Digital UNIX

Guide to Realtime Programming

Order Number: AA–PS33D–TE

March 1996

This guide describes how to use POSIX 1003.1b functions to write realtime applications that run on Digital UNIX systems. This guide is intended for experienced application programmers.

Operating System and Version: Digital UNIX Version 4.0 or higher

Digital Equipment Corporation
Maynard, Massachusetts

Revised, March 1993
Revised, August 1994
Revised, March 1996

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1992, 1993, 1994, 1996.

All Rights Reserved.
Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation: CDA, DDIF, DDIS, DEC, DECdts, DECnet, DECstation, DECsystem, DECthreads, DEC OSF/1, DECUS, DECwindows, Digital UNIX, DTIF, MASSBUS, MicroVAX, PrintServer 40, Q–bus, ReGIS, ULTRIX, ULTRIX Mail Connection, ULTRIX Worksystem Software, UNIBUS, VAX DOCUMENT, VT, XUI, and the DIGITAL logo.

The following are third-party trademarks:

X Window System, Version 11 and its derivations (X, X11, X Version) are trademarks of the Massachusetts Institute of Technology.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Ltd.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

POSTSCRIPT® and Adobe are registered trademarks of Adobe Systems Incorporated.

X/Open is a trademark of the X/Open Company, Ltd. in the U.K. and other countries.

System V and AT&T are registered trademarks of American Telephone & Telegraph Company in the U.S. and other countries.

BSD is a trademark of University of California, Berkeley.

NFS is a trademark of Sun Microsystems, Inc.

All other trademarks are registered trademarks are property of their respective holders.

This document is available on CD–ROM

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

2 The Digital UNIX Scheduler

3 Shared Memory

4 Memory Locking

5 Signals

6 Clocks and Timers

10 Messages

11 Realtime Performance and System Tuning

A Digital UNIX Realtime Functional Summary

Index

Examples

Figures

Tables

About This Guide

This guide is designed for programmers who are using systems running Digital UNIX® and want to use realtime functions. 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 1003.1b (formerly POSIX 1003.4 Draft 14) functions in combination with other system and library functions to write realtime applications. This guide does not attempt to teach programmers how to write applications.

The audience for this guide is application programmers or system engineers who are 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 guide does not present function syntax or reference information. The online reference pages present syntax and explanations of POSIX 1003.1b functions.

New and Changed Features

This guide has been revised to document all of the changes to realtime programming that are part of the current release, including conformance to the POSIX 1003.1b standard. It includes:

- A completely revised chapter about realtime signals, Chapter 5
- New examples of asynchronous I/O in Chapter 7
- The following new chapters:
  - Chapter 8, File Synchronization
  - Chapter 11, Realtime Performance and System Tuning

Structure of this Guide

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

- Chapter 1, Introduction to Realtime Programming, describes the realtime functionality supported by the Digital UNIX operating system.
- Chapter 2, The Digital UNIX Scheduler, describes the use of P1003.1b functions to determine and set priority for processes in your application. This chapter also describes the priority scheduling policies provided by the Digital UNIX operating system.
- Chapter 3, Shared Memory, describes the creation and use of P1003.1b shared memory for interprocess communication.
- Chapter 4, Memory Locking, describes the use of P1003.1b functions for locking and unlocking memory.
- Chapter 5, Signals, describes the creation and use of POSIX 1003.1b realtime signals for interprocess communication.
- Chapter 6, Clocks and Timers, describes use of P1003.1b functions for constructing and using high-resolution clocks and timers.
- Chapter 7, Asynchronous Input and Output, describes the use of P1003.1b functions for asynchronous input and output.
- Chapter 8, File Synchronization, describes the use of POSIX 1003.1b functions for synchronized input and output.
- Chapter 9, Semaphores, describes the creation and use of P1003.1b semaphores for interprocess synchronization. An example illustrates how to use semaphores and shared memory in combination.
- Chapter 10, Messages, describes the creation and use of message queues for interprocess communication and synchronization in realtime applications.
- Chapter 11, Realtime Performance and System Tuning, describes tuning techniques for improving realtime system performance.
- Appendix A, Digital UNIX Realtime Functional Summary, provides tables of commands and functions useful for realtime application development.

Related Documents

The following documents are relevant to writing realtime applications:

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

To view online reference pages for P1003.1b functions, use the `man` or `whatis` command.

The printed version of the Digital UNIX 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	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

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, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

Using the man Command

Additional information about system commands and library functions (including P1003.1b functions) 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 P1003.1b functions discussed in this guide. If you need help using the `man` command, use this 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 guide:

Convention	Meaning
%	The default user prompt is the user's system name followed by a right angle bracket. In this guide, 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.

Convention	Meaning
...	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 any comments and suggestions you have on this and other Digital UNIX manuals. You can send your comments in the following ways:

- FAX: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on line in the following location:

  `/usr/doc/readers_comment.txt`
- Mail:
  Digital Equipment Corporation
  UEG Publications Manager
  ZK03-3/Y32
  110 Spit Brook Road
  Nashua, NH 03062-9987

  A Reader's Comments form is located in the back of each printed manual. The form is postage paid, if mailed in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.

- The version of Digital UNIX that you are using. For example, Digital UNIX Version 4.0.
- If known, the type of processor that is running the Digital UNIX software. For example, AlphaServer 2000.

The Digital UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

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, Digital UNIX provides features that facilitate efficient interprocess communication and synchronization, a fast interrupt response time, asynchronous input and output (I/O), memory management functions, file synchronization, and facilities for satisfying timing requirements. Digital UNIX provides realtime facilities as part of the standard Digital UNIX kernel and optional subsets.

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 predictability of our technology.

This chapter includes the following sections:

- Realtime Overview, Section 1.1
- Digital UNIX Realtime System Capabilities, Section 1.2
- Process Synchronization, Section 1.3
- POSIX Standards, Section 1.4
- Enabling Digital UNIX Realtime Features, 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 a different definition 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 meteorologist 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. Realtime programmers must know the *worst-case* time requirements so that they can design an application that will always meet worst-case deadlines.

Realtime systems also use techniques to reduce the hazards 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 one 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 Digital UNIX Realtime System Capabilities

The Digital UNIX operating system supports facilities to enhance the performance of realtime applications. Digital UNIX 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 Digital UNIX operating system can respond to external events regardless of the impact on other executing tasks or processes.

Realtime applications written to run on the Digital UNIX 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
- File synchronization
- Queued realtime signals
- Process communication facilities

All of these realtime facilities work together to form the Digital UNIX 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 must 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 can interact 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 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 delay 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 Digital UNIX kernel with realtime preemption enabled, 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 Digital UNIX kernel can break out of kernel mode to honor the preemption request.

A preemptive kernel supports the concept of process synchronization with the ability to respond quickly to interrupts while maintaining data integrity. The kernel employs mechanisms to protect the integrity of kernel data structures, and defines restrictions on when the kernel can preempt execution.

The maximum process preemption latency for a preemptive kernel is the exact 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 only 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 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 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.

Figure 1–1 Nonpreemptive Kernel

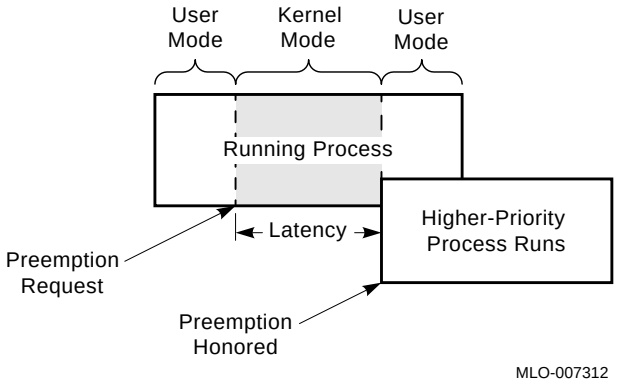
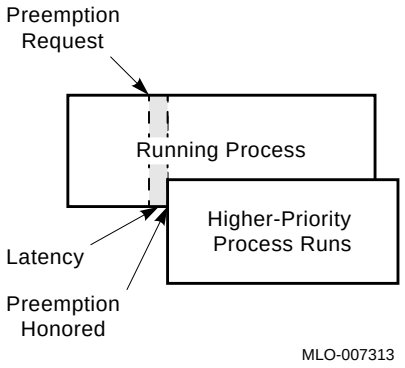


Figure 1–2 Preemptive Kernel



1.2.2 Fixed-Priority Scheduling Policies

The scheduler determines how CPU resources are allocated to executing processes. Each process has a priority that associates the process with a run queue. Each process begins execution 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 the process's priority and scheduling policy to control use of system resources.

If the realtime subset is installed on your system, the Digital UNIX 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. You have access to the realtime scheduling interface only if you have installed the realtime subset.

The Digital UNIX 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 (round-robin and first-in, first-out), 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 2–4 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 begins 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 P1003.1b 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 allotted time (quantum) is complete or the process is preempted by another, higher-priority process. When a process reaches the end of its quantum, it takes its place at the end of the run queue for processes that have the same priority. 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 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 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 timer.

The Digital UNIX system clock provides the timing base for realtime 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. An option is also available for using a high-resolution clock (see Section 6.1.5).

Clock and timer functions allow you to retrieve and set the system clock, suspend execution for a period of time, provide high-resolution timers, and use asynchronous signal and realtime signal notification.

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

1.2.4 Memory Locking

Memory locking is one of the primary tools available to the Digital UNIX realtime application designer for reducing 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 for guaranteeing 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 P1003.1b 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 memory-locking functions, refer to Chapter 4.

1.2.5 Asynchronous I/O

Digital UNIX asynchronous I/O allows the calling process to resume execution immediately after an I/O operation is queued, in contrast to synchronous I/O. Asynchronous I/O 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.

Notification of asynchronous I/O completion is optional and can be done without the overhead of calling signal functions by using the `aiocb` data structure, providing faster interprocess communication.

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

1.2.6 Synchronized I/O

Synchronized I/O may be preferable to asynchronous I/O when the integrity of data and files is critical to an application. Synchronized output assures that data that is written to a device is actually stored there. Synchronized input assures that data that is read from a device is a current image of data on that device. For both synchronized input and output, the function does not return until the operation is complete and verified.

Synchronized I/O offers two separate options:

- Ensure integrity of file data and file control information
- Ensure integrity of file data and only that file control information which is needed to access the data

For information on using synchronized I/O features, refer to Chapter 8.

1.2.7 Realtime Interprocess Communication

Interprocess communication (IPC) 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 use additional communication mechanisms for passing data.

Digital UNIX 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. Digital UNIX supports P1003.1b shared memory.

- Signals, Chapter 5

  Signals provide a means to communicate to a large number of processes. Signals for timer expiration and asynchronous I/O completion use a data structure, making signal delivery asynchronous, fast, and reliable. Posix 1003.1b realtime signals include:

  - A range of priority-ordered, application-specific signals from SIGRTMIN to SIGRTMAX.

- A mechanism for queueing signals for delivery to a process.
- A mechanism for providing additional information about a signal to the process to which it is delivered.
- Features that allow efficient signal delivery to a process when a POSIX 1003.1b timer expires, when a message arrives on an empty message queue, or when an asynchronous I/O operation completes.
- Functions that allow a process to respond more quickly to signal delivery.

- Semaphores, Chapter 9

  Semaphores are most commonly used to control access to system resources such as shared memory regions. Digital UNIX supports P1003.1b semaphores.

- Messages, Chapter 10

  Cooperating processes can communicate by accessing system-wide message queues. The message queue interface is a set of structures and data that allows processes to send and receive messages to a message queue.

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-level 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.

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 period of 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 confirmation or rejection of a transaction.

1.3.1 Waiting for a Specified Period of Time or an Absolute 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 these methods:

- 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 that controls metered access to the shared resource. **Metered access** means that up to a specified number of processes can access the resource simultaneously. Metered access is achieved through the use of counting 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 semaphore 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. 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 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 9.

1.3.3 Waiting for Communication

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

Interprocess communication facilitates application control over the following:

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

Section 1.2.7 introduced the forms of interprocess communication available to the realtime application designer. For further information on using interprocess communication facilities refer to Chapters 3, 5, 9, and 10.

1.3.4 Waiting for Another Process

Waiting for another process means waiting until that 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, refer to Chapter 5, and for information on using semaphores, refer to Chapter 9.

1.3.5 Realtime Needs and System Solutions

Table 1–1 summarizes the common realtime needs and the solutions available through P1003.1b functions and the Digital UNIX 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 and System Solutions

Realtime Need	Realtime System Solution
Change the availability of a process for scheduling	Use scheduler functions to set the scheduling policy and priority of the process
Keep critical code or data highly accessible	Use 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, or use asynchronous I/O
Perform higher throughput or special purpose I/O	Use asynchronous I/O
Ensure that data read from a device is actually a current image of data on that device, or that data written to a device is actually stored on that device	Use synchronized I/O
Share data between processes	Use shared memory, or use memory-mapped files
Synchronize access to resources shared between cooperating processes	Use 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, use semaphores, or 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 support creation of 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 Digital UNIX 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.

Formal standards to date include POSIX 1003.1 for basic system interfaces, and POSIX 1003.13 for assertions a vendor must test to claim conformance to POSIX 1003.1. Draft standards are not formal standards. They are working documents that will evolve over time into formal standards.

POSIX standards for the programming interface (P1003.1), POSIX threads (P1003.1c), and realtime programming extensions (P1003.1b) are supported by Digital UNIX.

- 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 1003.1c defines a set of thread functions that can be used in the design and creation of multithreaded realtime applications in the Digital UNIX environment.
- POSIX 1003.1b provides support for functions that support the needs of realtime applications, such as enhanced interprocess communication, scheduling and memory management control, asynchronous I/O operations, and file synchronization.

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 backward compatibility and formal support for older versions (drafts) will be less than that normally expected from a stable Digital product.

An application that strictly conforms to any combination of these standards can be developed on one system and then ported to another system that supports the same POSIX standards. (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 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 Enabling Digital UNIX Realtime Features

The files that make up the realtime facility are included with the base system software, and are installed when you choose the realtime option during installation. This provides extended features such as realtime and symmetric multiprocessing.

Note

If you install Digital UNIX with the default options, realtime preemption is disabled. See the *Installation Guide* for complete installation instructions.

1.6 Building Realtime Applications

To build a Digital UNIX 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, for example _POSIX_C_SOURCE, that are required to compile programs containing POSIX 1003.1b functions. If you need to exclude any of the standards definitions provided by the `unistd.h` header file, you should explicitly define those standards macros in the source file or on the compilation command line.

As a general rule, use specific definitions in your application *only* if your application must *exclude* certain definitions related to other unneeded standards, such as XPG3. For example, if you defined _POSIX_C_SOURCE (`#define _POSIX_C_SOURCE 199506L`) in your application, you would get *only* the definitions for POSIX 1003.1b and other definitions pulled in by that definition, such as, POSIX 1003.1.

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.

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 for the linker to use to locate the libraries. You can specify the shareable realtime library, `librt.so`, or the nonshareable library, `librt.a`.

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 if you do not want to use the default.

The following example specifies that the 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, include the threads library on the compile command line. The following example shows the specification required if your application uses asynchronous I/O.

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

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
```

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
The Digital UNIX Scheduler

On a single-processor system, only one process's code is executing at a time. Which process has control of the CPU is decided by the scheduler. The scheduler chooses which process should execute based on priority, therefore the highest priority process will be the one that is executing.

The scheduler has 64 priority levels; every process on the system is at one of these priority levels. The priority level at which a process is allowed to execute, its scheduling interactions with other processes at that level, and if or how it moves between priority levels are determined by its scheduling policy.

Digital UNIX provides two interfaces to the scheduler: the traditional UNIX timesharing interface (nice) and the POSIX 1003.1b realtime execution scheduling interface.

This chapter includes the following sections:

- Scheduler Fundamentals, 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 Scheduler Fundamentals

The terms and mechanisms needed to understand the Digital UNIX scheduler are explained in the following sections.

2.1.1 Schedulable Entities

The scheduler operates on threads. A thread is a single, sequential flow of control within a process. Within a single thread, there is a single point of execution. Most traditional processes consist of a single thread.

Using DECthreads, Digital's multithreading run-time library, a programmer can create several threads within a process. Threads execute independently, and within a multithreaded process, each thread has its own point of execution.

The scheduler considers all threads on the system and runs the one with the highest priority.

2.1.2 Thread States

Every thread has a state. The thread currently executing in the CPU is in the "run" state. Threads that are ready to run are in the "runnable" state. Threads that are waiting for a condition to be satisfied are in the "wait" state. Examples of conditions a thread may be waiting for are a signal from another process, a timer expiration, or an I/O completion.

The scheduler selects the highest priority thread in the running or runnable state to execute on the CPU. Thus the running thread will always be the one with the highest priority.

2.1.3 Scheduler Database

All runnable threads have entries in the scheduler database. The scheduler database is an array of 64 lists, one list for each priority level.

The scheduler orders the processes on each priority level list by placing the process that should run next at the head of the list, and the process that should wait the longest to run at the tail of the list.

2.1.4 Quantum

Each thread has a value associated with it, known as a quantum, that defines the maximum amount of contiguous CPU time it may use before being forced to yield the CPU to another thread of the same priority.

A thread's quantum is set according to its scheduling policy. The goal of the timesharing policy is to choose a short enough time so that multiple users all think the system is responsive while allowing a long enough time to do useful work. Some realtime policies have an infinite quantum since the work to be done is considered so important that it should not be interrupted by a process of equal priority.

2.1.5 Scheduler Transitions

A new thread is selected to run when one of the following events occurs:

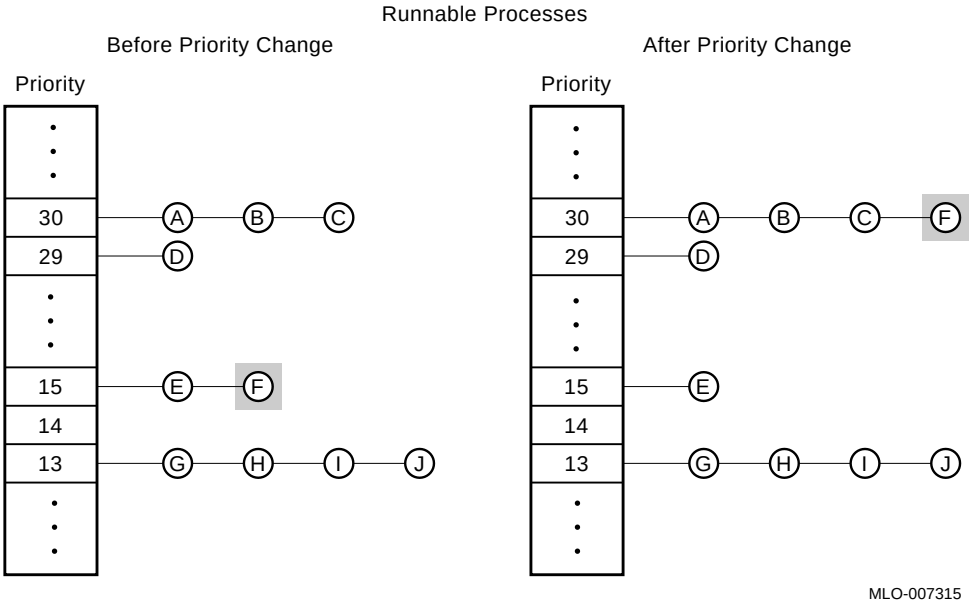
- The running process enters a wait state
- A higher priority process becomes runnable
- A process changes scheduling policy
- The quantum of the running process expires

When an event occurs, the scheduler updates the scheduler database. If a thread in the database now has priority higher than that of the currently running thread, the current thread is preempted, placed into the scheduler database, and the highest priority thread is made the running thread. A scheduler that works in this manner is known as a **preemptive priority scheduler**.

When a thread is placed into a priority list in the scheduler database, it is placed at the tail of the list unless it has just been preempted. If it has just been preempted, the threads scheduling policy determines whether it is inserted at the head (realtime scheduling policy) or the tail (timeshare scheduling policy).

Figure 2–1 illustrates the general principles of process scheduling.

Figure 2–1 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 2–1 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.

Figure 2–2 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.

Figure 2–2 Process Events

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

MLO-007316

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.

2.2 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

  Used for 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 P1003.1b realtime scheduling functions and select an appropriate scheduling policy for your process. Digital UNIX P1003.1b 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 P1003.1b 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.

Also, under the first-in first-out scheduling policy, 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.

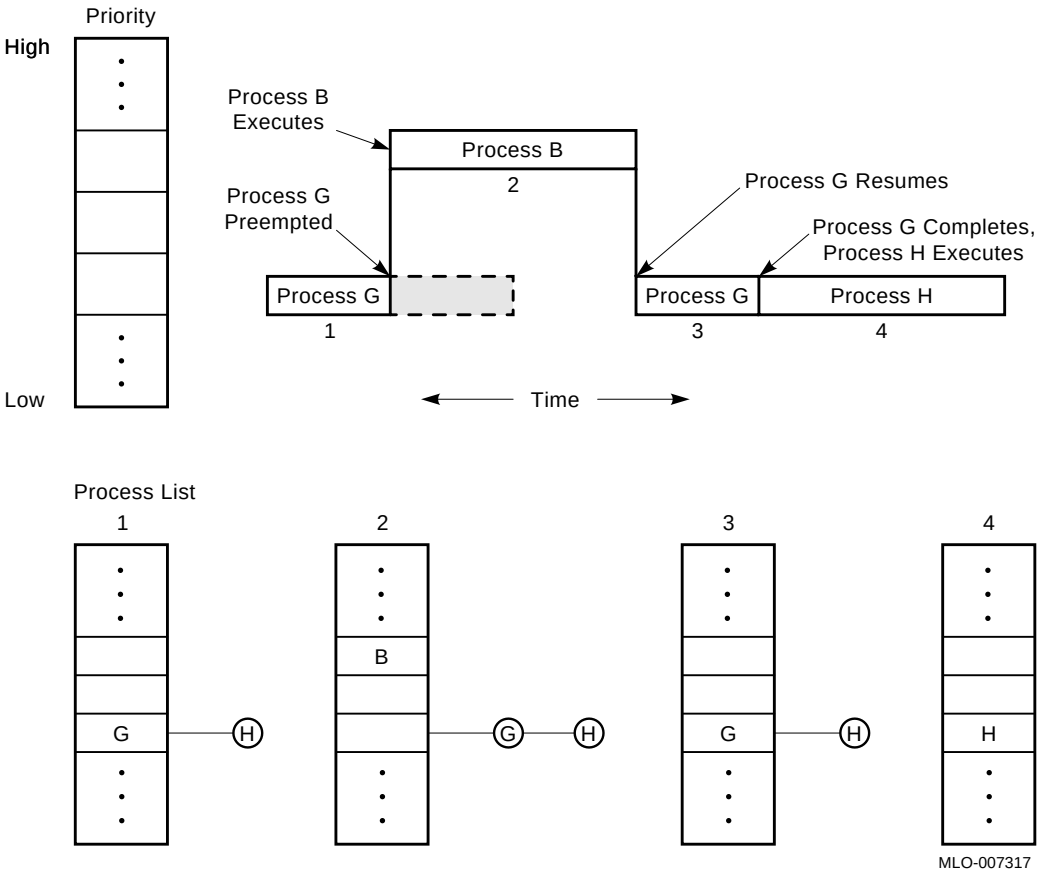
2.2.3.2 Round-Robin Scheduling

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 2–3 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 2–3 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_rr_get_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 P1003.1b 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 P1003.1b scheduling policies. Realtime priorities can be managed only by using the associated P1003.1b 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. Table 2–1 shows the `nice` interface priority ranges.

Table 2–1 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 `setpriority` 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 P1003.1b 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. Digital UNIX realtime priorities loosely map to the `nice` priority range, but provide a wider range of priorities. Processes using the P1003.1b scheduling policies must also use the Digital UNIX realtime interface priority scheme. Table 2–2 shows the Digital UNIX realtime priority ranges.

Table 2–2 Priority Ranges for the Digital UNIX Realtime Interface

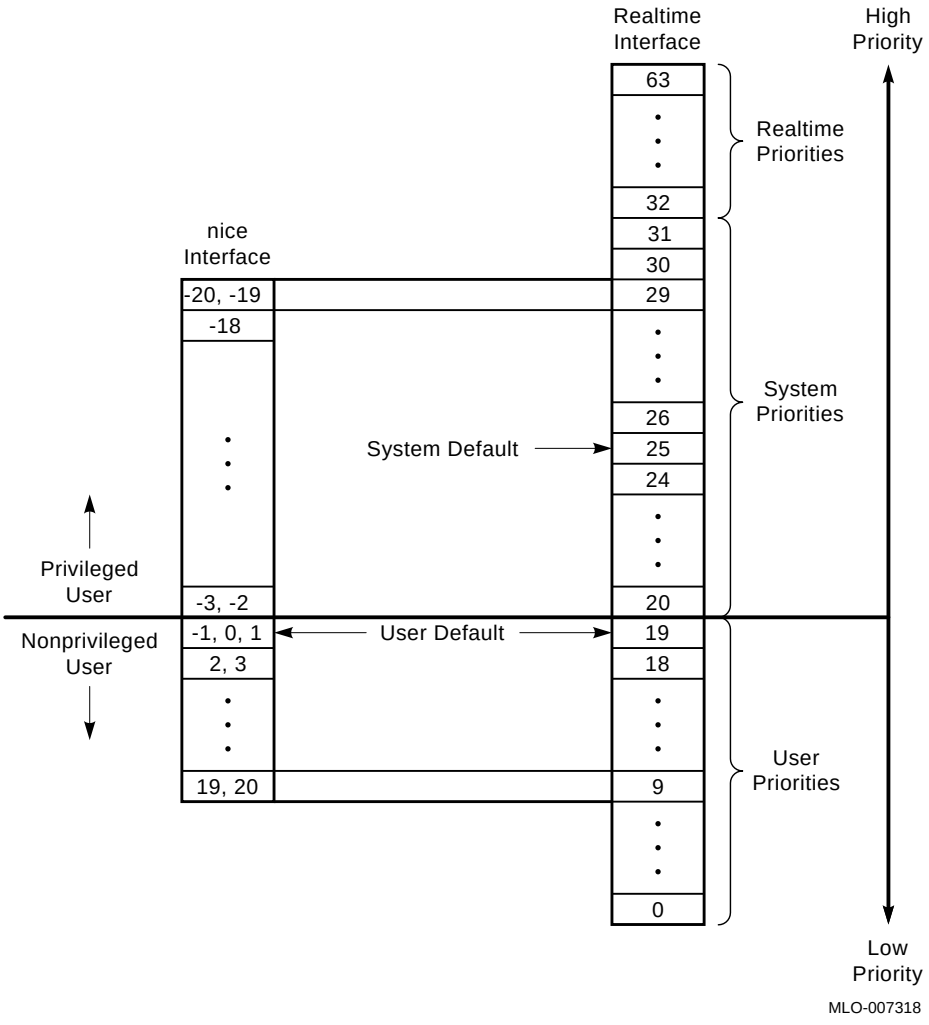
Range	Priority Level
Nonprivileged user	SCHED_PRIO_USER_MIN through SCHED_PRIO_USER_MAX
System	SCHED_PRIO_SYSTEM_MIN through SCHED_PRIO_SYSTEM_MAX
Realtime	SCHED_PRIO_RT_MIN through SCHED_PRIO_RT_MAX

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 2–4 illustrates the relationship between these two priority interfaces.

Figure 2–4 Priority Ranges for the nice and Realtime Interfaces



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

Digital UNIX does not support priority inheritance between processes. This is important to remember in prioritizing processes in such a way to avoid priority inversion. Priority inversion takes place when a higher priority process is blocked by the effects of a lower priority process.

For example, a client program running at a priority of 60 (realtime priority) blocks while waiting for the receipt of data. This allows a loop program to run at the lower priority of 40 (also realtime priority), but the network thread that dequeues the network packets is running at a system priority of 30. The loop program blocks the network thread, which in turn blocks the higher priority client process which is still waiting for the receipt of data.

In this case, the inversion may be resolved by running the network thread at a higher priority than the loop program. When running realtime processes at the exclusive realtime priority level, it is important to ensure that the processes give up the CPU in order for normal system processes to run.

2.3.3 Displaying Realtime Priorities

The ps command displays current process status and can be used to give realtime users snapshots of process priorities. 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 discrepancies.

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), POSIX scheduling priority (PPR), the state of the process (S), control terminal of the process (TTY), CPU time used by the process (TIME), and the process command (COMM).

The following example shows information regarding processes, with or without terminals, and displays timesharing and POSIX priorities. Note that the display indicates that the ps command is also running.

```
% ps -aeO psxpri
  PID PPR S    TTY            TIME COMMAND
    0  31 R <  ??          16:52:49 kernel idle
    1  19 I    ??          28:28.03 init
    7  19 I    ??           0:02.72 kloadsrv
   11  19 I    ??           0:00.94 dxterm
       .
       .
       .
14737  60 S<   p2           0:00.01 ./tests/work
13848  15 R    ttyv3        0:01.12 ps
```

In the example above, two processes are using realtime priorities. The first process (*PID* 0) is running at maximum system priority. The second realtime process (*PID* 14737) has been sleeping for less than twenty seconds at priority 60. The processes with *PID*s 1, 7, and 11 are idle at the maximum user priority.

For more information, see the reference page for the `ps` command.

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 P1003.1b 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 a `.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 Digital UNIX 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 P1003.1b functions, discussed in Section 2.4.3. You should adjust process priorities for your own application only. Adjusting system process priorities could have unexpected consequences.

2.4 Scheduling Functions

Realtime processes must be able to select the most appropriate priority level and scheduling policy dynamically. A realtime application often modifies the scheduling policy and priority of a process, performs some function, and returns the process to its previous priority. Realtime processes must also be able to yield system resources to each other in response to specified conditions. Eight P1003.1b functions, summarized in Table 2–3, satisfy these realtime requirements. Refer to the reference pages for a complete description of these functions.

Table 2–3 P1003.1b 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_rr_get_interval	Returns the current quantum for the round-robin scheduling policy
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

All the preceding functions, with the exception of the sched_yield function, require a process ID parameter (*pid*). In all P1003.1b 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 P1003.1b 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_rr_get_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 priority or scheduling policy.

If the *pid* is zero for either function, 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 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 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 (strictly conforming POSIX 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 P1003.1b 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 is 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. */
```

(continued on next page)

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

```
  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");

       /* 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 GID 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

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. Using the sched_yield function causes the scheduler to look for another process at the same priority level 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 */

stat = sched_rr_get_interval (my_pid, &rr_interval);
CHECK_STAT(stat, "sched_rr_get_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 */
```

(continued on next page)

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

```
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 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, 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 name as the controlling process when creating its own connections to the shared-memory object by also calling the shm_open function. The name can either be a string or a pathname, but in either case, processes must use the same name to refer to a specific shared-memory object.

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	Create an exclusive connection to a shared-memory object, when used with O_CREAT
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 remain 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 read the file only, specify read permission only on the open function. If you intend to read and write to the file, open the file with both read and write permission. After opening a file, call the mmap function to map the file into application address space.

When you have 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 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 protection options you specify 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 specified by POSIX 1003.1b. The MAP_ANONYMOUS, MAP_FILE, and MAP_VARIABLE flags are not part of the POSIX 1003.1b interface, but are supported by Digital UNIX. 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 use 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: 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 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 to 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 files
`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 `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. 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.

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, return a new file descriptor, or return 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 file control functions.

3.1.5 Controlling Memory-Mapped Files

Several 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 can 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 after 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 way, 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 specified in the call to 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 name 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 MCL_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 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>
```

(continued on next page)

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

```
main()
{
int fd;
caddr_t pg_addr;

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

     /* 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 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.

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 semaphore
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 9 for information on 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 includes 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 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 usually occupies 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. That 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. Digital UNIX memory-locking functions let you lock the entire process at the time of the function call and throughout the life of the application, or 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 nonrealtime tasks, such as opening files or allocating 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
`mlock`	Locks a specified region of a process's address space
`munlock`	Unlocks a specified region of a process's address space
`mlockall`	Locks all of a process's address space
`munlockall`	Unlocks all of a process's address space

You must have superuser privileges to call the memory locking functions.

4.2.1 Locking and Unlocking a Specified Region

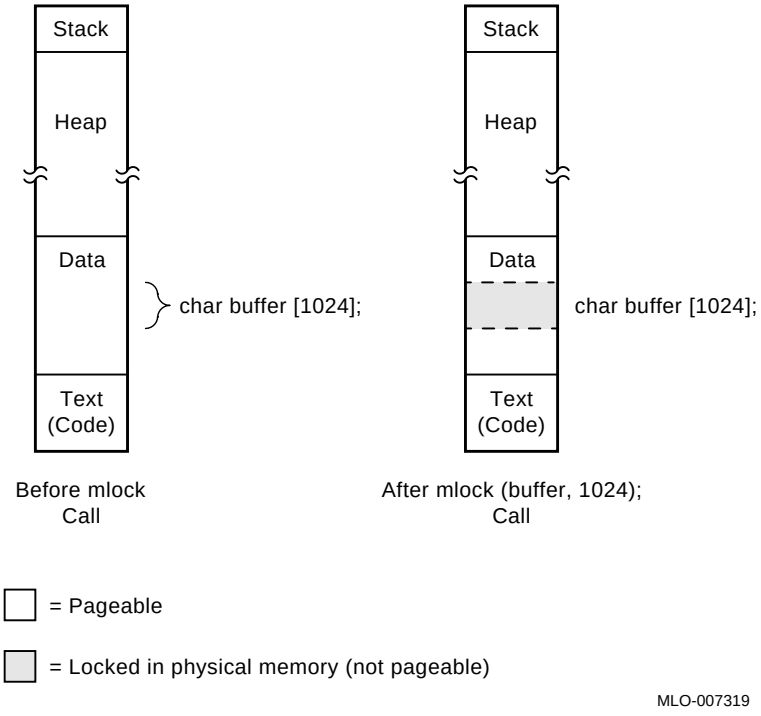
The `mlock` function locks a preallocated specified region. The address and size arguments of the `mlock` function determine the boundaries of the preallocated region. On a successful call to `mlock`, the specified region becomes locked. Memory is locked by the system according to system-defined pages. 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; in such cases, subsequent processes must wait for locked memory to become available. Realtime applications often cannot tolerate the latency introduced when a process must wait for lockable space to become available. Preallocating and locking regions is recommended for realtime applications.

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, and 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");
}
```

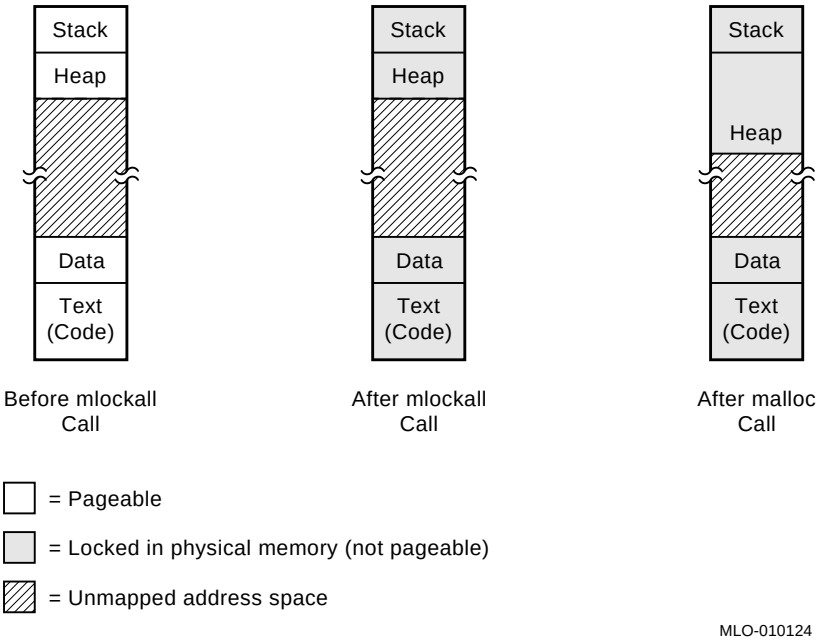
4.2.2 Locking and Unlocking an Entire Process Space

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 to be locked are those currently mapped, or if 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 area of the specified region is not locked into memory. If you specify the MCL_FUTURE flag only, 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 call the memory-locking functions again.

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 allocated in the future will be locked         */

}
```

5
Signals

The UNIX operating system uses signals as a means of notifying a process that some event, often unrelated to the process's current activity, has occurred that requires the process's attention. Signals are delivered to a process asynchronously; a process cannot predict when a signal might arrive.

Signals originate from a number of sources:

- An exception, such as a divide-by-zero or segmentation violation, may be detected by hardware, causing the UNIX kernel to generate an appropriate signal (such as SIGFPE or SIGSEGV) and send it to the current process.
- A user may press certain terminal keys, such as Ctrl/C, to control the behavior of the currently running program. This causes the terminal driver program to send a signal (such as SIGINT) to the user-level process in which the program is running. (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 currently associated with the terminal.)
- One user-level process may send a signal to another process. Traditionally, it does this using the `kill` function, although POSIX 1003.1b provides the `sigqueue` function for this purpose.
- A process may request a signal from the operating system when a timer expires, an asynchronous I/O operation completes, or a message arrives at an empty message queue.

The signal interface is also a traditional form of interprocess communication. Multitasking applications in particular take advantage of signals as a means of allowing components to coordinate activities across a number of processes. Because of the asynchronous nature of signals, a process can perform useful work while waiting for a significant event (for instance, it does not need to wait on a semaphore) and, when the event occurs, the process is notified immediately.

A process can specify what to do when it receives a signal. It can:

- Ignore the signal completely
- Handle the signal by establishing a function that is called whenever a particular signal is delivered
- Block the signal until it is able to deal with it. Typically the blocked signal has an established handler

An application can alternatively accept the default consequences of the delivery of a specific signal. These consequences vary from signal to signal, but can result in process termination, the process dumping core, the signal being ignored, or the process being restarted or continued. The default action of most signals is to terminate the process. If sudden process termination for the wide variety of conditions that cause signals is not desirable, an application should be prepared to deal with signals properly.

5.1 POSIX Signal Functions

POSIX 1003.1 standardized the reliable signal functions developed under 4.3BSD and SVR3. Table 5–1 lists the POSIX 1003.1 signal functions.

Table 5–1 POSIX 1003.1 Signal Functions

Function	Description
`sigemptyset`	Initializes a signal set such that all signals are excluded
`sigfillset`	Initializes a signal set such that all signals are included
`sigaddset`	Adds a signal to a signal set
`sigdelset`	Removes a signal from a signal set
`sigismember`	Tests whether a signal is a member of a signal set
`sigprocmask`	Sets the process's current blocked signal mask
`sigaction`	Specifies the action a process takes when a particular signal is delivered
`sigsuspend`	Replaces the process's current blocked signal mask, waits for a signal, and, upon its delivery, calls the handler established for the signal and returns
`sigpending`	Returns a signal set that represents those signals that are blocked from delivery to the process but are pending
`kill`	Sends a signal to a process or a group of processes

POSIX 1003.1b extended the POSIX 1003.1 definition to include better support for signals in realtime environments. Table 5–2 lists the POSIX 1003.1b signal

functions. A realtime application uses the sigqueue function instead of the kill function. It may also use the sigwaitinfo or sigtimedwait function instead of the sigsuspend function.

Table 5–2 POSIX 1003.1b Signal Functions

Function	Description
sigqueue	Sends a signal, plus identifying information, to a process.
sigwaitinfo	Waits for a signal and, upon its delivery, returns the signal number and any identifying information the signaling process provided.
sigtimedwait	Waits for a signal for the specified amount of time and, if the signal is delivered within that time, returns the signal number and any identifying information the signaling process provided.

To better explain the use of the POSIX 1003.1b extensions by realtime applications, this chapter first focuses on the basics of POSIX 1003.1 signal handling.

5.2 Signal Handling Basics

Example 5–1 shows the code for a process that creates a child that, in turn, creates and registers a signal handler, catchit.

Example 5–1 Sending a Signal to Another Process

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

#define SIG_STOP_CHILD SIGUSR1  1

main()
{
pid_t pid;
sigset_t newmask, oldmask;

    if ((pid = fork()) == 0) {  2    /*Child*/
        struct sigaction action;  3
        void catchit();
```

(continued on next page)

Example 5–1 (Cont.) Sending a Signal to Another Process

```
        sigemptyset(&newmask); 4
        sigaddset(&newmask, SIG_STOP_CHILD);  5
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);  6

        action.sa_flags = 0;  7
        action.sa_handler = catchit;

        if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {  8
            perror("sigusr: sigaction");
            _exit(1);
        }
        sigsuspend(&oldmask);  9
    }
    else {                              /* Parent */
        int stat;
        sleep(1);  10
        kill(pid, SIG_STOP_CHILD);  11
        pid = wait(&stat);  12
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
     _exit(0);
     }
}
void catchit(int signo)  13
{
       printf("Signal %d received from parent\n", signo);
       _exit(0);
}
```

In this example:

1 The program defines one of the two signals POSIX 1003.1 reserves for application-specific purposes (SIGUSR1) to be a SIG_STOP_CHILD signal.

2 The `main` program forks, creating a child process.

3 The child process declares a `sigaction` structure named `action` and a signal handler named `catchit`.

4 The child process initializes the `newmask` `sigset_t` structure to zero.

5 The child process calls the `sigaddset` function to set the bit corresponding to the SIG_STOP_CHILD signal in the *newmask* `sigset_t` structure.

6 The child process specifies the *newmask* `sigset_t` structure to a `sigprocmask` function call, thus blocking the SIG_STOP_CHILD signal.

7 The child process fills in the sigaction structure: first by calling the sigemptyset function to initialize the signal set to exclude all signals, then clearing the sa_flags member and moving the address of the catchit signal handler into the sa_handler member.

8 The child process calls the sigaction function to set up the catchit signal handler so that it is called when the process receives the SIG_STOP_ CHILD signal.

9 The child process calls the sigsuspend function. As a result, the SIG_ STOP_CHILD signal is unblocked and the child process pauses until the SIG_STOP_CHILD signal is delivered (and causes its catchit signal handler to run

10 The parent process sleeps for one second, allowing the child to run.

11 The parent process calls the kill function to send the SIG_STOP_CHILD signal to the child process.

12 It waits for the child process to terminate, printing the child's exit status when it does. Before this can occur, however, the child's catchit signal handler must run.

13 The catchit signal handler prints a message that acknowledges that the child received and handled the SIG_STOP_CHILD signal.

As in Example 5–1, under POSIX 1003.1, a process sends a signal to another process using the kill function. The first argument to the kill function is the process ID of the receiving process, or one of the following special values:

- 0

  Sends the signal to all processes with the same process group ID as that of the sender

- –1

  Sends the signal to all processes with a process group ID equal to the effective user ID of the sender

The second argument to the kill function is the name or number of the signal to be sent.

The permissions checking allowed by the first argument helps ensure that signals cannot be sent that arbitrarily or accidentally terminate any process on the system. Inasmuch as a process must have the identical user ID or effective user ID as the process it is signaling, it is often the case that it has spawned these processes or explicitly called the setuid function to set their effective user IDs. See the kill(2) reference page for additional discussion of the kill function.

The full set of signals supported by the Digital UNIX operating system is defined in `signal.h` and discussed in the `signal`(4) reference page. POSIX 1003.1 and POSIX 1003.1b require a subset of these signals; this subset is listed in Table 5–3.

Table 5–3 POSIX Signals

Signal	Description	Default Action
SIGABRT	Abort process (see `abort`(3))	Process termination and core dump
SIGALRM	Alarm clock expiration	Process termination
SIGFPE	Arithmetic exception (such as an integer divide-by-zero operation or a floating-point exception)	Process termination and core dump
SIGHUP	Hangup	Process termination
SIGILL	Invalid instruction	Process termination and core dump
SIGINT	Interrupt	Process termination
SIGKILL	Kill (cannot be caught, blocked, or ignored)	Process termination
SIGPIPE	Write on a pipe that has no reading process	Process termination
SIGQUIT	Quit	Process termination and core dump
SIGSEGV	Segmentation (memory access) violation	Process termination and core dump
SIGTERM	Software termination	Process termination
SIGUSR1	Application-defined	Process termination
SIGUSR2	Application-defined	Process termination
SIGCHLD	Child termination (sent to parent)	Ignored
SIGSTOP	Stop (cannot be caught, blocked, or ignored)	Process is stopped (suspended)
SIGTSTP	Interactive stop	Process is stopped (suspended)
SIGCONT	Continue if stopped (cannot be caught, blocked, or ignored)	Process is restarted (resumed)

(continued on next page)

Table 5–3 (Cont.) POSIX Signals

Signal	Description	Default Action
SIGTTOU	Background write attempted to controlling terminal	Process is stopped (suspended)
SIGTTIN	Background read attempted from controlling terminal	Process is stopped (suspended)
SIGRTMIN–SIGRTMAX	Additional application-defined signals provided by POSIX 1003.1b	Process termination

5.2.1 Specifying a Signal Action

The `sigaction` function allows a process to specify the action to be taken for a given signal. When you set a signal-handling action with a call to the `sigaction` function, the action remains set until you explicitly reset it with another call to the `sigaction` function.

The first argument to the `sigaction` function specifies the signal for which the action is to be defined. The second and third arguments, unless specified as NULL, specify `sigaction` structures:

- The second argument is a `sigaction` structure that specifies the action to be taken when the process receives the signal specified in the first argument. If this argument 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 third argument is a `sigaction` structure that receives from the `sigaction` function the action that was previously established for the signal. An application typically specifies this argument so that it can use it in a subsequent call to the `sigaction` function that restores the previous signal state. This allows you to activate handlers only when they are needed, and deactivate them when they may interfere with other handlers set up elsewhere for the same signal.

The sigaction structure has two different formats, defined in signal.h, distinguished by whether the *sa_handler* member specifies a traditional POSIX 1003.1 signal handler or a POSIX 1003.1b realtime signal handler:

- For POSIX 1003.1 signal handling:

```
struct sigaction (
    void (*sa_handler) (int);
    sigset_t sa_mask;
    int sa_flags;
    };
```

- For POSIX 1003.1b signal handling:

```
struct sigaction (
    void (*sa_sigaction) (int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    };
```

The remainder of this section focuses on the definition of a traditional signal handler in the *sa_handler* member of the sigaction structure. Note that, for realtime signals (those defined as SIGRTMIN through SIGRTMAX, you define the *sa_sigaction* member, not the *sa_handler* member. Section 5.3 describes the definition of a realtime signal handler in the *sa_sigaction* member.

Use the *sa_handler* member of the sigaction structure to identify the action associated with a specific signal, as follows:

- To ignore the signal, specify SIG_IGN. In this case, the signal is never delivered to the process. Note that you cannot ignore the SIGKILL or SIGSTOP signals.
- To accept the default action for a signal, specify SIG_DFL.
- To handle the signal, specify a pointer to a signal handling function. When the signal handler is called, it is passed a single integer argument, the number of 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.

The *sa_mask* field identifies the additional set of signals to be added to the process's current signal mask before the signal handler is actually called. This signal mask, plus the current signal, is active while the process's signal handler is running (unless it modified by another call to the sigaction function, or a call to the sigprocmask or sigsuspend functions). If the signal handler completes successfully, the original mask is restored.

The *sa_flags* member specifies various flags that direct the operating system's dispatching of a signal. For a complete listing of these flags and a description of their meaning, see the `sigaction`(2) reference page.

5.2.2 Setting Signal Masks and Blocking Signals

A process blocks a signal to protect certain sections of code from receiving signals when the code cannot be interrupted. Unlike ignoring a signal, blocking a signal postpones the delivery of the signal until the process is ready to handle it. A blocked signal is marked as pending when it arrives and is handled as soon as the block is released. Under POSIX 1003.1, multiple occurrences of the same signal are not saved; that is, if a signal is generated again while the signal is already pending, only the one instance of the signal is delivered. The signal queuing capabilities introduced in POSIX 1003.1b allow multiple occurrences of the same signal to be preserved and distinguished (see Section 5.3).

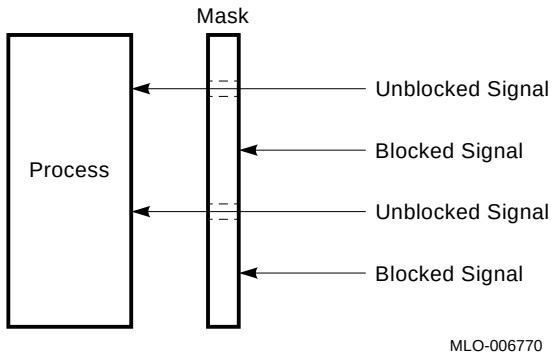
Each process has an associated signal mask that determines which signals are delivered to it and which signals are blocked from delivery. (A child process inherits its parent's signal mask when the parent forks.) Each bit represents a signal, as defined in the `signal.h` header file. For instance, if the *n*th bit in the mask is set, then signal *n* is blocked.

Note

As described in Chapter 2, the Digital UNIX operating system actually schedules threads, not processes. For multithreaded applications, a signal can be delivered to a thread, using the `pthread_kill` function, and a thread signal mask can be created using the `pthread_sigmask` function. These functions are provided in the DECthreads POSIX 1003.1c library (`libpthread.so`). See the appropriate reference pages and the *Guide to DECthreads* for a discussion of using signals with multithreaded applications.

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



The `sigprocmask` function lets you replace or alter the signal mask of the calling process; the value of the first argument to this function determines the action taken:

- SIG_BLOCK

  Adds the set of signals specified in the second argument to the process's signal mask

- SIG_UNBLOCK

  Subtracts the set of signals specified in the second argument from the process's signal mask

- SIG_SETMASK

  Replaces the process's signal mask with the set of signals specified in the third argument

The third argument to the `sigprocmask` function is a `sigset_t` structure that receives the process's previous signal mask.

Prior to calling the `sigprocmask` function, you use either the `sigemptyset` or `sigfillset` function to create the signal set (a `sigset_t` structure) that you provide as its second argument. The `sigemptyset` function creates a signal set with no signals in it. The `sigfillset` function creates a signal set containing all signals. You adjust the signal set you create with one of these functions by calling the `sigaddset` and `sigdelset` functions. You can determine whether a given signal is a member of a signal set by using the `sigismember` function.

The `sigprocmask` function is also 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)`.

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.

5.2.3 Suspending a Process and Waiting for a Signal

The sigsuspend function replaces a process's signal mask with the mask specified as its only argument and waits for the delivery of an unblocked signal. If the signal delivery causes a signal handler to run, the sigsuspend function returns after the signal handler completes, having restored the process's signal mask to its previous state. If the signal delivery causes process termination, the sigsuspend function does not return.

Because sigsuspend sets the signal mask and waits for an unblocked signal in one atomic operation, the calling process does not miss delivery of a signal that may occur just before it is suspended.

A process typically uses the sigsuspend function to coordinate with the asynchronous completion of some work by some other process. For instance, it may block certain signals while executing a critical section and wait for a signal when it completes:

```
     .
     .
     .
sigset_t newmask, oldmask;

sigemptyset(&newmask);
sigemptyset(&oldmask);
sigaddset(&newset, SIGUSR1);
sigaddset(&newset, SIGUSR2);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

      /* Code protected from SIGUSR1 and SIGUSR2 goes here */

      /* Release blocked signals and restore old mask */

sigsuspend(&oldmask);
     .
     .
     .
```

5.2.4 Setting Up an Alternate Signal Stack

The XPG4-UNIX specification defines the `sigaltstack` function to allow a process to set up a discrete stack area on which signals can be processed. The alternate signal stack is used if the *sa_flags* member of the `sigaction` structure for the signal specifies the SA_ONSTACK flag.

The `stack_t` structure supplied to a call to the `sigaltstack` function determines the configuration and use of the alternate signal stack by the values of the following members:

- The *ss_sp* member contains a pointer to the location of the signal stack.
- If the *ss_flags* member is not NULL, it can specify the SS_DISABLE flag, in which case the stack is disabled upon creation.
- The *ss_size* member specifies the size of the stack.

See the `sigaltstack`(2) reference page for additional information on the `sigaltstack`(2) function.

5.3 Realtime Signal Handling

Traditional signals, as defined by POSIX 1003.1, have several limitations that make them unsuitable for realtime applications:

- There are too few user-defined signals.

  There are only two signals available for application use, SIGUSR1 and SIGUSR2. For those applications that are constructed from various general-purpose and special-purpose components, all executing concurrently, the same signal could trigger different actions, depending on the sender. To avoid the risk of calling the wrong signal handler, code must become more complex and avoid asynchronous, unpredictable signal delivery.

- There is no priority ordering to the delivery of signals.

  When multiple signals are pending to a process, the order in which they are delivered is undefined.

- Blocked signals are lost.

  A signal can be lost if it is not delivered immediately. A single bit in a signal set is set when a blocked signal arrives and is pending delivery to a process. When the signal is unblocked and delivered, this bit is cleared. While it is set, however, multiple instances of the same signal can arrive and be discarded.

- The signal delivery carries no information that distinguishes the signal from others of the same type.

  From the perspective of the receiving process, there is no information associated with signal delivery that explains where the signal came from or how it is different from other such signals it may receive.

To overcome some of these limitations, POSIX 1003.1b extends the POSIX 1003.1 signal functionality to include the following facilitators for realtime signals:

- A range of priority-ordered, application-specific signals from SIGRTMIN to SIGRTMAX
- A mechanism for queuing signals for delivery to a process
- A mechanism for providing additional information about a signal to the process to which it is delivered
- Features that allow efficient signal delivery to a process when a POSIX 1003.1b timer expires, when a message arrives on an empty message queue, or when an asynchronous I/O operation completes
- Functions that allow a process to respond more quickly to signal delivery

Example 5–2 shows some modifications to Example 5–1 that allow it to process realtime signals more efficiently.

Example 5–2 Sending a Realtime Signal to Another Process

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

#define SIG_STOP_CHILD SIGRTMIN+1  1

main()
{
pid_t pid;
sigset_t newmask, oldmask;

    if ((pid = fork()) == 0) {  2    /*Child*/
        struct sigaction action;
        void catchit();

        sigemptyset(&newmask);
        sigaddset(&newmask, SIG_STOP_CHILD);
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);

        action.sa_flags = SA_SIGINFO;  3
        action.sa_sigaction = catchit;

        if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {  4
            perror("sigusr: sigaction");
            _exit(1);
        }
        sigsuspend(&oldmask);
    }
    else {                              /* Parent */
        union sigval sval;  5
        sval.sigev_value.sival_int = 1;
        int stat;
        sleep(1);  6
        sigqueue(pid, SIG_STOP_CHILD, sval);  7
        pid = wait(&stat);  8
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
        _exit(0);
    }
}
void catchit(int signo, siginfo_t *info, void *extra)  9
{
       void *ptr_val = info->si_value.sival_ptr;
       int int_val = info->si_value.sival_int;
       printf("Signal %d, value %d  received from parent\n", signo, int_val);
       _exit(0);
}
```

In this example:

1. The program defines one of the realtime signals defined by POSIX 1003.1b (SIGRTMIN+1) to be a SIG_STOP_CHILD signal.
2. The `main` program forks, creating a child process. The child process's initialization of the signal sets and creation of the process signal mask is the same as in the nonthreaded example in Example 5–1.
3. By specifying the SA_SIGINFO flag in the *sa_flags* member of the `sigaction` structure, the child process indicates that the associated signal will be using the realtime queued signaling behavior.
4. As in Example 5–1, the child process calls the `sigaction` function to set up the `catchit` signal handler so that it is called when the process receives the SIG_STOP_CHILD signal. It also calls the `sigsuspend` function to wait for a signal.
5. The parent process declares a `sigval` union. The member of this union can either be an integer or a pointer, depending on the value the parent sends to its child's signal handler. In this case, the value is an integer.
6. As in Example 5–1, the parent process sleeps for one second, allowing the child to run.
7. The parent process calls the `sigqueue` function to send the SIG_STOP_CHILD signal, plus a signal value, to the child process.
8. As in Example 5–1, the parent waits for the child process to terminate, printing the child's exit status when it does. Before this can occur, however, the child's `catchit` signal handler must run.
9. The `catchit` signal handler prints a message that acknowledges that the child received the SIG_STOP_CHILD signal and the signal value.

The following sections describe the POSIX 1003.1b extensions illustrated in this example.

5.3.1 Additional Realtime Signals

POSIX 1003.1 specified only two signals for application-specific purposes, SIGUSR1 and SIGUSR2. POSIX 1003.1b defines a range of realtime signals from SIGRTMIN to SIGRTMAX, the number of which is determined by the RTSIG_MAX constant in the `rt_limits.h` header file (which is included in the `limits.h` header file).

You specify these signals (in `sigaction` and other functions) by referring to them in terms of SIGRTMIN or SIGRTMAX: for instance, SIGRTMIN+1 or SIGRTMAX-1. Beware that SIGRTMIN and SIGRTMAX are not constants, so avoid compiler declarations that use them. You can determine the number of realtime signals on the system by calling `sysconf(_SC_RTSIG_MAX)`.

Although there is no defined delivery order for non-POSIX 1003.1b signals, the POSIX 1003.1b realtime signals are ranked from SIGRTMIN to SIGRTMAX (that is, the lowest numbered realtime signal has the highest priority). This means that, when these signals are blocked and pending, SIGRTMIN signals will be delivered first, SIGRTMIN+1 signals will be delivered next, and so on. Note that POSIX 1003.1b does not specify any priority ordering for non-realtime signals, nor does it indicate the ordering of realtime signals relative to nonrealtime signals.

If you want a function to use only these new realtime signal numbers, you can block the old POSIX 1003.1 signal numbers in process signal masks.

5.3.2 Queuing Signals to a Process

As shown in Section 5.2.1, the `sigaction` structure a realtime process passes to the `sigaction` function has the following format:

```
struct sigaction {
    void (*sa_sigaction) (int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
};
```

A process specifies the POSIX 1003.1b realtime signaling behavior (including signal queuing and the passing of additional information about the signal to its handler) by setting the SA_SIGINFO flag in the *sa_flags* member of this structure. Setting the SA_SIGINFO bit has the following effects:

- It causes the signal, if blocked, to be queued to the process, instead of being marked as pending in the process's pending signal set.
- It causes the signal handler defined in the *sa_sigaction* member of the `sigaction` structure to be called.
- It causes the signal handler to be called with two arguments in addition to the signal number.

5.3.2.1 The siginfo_t Structure

The second argument provided to the signal handler is a `siginfo_t` structure that provides information that identifies the sender of the signal and the reason why the signal was sent. The `siginfo_t` structure is defined in the `siginfo.h` header file (included by the `signal.h` header file), as follows:

```
typedef struct siginfo {
        int si_signo;
        int si_errno;
        int si_code;
        pid_t si_pid;
        uid_t si_uid;
        int si_status;
        union sigval si_value;
        void *si_addr;
        long si_band;
        int si_fd;
} siginfo_t;
```

The following list describes the members of the `siginfo_t` structure:

- The *si_signo* member contains the signal number. It is identical to the value passed as the first argument passed to the signal handler.
- The *si_errno* member contains the `errno` value that is associated with the signal.
- The *si_code* member provides information that identifies the source of the signal. For POSIX.1b signals, it can contain one of the following values:
  - SI_ASYNCIO

    The signal was sent on completion of an asynchronous I/O operation (see Section 5.3.3).
  - SI_MESGQ

    The signal was sent on arrival of a message to an empty message queue (see Section 5.3.3).
  - SI_QUEUE

    The signal was sent by the `sigqueue` function.
  - SI_TIMER

    The signal was sent because of a timer expiration (see Section 5.3.3).
  - SI_USER

    The signal was sent by `kill` function or a similar function such as `abort` or `raise`.

For XPG4-UNIX signals, this member can contain other values, as described in the siginfo(5) reference page.

- The *si_pid* member contains the process identification (PID) of the sending process.
- The *si_uid* member contains the user identification (UID) of the sending process. It is valid only when the *si_code* member contains the value SI_USER.
- The *si_status* member contains the exit value returned from the child process when a SIGCHLD signal is generated.
- The *si_value* member contains an application-specific value that has been passed to the signal handler in the last argument to the sigqueue function that generated the signal. The *si_value* member can contain either of the following members, depending upon whether the application-specific value is an integer or a pointer:

```
typedef union sigval {
 int  sival_int;
 void *sival_ptr;
} sigval_t;
```

- The *si_addr* member contains a pointer to the faulting instruction or memory reference. It is valid only for the SIGILL, SIGFPE, SIGSEGV, and SIGBUS signals.
- The *si_band* member contains the band event job-control character (POLL_OUT, POLL_IN, or POLL_MSG) for the SIGPOLL signal. See the poll(2) reference page for additional information on poll events.
- The *si_fd* member contains a pointer to the file descriptor of the poll event associated with the SIGPOLL signal.

5.3.2.2 The ucontext_t and sigcontext Structures

The third argument passed to a signal handler when the SA_SIGINFO flag is specified in the *sa_flags* member of the *sigaction* structure is defined by POSIX.1b as an "extra" argument. The Digital UNIX operating system uses this field to pass a ucontext_t structure to a signal handler in an XPG4-UNIX environment, or a sigcontext structure in a BSD environment.

Both structures contain the receiving process's context at the time at which it was interrupted by the signal. The sigcontext structure is defined in the signal.h header file. The ucontext_t structure is defined in the ucontext.h header file and is fully described in the ucontext(5) reference page.

5.3.2.3 Sending a Realtime Signal With the sigqueue Function

Where a process uses the kill function to send a nonrealtime signal to another process, it uses the sigqueue function to send a realtime signal. The sigqueue function resembles the kill function, except that it provides an additional argument, an application-defined signal value that is passed to the signal handler in the *si_value* member of the siginfo_t structure if the receiving process has enabled the SA_SIGINFO flag in the *sa_flags* member of the signal's sigaction structure.

The sigqueue function queues the specified signal to the receiving process. The permissions checking for the sigqueue function are the same as those applied to the kill function (see Section 5.2). Nonprivileged callers are restricted in the number of signals they can have actively queued at any time. This per-process quota value is defined in the rt_limits.h header file (which is included in the limits.h header file) as SIGQUEUE_MAX and is configurable by the system administrator. You can retrieve its value by calling sysconf(_SC_SIGQUEUE_MAX).

5.3.3 Asynchronous Delivery of Other Realtime Signals

Besides providing the sigqueue function to send realtime signals to processes, the POSIX 1003.1b standard defines additional features that extend realtime signal generation and delivery to functions that require asynchronous notification. Realtime functions are provided that automatically generate realtime signals for the following events:

- Asynchronous I/O completion (as initiated by the aio_read, aio_write, or lio_listio function)
- Timer expiration (for a timer established by the timer_create function)
- Arrival of a message to an empty message queue (for a message queue created by the mq_notify function)

When using the functions that trigger these events, you do not need to call a separate function to deliver signals. Realtime signal delivery for these events employs a sigevent structure, which is supplied as an argument (either directly or indirectly) to the appropriate function call. The sigevent structure contains information that describes the signal (or, prospectively, another mechanism of asynchronous notification to be used). It is defined in the signal.h header file and contains the following members:

```
int           sigev_notify;
union sigval  sigev_value;
int           sigev_signo;
```

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. There are two values defined for *sigev_notify* in POSIX 1003.1b:

- SIGEV_SIGNAL

  Indicates that a queued signal with an application-defined value is delivered when an event occurs.

- SIGEV_NONE Indicates that no asynchronous notification is delivered when an event occurs.

If the *sigev_notify* member contains SIGEV_SIGNAL, the other two members of the `sigevent` structure are meaningful.

The *sigev_value* member is an application-defined value to be passed to the signal catching function at the time of signal delivery. It can contain either of the following members, depending upon whether the application-specific value is an integer or a pointer:

```
typedef union sigval {
 int  sival_int;
 void *sival_ptr;
} sigval_t;
```

The *sigev_signo* member specifies the signal number to be sent on completion of the asynchronous I/O operation, timer expiration, or delivery of a message to the message queue. For any of these events, you must use the `sigaction` function to set up a signal handler to execute once the signal is received. Refer to Chapter 6 and Chapter 7 for examples of using signals with these functions.

5.3.4 Responding to Realtime Signals Using the sigwaitinfo and sigtimedwait Functions

The `sigsuspend` function, described in Section 5.2.3, allows a process to block while waiting for signal delivery. When the signal arrives, the process's signal handler is called. When the handler, completes, the process is unblocked and continues execution.

The `sigwaitinfo` and `sigtimedwait` functions, defined in POSIX 1003.1b, also allow a process to block waiting for signal delivery. However, unlike `sigsuspend`, they do not call the process's signal handler when a signal arrives. Rather, they immediately unblock the process, returning the number of the received signal as a status value.

The first argument to these functions is a signal mask that specifies which signals the process is waiting for. The process must have blocked the signals specified in this mask; otherwise, they will be dispatched to any established signal handler. The second argument is an optional pointer to a location to which the function returns `siginfo_t` structure that describes the signal.

The `sigtimedwait` function further allows you to specify a timeout value, allowing you to set a limit to the time the process waits for a signal.

Example 5–3 shows a version of Example 5–2 that eliminates the signal handler that runs when the child process receives a SIG_STOP_CHILD signal from its parent. Instead, the child process blocks the signal and calls the `sigwaitinfo` function to wait for its delivery.

Example 5–3 Using the sigwaitinfo Function

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

#define SIG_STOP_CHILD SIGRTMIN+1

main()
{
pid_t pid;
sigset_t newmask;
int rcvd_sig;  1
siginfo_t info;  2
    if ((pid = fork()) == 0) {       /*Child*/

        sigemptyset(&newmask);
        sigaddset(&newmask, SIG_STOP_CHILD);
        sigprocmask(SIG_BLOCK, &newmask, NULL);  3
```

(continued on next page)

Example 5–3 (Cont.) Using the sigwaitinfo Function

```
        while (1) {  4
            rcvd_sig = sigwaitinfo(&newmask, &info) 5
            if (rcvd_sig == -1) {
               perror("sigusr: sigwaitinfo");
               _exit(1);
            }
            else {  6
                 printf("Signal %d, value %d  received from parent\n",
                     rcvd_sig, info.si_value.sival_int);
               _exit(0);
            }
        }
    }
    else {                          /* Parent */
        union sigval sval;
        sval.sigev_value.sival_int = 1;
        int stat;
        sleep(1);
        sigqueue(pid, SIG_STOP_CHILD, sval);
        pid = wait(&stat);
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
        _exit(0);
    }
}
```

In this example:

1 The program defines a variable to which the sigwaitinfo call returns the value of the delivered signal (or returns –1, indicating an error).

2 The program defines a variable to which the sigwaitinfo call returns the siginfo_t structure that describes the received signal.

3 The child process sets up a signal mask to blocks the SIG_STOP_CHILD signal. Notice that it has not defined a signal handler to run when the signal is delivered. The sigwaitinfo function does not call a signal handler.

4 The child process loops waiting for signal delivery.

5 The child process calls sigwaitinfo function, specifying the newmask signal mask to block the SIG_STOP_CHILD signal and wait for its delivery.

6 When the signal is delivered, the child process prints a message indicating that it has received the signal. It also prints the signal value that may accompany the realtime signal.

An additional example using the sigwaitinfo function is shown in Example 5–4. In this example, the child process sends to its parent the maximum number of signals that the system allows to be queued. When a SIG signal is delivered to it, the parent counts it and prints an informative message. After it has received _SC_SIGQUEUE_MAX signals, the parent prints a message that indicates the number of signals it has received.

Example 5–4 Using the sigwaitinfo Function

```
#include <unistd.h>
#include <stdio.h>
#include <sys/siginfo.h>
#include <sys/signal.h>

main()
{
        sigset_t        set, pend;
        int             i, sig, sigq_max, numsigs = 0;
        siginfo_t       info;
        int             SIG = SIGRTMIN;

        sigq_max = sysconf(_SC_SIGQUEUE_MAX);
        sigemptyset(&set);
        sigaddset(&set, SIG);
        sigprocmask(SIG_SETMASK, &set, NULL);
        printf("\nNow create a child to send signals...\n");
        if (fork() == 0) {      /* child */
                pid_t parent = getppid();
                printf("Child will signal parent %d\n", parent);
                for (i = 0; i < sigq_max; i++) {
                        if (sigqueue(parent, SIG, i) < 0)
                                perror("sigqueue");
                }
                exit(1);
        }
        printf("Parent sigwait for child to queue signal...\n");
        sigpending(&pend);
        printf("Is signal pending: %s\n",
               sigismember(&pend, SIG) ? "yes" : "no");
        for (i = 0; i < sigq_max; i++) {
                sig = sigwaitinfo(&set, &info);
                if (sig < 0) {
                        perror("sigwait");
                        exit(1);
                }
                printf("Main woke up after signal %d\n", sig);
                printf("signo = %d, pid = %d, uid = %d, val = %d,\n",
                       info.si_signo, info.si_pid, info.si_uid, info.si_int);
```

(continued on next page)

Example 5–4 (Cont.) Using the sigwaitinfo Function

```
                numsigs++;
        }
        printf("Main: done after %d signals.\n", numsigs);
}
```

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.

Digital UNIX P1003.1b 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 P1003.1b timing functions perform the following tasks:

- Set a systemwide clock and obtain the current value of the clock
- 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, and use of multiple per-process timers should use the P1003.1b 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 that has elapsed since 00:00:00 January 1, 1970 Greenwich Mean Time (GMT). [1]

The CLOCK_REALTIME clock measures time in nanoseconds; clock resolution does not reflect fractions of nanoseconds. For example, when the resolution for CLOCK_REALTIME is calculated at 1 sec / 1024 Hz, the result is 976562.5 nanoseconds. The clock resolution returned by the call to `clock_getres` for CLOCK_REALTIME is 976562. The fractional nanoseconds are ignored. The system self-corrects at the end of every second and adjusts time to correct for disparities. See Section 6.1.4 for more information about system clock resolution.

[1] Otherwise known as the "Epoch."

Table 6–1 lists P1003.1b timing functions for a specified clock.

Table 6–1 Clock Functions

Function	Description
clock_getres	Returns the resolution of the specified clock
clock_gettime	Returns the current value for the specified clock
clock_settime	Sets the specified clock to the specified value

Use the name CLOCK_REALTIME as the *clock_id* argument in all P1003.1b clock functions.

The clock_getres function returns the clock resolution. Note that you cannot set the resolution of the specified clock, although you can specify a high-resolution option that gives the appearance of higher resolution (see Section 6.1.5).

The values returned by the clock_gettime function can be used to determine values for the creation of realtime timers.

When the clock_settime function is called, the *time* argument is truncated to a multiple of the clock resolution, if it is not already a multiple of the clock resolution. Similarly, the clock resolution is used when setting interval timers.

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, 876,764,530 seconds is returned from the time function, and 876,764,530 seconds and 000,0674,633 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.

If timers are pending execution, use the adjtime function to adjust the clock slowly; armed timers are not affected by this function. Refer to the reference page for adjtime for complete information about this function.

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

6.1.3 Converting Time Values

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

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

The C language provides a number of functions to convert and store time in both a tm structure and an ASCII format. Note that although these C routines use seconds as the smallest unit of time, they provide users with a readable format.

When you pass the time in seconds to these functions, some functions return a pointer to a tm structure. This structure breaks down time into units such as hours, minutes, and seconds, and stores the data in the appropriate fields.

Digital UNIX provides date and time functions that deal with these time units and calendar time, making conversions as necessary. Table 6–2 describes date and time conversion functions. To select the most appropriate time conversion function for your application, refer to the reference pages for each of these functions.

Table 6–2 Date and Time Conversion Functions

C Function	Description
asctime	Converts time units (hours, minutes, and seconds) into a 26-character string
ctime	Converts a time in seconds since the Epoch to an ASCII string in the form generated by asctime
difftime	Computes the difference between two calendar times (time1–time0) and returns the difference expressed in seconds

(continued on next page)

Table 6–2 (Cont.) Date and Time Conversion Functions

C Function	Description
gmtime	Converts a calendar time into time units, expressed as GMT
localtime	Converts a time in seconds since the Epoch into time units
mktime	Converts the time units 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 time zone names

The converted time values for the functions listed in Table 6–2 are placed in a time structure (tm) defined in the time.h header file, as follows:

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

6.1.4 System Clock Resolution

System clock resolution on Digital Alpha systems is 1/1024 second, or roughly 976 microseconds. The system maintains time by adding 976 microseconds at every clock interrupt. The actual time period between clock ticks is exactly 1/1024 second = 976.5625 microseconds.

The missing 576 microseconds (1024 * .5625) are added at the end of the 1024th tick (that is, every second), to make sure that the system time matches with the observed "wall-clock" time.

This implies that each clock tick increments the system time by 976 microseconds except the 1024th one, which advances the time by 1552 microseconds (976 + 576). Thus there is a spike in the time as maintained by Digital UNIX.

The POSIX 1003.1a specification mandates that the system quantize all timer values passed by a program to the next multiple of the clock tick. If an application program requests a timer value that is not an exact multiple of the system clock resolution (an exact multiple of 976.5625 microseconds), the actual time period counted down by the system will be slightly larger than the requested time period.

A program that asks for a periodic timer of 50 milliseconds will actually get a time period of 50.78 milliseconds (.976562 * 52). Unless accounted for, the additional .78 milliseconds every 50 milliseconds will result in a wrong calculation of the elapsed time as calculated by the program.

Possible solutions to the above anomaly are to either always ask for time periods that are integral multiples of the system clock resolution, or to not use the periodic timer for the purpose of time keeping.

6.1.5 High-Resolution Clock

Version 4.0 of Digital UNIX adds the capability of an optional high-resolution clock. To enable the high-resolution clock, add the following line to the kernel configuration file and rebuild the kernel:

```
options MICRO_TIME
```

The system clock (CLOCK_REALTIME) resolution as returned by `clock_getres(3)` will not change; timer resolution remains the same. However, time as returned by the `clock_gettime(3)` routine will now be extrapolated between the clock ticks. The granularity of the time returned will now be in microseconds. The time values returned are SMP safe, monotonically increasing, and have 1 microsecond as the apparent resolution.

The high-resolution clock can be used for time stamping and for measuring events which are of the order of microseconds, such as time spent in some critical code path.

6.2 Types of Timers

Two types of timers are provided to support realtime timing facilities: one-shot timers 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 at 12-minute 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 specify an initial expiration value and an interval value. 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 after 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 you do not choose to use realtime signals, then identical signals delivered from multiple timers are compressed into a single signal. In this case, you may need to specify a different signal for each timer. If you use realtime signals, identical signals are queued to the calling process. Refer to Chapter 5 for more information on signals and signal handling.

6.4 Data Structures Associated with Timing Facilities

The timespec and itimerspec data structures in the timers.h header file are used in many of the P1003.1b 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 P1003.1b 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 P1003.1b 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  */
} timespec_t;
```

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 P1003.1b functions to set and return the specified clock, return the resolution of the clock, set and return timer values, and specify nanosleep values.

6.4.2 Using the itimerspec Data Structure

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

```
struct itimerspec {
       struct timespec it_interval;   /* Timer interval     */
       struct timespec it_value;      /* Initial 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 6–3 summarizes the ways that values for the two members of the itimerspec structure are used to specify timers.

Table 6–3 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.

TIMER_ABSTIME is a flag which, when set, makes the timer 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.

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.

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; after the first expiration 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 as fractions of a second, you could define a symbolic constant:

```
#define NSECS_PER_SEC 1000000000;
```

After defining this constant, you could specify 1/4 second as follows:

```
mytimer.it_value.tv_nsec = NSECS_PER_SEC/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, which 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:

```
union sigval   sigev_value;   /* Application-defined value */
int            sigev_signo;   /* Signal to raise */
int            sigev_notify;  /* Notification type */
```

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.

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 when the signal is received. You can use the `sigaction` function to specify the action required. Refer to Chapter 5 for more information about the `sigaction` function.

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. There are two values defined for *sigev_notify* in P1003.1b: SIGEV_NONE and SIGEV_SIGNAL. SIGEV_NONE indicates that no asynchronous notification is delivered when an event occurs. SIGEV_ SIGNAL indicates that a queued signal with an application-defined value is delivered when an event occurs.

6.5 Timer Functions

Clocks and timers allow an application to synchronize and coordinate activities according to a user-defined schedule. Digital UNIX P1003.1b timers have the ability to issue periodic timer requests initiated by a single call from the application.

Table 6–4 lists the P1003.1b timing functions available for realtime applications.

Table 6–4 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_getoverrun`	Returns the timer expiration overrun count for the 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 that 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 timer. 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, if non-NULL, 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. If the *sigev_notify* member of *evp* is SIGEV_SIGNAL, the structure must contain the signal number and data value to send to the process when the timer expires. If the *sigev_notify* member is SIGEV_NONE, no notification will be sent.

If the *evp* argument is NULL, the default signal SIGALRM is used.

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 interval you specify is rounded up to the next integral multiple of the system clock resolution. See Section 6.1.4 for more information about system clock resolution.

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 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 delete 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 Getting the Overrun Count

Under POSIX.1b, timer expiration signals for a specific timer are not queued to the process. If multiple timers are due to expire at the same time, or a periodic timer generates an indeterminate number of signals with each timer request, a number of signals will be sent at essentially the same time. There may be instances where the requesting process can service the signals as fast as they occur, and there may be other situations where there is an overrun of the signals.

The `timer_getoverrun` function helps track whether or not a signal was delivered to the calling process. Digital UNIX P1003.1b timing functions keep a count of timer expiration signals for each timer created. The `timer_getoverrun` function returns the counter value for the specified timer ID. If a signal is sent, the overrun count is incremented, even if the signal was not delivered or if it was compressed with another signal. If the signal cannot be delivered to the calling process or if the signal is delayed for some reason, the overrun count contains the number of extra timer expirations that occurred during the delay. A signal may not be delivered if, for instance, the signal is blocked or the process was not scheduled. Use the `timer_getoverrun` function to track timer expiration and signal delivery as a means of determining the accuracy or reliability of your application.

If the signal is delivered, the overrun count is set to zero and remains at zero until another overrun occurs.

6.5.5 Disabling Timers

When 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.

6.6 High-Resolution Sleep

To suspend process execution temporarily using the P1003.1b 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 P1003.1b 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 1003.1b 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;

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

(continued on next page)

Example 6–2 (Cont.) Using Timers

```
/*
 * 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;



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;

        /*
         *  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.
         */
```

(continued on next page)

Example 6–2 (Cont.) Using Timers

```
        for (i = 0; i < timers_available; i++) {
                status = timer_create(clock_id, &timer_values[i].evp,
   &timerid[i]);
                if (status == 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)
        */

        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();

}
```

(continued on next page)

Example 6–2 (Cont.) Using Timers

```
/*
 *  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 */

        }
        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 passed through the `aiocb` structure for the asynchronous I/O function. Alternatively, the calling process can poll the `aiocb` structure for completion status.

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 to Raw Devices, Section 7.3
- Asynchronous I/O Examples, Section 7.4

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 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 request to read data from one channel and immediately be free to receive the next data transmission from another channel. 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 operation requests 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 P1003.1b 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 P1003.1b 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, 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;    /* File offset                     */
volatile void   *aio_buf;      /* Pointer to buffer               */
size_t          aio_nbytes;    /* Number of bytes to transfer     */
int             aio_reqprio;   /* Request priority offset         */
struct sigevent aio_sigevent;  /* 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 queue 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, call 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:

```
union sigval   sigev_value;   /* Application-defined value */
int            sigev_signo;   /* Signal to raise */
int            sigev_notify   /* Notification type */
```

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. There are two values defined for *sigev_notify* in P1003.1b: SIGEV_NONE and SIGEV_SIGNAL. SIGEV_NONE indicates that no asynchronous notification is delivered when an event occurs. SIGEV_SIGNAL indicates that the signal number specified in *sigev_signo* and the application-defined value specified in *sigev_value* are queued when an event occurs. When the signal is queued to the process, the value of *aio_sigevent.sigev_value* will be the *si_value* component of the generated signal. See Chapter 5 for more information.

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, if *sigev_notify* equals SIGEV_SIGNAL. 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 retrieved using the `aio_error` and `aio_return` functions.

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 typically calls the `lseek` function, performs the I/O operation, and then waits to receive 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_fsync`	Asynchronously writes system buffers containing a file's modified data to permanent storage
`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

(continued on next page)

Table 7–1 (Cont.) Asynchronous I/O Functions

Function	Description
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 functions aio_read and aio_write perform the same I/O operations as the read and write functions. However, the aio_read and aio_write functions return control to the calling process once 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, aio_write, or aio_fsync 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.

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

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.

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 waits 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 lio_listio operation is complete.

Completion means that all the individual operations in the list have completed, either successfully or unsuccessfully. In either case, the return value indicates only the success or failure of the lio_listio function call, not the status of individual I/O requests. In some cases one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, examine the error status associated with each lio_aiocb control block.

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 after the lio_listio function completes, or to catch a signal returned on completion of the entire list of I/O operations in the lio_listio request.
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 function provides 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 subsequent 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 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 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 after 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 take 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 had completed when the call to the aio_suspend function was made, the function returns without suspending the calling process. Before using the aio_suspend function, your application must already have initiated an I/O request with a call to the aio_read, aio_write, aio_fsync, or lio_listio function.

7.2.6 Asynchronous File Synchronization

The aio_fsync function is similar to the fsync function; however, it executes in an asynchronous manner, in the same way that aio_read performs an asynchronous read.

The aio_fsync function requests that all I/O operations queued to the specified file descriptor at the time of the call to aio_fsync be forced to the synchronized I/O completion state. Unlike fsync, aio_fsync returns control to the calling process once the operation has been initiated, rather than after the operation is complete. I/O operations that are subsequently initiated on the file descriptor are not guaranteed to be completed by any previous calls to aio_fsync.

Like the aio_read and aio_write functions, aio_fsync takes an *aiocbp* value as an argument, which can then be used in subsequent calls to aio_error and aio_return in order to determine the error and return status of the asynchronous operation. In addition, the *aio_sigevent* member of *aiocbp* can be used to define the signal to be generated when the operation is complete.

Note that the aio_fsync function will force to completion *all* I/O operations on the specified file descriptor, whether initiated by synchronous or asynchronous functions.

7.3 Asynchronous I/O to Raw Devices

You may have applications which call for performing asynchronous I/O operations by reading to and writing from raw partitions. Digital UNIX provides the `libaio_raw.a` library for those applications which will only perform asynchronous I/O operations to raw devices, When using this library, you are not required to link with pthreads, libmach, or libc_r.

If you attempt to perform asynchronous I/O operations to a file when linked with `libaio_raw.a`, the request fails with an error of ENOSYS.

The syntax for compiling or linking with `libaio_raw.a` is as follows:

```
% cc -o binary_name my_program -laio_raw
```

7.4 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; Example 7–2 uses the `lio_listio` function.

7.4.1 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 -laio -pthread
 */

           /* * * *  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 */
/* To run completion code in the signal handler, define the following: */
#define COMPLETION_IN_HANDLER

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

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

/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
void sig_action(signo,info,context)
int signo;
siginfo_t *info;
void *context;
{
        printf("Entered sig_action\n");
        printf("  signo = %d \n",signo);
        printf("  si_code  = %d \n",info->si_code);

#ifndef COMPLETION_IN_HANDLER
        printf("  si_value.sival_int = %d decimal
\n",info->si_value.sival_int);

#else
        printf("  si_value.sival_ptr = %lx hex \n",info->si_value.sival_ptr);

        /* Call aio_error and aio_return from the signal handler.
         * Note that si_value is the address of the write aiocb.
         */
        while (aio_error((struct aiocb *)info->si_value.sival_ptr) ==
EINPROGRESS);
```

(continued on next page)

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

```
        /* * * * Update total bytes written to set new file offset * * * */
        total += aio_return((struct aiocb *)info->si_value.sival_ptr);
#endif
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        sigcnt++;
        return;
}

void sig_handler(signo)
int signo;
{
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        printf("Entered sig_handler, signo = %d \n",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, 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");
```

(continued on next page)

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

```
           /* * * * 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 * * * */

        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  * * * */
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        sigemptyset(&sig_act.sa_mask);  /* block only current signal */

        /*  If the SA_SIGINFO flag is set in the sa_flags field then
         *  the sa_sigaction field of sig_act structure specifies the
         *  signal catching function:
         */
        sig_act.sa_flags = SA_SIGINFO;
        sig_act.sa_sigaction = sig_action;

        /*  If the SA_SIGINFO flag is NOT set in the sa_flags field
         *  then the the sa_handler field of sig_act structure specifies
         *  the signal catching function, and the signal handler will be
            *  invoked with 3 arguments instead of 1:
         *     sig_act.sa_flags = 0;
         *     sig_act.sa_handler = sig_handler;
         */

           /* * * *  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) * * * */
```

(continued on next page)

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

```
        a_write.aio_fildes = out_file;
        a_write.aio_offset = 0;                   /* write from current */
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        a_write.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
        a_write.aio_sigevent.sigev_signo  = SIGUSR1; /* completion signal */
#ifdef COMPLETION_IN_HANDLER
        /* Fill in a user-specified value which will be the si_value
         * component of the generated signal.  sigev_value is a union
         * of either an int (sival_int) or a void * (sival_ptr).
         * In this example, we use the sival_ptr field, and pass
          * the address of the aiocbp into the signal handler, so
         * that the signal handler can call aio_error and aio_return directly:
         */
        a_write.aio_sigevent.sigev_value.sival_ptr = &a_write;
#else
        /* Pass an integer value into the signal handler: */
        a_write.aio_sigevent.sigev_value.sival_int = 1;
#endif
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/

           /* * * * 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);

#ifdef COMPLETION_IN_HANDLER
                if (rec_cnt) {  /* will be >1 on all but first write... */
                        aiocb_t *wait_list = &a_write;

                        /* Wait until previous write completes */
                                aio_suspend(&wait_list,1,NULL);

                }  /* if (rec_cnt) */

#else

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

(continued on next page)

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

```
                        /* previous write completed? If not, wait */
                        while (aio_error(&a_write) == EINPROGRESS) {
                                aiocb_t *wait_list = &a_write;
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
                                /* No timeout specified */
                                aio_suspend(&wait_list,1,NULL);
                                /* aio_suspend(1, &wait_list); */
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
                        }
           /* * * * Update total bytes written to set new file offset * * * */

                        total += aio_return(&a_write);
                }  /* if (rec_cnt) */
#endif

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

                if (buf_len <= 0)
                        break;

           /* * * * 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;
        }

        printf("total number of bytes written to output file = %d\n",total);

           /* * * * 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);
}
```

7.4.2 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 -non_shared -O0 -L/usr/ccs/lib \
 *       -laio -pthread
 */

/* * * *  lio_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 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 * * * */
```

(continued on next page)

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

```
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 * * * */

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 * * * */
```

(continued on next page)

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

```
        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 * * * */

        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 * * * */
```

(continued on next page)

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

```
                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");

      /* * * 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);
}
```

Note

Use of the `printf` function in this example is for illustrative purposes only. You should avoid using `printf` and any similar functions in signal handlers because they can affect scheduling characteristics.

8

File Synchronization

By default, UNIX systems read from and write to a buffer cache that is kept in memory, and avoid actually transferring data to disk until the buffer is full, or until the application calls a sync function to flush the buffer cache. This increases performance by avoiding the relatively slow mechanical process of writing to disk more often then necessary.

Realtime input and output operations are of two types:

- Asynchronous I/O, which frees the application to perform other tasks while input is written or read (see Chapter 7)
- Synchronized I/O, which performs the write or read operation and verifies its completion before returning

Digital UNIX supports POSIX 1003.1b file synchronization for the UFS and AdvFS file systems, as described in this chapter. However, Digital recommends use of the UFS file system for better realtime performance.

Synchronized I/O is useful when the integrity of data and files is critical to an application. Synchronized output assures that data that is written to a device is actually stored there. Synchronized input assures that data that is read from a device is a current image of data on that device.

Two levels of file synchronization are available, data integrity and file integrity:

- Data integrity
  - Write operations: data in the buffer is transferred to disk, along with file system information necessary to retrieve the data.
  - Read operations: any pending write operations relevant to the data being read complete with data integrity before the read operation is performed.
- File integrity
  - Write operations: data in the buffer and all file system information related to the operation are transferred to disk.

– Read operations: any pending write operations relevant to the data being read complete with file integrity before the read operation is performed.

8.1 How to Assure Data or File Integrity

You can assure data integrity or file integrity at specific times by using function calls, or you can set file status flags to force automatic file synchronization for each `read` or `write` call associated with that file.

Use of synchronized I/O may degrade system performance; see Chapter 11.

8.1.1 Using Function Calls

You can choose to write to buffer cache as usual, and call functions explicitly when you want the program to flush the buffer to disk. For instance, you may want to use the buffer cache when a lot of I/O is occurring, and call these functions when activity slows down. Two functions are available:

- `fdatasync` — flushes data only, providing data integrity completion
- `fsync` — flushes data and file control information, providing file integrity completion

Refer to online reference pages for a complete description of these functions.

8.1.2 Using File Descriptors

If you want to write data to disk in all cases automatically, you can set file status flags to force this behavior instead of making explicit calls to `fdatasync` or `fsync`.

To set this behavior, use these flags with the `open` or `fcntl` function:

- O_DSYNC — forces data synchronization for each write operation.

  Example:

  ```
  fd = open("my_file", O_RDWR | O_CREAT | O_DSYNC, 0666);
  ```

- O_SYNC — forces file and data synchronization for each write operation.

  Example:

  ```
  fd = open("my_file", O_RDWR | O_CREAT | O_SYNC, 0666);
  ```

- O_RSYNC — when either of the other two flags is in effect, forces the same file synchronization level for each read as well as each write operation. Use of O_RSYNC has no effect in the absence of O_DSYNC or O_SYNC.

Examples:

```
fd = open("my_file", O_RDWR | O_CREAT | O_SYNC | O_RSYNC, 0666);
fd = open("my_file", O_RDWR | O_CREAT | O_DSYNC | O_RSYNC, 0666);
```

If both the O_DSYNC and O_SYNC flags are set using the `open` or `fcntl` function, O_SYNC takes precedence.

9
Semaphores

POSIX 1003.1b semaphores provide an efficient form of interprocess communication. Cooperating processes can use 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 prevents 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:

- Overview of Semaphores, Section 9.1
- The Semaphore Interface, Section 9.2
- Semaphore Example, Section 9.3

9.1 Overview of Semaphores

Semaphores are used to control access to shared resources by processes. Counting semaphores have a positive integral value representing the number of processes that can concurrently lock the semaphore.

There are named and unnamed semaphores. Named semaphores provide access to a resource between multiple processes. Unnamed semaphores provide multiple accesses to a resource within a single process or between related processes. Some semaphore functions are specifically designed to perform operations on named or unnamed semaphores.

The semaphore lock operation checks to see if the resource is available or is locked by another process. If the semaphore's value is a positive number, the lock is made, the semaphore value is decremented, and the process continues execution. If the semaphore's value is zero or a negative number, the process requesting the lock waits (is blocked) until another process unlocks the resource. Several processes may be blocked waiting for a resource to become available.

The semaphore unlock operation increments the semaphore value to indicate that the resource is not locked. A 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.

Semaphores are global entities and are not associated with any particular process. In this sense, 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 incrementing the semaphore value when relinquishing the resource. Since semaphores lack owners, there is no way to determine whether one of the cooperating processes has become uncooperative. Applications using semaphores must carefully detail cooperative tasks. All of the processes that share a resource must agree on which semaphore controls the resource.

POSIX 1003.1b semaphores are persistent. The value of the individual semaphore is preserved after the semaphore is no longer open. For example, a semaphore may have a value of 3 when the last process using the semaphore closes it. The next time a process opens that semaphore, it will find the semaphore has a value of 3. For this reason, cleanup operations are advised when using semaphores.

Note that because semaphores are persistent, you should call the `sem_unlink` function after a system reboot. After calling `sem_unlink`, you should call the `sem_open` function to establish new semaphores.

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

9.2 The Semaphore Interface

Table 9–1 lists the functions that allow you to create and control P1003.1b semaphores.

Table 9–1 Semaphore Functions

Function	Description
sem_close	Deallocates the specified named semaphore
sem_destroy	Destroys an unnamed semaphore
sem_getvalue	Gets the value of a specified semaphore
sem_init	Initializes an unnamed semaphore
sem_open	Opens/creates a named semaphore for use by a process
sem_post	Unlocks a locked semaphore
sem_trywait	Performs a semaphore lock on a semaphore only if it can lock the semaphore without waiting for another process to unlock it
sem_unlink	Removes a specified named semaphore
sem_wait	Performs a semaphore lock on a semaphore

You create an unnamed semaphore with a call to the sem_init function, which initializes a counting semaphore with a specific value. To create a named semaphore, call sem_open with the O_CREAT flag specified. The sem_open function establishes a connection between the named semaphore and a process.

Semaphore locking and unlocking operations are accomplished with calls to the sem_wait, sem_trywait, and sem_post functions. You use these functions for named and unnamed semaphores. To retrieve the value of a counting semaphore, use the sem_getvalue function.

When the application is finished with an unnamed semaphore, the semaphore name is destroyed with a call to sem_destroy. To deallocate a named semaphore, call the sem_close function. The sem_unlink function removes a named semaphore. The semaphore is removed only when all processes using the semaphore have deallocated it using the sem_close function.

9.2.1 Creating and Opening a Semaphore

A call to the sem_init function creates an unnamed counting semaphore with a specific value. If you specify a non-zero value for the pshared argument, the semaphore can be shared between processes. If you specify the value zero, the semaphore can be shared among threads of the same process.

The sem_open function establishes a connection between a named semaphore and the calling process. Two flags control whether the semaphore is created or only accessed by the call. Set the O_CREAT flag to create a semaphore if it does not already exist. Set the O_EXCL flag along with the O_CREAT flag to indicate that the call to sem_open should fail if the semaphore already exists.

Subsequent to creating a semaphore with either sem_init or sem_open, the calling process can reference the semaphore by using the semaphore descriptor address returned from the call. The semaphore is available in subsequent calls to the sem_wait, sem_trywait, and sem_post functions, which control access to the shared resource. You can also retrieve the semaphore value by calls to sem_getvalue.

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 by a call to the sem_open function. After the initial call to the sem_init or sem_open function to establish the semaphore, 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. It is strongly recommended that the full pathname be used, such as /tmp/mysem1. The directory must exist for the call to succeed.

On the first call to the sem_init or sem_open function, the semaphore is initialized to the value specified in the call.

The following example initializes an unnamed semaphore with a value of 5, which can be shared among processes.

```
/*
 * Initializes an unnamed semaphore with a value of 5 which can be shared
 * between related processes
 */

#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <semaphore.h>

...
```

```
sem_t   mysem;
int     pshared = TRUE;
unsigned int value = 5;
int     sts;

...

sts = sem_init(&mysem, pshared, value);
if (sts) {
  perror("sem_init() failed");
}
```

The following example creates a semaphore named /tmp/mysem with a value of 3:

```
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/stat.h>

...

sem_t   *mysemp;
int     oflag = O_CREAT;
mode_t  mode = 0644;
const char semname[] = "/tmp/mysem"
unsigned int value = 3;
int     sts;

...

mysemp = sem_open(semname, oflag, mode, value);
if (mysemp == (void *)-1) {
  perror(sem_open() failed ");
}
```

To access a previously created semaphore, a process must call the sem_open function using the name of the semaphore.

To determine the value of a previously created semaphore, use the sem_getvalue function. Pass the semaphore and the location for storing the value to the function; it returns the value of the semaphore specified when the sem_init or sem_open function was called.

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

9.2.2 Locking and Unlocking Semaphores

After you create the semaphore with a call to the sem_init or sem_open function, you can use the sem_wait, sem_trywait, and sem_post functions to lock and unlock the semaphore.

Using semaphores to share resources among processes works only if processes unlock a resource immediately after they finish using it. As you code your application, do not attempt to unlock a semaphore you did not previously lock.

To lock a semaphore, you can use either the sem_wait or sem_trywait function. If the semaphore value is greater than zero, the sem_wait function locks the specified semaphore. If the semaphore value is less than or equal to zero, the process is blocked (sleeps) and must wait for another process to release the semaphore and increment the semaphore value.

To be certain that the process is not blocked while waiting for a semaphore to become available, use the sem_trywait function. The sem_trywait 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_trywait function. If not, the sem_trywait function returns a –1 and errno is set to EAGAIN.

Example 9–1 locks a semaphore by using the sem_trywait function.

Example 9–1 Locking a Semaphore

```
...
int  oflag = 0;  /* open an existing semaphore; do not create new one */
...
mysemp = sem_open(semname, oflag, mode, value);
if (mysemp == (void *)-1) {
  perror(sem_open() failed ");
}
sts = sem_trywait(mysemp);
if (sts == 0)
     printf("sem_trywait() succeeded!\n");
else if (errno == EAGAIN)
     printf("semaphore is locked\n");
else
  perror("sem_trywait() failure");
```

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 semaphore, the highest priority process is allowed access to the semaphore first.

9.2.3 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 semaphore value is equal to or less than zero. 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 semaphores. Therefore, semaphore locks are separate from process priorities. Be careful when designing the use of semaphores in your application.

9.2.4 Closing a Semaphore

When an application is finished using an unnamed semaphore, it should destroy the semaphore with a call to the `sem_destroy` function. For named semaphores, the application should deallocate the semaphore with a call to the `sem_close` function. The semaphore name is disassociated from the process. A named semaphore is removed using the `sem_unlink` function, which takes effect once all processes using the semaphore have deallocated the semaphore with calls to `sem_close`. If needed, the semaphore can be reopened for use through a call to the `sem_open` function. Since semaphores are persistent, the state of the semaphore is preserved, even though the semaphore is closed. When you reopen the semaphore, it will be in the state it was when it was closed, unless altered by another process.

As with other interprocess communication methods, you can set up a signal handler to remove the semaphore as one of the tasks performed by the last process in your application.

When the controlling process is finished using an unnamed semaphore, remove the semaphore from memory as follows:

```
/*
 * Removing unnamed semaphore
 */

...

sts = sem_destroy(&mysem);
```

When the controlling process is finished using a named semaphore, close and unlink the semaphore as follows:

```
/*
 * Closing named semaphore and then unlinking it
 */

...

sts = sem_close(mysemp);
sts = sem_unlink(semname);
```

9.3 Semaphore Example

It is important that two processes not write to the same area of shared memory at the same time. 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 9–2 consists of two programs, both of which open the shared-memory object. The two processes, writer and reader, use semaphores to ensure that they have exclusive, alternating access to a shared memory region.

The `writer.c` program creates the semaphore with a call to the `sem_open` function. The `reader.c` program opens the semaphore previously created by the `writer.c` program. Because the `writer.c` program creates the semaphore, `writer.c` needs to be executed before `reader.c`.

Example 9–2 Using Semaphores and Shared Memory

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

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

#include <unistd.h>
#include <semaphore.h>
#include <errno.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;
  sem_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);

  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 in locked state */
```

(continued on next page)

Example 9–2 (Cont.) Using Semaphores and Shared Memory

```
 sem_des = sem_open(sem_fn, O_CREAT, 0644, 0);

 if(sem_des == (void*)-1){
   perror("sem_open failure");
   exit();
 }

    /* 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);
  munmap(shmptr, SHM_SIZE);

  /* Close the shared memory object */

  close(shmdes);

  /* Close the Semaphore */

  sem_close(semdes);

  /* Delete the shared memory object */

  shm_unlink(shm_fn);

}
/*****************************************************************
*****************************************************************/

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

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

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

/**** READER ****/
```

(continued on next page)

Example 9–2 (Cont.) Using Semaphores and Shared Memory

```
main(){
  caddr_t shmptr;
  int shmdes, index;
  sem_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 Semaphore */

 semdes = sem_open(sem_fn, 0, 0644, 0);

 if(semdes == (void*) -1){
   perror("sem_open failure");
   exit();
 }

 /* Lock the semaphore */

 if(!sem_wait(semdes)){

  /* 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);

  }

  munmap(shmptr, SHM_SIZE);

  /* Close the shared memory object */

  close(shmdes);

  /* Close the Semaphore */
```

(continued on next page)

Example 9–2 (Cont.) Using Semaphores and Shared Memory

```
  sem_close(semdes);
  sem_unlink(sem_fn);
}
```

10
Messages

Message queues work by exchanging data in buffers. Any number of processes can communicate through message queues, regardless of whether they are related; if a process has adequate access permission, it can send or receive messages through the queue. Message notification can be synchronous or asynchronous. Message queues can store multiple messages, be accessed by multiple processes, be read in any order, and be prioritized according to application needs.

This chapter includes the following sections:

- Message Queues, Section 10.1
- The Message Interface, Section 10.2
- Message Queue Examples, Section 10.3

10.1 Message Queues

The POSIX 1003.1b message passing facilities provide a deterministic, efficient means for interprocess communication (IPC). Realtime message passing is designed to work with shared memory in order to accommodate the needs of realtime applications with an efficient, deterministic mechanism to pass arbitrary amounts of data between cooperating processes. Predictability is the primary emphasis behind the design for realtime message passing.

Cooperating processes can send and receive messages by accessing system-wide message queues. These message queues are accessed through names that may be pathnames.

The maximum size of each message is defined by the system to optimize the message sending and receiving functions. Message buffers are preallocated, ensuring the availability of resources when they are needed.

If your application involves heavy message traffic, you can prioritize the order in which processes receive messages by assigning a priority to the message or controlling the priority of the receiving process.

Asynchronous notification of the availability of a message on a queue allows a process to do useful work while waiting to receive a message.

Message passing operations that contribute to kernel overhead have been eliminated in the realtime message queue interface. If your application requires the ability to wait on multiple message queues simultaneously or the broadcast of a single message to multiple queues, you may need to write this functionality into your application.

10.2 The Message Interface

The message queue interface is a set of structures and data that allows you to use a message queue for sending and receiving messages. The message queue is a linked list that serves as a holding place for messages being sent to and received by processes sharing access to the message queue.

Table 10–1 lists the POSIX 1003.1b message queue functions that allow you controlled access to messaging operations on a message queue.

Table 10–1 Message Functions

Function	Description
`mq_close`	Closes a message queue
`mq_unlink`	Removes a message queue
`mq_getattr`	Retrieves the attributes of a message queue
`mq_notify`	Requests that a process be notified when a message is available on a queue
`mq_open`	Opens a message queue
`mq_receive`	Receives a message from a queue
`mq_send`	Sends a message to a queue
`mq_setattr`	Sets the attributes of a message queue

General usage for message queues is as follows:

1. Get a message queue descriptor with a call to the `mq_open` function.
2. Send and receive messages with calls to the `mq_send` and `mq_receive` functions.
3. Close the message queue with a call to the `mq_close` function.
4. Remove the message queue with a call to the `mq_unlink` function.

Data written to a message queue created by one process is available to all processes that open the same message queue. Message queues are persistent; once unlinked, their names and contents remain until all processes that have opened the queue call the mq_close function. Child processes inherit the message queue descriptor created by the parent process. Once the message queue is opened, the child process can read or write to it according to access permissions. Unrelated processes can also use the message queue, but must first call the mq_open function to establish the connection.

You can identify message queue attributes with a call to the mq_getattr function. You can specify whether the message operation is blocking or non-blocking by calling the mq_setattr function.

A call to the mq_receive function receives the oldest, highest-priority message on the queue. If two or more processes are waiting for an incoming message on the same queue, the process with the highest priority that has been waiting the longest receives the next message.

Often message queues are created and used only while an application is executing. The mq_unlink function removes (deletes) the message queue and its contents, unless processes still have the queue open. The message queue is deleted only when all processes using it have closed the queue.

10.2.1 Opening a Message Queue

To set up a message queue, first create a new message queue or open an existing queue using the mq_open function. If a message queue of the specified name does not already exist, a new message queue is allocated and initialized. If one already exists, the mq_open function checks permissions.

A process can create and open message queues early in the life of the application. Use the mq_open function to open (establish a connection to) a message queue. After a process opens the message queue, each process that needs to use it must call the mq_open function specifying the same pathname.

The mq_open function provides a set of flags that prescribe the characteristics of the message queue for the process and define access modes for the message queue. Message queue access is determined by the OR of the file status flags and access modes listed in Table 10–2.

Table 10–2 Status Flags and Access Modes for the mq_open Function

Flag	Description
O_RDONLY	Open for read access only
O_WRONLY	Open for write access only
O_RDWR	Open for read and write access
O_CREAT	Create the message queue, if it does not already exist
O_EXCL	When used with O_CREAT, creates and opens a message queue if a queue of the same name does not already exist. If a message queue of the same name exists, the message queue is not opened.
O_NONBLOCK	Determines whether a send or receive operation is blocking or nonblocking

The first process to call the `mq_open` function should use the O_CREAT flag to create the message queue, to set the queue's user ID to that of the calling process, and to set the queue'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 message queue. All processes that subsequently open the message queue must have the same access permission as the creating process.

Each process that uses a message queue must begin by calling the `mq_open` function. This call can accomplish several objectives:

- Create and open the message queue, if it does not yet exist (specify the O_CREAT flag).
- Open an existing message queue.
- Attempt to create and open the queue but fail if the queue already exists (specify both the O_CREAT and O_EXCL flags).
- Open access to the queue for the calling process and establish a connection between the queue and a descriptor. All threads within the same process using the queue use the same descriptor.
- Specify the access mode for the process:
  - — Read only
  - — Write only
  - — Read/write

- Specify whether the process will block or fail when unable to send a message (the queue is full) or receive a message (the queue is empty) with the *oflags* argument.

The *mode* bit is checked to determine if the caller has permission for the requested operation. 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.

For example, the following code creates a message queue and, if it does not already exist, opens it for read and write access.

```
fd = mq_open("new_queue", (O_CREAT|O_EXCL|O_RDWR);
```

Once a message queue is created, its name and resources are persistent. It exists until the message queue is unlinked with a call to the `mq_unlink` function and all other references to the queue are gone.

The message flag parameter is either 0 or O_NONBLOCK. If you specify a flag of 0, then a sending process sleeps if the message cannot be sent to the specified queue, due to the queue being full. The process will sleep until other messages have been removed from the queue and space becomes available. When the flag is specified as O_NONBLOCK, the `mq_send` function returns immediately with an error status.

Example 10–1 shows the code sequence to establish a connection to a message queue descriptor.

Example 10–1 Opening a Message Queue

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

main ()
      int md;
      int status;
                                        /* Create message queue */
      md = mq_open ("my_queue", O_CREAT|O_RDWR);

/*
 * code to close and unlink the message queue goes here
 */
```

(continued on next page)

Example 10–1 (Cont.) Opening a Message Queue

```
status = mq_close(md);              /* Close message queue  */
status = mq_unlink("my_queue");     /* Unlink message queue */
```

Use the same access permissions that you would normally use on a call to the file `open` function. If you intend to only read the queue, specify read permission only on the `mq_open` function. If you intend to read and write to the queue, open the queue with both read and write permissions.

When finished using a message queue, close the queue with the `mq_close` function, and remove the queue by calling the `mq_unlink` function.

10.2.2 Sending and Receiving Messages

For an application in which the intended recipients of messages might be ambiguous because they all use a single message queue, you can establish multiple queues. In some cases you may need to provide a separate queue for each process that receives a message. Two processes that carry on two-way communication between them normally require two message queues:

- Process X sends messages to queue A; process Y receives from it
- Process Y sends messages to queue B; process X receives from it

Use of a single queue by multiple processes could be appropriate for an application that collects and processes data. Consider an application that consists of five processes that monitor data points and a sixth process that accumulates and interprets the data. Each of the five monitoring processes could send information to a single message queue. The sixth process could receive the messages from the queue, with assurance that it is receiving information according to the specified priorities of the incoming messages, in first-in first-out order within each priority.

When a process receives a message from a queue, it removes that message from the queue. Therefore, an application that requires one process to send the same message to several other processes should choose one of the following communication methods:

- Set up a message queue for each receiving process, and send each message to each queue
- Communicate by using signals and shared memory

Once a message queue is open, you can send messages to another process using the mq_send function. The mq_send function takes four parameters, including: the message queue descriptor, a pointer to a message buffer, the size of the buffer, and the message priority. The read/write permissions are checked along with the length of the message, the status of the message queue, and the message flag. If all checks are successful, the message is added to the message queue. If the queue is already full, the sending process can block until space in the queue becomes available, or it can return immediately, according to whether it set the O_NONBLOCK flag when it called the mq_open function.

Once a message has been placed on a queue, you can retrieve the message with a call to the mq_receive function. The mq_receive function includes four parameters: the message queue descriptor, a pointer to a buffer to hold the incoming message, the size of the buffer, and the priority of the message received (the priority is returned by the function). The size of the buffer must be at least the size of the message queue's size attribute.

As with the mq_send function, the read/write operation permissions are checked on a call to the mq_receive function. If more than one process is waiting to receive a message when a message arrives at an empty queue, then the process with the highest priority that has been waiting the longest is selected to receive the message.

When a process uses the mq_receive function to read a message from a queue, the queue may be empty. The receiving process can block until a message arrives in the queue, or it can return immediately, according to the state of the O_NONBLOCK flag established with a preceding call to the mq_open function.

10.2.3 Asynchronous Notification of Messages

A process that wants to read a message from a message queue has three options:

- Set the queue to blocking mode, and wait for a message to be received by calling mq_receive
- Set the queue to non-blocking mode, and call mq_receive multiple times until a message is received
- Set the queue to non-blocking mode, and call mq_notify specifying a signal to be sent when the queue goes from empty to non-empty

The last option is a good choice for a realtime application. The mq_notify function is used to register a request for asynchronous notification by a signal when a message becomes available on a previously empty queue. The process can then do useful work until a message arrives, at which time a signal is sent according to the signal information specified in the *notification* argument of

the mq_notify function. After notification, the process can call mq_receive to receive the message.

Only one notification request at a time is allowed per message queue descriptor. The previous notification request is canceled when another signal is sent; thus, the request must be re-registered by calling mq_notify again.

10.2.4 Prioritizing Messages

A process can control the relative priority of messages it sends to a specified queue by setting the *msg_prio* parameter in the mq_send function.

If *msg_prio* is specified on the mq_send function, the message is inserted into the message queue according to its priority relative to other messages on the queue. A message with a larger numeric value (higher priority) is inserted into the queue before messages with a lower numeric value. The mq_receive function always returns the first message on the queue, so if you assign higher priorities to messages of higher importance, you can receive the most important messages first. If you assign lower priorities to less important messages, you can delay delivery of the messages as more important messages are sent. Messages of equal priority are inserted in a first-in, first-out manner. The ability to assign priorities to messages on the queue reduces the possibility of priority inversion in the realtime messaging interface.

10.2.5 Using Message Queue Attributes

Use the mq_getattr function to determine the message queue attributes of an existing message queue. The attributes are as follows:

- *mq_flags* — The message queue flags
- *mq_maxmsg* — The maximum number of messages allowed
- *mq_msgsize* — The maximum message size allowed for the queue
- *mq_curmsgs* — The number of messages on the queue

The *mq_curmsgs* attribute describes the current queue status. If necessary, call the mq_setattr function to reset the flags. The *mq_maxmsg* and *mq_msgsize* attributes cannot be modified after the initial queue creation. The mqueue.h header file contains information concerning system-wide maximums and other limits pertaining to message queues.

10.2.6 Closing and Removing a Message Queue

Each process that uses a message queue should close its access to the queue by calling the `mq_close` function before exiting. When all processes using the queue have called this function, the software removes the queue.

A process can remove a message queue by calling the `mq_unlink` function. However, if other processes still have the message queue open, the `mq_unlink` function returns immediately and destruction of the queue is postponed until all references to the queue have been closed.

10.3 Message Queue Examples

Example 10–2 creates a message queue and sends a loop of messages. The message queue is created using O_CREAT.

Example 10–2 Using Message Queues to Send Data

```
/*
 * test_send.c
 *
 * This test goes with test_receive.c.
 * test_send.c does a loop of mq_sends,
 * and test_receive.c does a loop of mq_receives.
 */
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <sys/fcntl.h>
#include <signal.h>
#include <sys/rt_syscall.h>
#include <mqueue.h>
#include <errno.h>

#define PMODE 0666
extern int errno;
```

(continued on next page)

Example 10–2 (Cont.) Using Message Queues to Send Data

```
int main()
{
int i;
int status = 0;
mqd_t mqfd;
char msg_buffer[P4IPC_MSGSIZE];
struct mq_attr attr;
int open_flags = 0;
int num_bytes_to_send;
int priority_of_msg;

printf("START OF TEST_SEND \n");

/* Fill in attributes for message queue */
attr.mq_maxmsg = 20;
attr.mq_msgsize = P4IPC_MSGSIZE;
attr.mq_flags   = 0;

/* Set the flags for the open of the queue.
 * Make it a blocking open on the queue, meaning it will block if
 * this process tries to send to the queue and the queue is full.
 * (Absence of O_NONBLOCK flag implies that the open is blocking)
 *
 * Specify O_CREAT so that the file will get created if it does not
 * already exist.
 *
 * Specify O_WRONLY since we are only planning to write to the queue,
 * although we could specify O_RDWR also.
 */
open_flags = O_WRONLY|O_CREAT;

/* Open the queue, and create it if the receiving process hasn't
 * already created it.
 */
mqfd = mq_open("myipc",open_flags,PMODE,&attr);
if (mqfd == -1)
    {
    perror("mq_open failure from main");
    exit(0);
    };
```

(continued on next page)

Example 10–2 (Cont.) Using Message Queues to Send Data

```
/* Fill in a test message buffer to send */
msg_buffer[0] = 'P';
msg_buffer[1] = 'R';
msg_buffer[2] = 'I';
msg_buffer[3] = 'O';
msg_buffer[4] = 'R';
msg_buffer[5] = 'I';
msg_buffer[6] = 'T';
msg_buffer[7] = 'Y';
msg_buffer[8] = '1';
msg_buffer[9] = 'a';

num_bytes_to_send = 10;
priority_of_msg = 1;

/* Perform the send 10 times */
for (i=0; i<10; i++)
    {
    status = mq_send(mqfd,msg_buffer,num_bytes_to_send,priority_of_msg);
    if (status == -1)
 perror("mq_send failure on mqfd");
    else
        printf("successful call to mq_send, i = %d\n",i);
    }

/* Done with queue, so close it */
if (mq_close(mqfd) == -1)
    perror("mq_close failure on mqfd");

printf("About to exit the sending process after closing the queue \n");

}
```

Example 10–3 creates a message queue and receives a loop of messages. The message queue is created using O_CREAT.

Example 10–3 Using Message Queues to Receive Data

```
/*
 * test_receive.c
 *
 * This test goes with test_send.c.
 * test_send.c does a loop of mq_sends,
 * and test_receive.c does a loop of mq_receives.
 */
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <sys/fcntl.h>
#include <signal.h>
#include <sys/rt_syscall.h>
#include <mqueue.h>
#include <errno.h>

#define PMODE 0666
extern int errno;

int main()
{
int i;
mqd_t mqfd;
/* Buffer to receive msg into */
char msg_buffer[P4IPC_MSGSIZE];
struct mq_attr attr;
int open_flags = 0;
ssize_t num_bytes_received = 0;
msg_buffer[10] = 0; /* For printing a null terminated string for testing */

printf("START OF TEST_RECEIVE \n");

/* Fill in attributes for message queue */
attr.mq_maxmsg = 20;
attr.mq_msgsize = P4IPC_MSGSIZE;
attr.mq_flags   = 0;
```

(continued on next page)

Example 10–3 (Cont.) Using Message Queues to Receive Data

```
/* Set the flags for the open of the queue.
 * Make it a blocking open on the queue,
 * meaning it will block if this process tries to
 * send to the queue and the queue is full.
 * (Absence of O_NONBLOCK flag implies that
 * the open is blocking)
 *
 * Specify O_CREAT so that the file will get
 * created if it does not already exist.
 *
 * Specify O_RDONLY since we are only
 * planning to write to the queue,
 * although we could specify O_RDWR also.
 */
open_flags = O_RDONLY|O_CREAT;

/* Open the queue, and create it if the sending process hasn't
 * already created it.
 */
mqfd = mq_open("myipc",open_flags,PMODE,&attr);
if (mqfd == -1)
    {
    perror("mq_open failure from main");
    exit(0);
    };

/* Perform the receive 10 times */
for (i=0;i<10;i++)
    {
    num_bytes_received = mq_receive(mqfd,msg_buffer,P4IPC_MSGSIZE,0);
    if (num_bytes_received == -1)
 {
 perror("mq_receive failure on mqfd");
 }
    else
        printf("data read for iteration %d = %s \n",i,msg_buffer);
    }

/* Done with queue, so close it */
if (mq_close(mqfd) == -1)
    perror("mq_close failure on mqfd");

/* Done with test, so unlink the queue,
 * which destroys it.
 * You only need one call to unlink.
 */
if (mq_unlink("myipc") == -1)
    perror("mq_unlink failure in test_ipc");
```

(continued on next page)

Example 10–3 (Cont.) Using Message Queues to Receive Data

```
printf("Exiting receiving process after closing and unlinking queue \n");
}
```

11

Realtime Performance and System Tuning

Chapter 1 describes the basic issues that concern a realtime application, and what services a realtime operating system can provide to users to help meet their realtime needs. It mainly describes issues within the scope of the user's application code itself, such as how to set priority and scheduling priorities, how to lock down process memory, and how to use asynchronous I/O. Chapter 1 also discusses the value of a preemptive kernel in reducing the process preemption latency of a realtime application.

This chapter explores more deeply the latency issues of a system and how they affect the realtime performance of an application. This involves a greater understanding of the interaction of the application with the underlying UNIX system, and with devices involved directly or indirectly with the application. Section 11.2 outlines some ways that a user can improve application performance.

11.1 Realtime Responsiveness

Realtime applications require a predictable response time to external events, such as device interrupts. A typical realtime application involves:

- An interrupt-generating device
- An interrupt service routine that collects data from the device
- User-level code that processes the collected data

Realtime responsiveness is a characterization of how quickly an operating system and an application, working together, can respond to external events. One way of measuring responsiveness is through a system's latency. The time it takes for hardware and the operating system to respond to external events is latency, and is expressed as a delay time. Understanding the causes of latency and minimizing their effects is a key to successful realtime program design, and is the focus of this chapter.

Two types of latency are described in the following sections:

- Interrupt service routine (ISR) latency

- Process dispatch latency (PDL)

11.1.1 Interrupt Service Routine Latency

A system's interrupt service routine (ISR) latency is the elapsed time from when an interrupt occurs until execution of the first instruction in the interrupt service routine. The system must first recognize that an interrupt has occurred, and then dispatch to the ISR code. If critical postprocessing is done in the ISR, then the user must be concerned with completion time of the ISR code, not just the time it takes to begin execution of its first instruction. Thus there are two concerns: ISR latency and ISR execution. There are factors that cause ISR latency and ISR execution to vary in duration, and these factors make it more difficult to assign latency a deterministic value.

The most important factor is the relative interrupt priority level (IPL) at which the ISR executes. When there are other ISRs of equal or greater interrupt priority level running at the time that the realtime device interrupts, the realtime device ISR is blocked from running until the current ISR is finished.

There could be multiple ISRs waiting to execute that have an equal or higher IPL at the time of the realtime interrupt, and all will hold off the realtime ISR until they complete. In addition, once the realtime ISR is running, it can be preempted or held off by one or more devices of higher IPL, and the realtime ISR will be delayed by the collective duration of these ISRs. Thus, it is important to know the relative IPLs of all the devices that could potentially interrupt during critical realtime processing, including system-provided devices such as a network driver or disk driver.

11.1.2 Process Dispatch Latency

Process Dispatch Latency (PDL) is the time it takes from when an interrupt occurs until a process that was blocked waiting on the interrupt executes. Process dispatch latency includes:

- ISR latency
- ISR execution time
- Time required to return from the ISR
- Time required for the context switch back to the process-level code which is waiting on the interrupt

There are many more factors that can potentially increase the process dispatch latency of a realtime application. Any process that is currently executing code that holds a simple lock, is funneled to the master process, or has its IPL raised, will not be preemptable by the realtime process and thus will hold off the realtime process from running. (Note that a user process cannot hold a

simple lock, be funneled to the master process, or have its IPL raised, except through a system call.) Once a process is able to run, it must compete against other processes in order to actually run, and the process with the highest priority will run.

Note that process priority can affect PDL, but cannot affect ISR latency. In other words, no matter how high the priority of an application process, even if it is in the realtime priority range, all ISRs that need servicing at the time that the realtime device's ISR needs servicing will be serviced before process code can execute, no matter in what order or at what interrupt priority level the ISRs run.

11.2 Improving Realtime Responsiveness

This section contains guidelines for improving realtime responsiveness.

Minimize Paging by Locking Down Memory

Be sure that there is sufficient memory on your system, and always lock down memory in the user process to reduce paging. Paging will occur when there are many threads and processes running on the system that do not collectively fit into system memory, and must be paged in and out as necessary. Application code and data that are locked in memory will not be paged. Paging affects process dispatch latency because it executes code in the kernel that is protected by simple locks, and thus cannot be preempted. Note that certain system daemons are not locked in memory, so a secondary effect is paging from those systems.

Turn On Kernel Preemption

Turn on kernel preemption and set your application code scheduling priority to SCHED_FIFO. This is described in Chapter 2.

Manage Priorities

Always consider the process priority level of your application in terms of relative importance in the overall system. You may need to use priorities in the realtime range. This affects process dispatch latency when there are other processes ready to run at the same time that the realtime application is ready to run. The process with the highest priority that has been waiting the longest among the waiting processes of that priority will run first.

Note, however, that always running in the realtime priority range is not necessarily what you should do. If you need to interact with system services that have threads or processes associated with them such as the network, you need to run at a priority at or below the priority of those threads or processes, as well as the priority of anything on which those threads or processes depend.

In the kernel, there are multiple threads. The purpose of these threads is to perform activities that have the potential of blocking, and thus serve as the delivery mechanism of information between ISRs and user processes. These kernel threads do not have much of the state information that processes have.

Kernel threads use the first-in first-out scheduling policy, and are scheduled along with POSIX processes. The kernel sets priorities as Mach priorities, which are the inverse of POSIX priorities: 0 is the highest priority Mach thread and 63 is the lowest. Under POSIX, 64 is the highest priority and 0 is the lowest.

You can use the `ps` command to display thread priorities. Because the `ps` program predates the use of threads, its ability to display information clearly about threads is limited. The following example shows an example of using the command `ps axm -o L5FMT,psxpri` to display L5FMT format and append the POSIX priority field:

```
% ps axm -o L5FMT,psxpri
       F S     UID   PID  PPID    C PRI  NI   ADDR   SZ WCHAN    TTY            TIME CMD          PPR
       3 R <     0     0     0  0.0  32 -12      0 3.4M *        ??         05:02:40 kernel idle   31
         R N                    0.0  63  19             -                    0:00.00                0
         U <                    0.0  38  -6           malloc_                0:00.51               25
         U <                    0.0  32 -12           402cb0                 0:49.47               31
         U <                    0.0  32 -12           402eac                 0:00.00               31
         S <                    0.0  33 -11           netisr                05:01:23               30
         U <                    0.0  32 -12           3e3f18                 0:00.00               31
         U <                    0.0  38  -6           4c3b80                 0:00.00               25
         U                      0.0  42   0           ubc_dir                0:00.52               21
         U <                    0.0  37  -7           4c2678                 0:00.01               26
         U <                    0.0  37  -7           4c2680                 0:03.77               26
         U <                    0.0  38  -6           4c33b0                 0:12.69               25
         U <                    0.0  32 -12           4e36d8                 0:00.01               31
         U <                    0.0  37  -7           4e36d8                 0:00.12               26
         U <                    0.0  37  -7           4ba2d8                 0:00.00               26
         U <                    0.0  38  -6           4e3078                 0:00.00               25
         U <                    0.0  42  -2           24ce30                 0:00.03               21
         I                      0.0  42   0           nfsiod_                0:01.49               21
         I                      0.0  42   0           nfsiod_                0:01.65               21
         I                      0.0  42   0           nfsiod_                0:01.82               21
         I                      0.0  42   0           nfsiod_                0:00.61               21
         I                      0.0  42   0           nfsiod_                0:01.71               21
         I                      0.0  44   0           nfsiod_                0:01.26               19
         I                      0.0  42   0           nfsiod_                0:01.78               21
80048001 I       0     1     0  0.0  44   0      0  40K pause    ??          0:03.12 init          19
    8001 IW      0     3     1  0.0  44   0      0   0K sv_msg_  ??          0:00.12 kloadsrv      19
    8001 S       0    17     1  0.0  44   0      0  48K pause    ??         03:58:06 update        19
    8001 I       0    81     1  0.0  44   0      0 120K event    ??          0:02.64 syslogd       19
    8001 IW      0    83     1  0.0  42   0      0   0K event    ??          0:00.03 binlogd       21
    8001 S       0   135     1  0.0  44   0      0  80K event    ??          8:13.21 routed        19
    8001 S       0   226     1  0.0  44   0      0 104K event    ??          8:25.31 portmap       19
    8001 IW      0   234     1  0.0  44   0      0   0K event    ??          0:00.21 ypbind        19
    .
    .
    .
```

You can use the dbx command from a root account to display more information about kernel threads, as follows:

```
# dbx -k /vmunix
(dbx) set $pid=0
(dbx) tlist              [shows kernel threads]
(dbx) tset thread-name;t [shows which routine a thread is running]
(dbx) p thread->sched_pri [shows Mach priority for the current thread]
```

The following example shows use of the dbx command:

```
# dbx -k /vmunix
dbx version 3.11.8
Type 'help' for help.

stopped at  [thread_block:2020 ,0xfffffc00002a1da0]      Source not available

warning: Files compiled -g3: parameter values probably wrong
(dbx) set $pid=0
(dbx) tlist
thread 0xfffffc0003fd1be8 stopped at [thread_run:2388 ,0xfffffc00002a2560]   Source not available
thread 0xfffffc0003fd6000 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd62c0 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd6580 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd6dc0 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd7080 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd7340 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd7600 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd78c0 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd7b80 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6a000 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6a2c0 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6a580 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6a840 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6ab00 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6adc0 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003fd1950 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6b080 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6b340 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6b600 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6b8c0 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0003f6bb80 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
thread 0xfffffc0000926000 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
(dbx) tset 0xfffffc0003f6bb80;t
thread 0xfffffc0003f6bb80 stopped at [thread_block:2020 ,0xfffffc00002a1da0] Source not available
>  0 thread_block() ["/usr/sde/osf1/build/ptos.bl8/src/kernel/kern/sched_prim.c":2017,
      0xfffffc00002a1d9c]
   1 async_io_thread(0x0, 0x0, 0x0, 0x0, 0x0) ["../../../../src/kernel/nfs/nfs_vnodeops.c":2828,
      0xfffffc00002f4898]
(dbx) p thread->sched_pri
44
```

Manage Physical Memory

By default, the parameter *ubc_maxpercent* in the file /sys/conf/param.c is set to 100. That means that up to 100% of physical memory can be consumed by the Unified Buffer Cache (UBC) for buffering file data. Some systems perform better when not all physical memory is allowed to be taken by the UBC.

For improved realtime responsiveness, change this the value of /sys/conf /param.c to between 50 and 80, depending on the amount of file system activity done on the system. This can improve system realtime latency, because when the UBC has consumed its maximum allocation of memory for buffering file data, the least recently used buffers must be flushed to disk if they are modified. Flushing these buffers is done with a simple lock held, and therefore can effect process dispatch latency. The more memory that the UBC is allowed to use before flushing, the longer it will take to perform the flushing. Lowering the value of the *ubc_maxpercent* parameter will cause the flushing to occur more frequently, but take less time.

Write Effective Device Drivers

When writing device drivers, follow these guidelines:

- Avoid holding locks for long periods

  Holding a lock prevents context switches from occurring.

- Avoid funneling

  Funneled device drivers take a lock upon entry.

- Interrupt service routines should be brief

  Consider use of a kernel thread to do ISR postprocessing. While an ISR is executing, other interrupts of equal or lower IPL are delayed, and no process can run until all ISR activity is completed. Consider use of the rt_post_callout function for ISR postprocessing that needs to execute before any process code, but after any ISRs. See *Writing Device Drivers: Tutorial* for information about the rt_post_callout function.

Avoid Configuring Peripheral Devices in the System

Use devices with care that could interfere with realtime responsiveness, such as:

a. The network driver

   Do not configure the network driver into your system if it is not a necessary part of your realtime application. If it is necessary, then be sure that it is used only in postprocessing, and not during critical phases of your application when you are attempting to minimize latency.

b. The disk driver

   Be sure that postprocessing data is written to permanent storage during non-critical sections of your application, and that all data is properly flushed and synchronized to disk at appropriate times. See Chapter 8 for more information about synchronized I/O.

In general, keep all peripheral devices that can cause spurious interrupts out of the configuration of the most critical systems. Other devices can possibly cause interrupt latency as well as bus contention with the critical devices. If other devices are a necessary part of the system, analyze the interrupt rate and attempt to avoid interrupt overload on the system.

Consider Use of Symmetrical Multiprocessing

Consider a symmetrical multiprocessing (SMP) system as a possible means of improving realtime responsiveness, by dividing the application across multiple processors using the `runon` command.

A

Digital UNIX Realtime Functional Summary

This appendix 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 POSIX 1003.1b. 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
- P1003.1b Priority Scheduling
- P1003.1b Clocks
- Date and Time Conversion
- P1003.1b Timers
- BSD Clocks and Timers
- P1003.1b Memory Locking
- System V Memory Locking
- P1003.1b Asynchronous I/O
- POSIX Synchronized I/O
- BSD Synchronized I/O
- P1003.1b Messages
- P1003.1b Shared Memory
- P1003.1b Semaphores
- POSIX 1003.1b Realtime Signals
- Signal Control and Other Signal Operations
- sigsetops Primitives

- Process Ownership
- Input and Output
- Device Control
- System Database

Table A–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 A–2 P1003.1b 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_rr_get_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 A–3 P1003.1b Clocks

Function	Purpose
clock_gettime	Returns the current value for the specified clock
clock_getres	Returns the resolution and maximum value of the specified clock
clock_settime	Sets the specified clock to the specified value

Table A–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
difftime	Computes the difference between two calendar times (time1–time0) and returns the difference expressed in seconds

(continued on next page)

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

Function	Purpose
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 A–5 P1003.1b 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_getoverrun	Returns the timer expiration overrun count for the 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 A–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 A–6 (Cont.) BSD Clocks and Timers

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

Table A–7 P1003.1b 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 A–8 System V Memory Locking

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

Table A–9 P1003.1b 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_fsync	Asynchronously writes changes in a file to permanent storage
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

(continued on next page)

Table A–9 (Cont.) P1003.1b Asynchronous I/O

Function	Purpose
aio_write	Initiates a write request to the specified file descriptor
lio_listio	Initiates a list of requests

Table A–10 POSIX Synchronized I/O

Function	Purpose
fcntl	Performs controlling operations on the specified open file
fdatasync	Writes changes to a file to permanent storage — saves all modified data, and only file system information needed to access the data
fsync	Writes changes to a file to permanent storage — saves all modified data and file control information

Table A–11 BSD Synchronized I/O

Function	Purpose
sync	Updates all file systems — all information in memory that should be on disk is written out

Table A–12 P1003.1b Messages

Function	Purpose
mq_close	Closes a message queue
mq_getattr	Returns the status and attributes of a message queue
mq_notify	Attaches a request for asynchronous signal notification to a message queue
mq_open	Opens a message queue
mq_receive	Receives the oldest, highest-priority message from the message queue

(continued on next page)

Table A–12 (Cont.) P1003.1b Messages

Function	Purpose
mq_send	Places a message in the message queue
mq_setattr	Sets the attributes associated with a message queue
mq_unlink	Removes a message queue

Table A–13 P1003.1b 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 A–14 P1003.1b Semaphores

Function	Purpose
sem_close	Deallocates the specified semaphore
sem_destroy	Removes or destroys the specified semaphore
sem_getvalue	Gets the value of a specified semaphore
sem_trywait	Conditionally performs a semaphore lock on a semaphore
sem_init	Creates a new semaphore
sem_open	Opens a semaphore for use by a process
sem_post	Releases a locked semaphore
sem_wait	Performs a semaphore lock on a semaphore

Table A–15 POSIX 1003.1b Realtime Signals

Function	Purpose
sigaction	Examines or specifies the action taken on delivery of a specified signal

(continued on next page)

Table A–15 (Cont.) POSIX 1003.1b Realtime Signals

Function	Purpose
sigqueue	Sends and queues the specified signal with optional data delivery to the specified process
sigtimedwait	For a specified period of time, suspends a calling thread until a signal arrives
sigwaitinfo	Suspends a calling thread until a signal arrives

Table A–16 Signal Control and Other Signal Operations

Function	Purpose
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
sigwait	Suspends a calling thread until a signal arrives

Table A–17 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
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 A–18 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 A–19 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
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 A–20 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 A–21 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
getpwuid	Returns user information when passed a user ID

Index

A

C

D

E

F

G

M

Q

R

S

T

U

W

Y