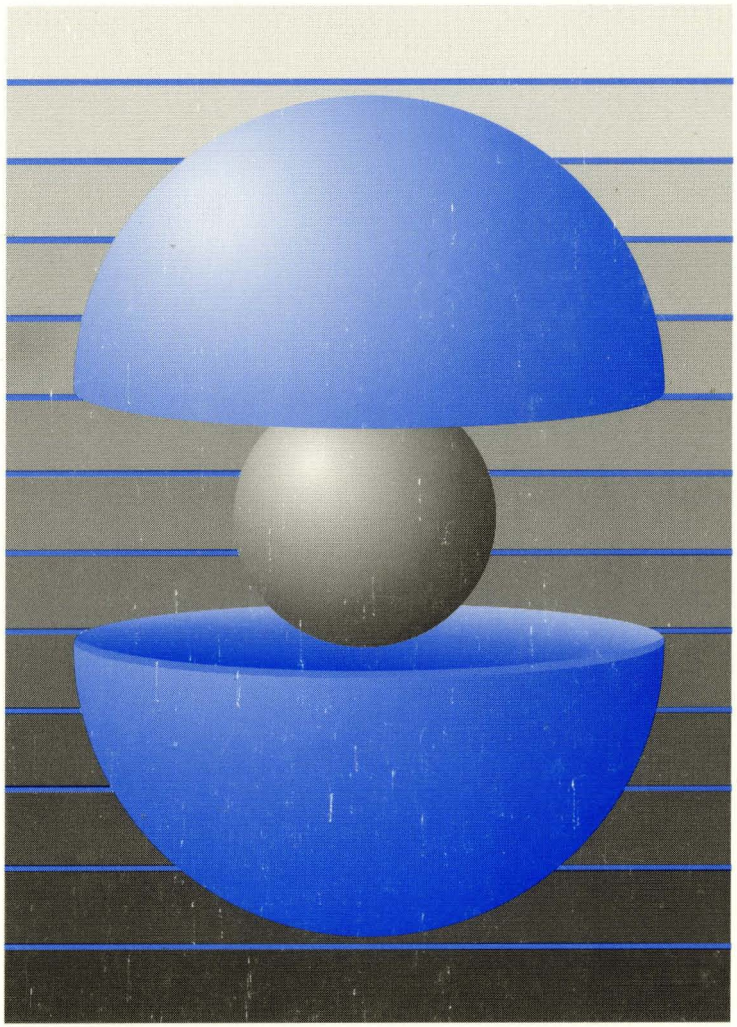
DEC OSF/1

digital

Guide to Realtime Programming



Part Number: AA-PS33B-TE

DEC OSF/1

Guide to Realtime Programming

Order Number: AA–PS33B–TE

February 1994

This guide describes how to use POSIX 1003.4 Draft 11 (P1003.4/D11) functions to write realtime applications that run on DEC OSF/1 systems. This guide is intended for experienced application programmers.

Product and Version: DEC OSF/1 Version 2.0 or higher

Digital Equipment Corporation
Maynard, Massachusetts

Revised, February 1994

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

S1857

This document is available on CD–ROM

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

2 Process Scheduling and Priorities

3 Shared Memory

4 Memory Locking

5 Signals

6 Clocks and Timers

7 Asynchronous Input and Output

8 Binary Semaphores

9 Messages

10 Pipes

A Summary of Differences Between P1003.4/D10 and P1003.4/D11

B DEC OSF/1 Realtime Functional Summary

Index

Examples

Figures

Tables

About This Guide

This guide is designed for programmers who are using systems running DEC OSF/1 with the realtime kernel. 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 11 (P1003.4/D11) 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 realtime kernel for the DEC OSF/1 operating system.
- Chapter 2, Process Scheduling and Priorities, describes use of the P1003.4/D11 functions to determine and set priority for processes in your application. This chapter also describes the priority scheduling policies provided by the realtime kernel for the DEC OSF/1 operating system.

- Chapter 3, Shared Memory, describes the creation and use of P1003.4/D11 shared memory for interprocess communication.
- Chapter 4, Memory Locking, describes the use of P1003.4/D11 functions for locking and unlocking memory.
- Chapter 5, Signals, describes the creation and use of signals for interprocess communication.
- Chapter 6, Clocks and Timers, describes use of the P1003.4/D11 functions for constructing and using high-resolution clocks and timers.
- Chapter 7, Asynchronous Input and Output, describes the use of P1003.4/D11 functions for asynchronous input and output.
- Chapter 8, Binary Semaphores, describes the creation and use of P1003.4/D11 semaphores for interprocess synchronization. An example illustrates how to use semaphores and shared memory in combination.
- Chapter 9, Messages, describes the creation and use of System V message queues for interprocess communication and synchronization in realtime applications.
- Chapter 10, Pipes, describes the creation and use of pipes and named pipes for interprocess communication.
- Appendix A, Summary of Differences Between P1003.4/D10 and P1003.4 /D11, lists the differences between POSIX 1003.4 Drafts 10 and 11.
- Appendix B, 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/D11 functions, use the `man` or `whatis` commands.

The printed version of the DEC OSF/1 documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General Users	G	Teal
System Administrators	S	Red
Network Administrators	N	Yellow
Programmers	P	Blue
Reference Page Users	R	Black

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the books in the DEC OSF/1 documentation set.

Using the man Command

System commands and library functions (including P1003.4/D11 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/D11 functions discussed in this manual. If you need help in 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, function, 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.
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.

Reader's Comments

Digital welcomes your comments on this or any other DEC OSF/1 manual. You can send your comments in the following ways:

- Internet electronic mail: `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32

- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of each printed manual.

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

1

Introduction to Realtime Programming

A realtime application is one in which the correctness of the application depends on the timeliness and predictability of the application as well as the results of computations. To assist the realtime application designer in meeting these goals, the DEC OSF/1 realtime kernel provides features that facilitate efficient interprocess communication and synchronization, a fast interrupt response time, asynchronous input and output (I/O), memory management functions, and facilities for handling timing constraints.

Realtime applications are becoming increasingly important in our daily lives and can be found in diverse environments such as the automatic braking system on an automobile, a lottery ticket system, or robotic environmental samplers on a space station. The use of realtime programming techniques is rapidly becoming a common means for improving the predicability of our technology.

This chapter includes the following sections:

- Realtime Overview, Section 1.1
- DEC OSF/1 Realtime System Capabilities, Section 1.2
- Process Synchronization, Section 1.3
- POSIX Standards, Section 1.4
- Installing the Realtime Kernel, Section 1.5
- Building Realtime Applications, Section 1.6

1.1 Realtime Overview

Realtime applications provide an action or an answer to an external event in a timely and predictable manner. While many realtime applications require high-speed compute power, realtime applications cover a wide range of tasks with differing time dependencies. "Timeliness" has different definitions in each realtime 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, the success of both applications depends on well-defined time requirements.

The concept of "predictability" has many connotations, but for realtime applications it generally means that a task or set of tasks can always be completed within a predetermined amount of time. Depending on the situation, 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.

To have control over the predictability of an application, the programmer must understand which time bounds are significant. For example, an understanding of the *average* time it takes for a context switch does not guarantee task completion within a predictable timeframe. The realtime programmer must know the *worst-case* time requirements so that he can design an application that will always meet worst-case deadlines.

Realtime systems also use techniques to reduce the numbers associated with a worst-case scenario. In some situations, a worst-case realtime deadline may be significantly faster than the non-realtime, average time.

Realtime applications can be classified as either hard or soft realtime. Hard realtime applications require a response to events within a predetermined amount of time for the application to function properly. If a hard realtime application fails to meet specified deadlines, the application fails. While many hard realtime applications require high-speed responses, the granularity of the timing is not the central issue in a hard realtime application. An example of a hard realtime application is a missile guidance control system where a late response to a needed correction leads to disaster.

Soft realtime applications do not fail if a deadline is missed. Some soft realtime applications can process large amounts of data or require a very fast response time, but the key issue is whether or not meeting timing constraints is a condition for success. An example of a soft realtime application is an airline reservation system where an occasional delay is tolerable.

Many 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 a 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 the acquisition of 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 about 1 second while simultaneously handling data from, perhaps, ten external sources.

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

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.

Realtime applications are usually characterized by a blend of requirements. Some portions of the application may consist of hard, critical tasks (all of which must meet their deadlines). Other parts of the application may require heavy data throughput. Many parts of a realtime application can easily run at a lower priority and require no special realtime functionality. The key to a successful realtime application is the developer's ability to accurately define application requirements at every point in the program. Resource allocation and realtime priorities are used only when necessary so that the application is not overdesigned.

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
- Reliable, asynchronous signals
- 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 facilities, as summarized in Section 1.3.

1.2.1 The Value of a Preemptive Kernel

The responsiveness of the operating system to asynchronous events is a critical element of realtime systems. Realtime systems must be capable of meeting the demands of hard realtime tasks with tight deadlines. To do this, the operating system's reaction time has to be short and the scheduling algorithm must be simple and efficient.

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, the worst-case scenario.

Every application interacts with the operating system in two modes: user mode and kernel mode. User-mode processes call utilities, library functions, and other user applications. A process running in user mode can be preempted by a higher-priority process. During execution, a user-mode process often makes system calls, switching the context from user to kernel mode where the process interacts with the operating system. Under the traditional timesharing scheduling algorithm, a process running in kernel mode cannot be preempted.

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, and the higher-priority process displaces the running, lower-priority process.

1.2.1.1 Nonpreemptive Kernel

The standard UNIX kernel is a nonpreemptive kernel; it 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 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 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, kernel-mode 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 quickly to a process preemption request. When a realtime user process engages one of the fixed-priority scheduling policies, 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

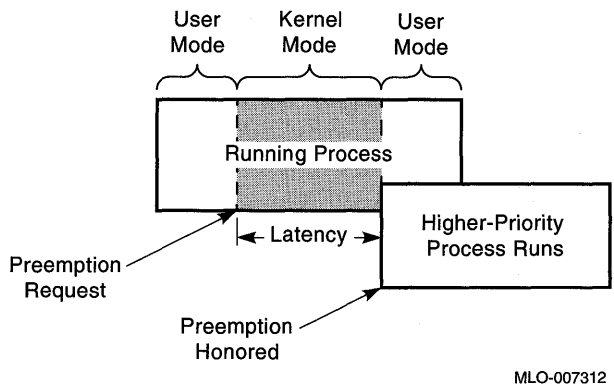


Figure 1–1 shows the expected latency of a nonpreemptive kernel. In this situation, the currently running process moves back and forth between user and kernel mode as it executes. The higher-priority, realtime process advances to the beginning of the priority process list, but cannot preempt the running 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.

Figure 1–2 Preemptive Kernel

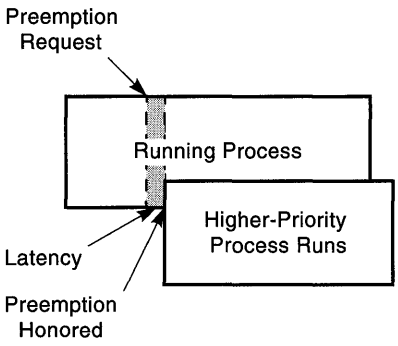


Figure 1–2 shows the expected latency of a preemptive kernel. In this situation the running process is quickly preempted and the higher-priority, realtime process takes its place on the run queue. With a preemptive kernel, latency is minimized and can be measured in terms of milliseconds.

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 a base 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 (first-in, first-out and round robin), 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. Figure 2–5 illustrates the priority ranges for both the `nice` and realtime scheduling interfaces.

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/D11 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. The user can raise the priority of a running process to avoid its 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. At the end of the quantum, a process of equal priority will take its place at the end of the run queue. Processes at that priority continue to execute as long as the waiting processes are 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

Realtime timers often schedule tasks and events in time increments considerably smaller than the traditional one-second timeframe. Because the system-wide 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-millisecond 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 use timers to coordinate and monitor the correctness of a realtime application. Some applications may require only one per-process timer; others may require multiple timers. Each timer is created and armed independently, which means that the application designer controls the action of each and every timer.

The DEC OSF/1 systemwide clock provides the timing base for per-process timers and is the source for timer synchronization. This clock maintains user and system time as well as the current time and date. The resolution of the 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, retrieve and correct for clock drift rate, suspend execution for a period of time, provide high-resolution timers, and use asynchronous signal notification.

For information on using the clock and timer functions, refer to Chapter 6.

1.2.4 Memory Locking

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 caused by paging would introduce involuntary and unpredictable time delays at runtime.

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. 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.

The P1003.4/D11 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

With synchronous I/O the process waits (is blocked) for the I/O operations to complete before continuing execution. DEC OSF/1 asynchronous I/O allows the calling process to resume execution immediately 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. The process immediately continues execution, thus overlapping operations.

Often, one process simultaneously performs multiple I/O functions while other processes continue execution. For example, an application may 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.

For information on using the asynchronous I/O functions, refer to Chapter 7.

1.2.6 Interprocess Communication

Interprocess communication is the exchange of information between two or more processes. In single-process programming, modules within a single process communicate by using global variables and function calls with data passing between the functions and the callers. In multiprocess programming with images running in separate address space, you need to have additional communication mechanisms for passing data.

DEC OSF/1 interprocess communication facilities allow the realtime application designer to synchronize 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:

- Shared memory, Chapter 3

  Shared memory is the fastest form of interprocess communication. As soon as one process writes data to the shared memory area, it is available to other processes using the same shared memory. DEC OSF/1 supports P1003.4/D11 shared memory.

- Signals, Chapter 5

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

- Semaphores, Chapter 8

  Semaphores are most commonly used to control access to system resources, such as shared memory regions. DEC OSF/1 supports P1003.4/D11 semaphores.

- Messages, Chapter 9

  Messages consist of user-defined structures that specify the length and type of message as well as carry the message text. DEC OSF/1 supports System V message queues.

- Pipes, Chapter 10

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

  Named pipes are like pipes, except that named pipes use file descriptors.

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.

To provide rapid signal communication on timer expiration and asynchronous I/O completion, these functions send signals through a common data structure. It is not necessary to call signal functions.

For information on using asynchronous signals for interprocess communication, refer to Chapter 5.

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 a "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 process purchases simultaneously 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 facilities 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 allows a process 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, which enforces exclusive access to the shared resource. Exclusive access means that only one process can access the resource at a time. 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 that 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 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 3 and Chapter 8.

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 introduced 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. For example, a parent process can wait for a child process or thread to terminate. The parent process creates a child process which 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:

1. The parent process creates the child process.
2. The parent process synchronizes with the child process.
3. The child process executes until it terminates.
4. The termination of the child process signals the parent process.

5. The parent process resumes execution.

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 signals and semaphores while the child process executes.

For information on using signals and semaphores and signals, refer to Chapter 5 and Chapter 8.

1.3.5 Realtime Needs and System Features

Table 1–1 summarizes the common realtime needs and the features or capabilities available through the P1003.4/D11 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 higher throughput or special purpose I/O	Use asynchronous I/O
Share data between processes	Use shared memory
	Use memory-mapped files
Synchronize access to resources shared between cooperating processes	Use binary (POSIX) or counting (System V) semaphores
Communicate between processes	Use messages, semaphores, shared memory, signals, pipes, and named 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

(continued on next page)

Table 1–1 (Cont.) Realtime Needs Summary

Realtime Need	Realtime Feature
	Use semaphores
	Cause the process to sleep and to awaken when needed

1.4 POSIX Standards

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.

The POSIX standards and draft 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.

POSIX is a set of standards generated and maintained by standards organizations — they are developed and approved by the Institute of Electrical and Electronics Engineers, Inc. (IEEE) and adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). Digital's POSIX implementations follow the standards and drafts defined by the POSIX standards.

The only formal standards to date are POSIX 1003.1 for basic system interfaces and POSIX 1003.13, the test assertions a vendor must test to claim conformance to POSIX 1003.1. Draft standards, such as the realtime extensions (1003.4/D11), are not formal standards. They are working documents that will evolve over time into formal standards.

POSIX standards for the programming interface (Standard 1003.1), POSIX threads (P1003.4a/D4), and realtime programming extensions (P1003.4/D11) are supported by DEC OSF/1.

POSIX 1003.1 defines the standard for basic system services on an operating system, and describes how system services can be used by POSIX applications. These services allow an application to perform operations such as process creation and execution, file system access, and I/O device management.

POSIX 1003.4a (draft standard) defines a set of thread functions that can be used in the design and creation of multithreaded realtime applications in the DEC OSF/1 environment. For this version of DEC OSF/1, Draft 4 of the proposed POSIX 1003.4a standard (P1003.4a/D4) was used.

POSIX 1003.4 (draft standard) provides support for functions that support the needs of realtime applications, such as enhanced interprocess communication, scheduling and memory management control, and asynchronous I/O operations. For this version of DEC OSF/1, Draft 11 of the proposed POSIX 1003.4 standard (P1003.4/D11) was used.

As Digital adds support for evolving and final standards, customers should modify their POSIX applications to conform to the latest version of these standards. Because draft standards are working documents and not formal standards, the level of backwards compatibility and formal support for older versions (drafts) will be less than that normally expected from a stable Digital product.

It is important to note that while the POSIX standards are evolving, DEC OSF/1 realtime will provide backwards compatibility for only one draft of the 1003.4 standard. The current version of DEC OSF/1 provides support for both P1003.4/D10 and P1003.4/D11. The next release of DEC OSF/1 will no longer support P1003.4/D10.

An application that strictly conforms to any combination of these standards and drafts can be developed on one system and then ported to another system that supports the same POSIX standards or drafts. (A strictly conforming application uses only the facilities within the applicable standards.) Similarly, an application developed on a non-Digital platform, if it strictly conforms to the POSIX standards and drafts supported by Digital systems, can be ported and run on a Digital system on which the POSIX software is installed.

It is the source code of an application that is portable. Most applications written for a POSIX environment use the C programming language. Each system that supports a POSIX environment includes POSIX runtime libraries as well as C runtime libraries. A portable application that requires an executable image must be compiled and linked on a system after being ported. It is important that you compile and link your POSIX applications against the runtime libraries on the system where they will be run.

The POSIX standards are based on the UNIX environment. However, POSIX specifies an interface to an operating system, not the operating system itself. 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.

1.5 Installing the Realtime Kernel

The files needed to build the realtime kernel are included with the base system software and are installed as an optional software subset (Kernel Build Environment) during an Advanced Installation.

After installing the realtime subsets, you must configure and reboot the system. See the *Guide to Installing DEC OSF/1* for complete installation instructions.

To check whether your system is running the realtime kernel, enter the following command:

```
% /etc/motd
DEC OSF/1 [RT] V1.2; Wed Mar 3 14:26:18 EST 1993
DEC OSF/1 V1.2 Worksystem Software
```

The `[RT]` in the first line indicates that your system is running the realtime kernel.

If your realtime application errors include ENOSYS, check whether the realtime kernel is installed.

1.6 Building Realtime Applications

To build a DEC OSF/1 realtime application you must first define the POSIX environment, then compile the application with the appropriate compile command switches. These steps draw POSIX header information and realtime libraries into your code.

1.6.1 Defining the POSIX Environment

Realtime applications should include the `unistd.h` header file before any other header files are included in the application. This header file defines the standard macros, _POSIX_4SOURCE, _POSIX_SOURCE, etc., that are required to compile programs containing POSIX 1003.4 functions. If, for some reason, you need to exclude any of the standards definitions provided by the `unistd.h` header file, you can explicitly define those standards macros in the source file or on the compilation command line.

The following example shows the code you would include as the first line of code in either your local header file or your application code:

```
#include <unistd.h>
```

Because the `unistd.h` header file defines all the standards needed for realtime applications, it is important that this `#include` is the first line of code in your application.

In addition to all the other symbols used by realtime applications, the symbols for both _POSIX_4SOURCE and _POSIX_SOURCE are included in the unistd.h header file. If you defined _POSIX_4SOURCE (#define _POSIX_4SOURCE) elsewhere in your application, you would get *only* the definitions for POSIX 1003.4 Draft 11 and other definitions pulled in by that definition, such as, POSIX 1003.1. Therefore, use specific definitions *only* if your application must *exclude* certain definitions related to other unneeded standards, such as XPG3.

1.6.2 Compiling Realtime Applications

You must explicitly load the required realtime runtime libraries when you compile realtime applications. The -l switch forces the linker to include the specified library and the -L switch indicates the search path the linker uses to locate the libraries.

To find the realtime library, the ld linker expands the command specification by replacing the -l with lib and adding the specified library 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.

Note that you cannot use shared libraries with the realtime functions. Make sure that you also specify -non_shared on the compile (or link) command.

The following example specifies that realtime archive library, librt.a, is to be included from the /usr/ccs/lib directory.

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

When you compile an application that uses asynchronous I/O, you must also include libraries used by pthreads; libpthreads.a, libmach.a, and libc_r.a. The following example shows the specification required if your application uses asynchronous I/O.

```
# cc -non_shared myprogram.c -L/usr/ccs/lib \
  -laio -lpthreads -lmach -lc_r
```

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

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.

If, for some reason, you want to just link your realtime application, you must explicitly 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 after librt.a. For example, you would link an application with the realtime library, librt.a, as follows:

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

For a limited amount of time, you will be able to run applications containing P1003.4/D10 functions on a DEC OSF/1 Version 1.2 system. Source compatibility between the two versions exists for realtime scheduling, clocks and timers, asynchronous I/O, memory-locking, and signal operations if the symbol for POSIX 1003.4/D10 is defined. You can define the POSIX_4D10 symbol on the command line as follows:

```
% cc -non_shared myprogram -DPOSIX_4D10
```

If your application fails to compile, you may need to check your programming environment to make sure that the realtime options are installed on your system. The lack of the realtime software 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/D11 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 run time, 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.

The DEC OSF/1 scheduler schedules threads rather than traditional UNIX processes. A UNIX process is identified with a single thread, the first thread in the process. In the following discussion of scheduling, the term "process" represents the first thread. In multithreaded applications (using POSIX threads, `pthreads`), all threads are treated equally by the scheduler. In fact, throughout the system, the highest-priority runnable thread is selected to run. Section 2.1.4 discusses the `pthread` package.

2.1.1 Process States

At run time, 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 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.

(continued on next page)

Table 2–1 (Cont.) Process States

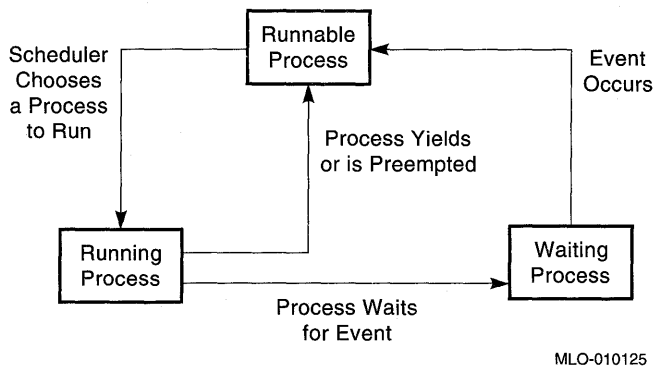
State	Description
Waiting	The process has given up eligibility to run until a condition or set of conditions is satisfied. A process may be waiting for a signal from another process, a wakeup call, a timer expiration, I/O completion, or any number of other events to occur.

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 user processes are locked into memory, pages may be paged out. The user process then must wait for pages to be paged back into memory, thus causing some latency in program execution. To guard against unwanted paging, realtime applications should use the P1003.4/D11 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

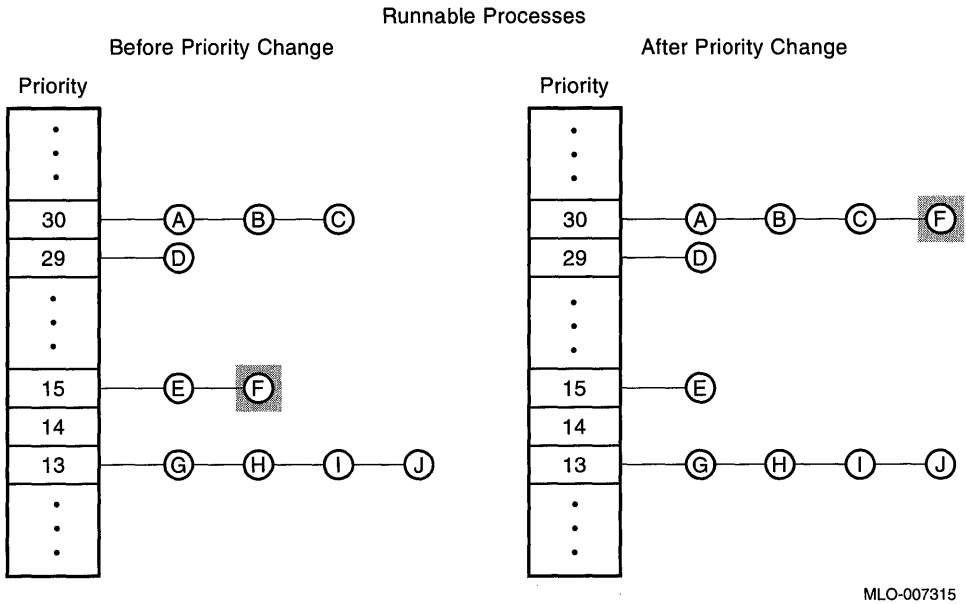
The primary function of the scheduler is to make scheduling decisions for the kernel. The scheduler makes certain that the highest-priority runnable process executes. The scheduler also maintains the kernel's scheduling database, representing the state of the system, in a consistent and accurate state. 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 in the 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 its 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 no more processes remain 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.
- A quantum expires for a running process.

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 running under the SCHED_RR scheduling policy move in and out of the running state.

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.

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/D11 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, (SCHED_OTHER)
- Supports priorities in the 20 through –20 range
- Uses a default priority of 0

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]	—Ⓗ—Ⓘ
A is a higher priority, becomes runnable, and preempts G	G preempted - goes to the beginning of the queue		
		[A]	—Ⓖ—Ⓗ—Ⓘ
A yields or enters waiting state	G runs again to finish its quantum		
		[G]	—Ⓗ—Ⓘ
G finishes its quantum	G goes to the end of the queue H moves to running		
		[H]	—Ⓘ—Ⓖ
A is a higher priority, becomes runnable, and preempts H	H preempted - goes to the beginning of the queue		
		[A]	—Ⓗ—Ⓘ—Ⓖ
A raises priority of K	K changes priority K goes to the end of the queue		
		[A]	—Ⓗ—Ⓘ—Ⓖ—Ⓚ

MLO-007316

- 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 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/D11 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/D11 functions, `sched_setparam` 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/D11 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.

2.1.4 Threads in Realtime Scheduling

The DEC OSF/1 realtime kernel supports the pthread interface, which is based on POSIX 1003.4a, Draft 4. While the standard is still in draft form, threads are useful in driving slow devices such as disks, networks, terminals, and printers. A multithreaded application can perform other useful work while waiting for the device to produce its next event.

To start a thread, you create it using the `pthread_create` routine. A thread is a single, sequential flow within an application. It is the active execution of a designated routine, including any nested invocations. A thread object defines and controls the executing thread. Each thread has its own thread identifier, scheduling policy and priority, thread-specific data values, and the required system resources to support a flow of control (stack size).

By default, a new thread's scheduling policy and priority are inherited from the creating thread (whether the result of a call to the `fork` function or a call to the `pthread_create` routine.) To create a thread that uses different attributes, first disable the inherit scheduling attribute using the `pthread_attr_setinheritsched` routine. Then, set the scheduling attributes individually using the `pthread_attr_setsched` and `pthread_attr_setprio` or `pthread_setsched` and `pthread_setprio`.

Threads are created in the ready state and therefore might begin executing immediately. The newly created thread will begin running before the `pthread_create` completes if the new thread follows either the SCHED_FIFO or SCHED_RR scheduling policy or has a priority higher than the creating thread, or both.

For additional information on using threads, refer to the *Guide to DECthreads*.

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/D11 realtime scheduling functions and select an appropriate scheduling policy for your process. DEC OSF/1 P1003.4/D11 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/D11 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 priority 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 equal or 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, 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_setparam` 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 voluntarily yields or is preempted by a higher-priority process.

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.

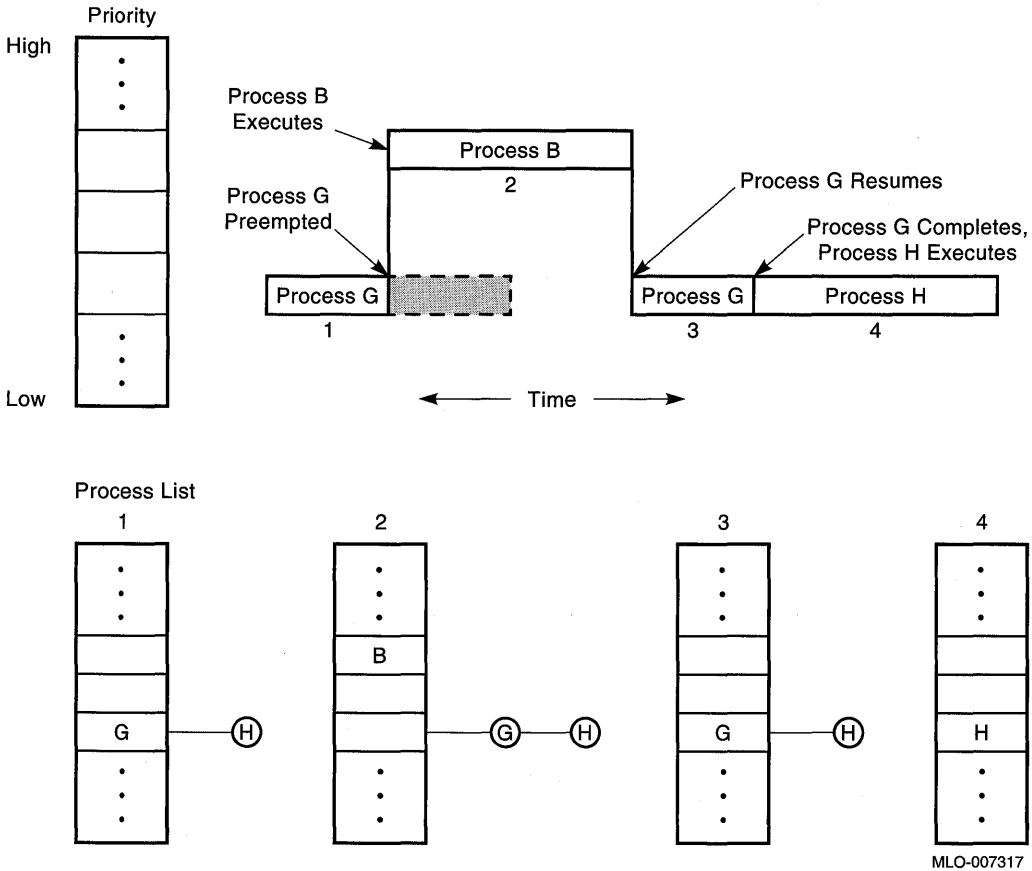
2.2.3.2 Round-Robin Scheduling

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.

Figure 2–4 Preemption—Finishing a Quantum



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 the quantum used in round-robin scheduling. If a process, running under the round-robin scheduling policy, runs without blocking or yielding for more than this amount of time, it may be preempted by another runnable process at the same priority.

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/D11 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/D11 scheduling policies. Realtime priorities can be managed only by using the associated P1003.4/D11 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 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/D11 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: the highest range is reserved for realtime, the middle range is used by the operating system, and the low range is 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/D11 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

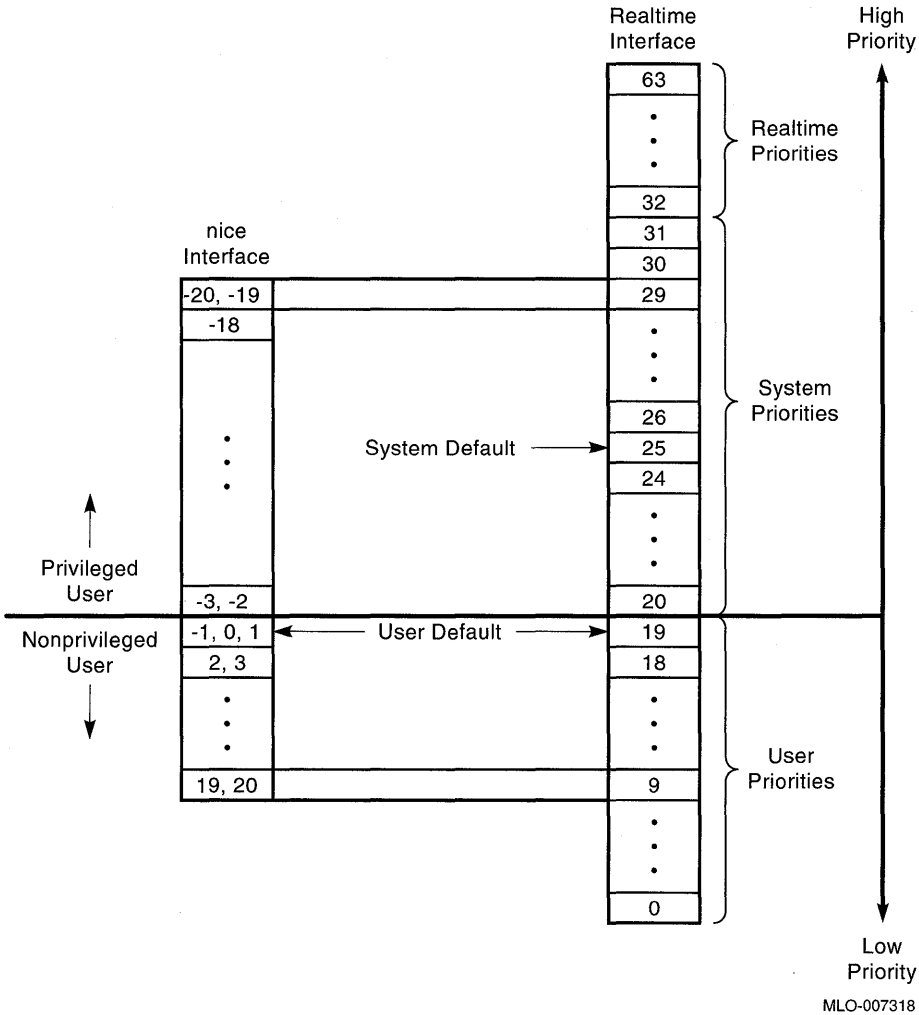
Range	Priority Level
Nonprivileged user	SCHED_PRIO_USER_MIN through SCHED_PRIO_USER_MAX
System	SCHED_PRIO_SYSTEM_MIN through SCHED_PRIO_SYSTEM_MAX
Realtime	SCHED_PRIO_RT_MIN through SCHED_PRIO_RT_MAX

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.

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



Note that hardware interrupts are unaffected by process priorities, even the highest realtime priority.

2.3.3 Displaying Realtime Priorities

The ps command displays current process status and can be used to give realtime users snapshots of process priorites. Realtime users can use POSIX realtime functions to change process priority. Therefore, the ps command is a useful tool for determining if realtime processes are running at the expected priority.

The ps command captures the states of processes, but the time required to capture and display the data from the ps command may result in some minor discrepencies.

Priorities used in the realtime scheduling interface are displayed when you use the specifier *psxpri* in conjunction with the *-o* or *-O* switch on the ps command. Fields in the output format include the process ID (*pid*), control terminal of the process (*tname*), user and system CPU time used by the process (*cputime*), the state of the process (*state*), and an indication that the ps command that is running (*comm*).

The following example requests information regarding processes, with or without terminals, displays timesharing and POSIX priorities, as well as the *pid*, *tname*, *cputime*, *state*, and *comm* for current processes.

```
% ps -aeO psxpri
PID  PPR  STAT  TT  TIME      COMMAND
0     31  R<    ??  29171:32.53 [kernel idle]
1     18  I     ??    0:17.37 /sbin/init -a
2     19  I     ??    0:00.00 [device server]
3     19  I     ??    0:00.00 [exception hdlr]
      .
      .
      .
7206 60   S<    p2    0:00.01 ./tests/work
7207 18   R     p2    0:00.06 ps -aeO psxpri
```

In the example above, two processes are using realtime priorities. The first process (*pid* 0) is running at maximum system priority. The processes with *pid*s 2 and 3 are idle at the maximum user priority. The second realtime process (*pid* 7206) has been sleeping for less than twenty seconds at priority 60.

See the reference page for the ps command for further information.

2.3.4 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 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/D11 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 overrides the default priority, or using environment variables, which override the default priority
3. Adjusting priority during initialization by calling the `sched_setparam` 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.

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_getparam` and `sched_setparam` P1003.4/D11 functions, discussed in Section 2.4.3. You should only adjust process priorities for your own application. Adjusting system process priorities could have unexpected consequences.

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/D11 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/D11 Process Scheduling Functions

Function	Description
`sched_getscheduler`	Returns the scheduling policy of a specified process
`sched_getparam`	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 current quantum for the round-robin scheduling policy
`sched_setscheduler`	Sets the scheduling policy and priority of a specified process

(continued on next page)

Table 2–4 (Cont.) P1003.4/D11 Process Scheduling Functions

Function	Description
sched_setparam	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/D11 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.

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. You must have superuser privileges to change the realtime priorities or scheduling policies of a process.

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/D11 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_getparam 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_getparam function to determine the initial priority of a process; use the sched_setparam 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_setparam 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_setparam function takes two parameters: *pid* and *param*. 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_setparam or sched_setscheduler. Example 2–1 shows how a process can initialize the fields using only constructs provided by the P1003.4/D11 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 <unistd.h>
#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_getparam(my_pid, &param);
  CHECK(sts,"sched_getparam");

       /* Keep track of the old priority. */

  old_priority = param.sched_priority;

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

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

(continued on next page)

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

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

    param.sched_priority = low_priority;
    sts = sched_setparam(my_pid, &param);
    CHECK(sts,"sched_setparam");

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

    param.sched_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.
- The real user UID or the effective user UID of the calling process is equal to the real user UID or the saved-set user UID of the target process.
- The real group GID or the effective group GID of the calling process is equal to 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 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 shows how the amount of time in a round-robin quantum can be determined, the current scheduling parameters saved, and a realtime priority set. 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 <unistd.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;
     volatile int tmp_nbr;

      /* Determine the round-robin quantum */
```

(continued on next page)

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

```
stat = sched_get_rr_interval (my_pid, &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 */

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

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

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

3
Shared Memory

Shared memory and memory-mapped files allow processes to communicate by incorporating data directly into process address space. Processes communicate by sharing portions of their address space. When one process writes to a location in the shared area, the data is immediately available to other processes sharing the area. Communication is fast because there is none of the overhead associated with system calls. Data movement is reduced because it is not copied into buffers.

This chapter includes the following sections:

- Memory Objects, Section 3.1
- Locking Shared Memory, Section 3.2
- Using Shared Memory with Semaphores, Section 3.3

A process manipulates its address space by mapping or removing portions of memory objects into the process address space. When multiple processes map the same memory object, they share access to the underlying data. Shared-memory functions allow you to open and unlink the shared-memory files.

3.1 Memory Objects

The memory-mapping and shared-memory functions allow you controlled access to shared memory so that the application can coordinate the use of shared address space.

When you use a shared, mapped file, the changes initiated by a single process or multiple processes are are reflected back to the file. Other processes using the same path and opening the connection to the memory object have a shared mapping of the file. Use memory-mapping or file control functions to control usage and access. If the mappings allow it, data written into the file through the address space of one process appears in the address space of all processes mapping the same portion of the file.

Memory-mapped objects are persistent; their names and contents remain until all processes that have accessed the object unlink the file.

Shared memory and memory mapped files follow the same general usage, as follows:

1. Get a file descriptor with a call to the open or shm_open function.
2. Map the object using the file descriptor with a call to the mmap function.
3. Unmap the object with a call to the munmap function.
4. Close the object with a call to the close function.
5. Remove the shared-memory object with a call to the shm_unlink function or optionally remove a memory-mapped file with a call to the unlink function.

Often shared-memory objects are created and used only while an application is executing. Files, however, may need to be saved and reused each time the application is run. The unlink and shm_unlink functions remove (delete) the file and its contents. Therefore, if you need to save a shared file, just close the file, but do not unlink it.

You can use memory-mapped files without using shared memory, but this chapter assumes that you will want to use them together. Table 3–1 summarizes the functions used to open and unlink shared memory.

Table 3–1 Shared-Memory Functions

Function	Description
shm_open	Opens a shared-memory object, returning a file descriptor
shm_unlink	Removes the name of the shared-memory object

Table 3–2 lists the functions for creating and controlling memory-mapped objects.

Table 3–2 Memory-Mapping Functions

Function	Description
mmap	Maps the memory object into memory
mprotect	Modifies protections of memory objects

(continued on next page)

Table 3–2 (Cont.) Memory-Mapping Functions

Function	Description
msync	Synchronizes a memory-mapped object
munmap	Unmaps a previously mapped region

A memory object can be created and opened by a call to the shm_open function. Then the object can be mapped into process address space. File control functions allow you to control access permissions, such as read and write permission or the timing of a file update.

Data written to an object through the address space of one process is available to all processes that map the same region. Child processes inherit the address space and all mapped regions of the parent process. Once the object is opened, the child process can map it with the mmap function to establish a map reference. If the object is already mapped, the child process also inherits the mapped region.

Unrelated processes can also use the object, but must first call the open or shm_open function (as appropriate) and then use the mmap function to establish a connection to the shared memory.

3.1.1 Opening a Shared-Memory Object

A process can create and open shared-memory regions early in the life of the application and then dynamically control access to the shared-memory object. Use the shm_open function to open (establish a connection to) a shared-memory object. After a process opens the shared-memory object, each process that needs to use the shared-memory object must use the same pathname as the controlling process when creating its own connections to the shared-memory object by also calling the shm_open function.

The shm_open function provides a set of flags that prescribe the action of the function and define access modes to the shared-memory object. Shared-memory access is determined by the OR of the file status flags and access modes listed in Table 3–3.

Table 3–3 Status Flags and Access Modes for the shm_open Function

Flag	Description
O_RDONLY	Open for read access only
O_RDWR	Open for read and write access
O_CREAT	Create the shared-memory object, if it does not already exist
O_EXCL	When used with O_CREAT, creates a shared-memory object
O_TRUNC	Truncate to zero length

The first process to call the shm_open function should use the O_CREAT flag to create the shared-memory object, to set the object's user ID to that of the calling process, and to set the object's group ID to the effective group ID of the calling process. This establishes an environment whereby the calling process, all cooperating processes, and child processes share the same effective group ID with the shared-memory object.

A process can create an exclusive connection to a shared-memory object by using the O_CREAT and O_EXCL flags. In this case, other processes attempting to create the shared-memory object at the same time will fail.

The *oflag* argument of the shm_open function requests specific actions from the shm_open code. For example, the following code creates an exclusive shared-memory object and opens it for read and write access.

```
fd = shm_open("all_mine", (O_CREAT|O_EXCL|O_RDWR), 0);
```

Once a shared-memory object is created, its state and name (including all associated data) are persistent. Its state and name persist until the shared-memory is unlinked with a call to the shm_unlink function and until all other references to the shared memory are gone.

Example 3–1 shows the code sequence to include shared-memory objects in an application.

Example 3–1 Including a Shared-Memory Object

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>

main ()
      int md;
      int status;
      long pg_size;
      caddr_t virt_addr;
                                        /* Create shared memory object */
      md = shm_open ("my_memory", O_CREAT|O_RDWR, 0);
      pg_size = sysconf(_SC_PAGE_SIZE);
     if((ftruncate(md, pg_size)) == -1){    /* Set the size */
          perror("ftruncate failure");
          exit();
                                             /* Map one page */
      virt_addr = mmap(0, pg_size, PROT_WRITE, MAP_SHARED, md, 0);
           .
           .
           .
      status = munmap(virt_addr, pg_size);  /* Unmap the page */
      status = close(md);                   /*   Close file   */
      status = shm_unlink("my_memory");     /* Unlink shared-memory object */
```

3.1.2 Opening Memory-Mapped Files

The open function points to the data you intend to use; the mmap function establishes how much of the data will be mapped and how it will be accessed. Use the same access permissions that you would normally use on any call to the open function. If you intend to only read the file, specify only read permission on the open function. If you intend to read and write to the file, open the file with both read and write permissions. After opening a file, call the mmap function to map the file into application address space.

When finished using a memory-mapped file, unmap the object by calling the munmap function, then close the object with the close function. Any memory locks resulting from a call to the mlock function associated with the address range are removed when the munmap function is called. The application could then remove the data file by calling the unlink function.

3.1.3 Mapping Memory-Mapped Files

The mmap function maps the memory by mapping data from a file into memory. The parameters to the mmap function specify the starting address and length in bytes for the new region, access permissions, attributes of the mapped region, file descriptor, and an offset for the address. The MAP_SHARED flag indicates the object will be accessible by other processes. A call to the munmap function unmaps the same region.

The address, length, and offset of the new mapped region should be a multiple of the page size returned by a call to the sysconf (_SC_PAGE_SIZE) function. If the length is not specified as a multiple of the page size returned by sysconf, then any reference to an address between the end of the region and the end of the page containing the end of the region is undefined. Note, too, that the offset must be aligned and sized properly. Other size parameters may also need to be aligned, depending on whether you specified MAP_FIXED.

The *prot* argument determines the type of access permitted to the data being mapped. As with other file permissions, the argument is constructed from the bitwise inclusive OR of one or more of the following flags:

PROT_READ — Data can be read.
PROT_WRITE — Data can be written.
PROT_EXEC — Data can be executed.
PROT_NONE — Data cannot be accessed.

Whatever of the protection options specified as the *prot* argument, the file descriptor must have been opened with at least read access. If you specify PROT_WRITE, the file descriptor must have been opened with write permission, unless MAP_PRIVATE is specified in the *flags* parameter.

The *flags* parameter provides additional information about how to handle mapped data. The *flags* parameter uses one of the following flags:

MAP_SHARED — Share changes
MAP_PRIVATE — Changes are private
MAP_FIXED — Interpret the *addr* argument exactly

MAP_SHARED, MAP_PRIVATE, and MAP_FIXED are the only flags allowed with the POSIX 1003.4 Draft 11 interfaces for shared memory. The MAP_ANONYMOUS, MAP_FILE, and MAP_VARIABLE flags are not supported by POSIX 1003.4 Draft 11, but are supported by DEC OSF/1. For more information on these flags, see the reference page for the mmap function.

The MAP_FIXED flag controls the location of the new region. No matter what flag is specified, a mapped region is never placed at address zero or at an address where it would overlap with an existing region. When multiple processes use the mapped object, the call to the `mmap` function can specify the address, and subsequent calls to the `mmap` function can use MAP_FIXED to request the same address in other processes. Cooperating processes must also use care to communicate this address among themselves. If you specify MAP_FIXED and for some reason the system is unable to place the new region at the specified address, the call fails.

The MAP_SHARED and MAP_PRIVATE flags control the visibility of modifications to the mapped file or shared-memory region. The MAP_SHARED flag specifies that modifications made to the mapped file region are immediately visible to other processes which are mapped to the same region and also used the MAP_SHARED flag. Changes to the region are written to the file.

The MAP_PRIVATE flag specifies that modifications to the region are not visible to other processes whether or not the other process used MAP_SHARED or MAP_PRIVATE. Modifications to the region are not written to the file.

Access to the mapped region or shared-memory region is controlled by the flags specified in the *prot* parameter. These flags function much the way they do for any other file descriptor in that access is specified as the OR of read, write, and execute with an additional flag to indicate that data cannot be accessed. The `mprotect` function changes the protection on a specified address range. That range should be within the range specified on the call to the `mmap` function. Protection flags can interact with the MAP_SHARED, MAP_PRIVATE, and MAP_FIXED flags. Refer to the online reference pages for `mmap` and `mprotect` for the specifics.

When you unmap a mapped region or shared memory, be sure to specify an address and length in the range of the parameters used in the call the `mmap` function.

3.1.4 Using File Functions

Shared-memory objects and memory-mapped files use the file system name space to map global names for memory objects. As such, POSIX.1 file control functions can be used on shared-memory objects and memory-mapped files, just as these functions are used for any other file control. Table 3–4 lists some of the file functions available.

Table 3–4 File Functions Used with Memory-Mapped Files

Function	Description
`fchmod`	Changes permissions on file descriptors
`fcntl`	Controls operations on files and memory objects
`flock`	Locks a file as shared or exclusive
`fstat`	Provides information about file status
`ftruncate`	Sets the length of a memory object

The `fstat` function returns information about the file, such as access permissions, link references, and type and size of file. You can use this function to obtain information for use in subsequent calls to other file control functions. The `fchmod` function can be used to change access permissions on a file. If you are the owner of the file or have superuser privileges, you can use the `fchmod` function to set the access mode and grant or deny permissions to the group, user, or others. The `fcntl` function can be used to retrieve and set the value of the close-on-exec flag, status flags and access modes, or set and clear locks. Using the `fcntl` function, you can override locks set with the `flock` function.

You can apply a lock to a shared-memory object or mapped file by using a variety of file control functions, including `fcntl` and `flock`. Both these functions apply a lock on an open file, but they differ in how the lock is performed and the range of other tasks they can perform.

Note that the locks applied with these functions are for files, not file descriptors. That means that under most circumstances, file locks are not inherited across a fork. If a parent process holds a lock on a file and the parent process forks, the child process will inherit the file descriptor, but not the lock on the file. A file descriptor that is duplicated with one of the dup functions does not inherit the lock.

The `fcntl` function is used for general file control. In addition to locking and unlocking an open file, the `fcntl` function is used to return or set status, a new file descriptor, or process IDs .

The `flock` function is limited to applying locks on a file and is not used for general file control.

Refer to the online reference pages for more information on using the file control functions.

3.1.5 Controlling Memory-Mapped Files

Numerous functions let you manipulate and control access to memory-mapped files and shared memory. These functions include msync and mprotect. Using these functions, you can modify access protections and synchronize writing to a mapped file.

The msync function synchronizes the caching operations of a memory-mapped file or shared-memory region. Using this function, you can ensure that modified pages in the mapped region are transferred to the file's underlying storage device or you can control the visibility of modifications with respect to file system operations.

Flags used on the msync function specify whether the cache flush is to be synchronous (MS_SYNC), asynchronous (MS_ASYNC), or invalidated (MS_INVALIDATE). Either the MS_SYNC or MS_ASYNC flag may be specified, but not both.

When you use the MS_SYNC flag, the msync function does not return until all write operations are complete and the integrity of the data is assured. All previous modifications to the mapped region are visible to processes using the read parameter.

When you use the MS_ASYNC flag, the msync function returns immediately once all of the write operations are scheduled.

When you invalidate previously cached copies of the pages, other users are required to get new copies of the pages from the file system the next time they are referenced. In this case, previous modifications to the file made with the write function are visible to the mapped region.

When using the msync function, you should use pages within the same address and length arguments used in the mmap function to ensure that the entire mapped region is synchronized.

The mprotect function changes the access protection of a mapped file or shared-memory region. When using the mprotect function, use pages within the same address and length specified in the call to the mmap function. Protection flags used on the mprotect function are the same as those used on the mmap function.

Note that use of the mprotect function modifies access only to the specified region. If the access protection of some pages within the range were changed by some other means, the call to the mprotect function may fail.

3.1.6 Removing Shared Memory

When a process has finished using a shared-memory segment, you can remove the pathname from the file system namespace with a call to the shm_unlink function, as shown in the following example:

```
status = shm_unlink("my_file");
```

The shm_unlink function unlinks the shared-memory object. Memory objects are persistent; which means the contents remain until all references have been unmapped and the shared-memory object has been unlinked with a call to the shm_unlink function.

Every process using the shared memory should perform the cleanup tasks of unmapping and closing.

3.2 Locking Shared Memory

You can lock and unlock a shared-memory segment into physical memory to eliminate paging. The MLOCK_FUTURE argument to the mlockall function causes new shared-memory regions to be locked automatically. See Chapter 4 for more information on using the mlock and mlockall functions.

Example 3–2 shows how to map a file into the address space of the process and lock it into memory. When the file is unmapped, the lock on the address is removed.

Example 3–2 Locking a Memory Object

```
/* This program locks the virtual memory address that */
/* was returned from the mmap() function into the memory. */

#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>

main()
{
int fd;
caddr_t pg_addr;

int size = 5000;
int mode =  S_IRWXO|S_IRWXG|S_IRWXU;
```

(continued on next page)

Example 3–2 (Cont.) Locking a Memory Object

```
     /* Create a file */

fd = shm_open("example", O_RDWR|O_CREAT, mode);
if(fd < 0){
   perror("open error ");
   exit();
}

     /* Set the size */

if((ftruncate(fd, size)) == -1){
     perror("ftruncate failure");
     exit();
}

     /* Map the file into the address space of the process */

pg_addr = (caddr_t) mmap(0, size, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_SHARED, fd, 0);

if(pg_addr == (caddr_t) -1){

  perror("mmap failure");
  exit();
}

     /* Lock the mapped region into the memory */

if(mlock(pg_addr,size) != 0){
  perror("mlock failure");
  exit();
}

     /* Unmap of the address region removes the memory lock */
     /* established on the address region by this process   */

if(munmap(pg_addr, size) < 0)
   perror("unmap error");

close(fd);

shm_unlink("example");
exit();
}
```

You can also lock the file so that other processes cannot use it, making it an exclusive resource for a process and its descendants. See Section 3.1.4 for more information on locking files.

3.3 Using Shared Memory with 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 provide an easy means of regulating access to a memory object and determining if the memory resource is available. Typically, an application will begin execution at a nonrealtime priority level, then perform the following tasks when using mapped or shared-memory objects and semaphores:

1. Create the shared-memory object
2. Determine the address and map the region into memory
3. Create a binary semaphore set
4. Adjust the process priority and scheduling policy as needed
5. Before a read or write operation, lock (reserve) the semaphore
6. After a read or write operation, unlock (release) the semaphore

A process can lock the semaphore associated with a mapped or shared-memory object to indicate that the process requires exclusive access. Cooperating processes normally wait until the semaphore is unlocked before accessing a region.

Refer to Chapter 8 for information on binary semaphores and for an example using semaphores and shared memory.

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 the user 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.

This chapter covers the following sections:

- Memory Management, Section 4.1
- Memory-Locking and Unlocking Functions, Section 4.2

Memory locking is one way to ensure that a process stays in main 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.

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 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 the 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 to a file area. This page is brought back into primary memory as needed and execution is delayed while the kernel retrieves the page.

Paging is generally transparent to the current process. The amount of paging can be decreased by increasing the size of physical memory or by locking the pages into memory. 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 efficiency.

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 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 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.

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 address space and the locks are removed.

Two functions, `mlock` and `mlockall`, are used to lock memory. The `mlock` function allows the calling process to lock a selected region of address space. 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

Table 4–1 lists the memory-locking functions.

Table 4–1 Memory-Locking Functions

Function	Description
`mlockall`	Locks all of a process's address space
`munlockall`	Unlocks all of 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 memory-locking functions support locking the entire address space or a selected range of one or more pages.

4.2.1 Locking and Unlocking a Specified Region

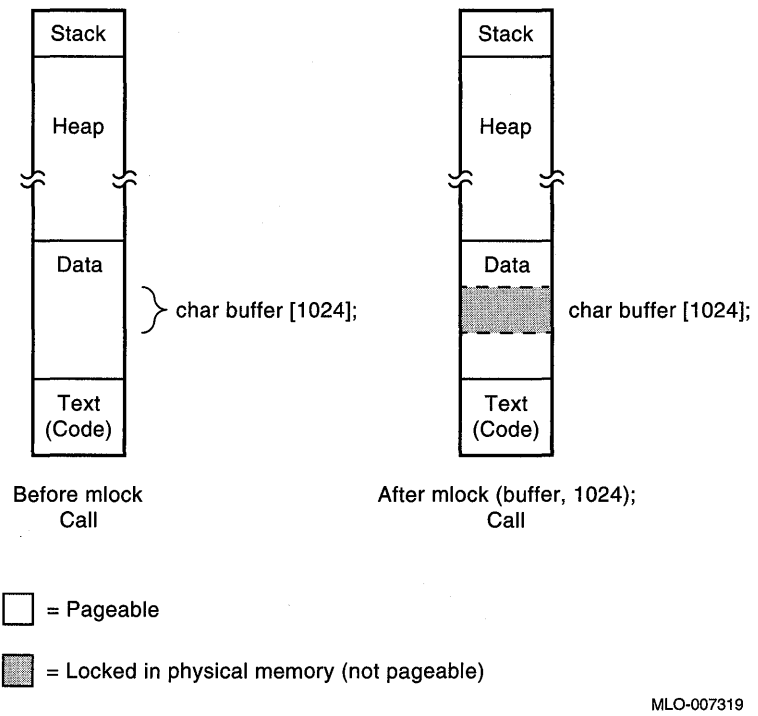
The `mlock` function locks a preallocated specified region. The address and size arguments of the `mlock` function determine the boundaries of the preallocated region. On a successful call to the `mlock` function, the specified region becomes locked. Memory is locked by the system according to system-defined 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 beyond 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.

Figure 4–1 Memory Allocation with mlock



The `mlock` function locks all pages defined by the range *addr* to *addr+len–1* (inclusive). The area locked is the same as if the *len* argument were rounded up to a multiple of the next page size before decrementing by 1. 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 sys/mman.h header file defines the maximum amount of memory that can be locked. Use the getrlimit function to determine the amount of total memory.

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.

Locked space is automatically unlocked when the process 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 can 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.

Example 4–1 shows how to lock and unlock memory segments. Each user-written function determines page size, adjusts boundaries, then either locks or unlocks the segment.

Example 4–1 Aligning and Locking a Memory Segment

```
#include <unistd.h>     /* Support all standards    */
#include <sys/mman.h>   /* Memory locking functions */

#define DATA_SIZE 2048

lock_memory(char   *addr,
            size_t  size)
{
  unsigned long    page_offset, page_size;

  page_size = sysconf(_SC_PAGE_SIZE);
  page_offset = (unsigned long) addr % page_size;

  addr -= page_offset;  /* Adjust addr to page boundary */
  size += page_offset;  /* Adjust size with page_offset */

  return ( mlock(addr, size) );  /* Lock the memory */
}
```

(continued on next page)

Example 4–1 (Cont.) Aligning and Locking a Memory Segment

```
unlock_memory(char   *addr,
              size_t  size)
{
  unsigned long    page_offset, page_size;

  page_size = sysconf(_SC_PAGE_SIZE);
  page_offset = (unsigned long) addr % page_size;

  addr -= page_offset;  /* Adjust addr to page boundary */
  size += page_offset;  /* Adjust size with page_offset */

  return ( munlock(addr, size) );  /* Unlock the memory */
}

main()
{
  char data[DATA_SIZE];

  if ( lock_memory(data, DATA_SIZE) == -1 )
    perror("lock_memory");

          /* Do work here */

  if ( unlock_memory(data, DATA_SIZE) == -1 )
    perror("unlock_memory");
}
```

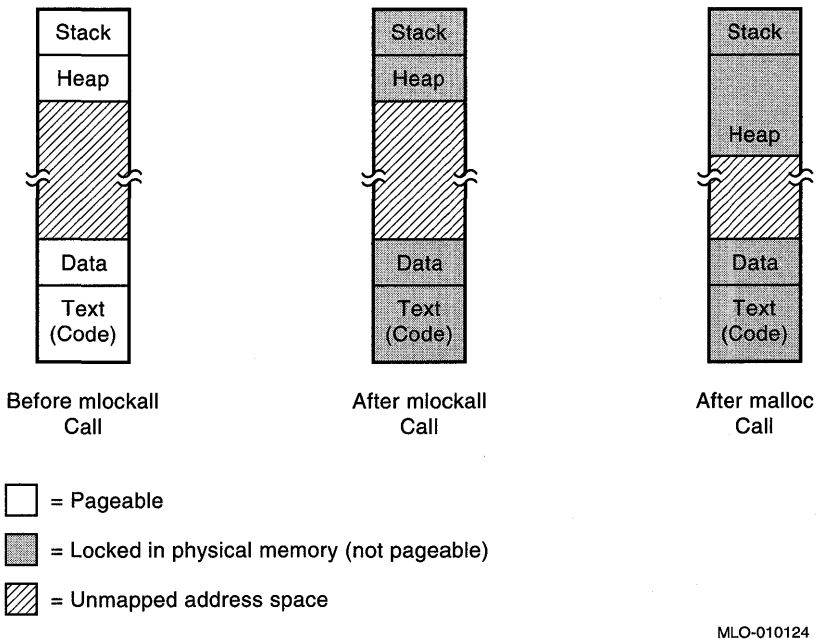
4.2.2 Locking an Unlocking an Entire Process Space

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 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 subsequent growth is automatically locked into memory.

Figure 4–2 shows memory allocation before and after a call to the `mlockall` function with both MCL_CURRENT and MCL_FUTURE flags. 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



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. The call to the `munlockall` function cancels the MCL_FUTURE flag. If you want additional locking later, you must again call on the the memory-locking functions.

Example 4–2 illustrates how the mlockall function might be used to lock current and future address space.

Example 4–2 Using the mlockall Function

```
#include <unistd.h>     /* Support all standards    */
#include <stdlib.h>     /* malloc support           */
#include <sys/mman.h>   /* Memory locking functions */

#define BUFFER 2048

main()
{
  void *p[3];  /* Array of 3 pointers to void */

  p[0] = malloc(BUFFER);

       /* Currently no memory is locked */

  if ( mlockall(MCL_CURRENT) == -1 )
    perror("mlockall:1");

      /* All currently allocated memory is locked */

  p[1] = malloc(BUFFER);

      /* All memory but data pointed to by p[1] is locked */

  if ( munlockall() == -1 )
    perror("munlockall:1");

      /* No memory is now locked */

  if ( mlockall(MCL_FUTURE) == -1 )
    perror("mlockall:2");

      /* Only memory allocated in the future */
      /*   will be locked */

  p[2] = malloc(BUFFER);

      /* Only data pointed to by data[2] is locked */

  if ( mlockall(MCL_CURRENT|MCL_FUTURE) == -1 )
    perror("mlockall:3");

  /* All memory currently allocated and all memory that  */
  /* gets allocted in the future will be locked          */

}
```

5
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/D11 Realtime Signals, Section 5.1
- The Signal Interface, Section 5.2

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.

5.1 P1003.4/D11 Realtime Signals

The P1003.4/D11 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/D11 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:

```
int             sigev_signo;   /* Signal sent on timer expiration */
union sigval    sigev_value;   /* Not supported - Specify as NULL    */
```

The *sigval* union contains at least the following members:

```
int     sival_int;             /* Used when sigev_value is of type int */
void    *sival_ptr;            /* Used when sigev_value is of type ptr */
```

The *sigev_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/D11 Realtime Signals, which are not fully implemented at this time. Specify a value of NULL for this member.

The *sigev_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 6 and Chapter 7 for examples of using signals with these functions.

5.2 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 5–1 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 5–1 Signal Control Functions

Function	Description
	Controlling a Signal Set
sigaction	Examines or specifies the action of a specific signal
sigpending	Returns 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 5–1 (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

5.2.1 Sending Signals

Signals are sent by a user process, the kernel, or a driver program. Four conditions 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 5–1, a parent process sends a signal to its child, which handles the signal and exits.

Example 5–1 Sending Signals Between Processes

```
/* The parent process sends SIGINT to a child process. */
/* The child process handles the signal and exits.     */

#include <unistd.h>
#include <signal.h>
#include <stdio.h>

main()
{
int pid;        /* The child's PID is returned by fork()   */

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                       */
      }
}

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             */
     }
```

5.2.2 Blocking Signals

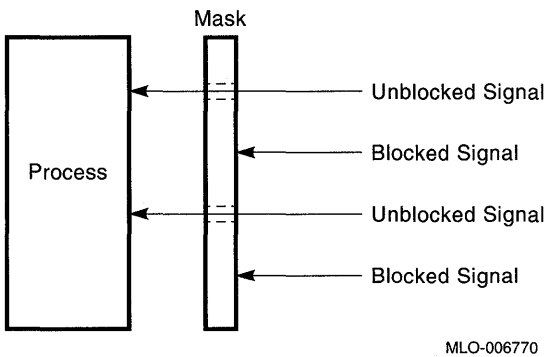
A signal can be blocked to protect certain sections of code from receiving signals when the work should not be interrupted. Unlike ignoring a signal, blocking a signal postpones the signal until the process is ready to handle it.

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 nine functions listed in Table 5–1 to manipulate and examine its signal mask, thus regulating the signals to which it is sensitive.

Figure 5–1 represents a mask blocking two signals. In this illustration, two signal bits are set, blocking signal delivery for the specified signals.

Figure 5–1 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 specify additional signals that you want blocked until an unblocked signal 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 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 SIGSYS 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 change the contents of the set. See Section 5.2.3.4 for more information on using the sigsetops primitive functions.

5.2.3 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 — Discard 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, 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 page for the signal function for a complete description of the default actions associated with individual signals.

5.2.3.1 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 the call can be used 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 5–2 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 5–2 Using the alarm Function

```
#include <unistd.h>
#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);
```

(continued on next page)

Example 5–2 (Cont.) Using the alarm Function

```
    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);
    }
```

5.2.3.2 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 values 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, to stop the process, or to terminate the process.

5.2.3.3 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 5–3 shows one way to handle the SIGINT signal. First it cleans up the condition that generated the SIGINT signal; then it stops the program.

Example 5–3 Handling Signals

```
/* This program prompts for input in file 'tmp'.    */
/* If interrupted by Ctrl/C, remove 'tmp' and exit. */

#include <unistd.h>
#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.        */

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() */
```

(continued on next page)

Example 5–3 (Cont.) Handling Signals

```
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 the `signal` function is called for SIGINT and its value is tested before declaring a handler for SIGINT in the previous example. If the program, `write_text.c`, declares that 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 5–4 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 5–4 Sending a Signal to Another Process

(continued on next page)

Example 5–4 (Cont.) Sending a Signal to Another Process

```
#include <unistd.h>
#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);
      }
```

5.2.3.4 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 on the set of signals known to the system. Table 5–2 lists the sigsetops primitive functions.

Table 5–2 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.

6

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 the need for either 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 to be completed within a 5 millisecond time frame.

DEC OSF/1 P1003.4/D11 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.

This chapter includes the following sections:

- Clock Functions, Section 6.1
- Types of Timers, Section 6.2
- Timers and Signals, Section 6.3
- Data Structures Associated with Timing Facilities, Section 6.4
- Timer Functions, Section 6.5
- High-Resolution Sleep, Section 6.6
- Clocks and Timers Example, Section 6.7

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/D11 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 to expire once or multiple times (arm the timers)
- Use asynchronous signals on timer expiration
- Retrieve the resolution of the systemwide clock
- 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.

Although non-POSIX functions are available for creating timers, application programmers striving for standards conformance, portability, multiple per-process timers, and flexibility in using high-resolution timers should use the P1003.4/D11 timing facilities described in this chapter.

6.1 Clock Functions

The supported time-of-day clock is the CLOCK_REALTIME clock, defined in the `time.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 (in nanoseconds) that has elapsed since 00:00:00:00 January 1, 1970 Greenwich Mean Time (GMT). [1]

Table 6–1 lists P1003.4/D11 timing functions for a specified clock.

Table 6–1 Clock Functions

Function	Description
`clock_getres`	Returns the resolution of the specified clock

(continued on next page)

[1] January 1, 1970 is otherwise known as the "Epoch."

Table 6–1 (Cont.) Clock Functions

Function	Description
clock_gettime	Returns the current value for the specified clock
clock_getdrift	Returns the value of the clock drift rate as set by the most recent call to clock_setdrift
clock_settime	Sets the specified clock to the specified value
clock_setdrift	Sets the drift rate of the specified clock, in parts per billion (nanoseconds), to the specified value

Use the name CLOCK_REALTIME as the *clock_id* argument in all P1003.4/D11 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. Note that you cannot set the resolution of the specified clock.

The following example calls the clock_getres function to determine clock resolution.

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

main()
{
struct timespec     clock_resolution;
int stat;

stat = clock_getres(CLOCK_REALTIME, &clock_resolution);

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

6.1.1 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 6–1 shows the difference between the time as returned by the time and clock_gettime functions.

Example 6–1 Returning Time

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

main()
{
struct timespec ts;

      /* Call time */

printf("time returns %d seconds\n", time(NULL);)

       /* Call clock_gettime */

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

In Example 6–1 876764530 seconds is returned from the `time` function and returns 876764530 seconds and 0000674633 nanoseconds is returned 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 receives a pointer to the `timespec` structure and returns the values in the *tv_sec* and *tv_nsec* members.

If you plan to write the current time to a device or file, you may want to convert the time format returned by the `clock_gettime` function.

6.1.2 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_setdrift` function to adjust the clock slowly. Armed timers are not affected by the `clock_setdrift` function.

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

6.1.3 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/D11 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/D11 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.

The `clock_setdrift` function fine-tunes the clock slowly by setting the clock's drift rate. Pending timeouts are not affected by drifts applied 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_setdrift` function. The `clock_getdrift` function returns the value of the clock drift rate as set by the most recent call to the `clock_setdrift` function.

The next example calls the `clock_setdrift` 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_setdrift` function. A call to the `clock_getdrift` function checks the new drift rate.

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

#define SUCCESS 0

main()
{
int clock_id = CLOCK_REALTIME;
int drift = 1000;
int odrift;

if(clock_setdrift(clock_id, drift, &odrift) == SUCCESS)
   printf("previous drift is %d\n", odrift);

if(clock_getdrift(clock_id, &drift) == SUCCESS)
   printf("current drift is %d\n", odrift);
}
```

Note that the clock drift functions should not be used in conjunction with Digital Distributed Time Service (DECdts).

6.2 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 timer becomes a periodic timer with the addition of a repetition value. The timer expires then loads the repetition interval, rearming the timer to expire after the repetition interval has elapsed.

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 expires at a calendar date and time.

Often, a timer uses both concepts of absolute and relative timers. You can establish a timer to fire as an absolute timer when it first expires, and set subsequent timer expirations 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 times are used to start and stop the data collection processes at midnight and 3:00 A.M. respectively. Relative time is 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 type. Refer to Section 6.5.2 for more information on the `timer_settime` function.

6.3 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 system sends the specified signal to the process that created 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 in 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 with the sigaction function.
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 5 for more information on signals and signal handling.

6.4 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 time.h are used in many of the P1003.4/D11 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/D11 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/D11 timer functions.

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

6.4.1 Using the timespec Data Structure

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

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

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 time interval described by the timespec structure is $(tv\_sec * 10^9) + tv\_nsec$ nanoseconds. (The minimum possible time interval is limited by the resolution of the specified clock.)

The timespec structure is used in P1003.4/D11 functions to set and return the specified clock and to return the resolution of clocks, timers, and nanosleep.

6.4.2 Using the itimerspec Data Structure

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

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

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 6–2 summarizes the ways that values for the two members of the itimerspec structure are used to specify timers.

Table 6–2 Values Used in Setting Timers

Member	Zero	Non-Zero
it_value	No expiration value	Expiration value
	Disarm the timer	Arm the timer
it_interval	No reload value	Interval reload value
	One-shot timer	Periodic 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 timer's first expiration. If the TIMER_ABSTIME flag is not set, the time until the next timer expiration is set equal to the interval specified by the *it_value* member, and the timer is a relative timer.

If the TIMER_ABSTIME flag is set, timer is an absolute timer. The time until the next timer expiration is specified in seconds and nanoseconds since the Epoch and is the difference between the absolute time specified by the *it_value* member and the current clock value. A zero value for the *it_value* member disarms the timer.

Once the timer expires 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 specify a timer that executes 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;
```

To arm a timer to execute 15 seconds from now and then at 0.5 second intervals, specify the following values:

```
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 6.5 for more information on relative and absolute timers.

6.4.3 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, that contains 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:

```
int             sigev_signo;   /* Signal sent on timer expiration */
union sigval    sigev_value;   /* Not supported - Specify as NULL   */
```

The *sigval* union contains at least the following members:

```
int     sival_int;             /* Used when sigev_value is of type int */
void    *sival_ptr;            /* Used when sigev_value is of type ptr */
```

The *sigev_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/D11 Realtime Signals, which are not currently supported. Specify a value of NULL for this member.

The *sigev_signo* member specifies the signal to be sent.

6.5 Timer Functions

Clocks and timers allow an application to synchronize and coordinate activities according to a user-defined schedule. DEC OSF/1 P1003.4/D11 timers have the ability to issue periodic timer requests initiated by a single call from the application.

Table 6–3 lists the P1003.4/D11 timing functions available for realtime applications.

Table 6–3 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_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

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 new image receive the signal. The newly created timer structures are inherited across a fork, but any pending timer signals will be delivered only to the parent process.

6.5.1 Creating Timers

The `timer_create` function allocates a timer and returns a timer ID which is unique within the calling process and exists for the life of that timer. 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, set up a signal handler to catch the signal when the timer expires, and arm the timers. To use timers in a realtime application, follow these steps:

1. Include `time.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.
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 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 and value to be sent to the calling process when the timer expires. You can either use the default signal, SIGALRM, or specify a particular signal.

6.5.2 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 *flags* argument determines whether the timer behaves as an absolute or relative timer.

If the TIMER_ABSTIME flag is set, the timer is set with a specified starting time (the timer is an absolute timer). If the TIMER_ABSTIME flag is not set, the timer is set relative to the current time (the timer is a relative timer).

3. The *value* argument points to an `itimerspec` structure, which contains the initial expiration value and repetition value for the timer.

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

     For absolute timers, 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.

     For relative timers, 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 that 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 reached, it is reloaded 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 points to an itimerspec structure that contains the time remaining on an active timer. 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.

You can use the timer_settime function to reuse an existing timer ID. If a timer is pending and you call the timer_settime function to pass in new expiration times, a new expiration time is established.

6.5.3 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.

6.5.4 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.

6.6 High-Resolution Sleep

To suspend process execution temporarily using the P1003.4/D11 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.

6.7 Clocks and Timers Example

Example 6–2 demonstrates the use of P1003.4/D11 realtime timers. The program creates both absolute and relative timers. The example demonstrates concepts using multiple signals to distinguish between timer expirations. The program loops continuously until the program is terminated by a Ctrl/C from the user.

Example 6–2 Using Timers

```
/*
 * The following program demonstrates the use of various types of
 * POSIX 1003.4 Realtime Timers in conjunction with 1003.1 Signals.
 *
 * The program creates a set of timers and then blocks waiting for
 * either timer expiration or program termination via SIGINT.
 * Pressing CTRL/C after a number of seconds terminates the program
 * and prints out the kind and number of signals received.
 *
 * To build:
 *
 * cc -g3 -O -non_shared -o timer_example timer_example.c -L/usr/ccs/lib -lrt
 */

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

/*
 * Constants and Macros
 */

#define FAILURE -1
#define ABS     TIMER_ABSTIME
#define REL     0
#define TIMERS  3

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

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

(continued on next page)

Example 6–2 (Cont.) Using Timers

```
/*
 * 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, { sigev_value, sigev_signo }, { it_iteration, it_value }
 *
 */

struct timer_definitions timer_values[TIMERS] = {
        { ABS, {0,SIGALRM}, {0,0, 3,0} },
        { ABS, {0,SIGUSR1}, {0,500000000, 2,0} },
        { REL, {0,SIGUSR2}, {0,0, 5,0} }
};

timer_t timerid[TIMERS];
int timers_available;                   /* number of timers available */
volatile int alrm, usr1, usr2;
sigset_t mask;

/*
 * This program demonstrates the use of various P1003.4 Timers.
 */

main()
{
        int status, i;
        int clock_id = CLOCK_REALTIME;
        struct timespec current_time;

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

        sigemptyset(&mask);
        sig_act.sa_handler = (void *)sig_handler;
        sig_act.sa_flags = 0;
        sigemptyset(&sig_act.sa_mask);
        alrm = usr1 = usr2 = 0;
```

(continued on next page)

Example 6–2 (Cont.) Using Timers

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

timers_available = MIN(sysconf(_SC_TIMER_MAX),TIMERS);

/*
 * Create "timer_available" timers, using a unique signal
 * type to denote the timer's 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("timer_create");
                exit(FAILURE);
        }
        sigaction(timer_values[i].evp.sigev_signo, &sig_act, 0);
}

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

sigaction(SIGINT, &sig_act, NULL);      /* catch crtl-c */

/*
 * Queue the following Timers: (see timer_values structure for details)
 *
 *   1.  An absolute one shot timer  (Notification is via SIGALRM).
 *   2.  An absolute periodic timer. (Notification is via SIGUSR1).
 *   3.  A relative one shot timer.  (Notification is via SIGUSR2).
 *
 * (NOTE: The number of TIMERS queued actually depends on
 *  timers_available)
*/
```

(continued on next page)

Example 6–2 (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 (CRTL/C will do this).
         */

        for(;;) pause();

}

/*
 *  Handle Timer expiration or Program Termination.
 */

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

        switch (signo) {
            case SIGALRM:
                alrm++;
                break;
            case SIGUSR1:
                usr1++;
                break;
            case SIGUSR2:
                usr2++;
                break;
            case SIGINT:
                for (i = 0; i < timers_available; i++)  /* delete timers */
                        status = timer_delete(timerid[i]);
                printf("ALRM: %d, USR1: %d, USR2: %d\n", alrm, usr1, usr2);
                exit(1);  /* exit if CRTL/C is issued */
```

(continued on next page)

Example 6–2 (Cont.) Using Timers

```
        }
        return;
}

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

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 += 1000000000;
        }
        if (t1->tv_nsec >= 1000000000) {
                t1->tv_sec++;
                t1->tv_nsec -= 1000000000;
        }
}
```

7

Asynchronous Input and Output

I/O operations on a file can be either synchronous or asynchronous. For synchronous I/O operations 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 or fails. For asynchronous I/O operations 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.

This chapter includes the following sections:

- Data Structures Associated with Asynchronous I/O, Section 7.1
- Asynchronous I/O Functions, Section 7.2
- Asynchronous I/O Examples, Section 7.3

Asynchronous I/O is most commonly used in realtime applications requiring high-speed or high-volume data collection and/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/D11 asynchronous I/O functions are designed to help meet these realtime needs.

You can perform asynchronous I/O operations using any open file descriptor.

7.1 Data Structures Associated with Asynchronous I/O

The P1003.4/D11 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, and the file descriptor on which the asynchronous I/O operation will be performed. 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_fildes;    /* File descriptor                  */
off_t           aio_offset;    /* Number of bytes in the offset    */
volatile char   *aio_buf;      /* Character pointer to buffer      */
size_t          aio_nbytes;    /* Number of bytes                  */
struct sigevent aio_sigevent;  /* Pointer to signal structure      */
int             aio_lio_opcode;/* Specifies type of I/O operation  */
```

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

7.1.1 Identifying the Location

When you call either the aio_read or aio_write function, you must specify how to locate the data to be read or to position the data to be written.

The *aio_offset* and *aio_nbytes* members of the aiocb structure provide information about the starting point and length of the data to be read or written. The *aio_buf* member provides information about where the information should be read or written in memory.

When you use the aio_write function to write to a new file, data is written to the end of a zero-length file. On additional write operations, if the O_APPEND flag is set, write operations are appended to the file in the same order as the calls to the aio_write function were made. If the O_APPEND flag is not set, write operations take place at the absolute position in the file as given by the *aio_offset* as if the lseek function were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET.

On a call to the aio_read function, the read operation takes place at the absolute position in the file as given by *aio_offset* as if the lseek function were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET.

After a successful call to queue an asynchronous write operation with O_APPEND or to an asynchronous read you must update the value of the offset with the value returned from the read or write operation. The file offset is not dynamically updated and failure to update the value of the offset can produce incorrect results.

To determine whether the read or write operation was successful, call the `aio_error` function. If the operation was successful, using the `aio_return` function to update the value of the *aio_offset* member after each successful read or write operation. See Section 7.2.3 for an example of using these functions to determine status.

7.1.2 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 5 for more information on signals and signal handling.

The *aio_sigevent* member refers to a `sigevent` structure that 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:

```
int             sigev_signo;    /* Signal sent on timer expiration */
union sigval    sigev_value;    /* Not supported - Specify as NULL    */
```

The *sigev_signo* member specifies the signal number to be sent on completion of the asynchronous I/O operation. Setting the *sigev_signo* member to a legal signal value will cause that signal to be posted when the operation is complete. Setting the value to NULL means that no signal is sent, but the error status and return value 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.

7.2 Asynchronous I/O Functions

The asynchronous I/O functions combine a number of tasks normally performed by the user during synchronous I/O operations. With synchronous I/O, the application calls the `lseek` function, performs the I/O operation, and then receives the return status.

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.
- Multiple read and write operations can be issued to multiple open file descriptors.
- Both sequential and random access devices can handle I/O requests.
- Outstanding I/O requests can be canceled.
- The process 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 7–1 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 7–1 Asynchronous I/O Functions

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

7.2.1 Reading and Writing

Asynchronous and synchronous I/O operations are logically parallel operations. The asynchronous function aio_read and aio_write perform the same I/O operations as the read and write functions. However, the aio_read or aio_write functions return control to the calling process once the the I/O is initiated rather than after the I/O operation is 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.

In contrast, 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 could be 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; or continue execution and poll for completion with aio_error; or continue execution until the signal arrives.
6. After completion, call the aio_return function to retrieve completion value.
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.

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

7.2.2 Using List-Directed Input/Output

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 control block structures for the individual read and write operations. The information contained in this structure is used during the operations. 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 immediately after the list of operations has been queued or wait until all the operations have been completed. The *mode* argument controls when the `lio_listio` function returns and can have one of the following values:

- LIO_NOWAIT, queues the operation, returns, and can 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 to the `lio_listio` function is a pointer to an array of `aiocb` structures.

The *aio_lio_opcode* member of the `aiocb` structure defines the I/O operation to be performed and the *aio_fildes* member identifies the file descriptor. The combination of these members makes it possible to specify individual read and write operations as if they had been submitted individually. Each read or write operation in list-directed asynchronous I/O has its own status, return value, and `sigevent` structure for signal delivery.

To use list-directed asynchronous I/O in your application, use the following steps:

1. Create and fill the `aiocb` control blocks.
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 upon completion of the `lio_listio` function.
4. Call 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.

7.2.3 Determining Status

Asynchronous I/O functions provide status values when the operation is successfully queued for servicing and provides both error and return 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 using the aio_error and aio_return functions in combination with each other.

The *aiocbp* argument to the aio_error or aio_return functions provide the address of an aiocb structure, unique for each asynchronous I/O operation. The aio_error function returns the error status associated with the specified *aiocbp*. The error status is the errno value that is set by the corresponding asynchronous I/O read or write operation.

The aio_error function returns EINPROGRESS if the operation is ongoing. Once the asynchronous I/O operation is complete, EINPROGRESS is not returned. A call to the aio_return function will show if the operation is successful.

Once you call the aio_return function, the system resources associated with the *aiocb* for the duration of the I/O operation are returned to the system. If the aio_return function is called for an *aiocb* with incomplete I/O, the result of the operation is undefined. To avoid losing data, use the aio_error function to ensure for completion before you call the aio_return function. Then use the aio_return function to retrieve the number of bytes read or written during the asynchronous I/O operation.

If you do not call the aio_return function, the number of asynchronous I/O resources available for use in your application is reduced by one for every completed asynchronous I/O operation that does not return data through a call to the aio_return function.

The following example shows how to use the aio_error and aio_return functions to track the progress of asynchronous write operations.

```
     .
     .
     .
return_value = aio_error(aiocbp);
if (return_value != EINPROGRESS)
     total = aio_return(aiocbp);
if (total == -1){
     errno = return_value;
     perror("aio_read");
     }
     .
     .
     .
```

In this example the variable *total* receives the number of bytes read in the operation. This variable is then be used to update the offset for the next read operation.

If you use list-directed asynchronous I/O, each asynchronous I/O operation in the list has an `aiocb` structure and a unique *aiocbp*.

7.2.4 Canceling I/O

Sometimes there is a need to 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. If, for some reason, the operation cannot be canceled, normal completion and notification takes place.

The `aio_cancel` function can return 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.

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.

7.2.5 Blocking to Completion

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 or 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 for the caller to wait for completion with a call to the aio_suspend function.

7.3 Asynchronous I/O Examples

The examples in this section demonstrate the use of the asynchronous I/O functions. Example 7–1 uses the aio functions; while Example 7–2 uses the lio_listio function.

7.3.1 Using the aio Functions

In Example 7–1, the input file (read synchronously) is copied to the output file (asynchronously) using the specified transfer size. A signal handler counts the number of completions, but is not required for the functioning of the program. A call to the aio_suspend function is sufficient.

Example 7–1 Using Asynchronous I/O

```
/*
 * Command line to build the program:
 * cc -o aio_copy aio_copy.c -g3 -non_shared -O0 -L/usr/ccs/lib \
 *      -laio -lpthreads -lmach -lc_r
 */

          /* * * *  aio_copy.c  * * * */

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

#define BUF_CNT 2     /* number of buffers */

struct sigaction sig_act;
volatile int sigcnt = 0;
```

(continued on next page)

Example 7–1 (Cont.) Using Asynchronous I/O

```
          /* * * *  Signal handler * * * */

void sig_handler(signo)
int signo;
{
        sigcnt++;
        return;
}

          /* * * * Main Routine * * * */

main(int argc, char **argv)
{
        int             in_file, out_file, rec_cnt = 0;
        typedef char   *buf_p;
        buf_p           buf[BUF_CNT];
        aiocb_t         a_write;
        size_t          xfer_size;
        int             buf_index, total = 0, ret;

        /* * * * Check number of input arguments * * * */

        if (argc < 4) {
         fprintf(stderr, "Usage: %s input-file output-file buf-size-in-Kb\n",
               argv[0]);
          exit(0);
        }

           /* * * * Open input file * * * */

        if ((in_file = open(argv[1], O_RDONLY)) == -1) {
              perror(argv[1]);
              exit(errno);
        }
        printf(" Opened Input File\n");

           /* * * * Open output file * * * */

        /* If O_APPEND is added to flags, all writes will appear at end */
        if ((out_file = open(argv[2], O_WRONLY|O_CREAT, 0777)) == -1) {
              perror(argv[2]);
              exit(errno);
        }
        printf("Opened Output File \n");

           /* * * * Calculate transfer size (# bufs * 1024) * * * */

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

           /* * * * Allocate buffers for file copy * * * */
```

(continued on next page)

Example 7–1 (Cont.) Using Asynchronous I/O

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

buf_index = 0;

   /* * * *  Init. signal action structure for SIGUSR1  * * * */

sig_act.sa_handler = (void *) sig_handler;/* handler for signal    */
sigemptyset(&sig_act.sa_mask);       /* block only current signal */
sig_act.sa_flags = 0;                /* no signal action flags    */

   /* * * *  Estab. signal handler for SIGUSR1 signal * * * */

printf("Establish Signal Handler for SIGUSR1\n");
if (ret = sigaction (SIGUSR1, /* Set action for SIGUSR1       */
      &sig_act,               /* Action to take on signal     */
       0))                    /* Don't care about old actions */
        perror("sigaction");

   /* * * * Init. aio control block (aiocb) * * * */

a_write.aio_fildes = out_file;
a_write.aio_offset = 0;                    /* write from current */
a_write.aio_sigevent.sigev_value.sival_ptr = NULL;
             /*not yet supported*/
a_write.aio_sigevent.sigev_signo = SIGUSR1;/* completion signal */

   /* * * * Copy from in_file to out_file * * * */

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

   /* * * * Read next buffer of information * * * */

        buf_len = read( in_file, buf[buf_index], xfer_size);

        if (rec_cnt) {  /* will be >1 on all but first write... */

                /* previous write completed? If not, wait */

                while (aio_error(&a_write) == EINPROGRESS) {
                        aiocb_t *wait_list = &a_write;
                        aio_suspend(1, &wait_list);
                }

   /* * * * Update total bytes written to set new file offset * * * */

                total += aio_return(&a_write);
        }

   /* * * * Check for end-of-file (won't have filled buffer) * * */

        if (buf_len <= 0)
                break;
```

(continued on next page)

Example 7–1 (Cont.) Using Asynchronous I/O

```
            /* * * * Set buffer up for next write * * * */

                 a_write.aio_nbytes = buf_len;
                 a_write.aio_buf = buf[buf_index];

                 /* if file is opened for append, can ignore offset field */

                 a_write.aio_offset = total;
                 ret = aio_write(&a_write);
                 if (ret) { perror ("aio_write"); exit(ret); }

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

                 rec_cnt++;
                 buf_index ^= 1;
        }

            /* * * * Close files * * * */

        close(in_file);
        printf("Closed Input File\n");
        close(out_file);
        printf("Closed Output File\n");
        printf("Copied: %d records, %d signals taken\n", rec_cnt, sigcnt);
}
```

7.3.2 Using the lio_listio Function

In Example 7–2 the input file is read synchronously to a specified number of output files (asynchronously) using the specified transfer size from the `lio_listio` function. After the list-directed I/O completes, it checks the return status and value for the write to each file and continues in a loop until the copy is complete.

Example 7–2 Using lio_listio in Asynchronous I/O

```
/*
 *
 * Command line to build the program:
 * cc -o lio_copy lio_copy.c -g3 -non_shared -O0 -L/usr/ccs/lib  \
 *      -laio -lpthreads -lmach -lc_r
 */
/* * * *  lio_copy.c  * * * */
```

(continued on next page)

Example 7–2 (Cont.) Using lio_listio in Asynchronous I/O

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

#define FOR_EACH_FILE   for (i = 0; i < out_cnt; i++)

#define BUF_CNT 2                       /* number of buffers */

/* * * * ------------------  Main Routine ------------------- * * * */

main(int argc, char **argv)
{
        register int    i, rec_cnt = 0, out_cnt = 0;
        char            outname[128], temp[8];
        int             in_file, out_file[AIO_LISTIO_MAX], len;
        typedef char   *buf_p;
        buf_p           buf[BUF_CNT];
        aiocb_t         a_write[AIO_LISTIO_MAX], *wait_list[AIO_LISTIO_MAX];
        size_t          xfer_size;
        int             buf_index, total[AIO_LISTIO_MAX], ret;
        struct sigevent lio_sigevent = {0,0};

        /* * * * Check the number of input arguments * * * */

        if (argc < 5) {
                fprintf(stderr, "Usage: %s in_file out_file buffsz-in-kb #-out-files\n",
                        argv[0]);
                exit(0);
        }

        /* * * * Open the input file * * * */

        if ((in_file = open(argv[1], O_RDONLY)) == -1) {
                perror(argv[1]);
                exit(errno);
        }
        printf("\tOpened Input File %s\n", argv[1]);

        /* * * * Open the output files * * * */
```

(continued on next page)

Example 7–2 (Cont.) Using lio_listio in Asynchronous I/O

```
out_cnt = atoi(argv[4]);
if ((out_cnt <=  0) || (out_cnt > AIO_LISTIO_MAX)) {
        fprintf(stderr, "Number of output files must be 1-%d.\n",
                AIO_LISTIO_MAX);
        exit(EINVAL);
}

outname[0] = '\0';
len = strlen(argv[2]);
strcpy(outname, argv[2]);

FOR_EACH_FILE {
        sprintf(&outname[len], "%d", i);
        /*
         * If O_APPEND is added to flags, all writes will appear at
         * end
         */
        if ((out_file[i] = open(outname, O_WRONLY|O_CREAT, 0777))
                        == -1) {
                perror(outname);
                exit(errno);
        }
        printf("\tOpened output file %s\n", outname);

}

/* * * * Calculate the transfer size (# bufs * 1024) * * * */

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

/* * * * Allocate buffers for file copy * * * */

for (buf_index = 0; buf_index < BUF_CNT; buf_index++) {
        buf[buf_index] = (buf_p) malloc(xfer_size);
        if (buf[buf_index] == NULL) {
                perror("malloc");
                exit(1);
        }
}

buf_index = 0;

/* * * * Init the aio control blocks and wait list * * * */
```

(continued on next page)

Example 7–2 (Cont.) Using lio_listio in Asynchronous I/O

```
        FOR_EACH_FILE {
                a_write[i].aio_fildes = out_file[i];
                a_write[i].aio_lio_opcode = LIO_WRITE;
                a_write[i].aio_sigevent.sigev_signo = 0;
                wait_list[i] = &a_write[i];
                total[i] = 0;
        }

        /* * * * Copy from in_file to out_file * * * * */

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

        /* * * * Read the next buffer of information * * * * */

                buf_len = read(in_file, buf[buf_index], xfer_size);

                if (rec_cnt) {  /* will be >1 on all but the first write... */

        /* * * * Update the bytes written to set new offset * * * * */

                        FOR_EACH_FILE {
                                errno = aio_error(&a_write[i]);
                                ret = aio_return(&a_write[i]);
                                if (ret == -1) {
                                        perror("Write error");
                                        exit(1);
                                } else {
                                        total[i] += ret;
                                }
                        }
                }

        /* * * * Check for end-of-file (won't have filled buffer) * * * */

                if (buf_len <= 0)
                        break;

        /* * * * Set the buffer up for the next write * * * * */

                FOR_EACH_FILE {
                        a_write[i].aio_nbytes = buf_len;
                        a_write[i].aio_buf = buf[buf_index];
                        /* if opened for append, ignore offset field */
                        a_write[i].aio_offset = total[i];
                }

                ret = lio_listio(LIO_WAIT, wait_list, out_cnt, &lio_sigevent);
                if (ret) /* report failure status, but don't exit yet */
                        perror("lio_listio");
```

(continued on next page)

Example 7–2 (Cont.) Using lio_listio in Asynchronous I/O

```
        /* * * Update record count, and position to next buffer * * */

                buf_index ^= 1;
                rec_cnt++;
          }

          /* * * * Close the files * * * */

          close(in_file);
          printf("\tClosed input file\n");
          FOR_EACH_FILE {
                close(out_file[i]);
          }
          printf("\tClosed output files\n");
          printf("Copied %d records to %d files\n", rec_cnt * out_cnt, out_cnt);
}
```

8

Binary Semaphores

Binary semaphores, as specified in P1003.4/D11, provide an efficient form of interprocess communication. Cooperating processes can use binary semaphores to synchronize access to resources, most commonly, shared memory. Semaphores can also 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.

This chapter includes the following sections:

- Binary Semaphores, Section 8.1
- The Semaphore Interface, Section 8.2
- Semaphore Example, Section 8.3

A binary semaphore is in either the locked or unlocked state. When a binary semaphore is unlocked, a process can successfully lock it and acquire exclusive control to any resources that the application associated with that semaphore. When a binary semaphore is locked, no process, including the current process, can lock that semaphore.

8.1 Binary Semaphores

Binary semaphores are used to protect shared resources from uncontrolled multiple access. For example, if a process has access to a shared memory object, the semaphore is set to SEM_LOCKED. If the process relinquishes access, the semaphore is set to SEM_UNLOCKED. For this reason, binary semaphores are often referred to as "mutual exclusion semaphores."

The lock operation checks to see if the resource is available or is locked by another process. If the resource is not already locked, the lock is made and the process continues execution. If the resource is locked, the process requesting the lock waits (is blocked) until the first process unlocks the resource. Several processes may be blocked waiting for a resource to become available.

The unlock operation sets the semaphore value to indicate that the resource is not locked. The waiting process, if there is one, is unblocked and it accesses the resource. Each semaphore keeps count of the number of processes waiting for access to the resource and the state of the semaphore, locked or unlocked.

Semaphores are global entities and are not associated with any particular process. In this sense, binary semaphores have no owners making it impossible to track semaphore ownership for any purpose, for example, error recovery.

Semaphore protection works only if all the processes using the shared resource cooperate by waiting for the semaphore when it is unavailable and resetting the semaphore to an unlocked state when relinquishing the resource. Since binary semaphores lack owners, there is no way to determine whether one of the cooperating processes has become uncooperative. Applications using binary semaphores must carefully detail cooperative tasks. All of the processes that share a resource must agree on which semaphore controls the resource.

POSIX P1003.4/D11 binary semaphores are persistent. The state of the individual binary semaphore is preserved after the semaphore set is no longer open. For example, a binary semaphore set containing ten semaphores may contain seven semaphores in the locked state and three in the unlocked state when the last process using the semaphore set closes it. The next time a process opens that semaphore set, it will find seven locked semaphores and three unlocked ones. For this reason, cleanup operations are advised when using binary semaphores.

Please note that because semaphores are persistent, you should call the `sem_destroy` function after a system reboot. Once this is done, you should call the `sem_mksem` function to establish a new semaphore set.

The semaphore descriptor is inherited across a `fork`. A parent process can create a semaphore set, open it, and fork. The child process does not need to open the semaphore set and can close the semaphore set if the application is done with it.

8.2 The Semaphore Interface

The functions relating to semaphores follow the same general logic as for P1003.4/D11 shared memory and memory mapped files. The sem_mksem function allocates a semaphore set, but the semaphore set is not ready for use until the the sem_open function is called.

Table 8–1 lists the functions that allow you to create and control P1003.4/D11 binary semaphores.

Table 8–1 Semaphore Functions

Function	Description
sem_close	Deallocates the binary semaphore set descriptor and makes it available for reuse
sem_destroy	Destroys a binary semaphore set
sem_getnsems	Returns the number of semaphores in the set
sem_ifwait	Locks a binary semaphore only if it can lock the semaphore without waiting for another process to unlock it
sem_mksem	Creates a new binary semaphore set
sem_open	Opens a set of binary semaphores
sem_post	Unlocks a binary semaphore
sem_wait	Locks a binary semaphore even if the process must wait for another process to unlock it

You create a semaphore set with a call to the sem_mksem function, which identifies pathname for the semaphore. A subsequent call to the sem_open function opens the semaphore set for use in the application. Semaphore operations of locking and unlocking are accomplished with calls to the sem_wait, sem_ifwait, and sem_post functions. When the application is finished with the semaphore set, the semaphore set name is deallocated with a call to the sem_close function and destroyed with a call to the sem_destroy function.

The sysconf system call can be used to determine the maximum number of semaphores that can be contained in a semaphore set sysconf(_SC_SEM_NSEMS_MAX) as well as an implementation-defined maximum number of semaphore sets sysconf(_SC_SEM_NSETS_MAX). The binsem.h header file also includes semaphore data structures and defines the possible binary semaphore states.

8.2.1 Creating and Opening a Semaphore Set

A call to the sem_mksem function allocates a specified number of semaphores in a set of semaphores and sets the initial state of the semaphores to either locked or unlocked. On a call to the sem_mksem function, the kernel opens a file with the name used in the call to the sem_mksem function. This file is created and opened with exclusive read and write access and each semaphore in the set is initialized to the state specified in the call to the sem_mksem function.

The sem_open function establishes a connection between the pathname of a binary semaphore set created by a call to the sem_mksem function and the binary semaphore set descriptor. After a call to the sem_open function, a process may reference all the binary semaphores in the specified set. These individual binary semaphores are used in subsequent calls to the sem_wait, sem_ifwait, and sem_post functions, which control access to the shared resource.

If your application consists of multiple processes that will use semaphores to synchronize access to a shared resource, each of these processes must first open the semaphore set by a call to the sem_open function. After the initial call to the sem_mksem function to establish the semaphore set and the sem_open function to link the pathname with the semaphore descriptor, each cooperating function must also call the sem_open function. If all cooperating processes are in the same working directory, just the name is sufficient. If the processes are contained in different working directories, the full pathname must be used.

On the first call to the sem_open function, the individual binary semaphores are initialized to the state specified in the call to the sem_mksem function. Subsequent calls to the sem_open function by cooperating processes do not change semaphore states.

The following example creates a semaphore set containing 25 semaphores and opens the semaphore set for use in an application.

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

int status;
int state;
binsemset_t semfd;
static char path[] = "sem_test";

state = SEM_LOCKED;
status = sem_mksem (path, 25, state);

status = sem_open (path, &semfd);
```

This example creates a semaphore set with the pathname "sem_test." The semaphore set contains an array of 25 semaphores, numbered from 0–24. The individual binary semaphores are referred to by number in subsequent operations. The initial state of each of the binary semaphores is specified in the *flags* parameter on the call to the sem_mksem function. The *flags* argument is constructed from the OR of the symbolic constants defined in the binsem.h header file. All of the semaphores in the semaphore set are created with the same initial state as specified by the *flags* argument. In the example above, each of the 25 new semaphores is initialized to the locked state. To access a previously allocated semaphore array, a process must call the sem_open function using the name of the semaphore set.

To determine the number of semaphores in a previously allocated semaphore set, use the sem_getnsems function. Pass the semaphore descriptor to the function and it returns the number of semaphores specified when the sem_mksem function was called. The sem_getnsems function would return the number 25 for the semaphore set created in the previous example.

The name of the semaphore set remains valid until the semaphore set is removed with a call to the sem_destroy function.

8.2.2 Locking and Unlocking Binary Semaphores

After you have created the semaphore set with a call to the sem_mksem function and opened the semaphore set with a call to the sem_open function, you can use the sem_wait, sem_ifwait, and sem_post functions to lock and unlock individual semaphores.

The use of semaphores to share resources among processes will work only if processes unlock the resource immediately after they finish using it. As you code your application, take care not to do an unlock operation on a semaphore you have not locked.

To lock a semaphore, you can use either the sem_wait or sem_ifwait function. The sem_wait function locks the specified binary semaphore. However, if the semaphore is already locked, the process is blocked (sleeps) until either the semaphore is unlocked or the process is interrupted by a signal.

If you want to make certain that the process is not blocked while waiting for a semaphore to become available, use the sem_ifwait function. The sem_ifwait function will lock the specified semaphore if, and only if, it can do so without waiting. That is, the specified semaphore must be available at the time of the call to the sem_ifwait function. If not, the sem_ifwait function returns a –1 and errno is set to EAGAIN.

Example 8–1 locks the 10th binary semaphore in the set by using the sem_ifwait function.

Example 8–1 Locking a Binary Semaphore

```
#include <unistd.h>
#include <sys/types.h>
#include <binsem.h>

int status;
int state;
binsemset_t semfd;
static char path[] = "sem_test";

status = sem_ifwait (semfd, 10);
```

The sem_post function unlocks the specified semaphore. Any process with access to the semaphore can call the sem_post function and unlock a semaphore. If more than one process is waiting for the binary semaphore, only the highest priority process is allowed access to the semaphore.

8.2.3 Priority Inversion with Semaphores

Process priority inversion can occur when using semaphores to lock a resource shared by processes of different priorities. If a low-priority process locks a semaphore to control access to a resource and a higher-priority process is waiting for the same resources, the higher-priority process is delayed. If the lower-priority process is then preempted by a medium-priority process, the higher-priority process is further delayed. In this situation, the higher-priority process is delayed while waiting for a resource locked by lower-priority processes, and the result is priority inversion.

Since semaphores are global in nature and lack owners, there is no mechanism for priority inheritance with binary semaphores. Therefore, semaphore locks are separate from process priorities. Care must be exercised when designing the use of semaphores in your application.

8.2.4 Closing a Semaphore Set

When an application is finished using the semaphore set, it should deallocate the semaphore descriptor with a call to the sem_close function. The semaphore set name is disassociated from the process, but the name and associated data structures remain in the system. If needed, the semaphore set can be reopened for use through a call to the sem_open function. Since semaphores are persistent, the state of the semaphores in the set is preserved, even though the semaphore set is closed. When you reopen the semaphore set, each semaphore

will be in the state it was when the set was closed, unless altered by another process using the same semaphore set.

To remove a semaphore set, its name, and associated data structures, use the sem_destroy function. In this case, a call to the sem_open function to open the semaphore set will fail. Once a semaphore set is destroyed, it cannot be opened again and the state of the semaphores in the set is not preserved. The state of the semaphore set is preserved until the last call to the sem_close function. You can, however, call the sem_mksem function and use the same semaphore set name to create an entirely new semaphore set by the same name.

The last process in your application to use the semaphore set should close it with a call to the sem_close function. Then, destroy the semaphore set with a call to the or sem_destroy function. If a process is waiting for a semaphore and you close or destroy the semaphore set, the waiting process will still have access to the semaphore set.

As with other interprocess communication methods, 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.

When the controlling process is finished using the semaphore set, remove the semaphores from memory as follows:

```
      .
      .
      .
status = sem_close(semfd);
status = sem_destory(path);
      .
      .
      .
```

8.3 Semaphore Example

It is important that two processes not write to the same area of shared memory at the same time. Binary semaphores protect access to resources such as shared memory. Before writing to a shared memory region, a process can lock the semaphore to prevent another process from accessing the region until the write operation is completed. When the process is finished with the shared memory region, the process unlocks the semaphore and frees the shared memory region for use by another process.

Example 8–2 uses semaphores to ensure that two processes, writer and reader, have exclusive, alternating access to a shared memory region.

Example 8–2 consists of two programs, both of which open the shared-memory object. The writer.c program creates the semaphore set with a call to the sem_mksem function, then opens the semaphore set. The reader.c program opens the semaphore set previously created by the writer.c program. Because the writer.c program creates the semaphore set, writer.c needs to be executed before reader.c.

Example 8–2 Using Semaphores and Shared Memory

```
/*
** The WRITER and READER are the examples that use semaphores to
** ensure that the writer and reader processes have exclusive,
** alternating access to the shared-memory region.
*/

/**********  writer.c  ***********/

#include <unistd.h>
#include <binsem.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/fcntl.h>

char shm_fn[] = "my_shm";
char sem_fn[] = "my_sem";

/**** WRITER ****/

main(){
  caddr_t shmptr;
  unsigned int mode;
  int shmdes, index;
  binsemset_t semdes;
  int SHM_SIZE;

  mode = S_IRWXU|S_IRWXG;

  /* Open the shared memory object */

  if ( (shmdes = shm_open(shm_fn,O_CREAT|O_RDWR|O_TRUNC, mode)) == -1 ) {
     perror("shm_open failure");
     exit();
   }

  /* Preallocate a shared memory area */

  SHM_SIZE = sysconf(_SC_PAGE_SIZE);
```

(continued on next page)

Example 8–2 (Cont.) Using Semaphores and Shared Memory

```
  if(ftruncate(shmdes, SHM_SIZE) == -1){
    perror("ftruncate failure");
    exit();
  }
  if((shmptr = mmap(0, SHM_SIZE, PROT_WRITE|PROT_READ, MAP_SHARED,
                shmdes,0)) == (caddr_t) -1){
    perror("mmap failure");
    exit();
  }
  /* Create a semaphore set */
  if(sem_mksem(sem_fn,1,SEM_LOCKED) == -1){
   perror("sem_mksem failure");
   exit();
  }
if(sem_open(sem_fn, &semdes) == -1){
  perror("sem_open failure");
  exit();
}
/* Lock the semaphore */
if(!sem_wait(semdes,0)){
   /* Access to the shared memory area */
   for(index = 0; index < 100; index++){
      printf("write %d into the shared memory shmptr[%d]\n", index*2, index);
      shmptr[index]=index*2;
      }
 }
 /* Release the semaphore lock */
 sem_post(semdes, 0);
 munmap(shmptr, SHM_SIZE);
 /* Close the shared memory object */
 close(shmdes);
 /* Delete the shared memory object */
 shm_unlink(shm_fn);
 /* Close the Binary Semaphore set */
```

(continued on next page)

Example 8–2 (Cont.) Using Semaphores and Shared Memory

```
  sem_close(semdes);

  /* Destroy the Binary Semaphore set */

  sem_destroy(sem_fn);
}

/*********************************************************************
*********************************************************************/

/**********  reader.c  ***********/

#include <sys/types.h>
#include <sys/mman.h>
#include <binsem.h>
#include <sys/stat.h>
#include <sys/fcntl.h>

char shm_fn[] = "my_shm";
char sem_fn[] = "my_sem";

/**** READER ****/

main(){
  caddr_t shmptr;
  int shmdes, index;
  binsemset_t semdes;
  int SHM_SIZE;

  /* Open the shared memory object */

  SHM_SIZE = sysconf(_SC_PAGE_SIZE);

  if ( (shmdes = shm_open(shm_fn, O_RDWR, 0)) == -1 ) {
     perror("shm_open failure");
     exit();
   }

  if((shmptr = mmap(0, SHM_SIZE, PROT_WRITE|PROT_READ, MAP_SHARED,
               shmdes,0)) == (caddr_t) -1){

     perror("mmap failure");
    exit();
  }

 /* Open the Binary Semaphore set */

 if(sem_open(sem_fn, &semdes) == -1){
   perror("sem_open failure");
   exit();
 }
```

(continued on next page)

Example 8–2 (Cont.) Using Semaphores and Shared Memory

```
 /* Lock the semaphore */
 if(!sem_wait(semdes,0)){
  /* Access to the shared memory area */
   for(index = 0; index < 100; index++)
        printf("The shared memory shmptr[%d] = %d\n", index,shmptr[index]);
  /* Release the semaphore lock */
   sem_post(semdes, 0);
  }
  munmap(shmptr, SHM_SIZE);
  /* Close the shared memory object */
  close(shmdes);
  /* Close the Binary Semaphore set */
  sem_close(semdes);
  sem_destroy(sem_fn);
}
```

9
Messages

Message queues are user-defined data structures that specify the length and type of message and carry the message text.[1] 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.

Message queues work by exchanging data in buffers, which means any number of other processes, regardless of whether they are related, can communicate through message queues. If a process has all the right access data, it can send or receive messages through the queue. The receiving process can also select incoming messages of a specified type.

This chapter includes the following sections:

- Data Structures Associated with Messages, Section 9.1
- The Message Interface, Section 9.2
- Message Queue Example, Section 9.3

9.1 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:

[1] This implementation uses System V messages, rather than POSIX messages.

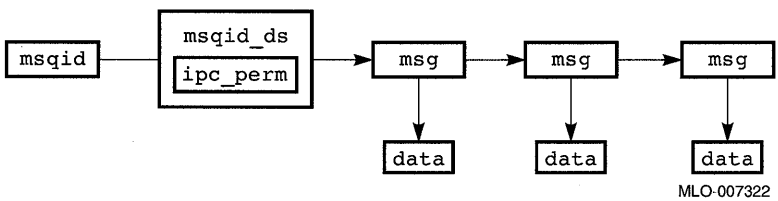
```
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>

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 9–1 shows how message structures relate to each other.

Figure 9–1 Representation of Message Data Structures



MLO-007322

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.

9.1.1 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                                */
```

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.

9.1.2 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 <unistd.h>
#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).

9.2 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 9–1 lists the functions that allow you to create and control messages.

Table 9–1 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

9.2.1 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 to take, 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, the `msgget` function checks permissions.

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

The *key* and the message queue ID are directly related — processes sharing the same *key* will also share the same message queue 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.

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 message queue. If the calling process is not the owner and is not in the group, 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.

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

```
#include <unistd.h>
#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.

Whenever you create a new message queue with the `msgget` 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 message queue 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 message queue. When you use the IPC_PRIVATE key, it is not necessary to use either the IPC_CREAT or IPC_EXCL flags.

When processes are unrelated, you must call the `msgget` 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 message queue 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 message queue ID or returns the existing one. If you want the call to fail if a message queue already exists, use the IPC_EXCL flag.

9.2.2 Using The ftok Function

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 `msgget` function.

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 System V IPC functions.

```
keyofmine = ftok ("/usr/users/data1", 'X');
```

If unrelated processes create the message queue or if multiple message queues are required, 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 message queue, or if you think the creating process may execute earlier than the other communicating processes, call the `ftok` function.

The following example calls the msgget function to create a new message queue. The call creates a new message queue, but the ftok call returns a key based on the */usr/users/data3* pathname and the character *X*. As other processes call the msgget function, *msq_id* is returned for them to use in other message calls.

```
msq_id = msgget (ftok ("/usr/users/data3", 'X'), IPC_CREAT|0600);
```

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

```
mykey = msgget (ftok ("/usr/users/data3", 'X'), IPC_CREAT|IPC_EXCL|0600);
```

Using these flags together ensures that an IPC key is created if none previously existed.

9.2.3 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 a *msgflg* of 0, then the sending process sleeps if the message cannot be sent to the specified queue. When the queue is full, the msgsnd function will sleep until other messages have been removed from the queue and space is available. When 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 there 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.

9.2.4 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 9–2 describes the command control flags.

Table 9–2 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 supply only 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.

9.3 Message Queue Example

Example 9–1 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 9–1 Using Message Queues

```
#include <unistd.h>
#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 9–1 (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");
}
```

10
Pipes

A pipe is a structure that facilitates interprocess communication, providing 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

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, you have a two-way pipe. If you open the pipe with either read or write, 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`)

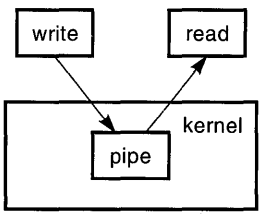
Subsequent file descriptors are allocated sequentially. A call to the `pipe` function, for example, returns two additional file descriptors as follows:

- 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, 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



MLO-007326

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.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);
```

(continued on next page)

Example 10–1 (Cont.) Creating a Child Process and a Pipe

```
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.

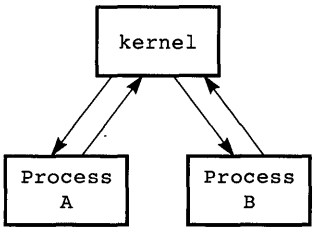
10.1.2 Redirecting stdin, stdout, stderr to Pipes

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`.

Figure 10–2 Two-Way Pipe



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

The popen function creates 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)[1] 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.

A named pipe provides a one-way flow of data between unrelated processes. Named pipes follow many of the same rules as regular pipes. 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.

[1] Named pipes are sometimes referred to as first-in, first-out pipes or FIFOs.

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.

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.

A

Summary of Differences Between P1003.4/D10 and P1003.4/D11

DEC OSF/1 V1.0 realtime software followed POSIX 1003.4 Draft 10; Version 2.0 follows POSIX 1003.4 Draft 11. If you are tracking the evolution of POSIX realtime standards, a summary of the differences between Draft 10 and Draft 11 might be useful. This section summarizes those differences.

Source compatibility between the two versions exists for realtime scheduling, clocks and timers, memory-locking, and signal operations if POSIX_4D10 is defined for compilations. You can define POSIX_4D10 on the command line as follows:

```
% cc my_program -DPOSIX_4D10
```

A.1 Scheduling Priorities and Policies

The following technical changes apply for realtime scheduling priorities and policies:

- These function and field names were changed:
  1. The `sched_set_sched_param` function became the `sched_setparam` function.
  2. The `sched_get_sched_param` function became the `sched_getparam` function.
  3. The *priority* field of the `sched_param` structure became the the *sched_priority* field of the `sched_param` structure
- The `sched_get_rr_interval` function now takes a *pid*, and returns a status of 0 if it succeeds and –1 (and sets `errno`) if it fails. This allows for support of a per-process quantum for the SCHED_RR scheduling policy.

A.2 Clocks and Timers

The following technical changes apply for realtime clocks and timers:

- Users are now required to include the `time.h` header file, which includes the earlier `timers.h` header file.
- The return protocol for the `nanosleep` function was changed to use (–1, [EINTR]) for interrupted sleep.
- The maximum value limits for timers, clocks, and the `nanosleep` function were removed and now specify the full range of a *time_t* data type.
- The `timer_getres` and `nanosleep_getres` functions were removed. If timers and the `nanosleep` function have different resolutions from the realtime clock, it is implementation-defined.
- The *maxval* argument was removed from the `clock_getres` function.
- The *abstime* boolean was changed to a flag word containing an TIMER_ ABSTIME flag.

A.3 Memory Locking

The following technical changes apply for memory locking:

- The standard now states that if a failure occurs during memory locking, no memory is locked.
- The range locked is now described as *addr* through *(addr + len) –1*.

A.4 Asynchronous I/O

The following technical changes apply for asynchronous I/O:

- All parameters to the asynchronous I/O functions are now located in the `aiocb` structure.
- The functions of the `aiocb` and `liocb` structures are combined into a single data structure. Specifically, the following changes were made to the `aiocb` structure:
  1. The *aio_handle* element was removed. (The `aio_error` and `aio_return` functions take a pointer to the `aiocb` structure as an argument instead of the handle.)
  2. The *aio_whence* field was eliminated. The *aio_offset* field must be explicitly maintained.
  3. The *aio_lio_opcode* element was added.

4. The `liocb` structure was eliminated and the *lio_opcode* field was moved into the *aio_lio_opcode* element of the `aiocb` structure.

- The `aio_return` function has been defined such that it should be called only once after the operation completes.
- The `lio_listio` function takes a list of `aiocb` structures instead of a list of `liocb` structures.

B

DEC OSF/1 Realtime Functional Summary

Appendix B 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/D11. The tables given in this appendix serve as a guide in application development, but you may need to consult the online reference pages for additional information or pointers to additional functions and commands.

The function tables are arranged according to the following categories:

- Process Control
- P1003.4/D11 Priority Scheduling
- P1003.4/D11 Clocks
- Date and Time Conversion
- P1003.4/D11 Timers
- BSD Clocks and Timers
- P1003.4/D11 Memory Locking
- System V Memory Locking
- P1003.4/D11 Asynchronous I/O
- BSD Synchronous I/O
- System V Messages
- P1003.4/D11 Shared Memory
- P1003.4/D11 Semaphores
- POSIX Signal Control
- sigsetops Primitives
- Process Ownership
- Input and Output

- Device Control
- System Database

Table B–1 Process Control

Function	Purpose
`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
`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

Table B–2 P1003.4/D11 Priority Scheduling

Function	Purpose
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_getparam	Returns the scheduling priority of a specified process
sched_setscheduler	Sets the scheduling policy and priority of a specified process
sched_setparam	Sets the scheduling priority of a specified process
sched_yield	Yields execution to another process

Table B–3 P1003.4/D11 Clocks

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

Table B–4 Date and Time Conversion

Function	Purpose
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

(continued on next page)

Table B–4 (Cont.) Date and Time Conversion

Function	Purpose
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

Table B–5 P1003.4/D11 Timers

Function	Purpose
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 to either an offset from the current clock setting or an absolute value

Table B–6 BSD Clocks and Timers

Function	Purpose
getitimer	Returns the amount of time before the timer expires and the repetition value
gettimeofday	Gets the time of day

(continued on next page)

Table B–6 (Cont.) BSD Clocks and Timers

Function	Purpose
setitimer	Sets the value of the specified timer
settimeofday	Sets the time of day

Table B–7 P1003.4/D11 Memory Locking

Function	Purpose
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

Table B–8 System V Memory Locking

Function	Purpose
plock	Locks and unlocks a process, text, or data in memory

Table B–9 P1003.4/D11 Asynchronous I/O

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

(continued on next page)

Table B–9 (Cont.) P1003.4/D11 Asynchronous I/O

Function	Purpose
lio_listio	Initiates a list of requests

Table B–10 BSD Synchronous I/O

Function	Purpose
fcntl	Performs controlling operations on the specified open file
fsync	Writes changes to a file to permanent storage—saves all modified data
sync	Updates all file systems—all information in memory that should be on disk is written out

Table B–11 System V Messages

Function	Purpose
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

Table B–12 P1003.4/D11 Shared Memory

Function	Purpose
shm_open	Opens a shared memory object, creating the object if necessary
shm_unlink	Removes a shared memory object created by a call to shm_open

Table B–13 P1003.4/D11 Semaphores

Function	Purpose
sem_close	Deallocates the specified binary semaphore set descriptor
sem_destroy	Removes or destroys the specified binary semaphore set
sem_getnsems	Returns the number of binary semaphores in the binary semaphore set
sem_ifwait	Conditionally performs a semaphore lock on a binary semaphore
sem_mksem	Creates a new binary semaphore set
sem_open	Opens a set of binary semaphores
sem_post	Releases a locked binary semaphore
sem_wait	Performs a semaphore lock on a binary semaphore

Table B–14 POSIX Signal Control

Function	Purpose
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

Table B–15 sigsetops Primitives

Function	Purpose
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

(continued on next page)

Table B–15 (Cont.) sigsetops Primitives

Function	Purpose
`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

Table B–16 Process Ownership

Function	Purpose
`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

Table B–17 Input and Output

Function	Purpose
`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

(continued on next page)

Table B–17 (Cont.) Input and Output

Function	Purpose
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

Table B–18 Device Control

Function	Purpose
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

Table B–19 System Database

Function	Purpose
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

(continued on next page)

Table B–19 (Cont.) System Database

Function	Purpose
`getpwuid`	Returns user information when passed a user ID

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Y

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[a]	————	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Guide to Realtime Programming
AA-PS33B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ----------

Cut Along Dotted Line

Reader's Comments

DEC OSF/1
Guide to Realtime Programming
AA-PS33B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line