pid, SIGUSR1);
     pid = wait(&stat);
     printf("Child exit status = %d\n", WEXITSTATUS(stat));
      _exit(0);
     }
}

void catchit(signo)
      int signo;
      {printf("Signal %d received from parent\n", signo);
       _exit(17);
      }
```

11.2.3.4 Using the sigsetops Primitives

The sigsetops primitives are used to manipulate signal sets that are blocked from delivery or the set of signals pending for a process. Primitives operate on data objects addressable by the application rather than the set of signals known to the system. Table 11–2 lists the sigsetops primitive functions.

Table 11–2 The sigsetops Primitive Functions

Primitive	Description
sigaddset	Adds the specified signal to the signal set
sigdelset	Deletes the specified signal from the signal set
sigemptyset	Initializes the signal set to exclude all signals given in POSIX 1003.1
sigfillset	Initializes the signal set to include all signals given in POSIX 1003.1
sigismember	Tests whether the specified signal is a member of the signal set

Before calling the sigaddset, sigdelset, or sigismember primitives, you should initialize the signal set with a call to either the sigemptyset or sigfillset primitive. See the reference pages for additional information about using the sigsetops primitives.

A

DEC OSF/1 Realtime Functional Summary

Table A–1 summarizes the functions that are of particular interest to realtime application developers. The source of these functions ranges from System V to POSIX 1003.1 and P1003.4/D10. This table are not exhaustive, but may serve as a guide in application development. Use the `man` command to get complete reference information concerning these functions.

The functions are arranged by category.

Table A–1 Summary of Functions

Function	Purpose
Process Control	
`alarm`	Sends the calling process a SIGALRM signal after a specified number of seconds
`exit`	Terminates the calling process
`exec`	Runs a new image, replacing the current running image
`fork`	Creates a new process
`getenv`	Reads an environment list
`isatty`	Verifies whether a file descriptor is associated with a terminal
`kill`	Sends a signal to a process or a group of processes
`malloc`	Allocates memory
`pause`	Suspends the calling process until a signal of a certain type is delivered
`sleep`	Suspends the current process either for a specified period or until a signal of a certain class is delivered

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
Process Control	
sysconf	Gets the current value of a configurable system limit or option
uname	Returns information about the current state of the operating system
wait	Lets a parent process get status information for a child that has stopped, and delays the parent process until a signal arrives
waitpid	Lets a parent process get status information for a specific child that has stopped and delays the parent process until a signal arrives from that child or that child terminates
P1003.4/D10 Priority Scheduling	
sched_getscheduler	Returns the scheduling policy of a specified process
sched_get_priority_max	Returns the maximum priority allowed for a scheduling policy
sched_get_priority_min	Returns the minimum priority allowed for a scheduling policy
sched_get_rr_interval	Returns the interval time limit allowed for the round-robin scheduling policy
sched_get_sched_param	Returns the scheduling priority of a specified process
sched_setscheduler	Sets the scheduling policy and priority of a specified process
sched_set_sched_param	Sets the scheduling priority of a specified process
sched_yield	Yields execution to another process

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
P1003.4/D10 Clocks	
clock_gettime	Returns the current value for the specified clock
clock_gettimedrift	Returns the value of the clock drift rate as set by the most recent call to clock_settimedrift
clock_settime	Sets the specified clock to the specified value
clock_settimedrift	Sets the drift rate for the specified clock, in parts per billion (nanoseconds), to the specified value
P1003.4/D10 Timing Facility Resolution	
clock_getres	Returns the resolution and maximum value of the specified clock
nanosleep_getres	Returns the resolution and maximum value supported by nanosleep
timer_getres	Returns the resolution and maximum value of an absolute or relative timer

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
Date and Time Conversion	
asctime	Converts a broken-down time into a 26-character string
ctime	Converts a time in seconds since the Epoch to an ASCII string in the form generated by asctime
difftime	Computes the difference between two calendar times (time1–time0) and returns the difference expressed in seconds
gmtime	Converts a calendar time into a broken-down time, expressed as GMT
localtime	Converts a time in seconds since the Epoch into a broken-down time
mktime	Converts the broken-down local time in the tm structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by time
tzset	Sets the external variable *tzname*, which contains current timezone names
P1003.4/D10 Timers	
nanosleep	Causes the calling process to suspend execution for a specified period of time
timer_create	Returns a unique timer ID used in subsequent calls to identify a timer based on the systemwide clock
timer_delete	Removes a previously allocated, specified timer
timer_gettime	Returns the amount of time before the specified timer is due to expire and the repetition value
timer_settime	Sets the value of the specified timer either to an offset from the current clock setting or to an absolute value

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
BSD Clocks and Timers	
getitimer	Returns the amount of time before the timer expires and the repetition value
gettimeofday	Get the time of day
setitimer	Sets the value of the specified timer
settimeofday	Set the time of day
P1003.4/D10 Memory Locking	
mlock	Locks a specified region of a process's address space
mlockall	Locks a process's address space
munlock	Unlocks a specified region of a process's address space
munlockall	Unlocks a process's address space
System V Memory Locking	
plock	Locks and unlocks a process, text, or data in memory
P1003.4/D10 Asynchronous I/O	
aio_cancel	Cancels one or more requests pending against the file descriptor
aio_error	Returns the error status of a specified operation
aio_read	Queues a read request on the specified file descriptor
aio_return	Returns the status of an operation
aio_suspend	Suspends the calling process until at least one of the specified requests has completed
aio_write	Queues a write request to the specified file descriptor
lio_listio	Initiates a list of requests

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
BSD Synchronous I/O	
`fcntl`	Performs controlling operations on the specified open file
`fsync`	Writes changes to a file to permanent storage—saves all modified data
`sync`	Update all file systems—all information in memory that should be on disk is written out
System V Messages	
`msgctl`	Provides control for message operations and has options to return and set message descriptor parameters and remove the descriptor
`msgget`	Creates or returns a message queue identifier for use in other message functions
`msgrcv`	Reads a message from the queue associated with the message queue ID and places it in a user-defined structure
`msgsnd`	Sends a message to the queue associated with the message queue ID
System V Shared Memory	
`shmat`	Attaches the shared memory segment to the data segment of the calling process
`shmctl`	Provides shared memory control operations
`shmdt`	Detaches the shared memory from the data segment of the calling process
`shmget`	Returns the shared memory identifier

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
System V Semaphores	
semctl	Provides semaphore control operations
semget	Returns the semaphore identifier
semop	Performs an array of semaphore operations on the set of semaphores associated with the semaphore
POSIX Signal Control	
sigaction	Examines or specifies the action of a specific signal
signal	Changes the action of a signal
sigpending	Stores a set of pending signals in a specified space
sigprocmask	Examines or changes the signal mask of the calling process
sigsetops	Manipulates signal sets
sigsuspend	Replaces the signal mask of the calling process and then suspends the process
sigsetops Primitives	
sigaddset	Adds the specified signal to the signal set
sigdelset	Deletes the specified signal from the signal set
sigemptyset	Initializes the signal set to exclude all signals given in POSIX 1003.1
sigfillset	Initializes the signal set to include all signals given in POSIX 1003.1
sigismember	Tests if the specified signal is a member of the signal set

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
Process Ownership	
`geteuid`	Returns the effective user ID of the calling process
`getegid`	Returns the effective group ID of the calling process
`getgid`	Returns the real group ID of the calling process
`getpgrp`	Returns the process group ID of the calling process
`getpid`	Returns the process ID of the calling process
`getppid`	Returns the process ID of the parent of the calling process
`getuid`	Returns the real user ID of the calling process
`setgid`	Sets the group ID of the calling process
`setsid`	Creates a new session, for which the calling process is the session leader
`setuid`	Sets the user ID of the calling process
Input and Output	
`close`	Closes a file
`dup`	Duplicates a file descriptor
`dup2`	Duplicates a file descriptor
`fileno`	Retrieves a file descriptor
`lseek`	Moves a pointer to a record within a file
`mkfifo`	Creates fifo special files
`open`	Opens a file
`pipe`	Creates an interprocess channel
`read`	Reads the specified number of bytes from a file
`write`	Writes the specified number of bytes to a file

(continued on next page)

Table A–1 (Cont.) Summary of Functions

Function	Purpose
Device Control	
cfgetispeed	Retrieves the input baud rate for a terminal
cfgetospeed	Retrieves the output baud rate for a terminal
cfsetispeed	Sets the input baud rate for a terminal
cfsetospeed	Sets the output baud rate for a terminal
isatty	Verifies whether a file descriptor is associated with a terminal
tcdrain	Causes a process to wait until all output has been transmitted
tcflow	Suspends or restarts the transmission or reception of data
tcflush	Discards data that is waiting to be transmitted
tcgetattr	Retrieves information on the state of a terminal
tcsendbreak	Sends a break character for a specified amount of time
tcsetattr	Applies a set of attributes to a terminal
System Database	
getgrgid	Returns group information when passed a group ID
getgrnam	Returns group information when passed a group name
getpwnam	Returns user information when passed a user name
getpwuid	Returns user information when passed a user ID

Index

A

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Y