RedHawk™ Linux® User's Guide

0898004-300
April 2003

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2881 Gateway Drive, Pompano Beach, Florida, 33069. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- August 2002	000	RedHawk Linux Release 1.1
Previous Release -- December 2002	200	RedHawk Linux Release 1.2
Current Release -- April 2003	300	RedHawk Linux Release 1.3

Preface

Scope of Manual

This manual consists of three parts. The information in Part 1 is directed towards real-time users. Part 2 is directed towards system administrators. Part 3 consists of backmatter: appendix and index. An overview of the contents of the manual follows.

Structure of Manual

This guide consists of the following sections:

Part 1 - Real-Time User

- Chapter 1, *Introduction to RedHawk Linux*, provides an introduction to the RedHawk Linux operating system and gives an overview of the real-time features included.
- Chapter 2, *Real-Time Performance*, discusses issues involved with achieving real-time performance including interrupt response, process dispatch latency and deterministic program execution. The shielded CPU model is described.
- Chapter 3, *Real-Time Interprocess Communication*, discusses procedures for using the POSIX and System V message-passing facilities.
- Chapter 4, *Process Scheduling*, provides an overview of process scheduling and describes POSIX® scheduling policies and priorities.
- Chapter 5, *Interprocess Synchronization*, describes the interfaces provided by RedHawk Linux for cooperating processes to synchronize access to shared resources. Included are: POSIX counting semaphores, System V semaphores, rescheduling control tools and condition synchronization tools.
- Chapter 6, *Programmable Clocks and Timers*, provides an overview of some of the RCIM and POSIX timing facilities that are available under RedHawk Linux.
- Chapter 7, *System Clocks and Timers*, describes the per-CPU local timer and the system global timer.
- Chapter 8, *File Systems and Disk I/O*, explains the xfs journaling file system and procedures for performing direct disk I/O on the RedHawk Linux operating system.
- Chapter 9, *Memory Mapping*, describes the methods provided by RedHawk Linux for a process to access the contents of another process' address space.

Part 2 - Administrator

- Chapter 10, *Configuring and Building the Kernel*, provides information on how to configure and build a RedHawk Linux kernel.
- Chapter 11, *Linux Kernel Crash Dump (LKCD)*, provides guidelines for saving, restoring and analyzing the kernel memory image using LKCD.
- Chapter 12, *Pluggable Authentication Modules (PAM)*, describes the PAM authentication capabilities of RedHawk Linux.
- Chapter 13, *Device Drivers and Real Time*, describes real-time issues involved with writing device drivers for RedHawk Linux.

Part 3 - Common Material

- *Appendix A* contains an example program illustrating the System V message queue facility.
- The *Index* contains an alphabetical reference to key terms and concepts and the pages where they occur in the text.

Syntax Notation

The following notation is used throughout this manual:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *itali*cs.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appear in `list` type.
`[ ]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such options or arguments

Related Publications

Title	Pub No.
RedHawk Linux Release Notes Version x.x	0898003
RedHawk Linux Frequency-Based Scheduler (FBS) User's Guide	0898005
Real-Time Clock and Interrupt Module (RCIM) User's Guide	0898007

where *x.x* = release version

Contents

Chapter 3 Real-Time Interprocess Communication

Chapter 4 Process Scheduling

Chapter 5 Interprocess Synchronization

Chapter 6 Programmable Clocks and Timers

Chapter 7 System Clocks and Timers

Chapter 8 File Systems and Disk I/O

Chapter 9 Memory Mapping

Chapter 10 Configuring and Building the Kernel

Chapter 11 Linux Kernel Crash Dump (LKCD)

Chapter 12 Pluggable Authentication Modules (PAM)

Chapter 13 Device Drivers and Real Time

Screens

Illustrations

Tables

1
Introduction to RedHawk Linux

1
Introduction to RedHawk Linux

Overview

The RedHawk™ Linux® operating system is included with each Concurrent iHawk™ system. It is based on the Red Hat® Linux distribution, a unique Linux kernel that has been modified for deterministic real-time processing.

The iHawk Series 860 features from one to eight Intel® Pentium® Xeon™ processors in a single rackmount or tower enclosure. The iHawk 860G is a one or two processor, high-performance AGP/PCI-based platform for real-time imaging applications. The iHawk systems offer leading-edge integrated circuit and packaging technology. iHawks are true symmetric multiprocessors (SMPs) that run a single copy of the RedHawk Linux real-time operating system. All CPUs in a system are linked by a high-speed front-side processor bus and have direct, cache-coherent access to all of main memory.

Except for the kernel, all Red Hat components operate in their standard fashion. These include Linux utilities, libraries, compilers, tools and installer unmodified from Red Hat.

The components that are unique to RedHawk include a modified Linux kernel, some additional user-level libraries and commands and optionally, the NightStar™ real-time application development tools. The RedHawk Linux kernel is based on the kernel maintained by Linus Torvalds and utilizes the 2.4.21-pre4 version.

There are three classes of changes that have been applied to this Linux kernel:

- new kernel features added by Concurrent based on features in the real-time operating systems that Concurrent has been deploying for many years
- open source real-time patches not integrated into standard Linux which supply features or performance benefits for real-time applications
- performance improvements developed by Concurrent to improve worst-case process dispatch latency or to improve determinism in process execution

These kernel enhancements provide the support needed for developing complex real-time applications. The additional user-level libraries in RedHawk Linux provide interfaces to some of the real-time features present in the Red Hat Linux kernel. Descriptions of the real-time features in RedHawk Linux are provided in the section “Real-Time Features in RedHawk Linux” later in this chapter.

Linux conforms to many of the interface standards defined by POSIX, but does not fully conform to these standards. Red Hat Linux has the same level of POSIX conformance as other Linux distributions based on the 2.4 series of kernels. Linux on the Intel x86 architecture has defined a defacto binary standard of its own which allows shrink-wrapped applications that are designed to run on the Linux/Intel x86 platform to run on Concurrent’s iHawk platform.

RedHawk Linux Kernels

There are three flavors of RedHawk Linux kernels installed on the iHawk system. The system administrator can select which version of the RedHawk Linux kernel is loaded via the GRUB boot loader. The three flavors of RedHawk Linux kernels available are:

Kernel Name	Kernel Description
vmlinuz-2.4.21-pre4-RedHawk-*x.x*-trace	Default kernel with trace points but no debug checks
vmlinuz-2.4.21-pre4-RedHawk-*x.x*-debug	Kernel with both debug checks and kernel trace points
vmlinuz-2.4.21-pre4-RedHawk-*x.x*	Kernel with no trace points and no debug checks

Where *x.x* = RedHawk Linux version number; for example, "1.3".

The default RedHawk Linux trace kernel installed on the iHawk system has been built with kernel trace points enabled. The kernel trace points allow the NightTrace™ tool to trace kernel activity.

The debug kernel has been built with both debugging checks and kernel trace points enabled. The debugging checks are extra sanity checks that allow kernel problems to be detected earlier than they might otherwise be detected. However, these checks do produce extra overhead. If you are measuring performance metrics on your iHawk system, this activity would be best performed using a non-debug version of the kernel.

System Updates

RedHawk Linux updates can be downloaded from Concurrent's RedHawk Linux update website. Refer to the *RedHawk Linux Release Notes* for details.

NOTE

It is NOT recommended that the user download Red Hat updates.

Most user-level components on RedHawk Linux are provided by the Red Hat Linux 8.0 distribution. The RedHawk Linux kernel replaces the standard Red Hat kernel. While the RedHawk Linux kernel is likely to work with any version of the Red Hat 8.0 distribution, it is not recommended that the user download additional Red Hat updates as these could potentially destabilize the system. Red Hat updates will be made available to RedHawk Linux users as those updates are validated against the RedHawk Linux kernel.

Real-Time Features in RedHawk Linux

This section provides a brief description of the real-time features included in the RedHawk Linux operating system. It reflects features included in open source Linux patches applied to RedHawk Linux as well as new features and enhancements developed by Concurrent to meet the demands of Concurrent's real-time customers.

More detailed information about the functionality described below is provided in subsequent chapters of this guide. Online readers can display the information immediately by clicking on the chapters referenced.

Processor Shielding

Concurrent has developed a method of shielding selected CPUs from the unpredictable processing associated with interrupts and system daemons. By binding critical, high-priority tasks to particular CPUs and directing most interrupts and system daemons to other CPUs, the best process dispatch latency possible on a particular CPU in a multiprocessor system can be achieved. Chapter 2 presents a model for shielding CPUs and describes techniques for improving response time and increasing determinism.

Processor Affinity

In a real-time application where multiple processes execute on multiple CPUs, it is desirable to have explicit control over the CPU assignments of all processes in the system. This capability is provided by Concurrent through the **`mpadvise(3)`** library routine and the **`run(1)`** command. See Chapter 2 and the man pages for additional information.

User-level Preemption Control

When an application has multiple processes that can run on multiple CPUs and those processes operate on data shared between them, access to the shared data must be protected to prevent corruption from simultaneous access by more than one process. The most efficient mechanism for protecting shared data is a spin lock; however, spin locks cannot be effectively used by an application if there is a possibility that the application can be preempted while holding the spin lock. To remain effective, RedHawk provides a mechanism that allows the application to quickly disable preemption. See Chapter 5 and the **`resched_cntl(2)`** man page for more information about user-level preemption control.

Fast Block/Wake Services

Many real-time applications are composed of multiple cooperating processes. These applications require efficient means for doing inter-process synchronization. The fast block/wake services developed by Concurrent allow a process to quickly suspend itself awaiting a wakeup notification from another cooperating process. See Chapter 2, Chapter 5 and the **`postwait(2)`** and **`server_block(2)`** man pages for more details.

RCIM Driver

A driver has been added for support of the Real-Time Clock and Interrupt Module (RCIM). This multi-purpose PCI card has the following functionality:

- connection of four external device interrupts
- four real time clocks that can interrupt the system
- four programmable interrupt generators which allow generation of an interrupt from an application program

These functions can all generate local interrupts on the system where the RCIM card is installed. Multiple RedHawk Linux systems can be chained together, allowing up to eight of the local interrupts to be distributed to other RCIM-connected systems. This allows one timer or one external interrupt or one application program to interrupt multiple RedHawk Linux systems almost simultaneously to create synchronized actions. In addition, the RCIM contains a synchronized high-resolution clock so that multiple systems can share a common time base. See Chapter 6 of this guide and the *Real-Time Clock & Interrupt Module (RCIM) PCI Form Factor* manual for additional information.

Frequency-Based Scheduler

The Frequency-Based Scheduler (FBS) is a mechanism added to RedHawk Linux for scheduling applications that run according to a predetermined cyclic execution pattern. The FBS also provides a very fast mechanism for waking a process when it is time for that process to execute. In addition, the performance of cyclical applications can be tracked, with various options available to the programmer when deadlines are not being met. The FBS is the kernel mechanism that underlies the NightSim™ GUI for scheduling cyclical applications. See the *Frequency-Based Scheduler (FBS) User's Guide* and *NightSim User's Guide* for additional information.

/proc Modifications

Modifications have been made to the process address space support in **`/proc`** to allow a privileged process to read or write the values in another process' address space. This is for support of the NightProbe™ data monitoring tool and the NightView™ debugger.

Kernel Trace Facility

Support was added to RedHawk Linux to allow kernel activity to be traced. This includes mechanisms for inserting and enabling kernel trace points, reading trace memory buffers from the kernel, and managing trace buffers. The kernel trace facility is used by the NightTrace™ tool.

ptrace Extensions

The ptrace debugging interface in Linux has been extended to support the capabilities of the NightView debugger. Features added include:

- the capability for a debugger process to read and write memory in a process not currently in the stopped state
- the capability for a debugger to trace only a subset of the signals in a process being debugged
- the capability for a debugger to efficiently resume execution at a new address within a process being debugged
- the capability for a debugger process to automatically attach to all children of a process being debugged

Kernel Preemption

The ability for a high priority process to preempt a lower priority process currently executing inside the kernel is provided through an open source patch to RedHawk Linux. Under standard Linux the lower priority process would continue running until it exited from the kernel, creating longer worst case process dispatch latency. This patch leverages the data structure protection mechanisms built into the kernel to support symmetric multiprocessing.

Real-Time Scheduler

The O(1) scheduler patch created by Ingo Molnar was actually created for the 2.5 Linux development baseline. It provides fixed-length context switch times regardless of how many processes are active in the system. It also provides a true real-time scheduling class that operates on a symmetric multiprocessor.

Low Latency Patches

In order to protect shared data structures used by the kernel, the kernel protects code paths that access these shared data structures with spin locks and semaphores. The locking of a spin lock requires that preemption, and sometimes interrupts, be disabled while the spin lock is held. A study was made which identified the worst-case preemption off times. The low latency patches applied to RedHawk Linux modify the algorithms in the identified worst-case preemption off scenarios to provide better interrupt response times.

High Resolution Timing

In the standard Linux kernel, the system accounts for a process' CPU execution times using a very coarse-grained mechanism. This means that the amount of CPU time charged to a particular process can be very inaccurate. The high resolution timing provided by an open source patch to RedHawk Linux provides a mechanism for very accurate CPU execution time accounting, allowing better performance monitoring of applications. This facility is used by the Performance Monitor and can be utilized for system and user execution time accounting when the local timer interrupt is disabled on a CPU.

Capabilities Support

The Pluggable Authentication Modules (PAM) open source patch provides a mechanism to assign privileges to users and set authentication policy without having to recompile authentication programs. Under this scheme, a non-root user can be configured to run applications that require privileges only root would normally be allowed. For example, the ability to lock pages in memory is provided by one predefined privilege that can be assigned to individual users or groups.

Privileges are granted through roles defined in a configuration file. A role is a set of valid Linux capabilities. Defined roles can be used as a building block in subsequent roles, with the new role inheriting the capabilities of the previously defined role. Roles are assigned to users and groups, defining their capabilities on the system.

See Chapter 12 for information about the PAM functionality.

Kernel Debuggers

Two different kernel debuggers are supported through open source patches. Each provides a different set of features. The **kdb** kernel debugger is linked with the kernel and can be run as a native kernel debugger. The **kgdb** debugger allows the RedHawk Linux kernel to be debugged with **gdb** as if it were a user application. **gdb** runs on a separate system, communicating with the kernel being debugged through the console's serial port.

Kernel Core Dumps/Crash Analysis

This open source patch provides the support for dumping physical memory contents to a file as well as support for utilities that do a post mortem analysis of a kernel core dump. See Chapter 11 and the **lcrash(1)** man page for more information about crash dump analysis.

User-Level Spin Locks

RedHawk Linux busy-wait mutual exclusion tools include a low-overhead busy-wait mutual exclusion variable (a spin lock) and a corresponding set of macros that allow you to initialize, lock, unlock and query spin locks. To be effective, user-level spin locks must be used with user-level preemption control. Refer to Chapter 5 for details.

usermap and /proc mmap

The **usermap(3)** library routine, which resides in the l**ibccur_rt** library, provides applications with a way to efficiently monitor and modify locations in currently executing programs through the use of simple CPU reads and writes.

The **/proc** file system **mmap(2)** is the underlying kernel support for **usermap(3)**, which lets a process map portions of another process' address space into its own address space. Thus, monitoring and modifying other executing programs becomes simple CPU reads and writes within the application's own address space, without incurring the overhead of **/proc** file system **read(2)** and **write(2)** system service calls. Refer to Chapter 9 for more information.

Hyper-threading

Hyper-threading is a feature of the Intel Pentium Xeon processor that allows for a single physical processor to appear to the operating system as two logical processors. Two program counters run simultaneously within each CPU chip so that in effect, each chip is a dual-CPU SMP. With hyper-threading, physical CPUs can run multiple tasks "in parallel" by utilizing fast hardware-based context-switching between the two register sets upon things like cache-misses or special instructions. RedHawk Linux includes support for hyper-threading and supplies it as the default mode of operation. Refer to Chapter 2 for more information on how to effectively use this feature in a real-time environment.

XFS Journaling File System

The XFS journaling file system from SGI is implemented in RedHawk Linux. Journaling file systems use a journal (log) to record transactions. In the event of a system crash, the background process is run on reboot and finishes copying updates from the journal to the file system. This drastically cuts the complexity of a file system check, reducing recovery time. The SGI implementation is a multithreaded, 64-bit file system capable of large files and file systems, extended attributes, variable block sizes, is extent based and makes extensive use of Btrees to aid both performance and scalability. Refer to Chapter 8 for more information.

POSIX Real-Time Extensions

RedHawk Linux supports most of the interfaces defined by the POSIX real-time extensions as set forth in ISO/IEC 9945-1. The following functional areas are supported:

- user priority scheduling
- process memory locking
- memory mapped files
- shared memory
- message queues
- counting semaphores (note that named semaphores are not supported [sem_open, sem_close, sem_unlink])
- real-time signal behavior
- asynchronous I/O
- synchronized I/O
- timers (high resolution version is supported)

User Priority Scheduling

RedHawk Linux accommodates user priority scheduling—that is, processes scheduled under the fixed-priority POSIX scheduling policies do not have their priorities changed by the operating system in response to their run-time behavior. The resulting benefits are reduced kernel overhead and increased user control. Process scheduling facilities are fully described in Chapter 4.

Memory Resident Processes

Paging and swapping often add an unpredictable amount of system overhead time to application programs. To eliminate performance losses due to paging and swapping, RedHawk Linux allows you to make certain portions of a process' virtual address space resident. The **`mlockall(2)`**, **`munlockall(2)`**, **`mlock(2)`**, and **`munlock(2)`** POSIX system calls allow locking all or a portion of a process' virtual address space in physical memory.

Memory Mapping and Data Sharing

RedHawk Linux supports shared memory and memory-mapping facilities based on IEEE Standard 1003.1b-1993, as well as System V IPC mechanisms. The POSIX facilities allow processes to share data through the use of m*emory objects,* named regions of storage that can be mapped to the address space of one or more processes to allow them to share the associated memory. The term *memory object* includes POSIX shared memory objects, regular files, and some devices, but not all file system objects (terminals and network devices, for example). Processes can access the data in a memory object directly by mapping portions of their address spaces onto the objects. This is generally more efficient than using the **`read(2)`** and **`write(2)`** system calls because it eliminates copying the data between the kernel and the application.

Process Synchronization

RedHawk Linux provides a variety of tools that cooperating processes can use to synchronize access to shared resources.

Counting semaphores based on IEEE Standard 1003.1b-1993 allow multiple processes to synchronize their access to the same set of resources. A counting semaphore has a value associated with it that determines when resources are available for use and allocated. System V IPC semaphore sets are also available under RedHawk Linux.

In addition to semaphores, a set of real-time process synchronization tools developed by Concurrent provides the ability to control a process' vulnerability to rescheduling, serialize processes' access to critical sections with busy-wait mutual exclusion mechanisms, and coordinate client–server interaction among processes. With these tools, a mechanism for providing sleepy-wait mutual exclusion with bounded priority inversion can be constructed.

Descriptions of the synchronization tools and procedures for using them are provided in Chapter 5.

Asynchronous Input/Output

Being able to perform I/O operations asynchronously means that you can set up for an I/O operation and return without blocking on I/O completion. RedHawk Linux accommodates asynchronous I/O with a group of library routines based on IEEE Standard 1003.1b-1993. These interfaces allow a process to perform asynchronous read and write operations, initiate multiple asynchronous I/O operations with a single call, wait for completion of an asynchronous I/O operation, cancel a pending asynchronous I/O operation, and perform asynchronous file synchronization.

Synchronized Input/Output

RedHawk Linux also supports the synchronized I/O facilities based on IEEE Standard 1003.1b-1993. POSIX synchronized I/O provides the means for ensuring the integrity of an application's data and files. A synchronized output operation ensures the recording of data written to an output device. A synchronized input operation ensures that the data read from a device mirrors the data currently residing on disk.

Real-Time Signal Behavior

Real-time signal behavior specified by IEEE Standard 1003.1b-1993 includes specification of a range of real-time signal numbers, support for queuing of multiple occurrences of a particular signal, and support for specification of an application-defined value when a signal is generated to differentiate among multiple occurrences of signals of the same type. The POSIX signal-management facilities include the **sigtimedwait(2)**, **sigwaitinfo(2)**, and **sigqueue(2)** system calls, which allow a process to wait for receipt of a signal and queue a signal and an application-defined value to a process.

Clocks and Timers

Support for high-resolution POSIX clocks and timers is included in RedHawk Linux. There are four system-wide POSIX clocks that can be used for such purposes as time stamping or measuring the length of code segments. POSIX timers allow applications to use relative or absolute time based on a high resolution clock and to schedule events on a one-shot or periodic basis. Applications can create multiple timers for each process. In addition, high-resolution sleep mechanisms are provided which can be used to put a process to sleep for a very short time quantum and specify which clock should be used for measuring the duration of the sleep. See Chapter 6 for additional information.

Message Queues

POSIX message passing facilities based on IEEE Standard 1003.1b-1993 are included in RedHawk Linux, implemented as a file system. POSIX message queue library routines allow a process to create, open, query and destroy a message queue, send and receive messages from a message queue, associate a priority with a message to be sent, and request asynchronous notification when a message arrives. POSIX message queues operate independently of System V IPC messaging, which is also available under RedHawk Linux. See Chapter 3 for details.

2
Real-Time Performance

2
Real-Time Performance

This chapter discusses some of the issues involved with achieving real-time performance under RedHawk Linux. The primary focus of the chapter is on the *Shielded CPU Model*, which is a model for assigning processes and interrupts to a subset of CPUs in the system to attain the best real-time performance.

Key areas of real-time performance are discussed: interrupt response, process dispatch latency and deterministic program execution. The impact of various system activities on these metrics is discussed and techniques are given for optimum real-time performance.

Overview of the Shielded CPU Model

The shielded CPU model is an approach for obtaining the best real-time performance in a symmetric multiprocessor system. The shielded CPU model allows for both deterministic execution of a real-time application as well as deterministic response to interrupts.

A task has deterministic execution when the amount of time it takes to execute a code segment within that task is predictable and constant. Likewise the response to an interrupt is deterministic when the amount of time it takes to respond to an interrupt is predictable and constant. When the worst-case time measured for either executing a code segment or responding to an interrupt is significantly different than the typical case, the application's performance is said to be experiencing *jitter*. Because of computer architecture features like memory caches and contention for shared resources, there will always be some amount of jitter in measurements of execution times. Each real-time application must define the amount of jitter that is acceptable to that application.

In the shielded CPU model, tasks and interrupts are assigned to CPUs in a way that guarantees a high grade of service to certain important real-time functions. In particular, a high-priority task is bound to one or more shielded CPUs, while most interrupts and low priority tasks are bound to *other* CPUs. The CPUs responsible for running the high-priority tasks are shielded from the unpredictable processing associated with interrupts and the other activity of lower priority processes that enter the kernel via system calls, thus these CPUs are called *shielded CPUs*.

Some examples of the types of tasks that should be run on shielded CPUs are:

- tasks that require guaranteed interrupt response time
- tasks that require the fastest interrupt response time
- tasks that must be run at very high frequencies
- tasks that require deterministic execution in order to meet their deadlines
- tasks that have no tolerance for being interrupted by the operating system

There are several levels of CPU shielding that provide different degrees of determinism for the tasks that must respond to high-priority interrupts or that require deterministic execution. Before discussing the levels of shielding that can be enabled on a shielded CPU, it is necessary to understand how the system responds to external events and how some of the normal operations of a computer system impact system response time and determinism.

Overview of Determinism

Determinism refers to a computer system's ability to execute a particular code path (a set of instructions executed in sequence) in a fixed amount of time. The extent to which the execution time for the code path varies from one instance to another indicates the degree of determinism in the system.

Determinism applies not only to the amount of time required to execute a time-critical portion of a user's application but also to the amount of time required to execute system code in the kernel. The determinism of the process dispatch latency, for example, depends upon the code path that must be executed to handle an interrupt, wake the target process, perform a context switch, and allow the target process to exit from the kernel. (The section "Process Dispatch Latency" defines the term *process dispatch latency* and presents a model for obtaining the best process dispatch latency possible on a particular CPU in a multiprocessor system.)

The largest impact on the determinism of a program's execution is the receipt of interrupts. This is because interrupts are always the highest priority activity in the system and the receipt of an interrupt is unpredictable – it can happen at any point in time while a program is executing. Shielding from non-critical interrupts will have the largest impact on creating better determinism during the execution of high priority tasks.

Other techniques for improving the determinism of a program's execution are discussed in the section called "Procedures for Increasing Determinism."

Process Dispatch Latency

Real-time applications must be able to respond to a real-world event and complete the processing required to handle that real-world event within a given deadline. Computations required to respond to the real-world event must be complete before the deadline or the results are considered incorrect. A single instance of having an unusually long response to an interrupt can cause a deadline to be missed.

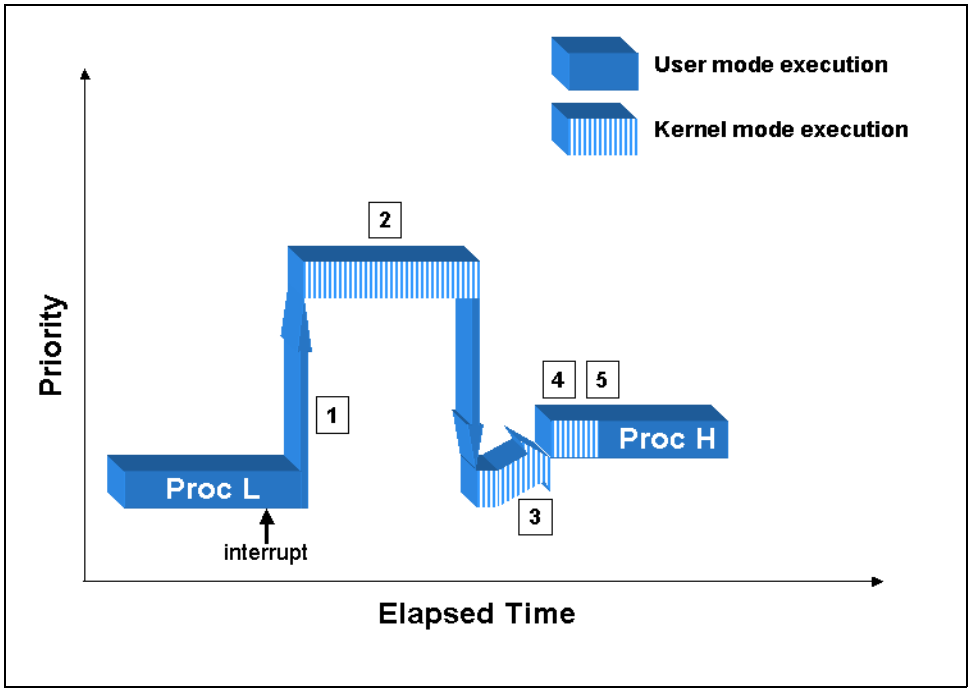
The term *process dispatch latency* denotes the time that elapses from the occurrence of an external event, which is signified by an interrupt, until the process waiting for that external event executes its first instruction in user mode. For real-time applications, the worst-case process dispatch latency is a key metric, since it is the worst-case response time that will determine the ability of the real-time application to guarantee that it can meet its deadlines.

Process dispatch latency comprises the time that it takes for the following sequence of events to occur:

1. The interrupt controller notices the interrupt and generates the interrupt exception to the CPU.
2. The interrupt routine is executed, and the process waiting for the interrupt (target process) is awakened.
3. The currently executing process is suspended, and a context switch is performed so that the target process can run.
4. The target process must exit from the kernel, where it was blocked waiting for the interrupt.
5. The target process runs in user mode.

This sequence of events represents the ideal case for process dispatch latency; it is illustrated by Figure 2-1. Note that events 1-5 described above, are marked in Figure 2-1.

Figure 2-1. Normal Process Dispatch Latency



The process dispatch latency is a very important metric for event–driven real–time applications because it represents the speed with which the application can respond to an external event. Most developers of real–time applications are interested in the worst-case process dispatch latency because their applications must meet certain timing constraints.

Process dispatch latency is affected by some of the normal operations of the operating system, device drivers and computer hardware. The following sections examine some of the causes of jitter in process dispatch latency.

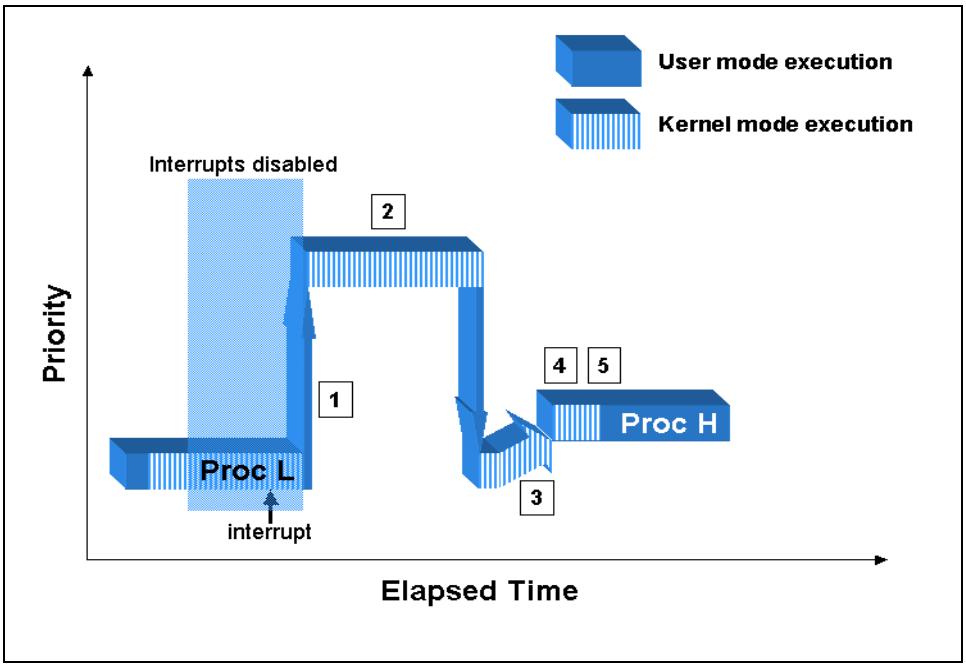
Effect of Disabling Interrupts

An operating system must protect access to shared data structures in order to prevent those data structures from being corrupted. When a data structure can be accessed at interrupt level, it is necessary to disable interrupts whenever that data structure is accessed. This prevents interrupt code from corrupting a shared data structure should it interrupt program level code in the midst of an update to the same shared data structure. This is the primary reason that the kernel will disable interrupts for short periods of time.

When interrupts are disabled, process dispatch latency is affected because the interrupt that we are trying to respond to cannot become active until interrupts are again enabled. In this case, the process dispatch latency for the task awaiting the interrupt is extended by the amount of time that interrupts remain disabled. This is illustrated in Figure 2-2. In this diagram, the low priority process has made a system call which has disabled interrupts. When the high priority interrupt occurs it cannot be acted on because interrupts are currently disabled. When the low priority process has completed its critical section, it enables interrupts, the interrupt becomes active and the interrupt service routine is called. The normal steps of interrupt response then complete in the usual fashion. Note that the numbers 1-5 marked in Figure 2-2 represent the steps of normal process dispatch latency as described earlier on page 2-3.

Obviously, critical sections in the operating system where interrupts are disabled must be minimized to attain good worst-case process dispatch latency.

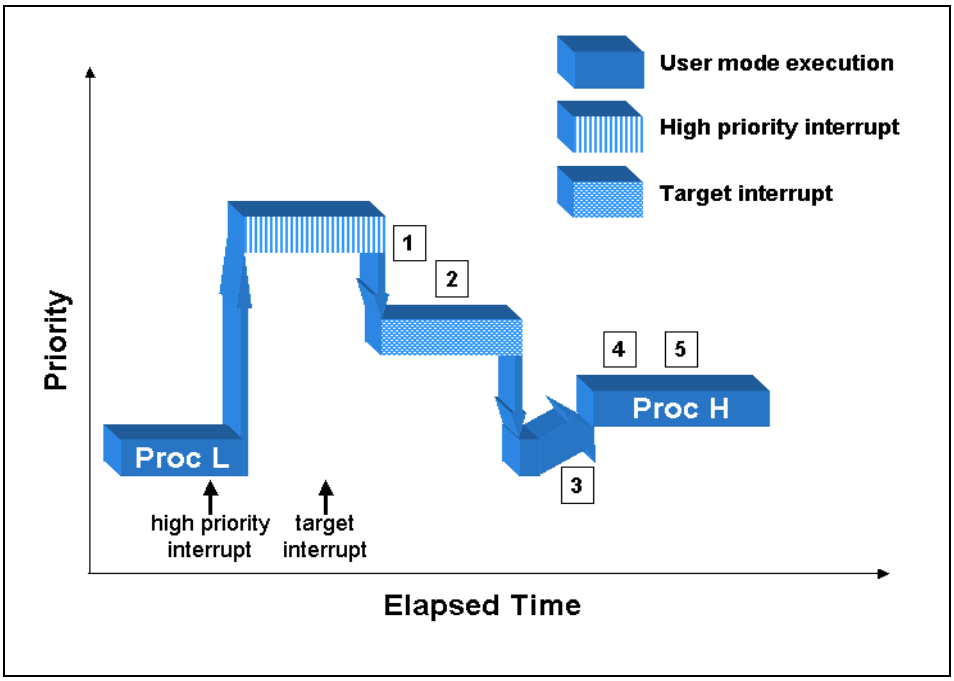
Figure 2-2. Effect of Disabling Interrupts on Process Dispatch Latency



Effect of Interrupts

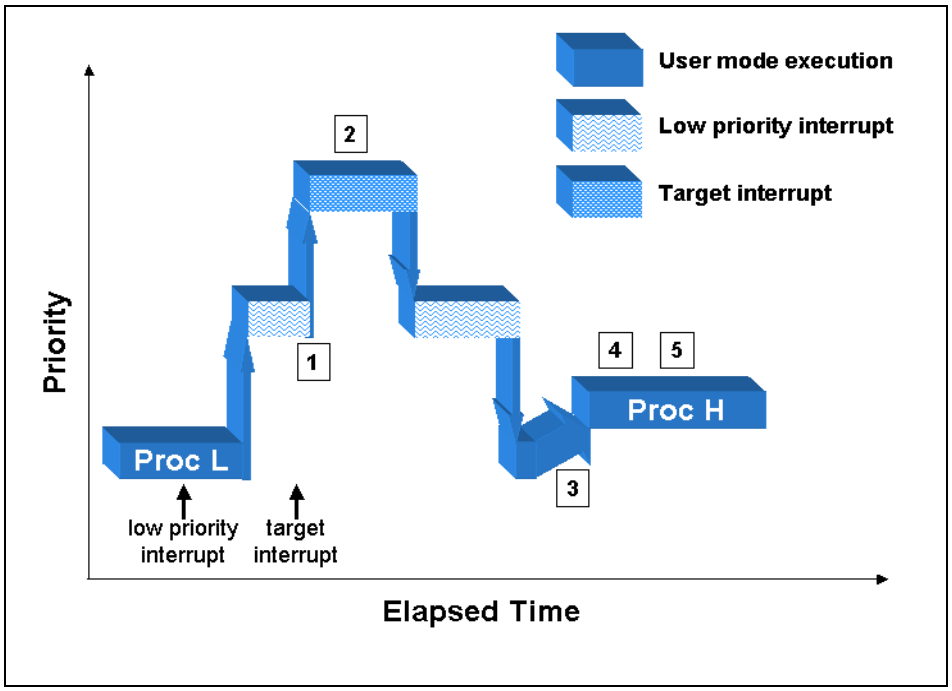
The receipt of an interrupt affects process dispatch latency in much the same way that disabling interrupts does. When a hardware interrupt is received, the system will block interrupts of the same or lesser priority than the current interrupt. The simple case is illustrated in Figure 2-3, where a higher priority interrupt occurs before the target interrupt, causing the target interrupt to be held off until the higher priority interrupt occurs. Note that the numbers 1-5 marked in Figure 2-3 represent the steps of normal process dispatch latency as described earlier on page 2-3.

Figure 2-3. Effect of High Priority Interrupt on Process Dispatch Latency



The relative priority of an interrupt does not affect process dispatch latency. Even when a low priority interrupt becomes active, the impact of that interrupt on the process dispatch latency for a high-priority interrupt is the same. This is because interrupts always run at a higher priority than user-level code. Therefore, even though we might service the interrupt routine for a high-priority interrupt, that interrupt routine cannot get the user-level context running until all interrupts have completed their execution. This impact of a low priority interrupt on process dispatch latency is illustrated in Figure 2-4. Note that the ordering of how things are handled is different than the case of the high-priority interrupt in Figure 2-3, but the impact on process dispatch latency is the same. Note that the numbers 1-5 marked in Figure 2-4 represent the steps of normal process dispatch latency as described earlier on page 2-3.

Figure 2-4. Effect of Low Priority Interrupt on Process Dispatch Latency

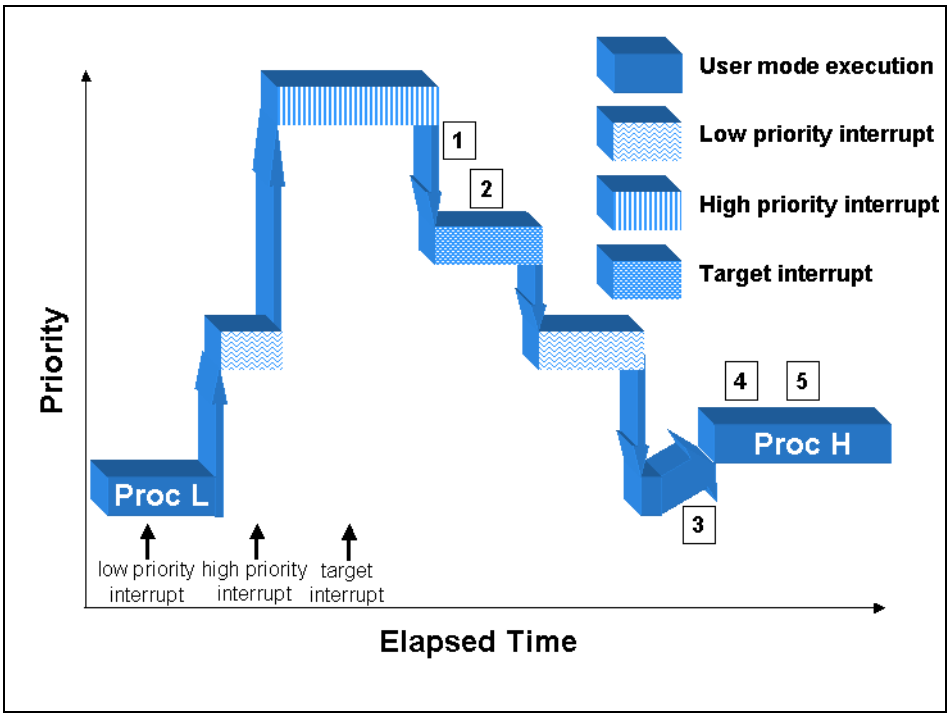


One of the biggest differences between the effect of disabling interrupts and receipt of an interrupt in terms of the impact on process dispatch latency is the fact that interrupts occur asynchronously to the execution of an application and at unpredictable times. This is important to understanding the various levels of shielding that are available.

When multiple interrupts can be received on a given CPU, the impact on worst-case process dispatch latency can be severe. This is because interrupts can stack up, such that more than one interrupt service routine must be processed before the process dispatch latency for a high priority interrupt can be completed. Figure 2-5 shows a case of two interrupts becoming active while trying to respond to a high priority interrupt. Note that the numbers 1-5 marked in Figure 2-5 represent the steps of normal process dispatch latency as described earlier on page 2-3. When a CPU receives an interrupt, that CPU will disable interrupts of lower priority from being able to interrupt the CPU. If a second interrupt of lower-priority becomes active during this time, it is blocked as long as the original interrupt is active. When servicing of the first interrupt is complete, the second interrupt becomes active and is serviced. If the second interrupt is of higher priority than the initial interrupt, it will immediately become active. When the second interrupt completes its processing, the first interrupt will again become active. In both cases, user processes are prevented from running until all of the pending interrupts have been serviced.

Conceivably, it would be possible for a pathological case where interrupts continued to become active, never allowing the system to respond to the high-priority interrupt. When multiple interrupts are assigned to a particular CPU, process dispatch latency is less predictable on that CPU because of the way in which the interrupts can be stacked.

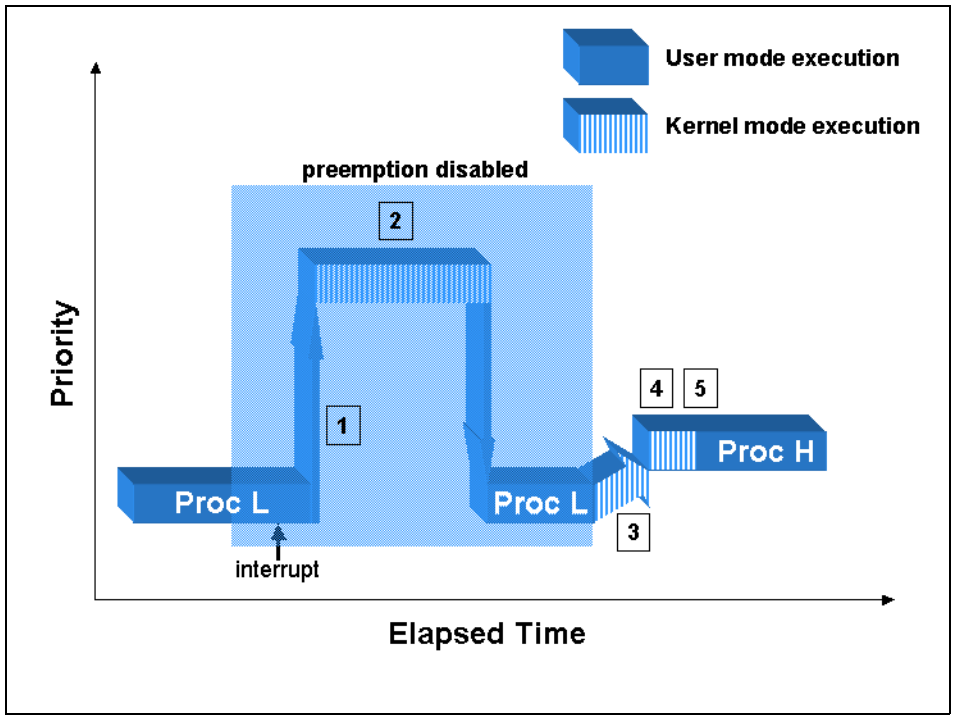
Figure 2-5. Effect of Multiple Interrupts on Process Dispatch Latency



Effect of Disabling Preemption

There are critical sections in RedHawk Linux that protect a shared resource that is never locked at interrupt level. In this case, there is no reason to block interrupts while in this critical section. However, a preemption that occurs during this critical section could cause corruption to the shared resource if the new process were to enter the same critical section. Therefore, preemption is disabled while a process executes in this type of critical section. Blocking preemption will not delay the receipt of an interrupt. However, if that interrupt wakes a high priority process, it will not be possible to switch to that process until preemption has again been enabled. Assuming the same CPU is required, the actual effect on worst-case process dispatch latency is the same as if interrupts had been disabled. The effect of disabling preemption on process dispatch latency is illustrated in Figure 2-6. Note that the numbers 1-5 marked in Figure 2-6 represent the steps of normal process dispatch latency as described earlier on page 2-3.

Figure 2-6. Effect of Disabling Preemption on Process Dispatch Latency



Effect of Open Source Device Drivers

Device drivers are a part of the Linux kernel, because they run in supervisor mode. This means that device drivers are free to call Linux functions that disable interrupts or disable preemption. Device drivers also handle interrupts, therefore they control the amount of time that might be spent at interrupt level. As shown in previous sections of this chapter, these actions have the potential to impact worst-case interrupt response and process dispatch latency.

Device drivers enabled in RedHawk Linux have been tested to be sure they do not adversely impact real-time performance. While open source device driver writers are encouraged to minimize the time spent at interrupt level and the time interrupts are disabled, in reality open source device drivers are written with very varied levels of care. If additional open source device drivers are enabled they may have a negative impact upon the guaranteed worst-case process dispatch latency that RedHawk Linux provides.

Refer to the “Device Drivers and Real Time” chapter for more information about real-time issues with device drivers.

How Shielding Improves Real-Time Performance

This section will examine how the different attributes of CPU shielding improve both the ability for a user process to respond to an interrupt (process dispatch latency) and determinism in execution of a user process.

When enabling shielding, all shielding attributes are enabled by default. This provides the most deterministic execution environment on a shielded CPU. Each of these shielding attributes is described in more detail below. The user should fully understand the impact of each of the possible shielding attributes, as some of these attributes do have side effects to normal system functions. There are three categories of shielding attributes currently supported:

- shielding from background processes
- shielding from interrupts
- shielding from the local interrupt

Each of these attributes is individually selectable on a per-CPU basis. Each of the shielding attributes is described below.

Shielding From Background Processes

This shielding attribute allows a CPU to be reserved for a subset of processes in the system. This shielding attribute should be enabled on a CPU when you want that CPU to have the fastest, most predictable response to an interrupt. The best guarantee on process dispatch latency is achieved when only the task that responds to an interrupt is allowed to execute on the CPU where that interrupt is directed.

When a CPU is allowed to run background processes, it can affect the process dispatch latency of a high-priority task that desires very deterministic response to an interrupt directed to that CPU. This is because background processes will potentially make system calls that can disable interrupts or preemption. These operations will impact process dispatch latency as explained in the sections "Effect of Disabling Interrupts" and "Effect of Disabling Preemption."

When a CPU is allowed to run background processes, there is no impact on the determinism in the execution of high priority processes. This assumes the background processes have lower priority than the high-priority processes. Note that background processes could affect the time it takes to wake a process via other kernel mechanisms such as signals or the **server_wake1(3)** interface.

Each process in the system has a CPU affinity mask. The CPU affinity mask determines on which CPUs the process is allowed to execute. The CPU affinity mask is inherited from the process' parent process and can be set via the **mpadvise(3)** library routine or the **sched_setaffinity(2)** system call. When a CPU is shielded from processes, that CPU will only run processes that have explicitly set their CPU affinity to a set of CPUs that only includes shielded CPUs. In other words, if a process has a non-shielded CPU in its CPU affinity mask, then the process will only run on those CPUs that are not shielded. To run a process on a CPU shielded from background processes, the process must have a CPU affinity mask that specifies ONLY shielded CPUs.

Shielding From Interrupts

This shielding attribute allows a CPU to be reserved for processing only a subset of interrupts received by the system. This shielding attribute should be enabled when it is desirable to have the fastest, most predictable process dispatch latency or when it is desirable to have determinism in the execution time of an application.

Because interrupts are always the highest priority activity on a CPU, the handling of an interrupt can affect both process dispatch latency and the time it takes to execute a normal code path in a high priority task. This is described in the section, "Effect of Interrupts".

Each device interrupt is associated with an IRQ. These IRQs have an associated CPU affinity that determines which CPUs are allowed to receive the interrupt. When interrupts are not routed to a specific CPU, the interrupt controller will select a CPU for handling an interrupt at the time the interrupt is generated from the set of CPUs in the IRQ affinity mask. IRQ affinities are modified by the **shield(1)** command or through **/proc/irq/***N***/smp_affinity.**

Shielding From Local Interrupt

The local interrupt is a special interrupt for a private timer associated with each CPU. Under RedHawk Linux, this timer is used for various timeout mechanisms in the kernel and at user level. This functionality is described in Chapter 7. By default, this interrupt is enabled on all CPUs in the system.

By default, this interrupt fires every ten milliseconds, making the local interrupt one of the most frequently executed interrupt routines in the system. Therefore, the local interrupt is a large source of jitter to real-time applications.

When a CPU is shielded from the local interrupt, the functions provided by the local timer associated with that CPU are no longer performed; however, they continue to run on other CPUs where the local interrupt has not been shielded. Some of these functions will be lost, while others can be provided via other means.

One of the functions that is disabled when the local timer is disabled on a particular CPU is the CPU execution time accounting. This is the mechanism that measures how much CPU time is used by each process that executes on this CPU. It is possible to continue to measure the amount of time used by processes at user and system levels by enabling the High Resolution Process Timing Facility.

When a CPU is shielded from the local interrupt, the local interrupt will continue to be used for POSIX timers and nanosleep functionality by processes biased to the shielded CPU. For this reason, if it is critical to totally eliminate local timer interrupts for optimum performance on a specific shielded CPU, applications utilizing POSIX timers or nanosleep functionality should not be biased to that CPU. If a process is not allowed to run on the shielded CPU, its timers will be migrated to a CPU where the process is allowed to run.

Refer to Chapter 7, “System Clocks and Timers” for a complete discussion on the effects of disabling the local timer.

Interfaces to CPU Shielding

This section describes both the command level and programming interfaces that can be used for setting up a shielded CPU. There is also an example that describes the common case for setting up a shielded CPU.

Shield Command

The **`shield(1)`** command sets specified shielding attributes for selected CPUs. The shield command can be used to mark CPUs as shielded CPUs. A shielded CPU is protected from some set of system activity in order to provide better determinism in the time it takes to execute application code.

The list of logical CPUs affected by an invocation of the **`shield`** command is given as a comma-separated list of CPU numbers or ranges.

The format for executing the **`shield`** command is:

`shield` [*OPTIONS*]

Options are described in Table 2-1.

In the options listed below, *CPULIST* is a list of comma separated values or a range of values representing logical CPUs. For example, the list of CPUs "**0-4,7**" specifies the following logical CPU numbers: **0,1,2,3,4,7**.

Table 2-1. Options to the shield(1) Command

Option	Description
--irq=*CPULIST*, **-i** *CPULIST*	Shields all CPUs in *CPULIST* from interrupts. The only interrupts that will execute on the specified CPUs are those that have been assigned a CPU affinity that would prevent them from executing on any other CPU.
--loc=*CPULIST*, **-l** *CPULIST*	The specified list of CPUs is shielded from the local timer. The local timer provides time-based services for a CPU. Disabling the local timer may cause some system functionality such as user/system time accounting and round-robin quantum expiration to be disabled. Refer to Chapter 7 for more a complete discussion.
--proc=*CPULIST*, **-p** *CPULIST*	The specified list of CPUs is shielded from extraneous processes. Processes that have an affinity mask that allows them to run on a non-shielded CPU only run on non-shielded CPUs. Processes that would be precluded from executing on any CPU other than a shielded CPU are allowed to execute on that shielded CPU.
--all=*CPULIST*, **-a** *CPULIST*	The specified list of CPUs will have all available shielding attributes set. See the descriptions of the individual shielding options above to understand the implications of each shielding attribute.
--help, -h	Describes available options and usage.
--version, -V	Prints out current version of the command.
--reset, -r	Resets shielding attributes for all CPUs. No CPUs are shielded.
--current, -c	Displays current settings for all active CPUs.

Shield Command Examples

The following command first resets all shielding attributes, then shields CPUs 0,1 and 2 from interrupts, then shields CPU 1 from local timer, shields CPU 2 from extraneous processes, and finally, displays all new settings after the changes:

```
shield -r -i 0-2 -l 1 -p 2 -c
```

The following command shields CPUs 1,2 and 3 from interrupts, local timer, and extraneous processes. CPU 0 is left as a “general purpose” CPU that will service all interrupts and processes not targeted to a shielded CPU. All shielding attributes are set for the list of CPUs.

```
shield --all=1-3
```

Exit Status

Normally, the exit status is zero. However, if an error occurred while trying to modify shielded CPU attributes, a diagnostic message is issued and an exit status of 1 is returned.

Shield Command Advanced Features

It is recommended that the advanced features described below should only be used by experienced users.

CPUs specified in the *CPULIST* can be preceded by a '+' or a '-' sign in which case the CPUs in the list are added to ('+') or taken out of ('-') the list of already shielded CPUs.

Options can be used multiple times. For example, “shield -i 0 -c -i +1 -c” shows current settings after CPU 0 has been shielded from interrupts and then displays current settings again after CPU 1 has been added to the list of CPUs shielded from interrupts.

/proc Interface to CPU Shielding

The kernel interface to CPU shielding is through the **/proc** file system using the following files:

/proc/shield/procs	process shielding
/proc/shield/irqs	irq shielding
/proc/shield/ltmrs	local timer shielding
/proc/shield/all	all of the above

All users can read these files, but only root or users with the **CAP_SYS_NICE** capability may write to them.

When read, an 8 digit ASCII hexadecimal value is returned. This value is a bitmask of shielded CPUs. Set bits identify the set of shielded CPUs. The radix position of each set bit is the number of the logical CPU being shielded by that bit.

For example:

`00000001` - bit 0 is set so CPU #0 is shielded

`00000002` - bit 1 is set so CPU #1 is shielded

`00000004` - bit 2 is set so CPU #2 is shielded

`00000006` - bits 1 and 2 are set so CPUs #1 and #2 are shielded

When written to, an 8 digit ASCII hexadecimal value is expected. This value is a bitmask of shielded CPUs in a form identical to that listed above. The value then becomes the new set of shielded CPUs.

See the **shield(5)** man page for additional information.

Assigning Processes to CPUs

This section describes the methods available for assigning a process to a set of CPUs. The set of CPUs where a process is allowed to run is known as a process' CPU affinity.

By default, a process can execute on any CPU in the system. Every process has a bit mask, or CPU affinity, that determines the CPU or CPUs on which it can be scheduled. A process inherits its CPU affinity from its creator during a **fork(2)** or a **clone(2)** but may change it thereafter.

You can set the CPU affinity for one or more processes by specifying the **MPA_PRC_SETBIAS** command on a call to **mpadvise(3)**, or the **-b** *bias* option to the **run(1)** command. Another set of interfaces for setting CPU affinity are **sched_setaffinity(2)** and **sched_getaffinity(2)**. The **sched_setaffinity(2)** function restricts the execution of some process to a subset of the available CPUs. **sched_getaffinity(2)** returns the set of CPUs that a process is restricted to.

To set the CPU affinity, the following conditions must be met:

- The real or effective user ID of the calling process must match the real or saved user ID of the process for which the CPU affinity is being set, or
- the calling process must have the **CAP_SYS_NICE** capability or be root.

To add a CPU to a process' CPU affinity, the calling process must have the **CAP_SYS_NICE** capability or be root.

A CPU affinity can be assigned to the **init(8)** process. All general processes are a descendant from **init**. As a result, most general processes would have the same CPU affinity as **init** or a subset of the CPUs in the **init** CPU affinity. Only privileged processes (as described above) are able to add a CPU to their CPU affinity. Assigning a restricted CPU affinity to **init** restricts all general processes to the same subset of CPUs as **init**. The exception is selected processes that have the appropriate capability who explicitly modify their CPU affinity. If you wish to change the CPU affinity of **init**, see the section "Assigning CPU Affinity to init" below for instructions.

The **mpadvise** library routine is documented in the section "Multiprocessor Control Using mpadvise" below and the **mpadvise(3)** man page. The **run** command is documented in the section "The run Command" in Chapter 4 and the **run(1)** man page. For information on **sched_setaffinity(2)** and **sched_getaffinity(2)**, see the **sched_affinity(2)** man page.

Multiprocessor Control Using mpadvise

mpadvise(3) performs a variety of multiprocessor functions. CPUs are identified by specifying a pointer to a `cpuset_t` object, which specifies a set of one or more CPUs. For more information on CPU sets, see the **cpuset(3)** man page.

Synopsis

```
#include <mpadvise.h>

int mpadvise (int cmd, int which, int who, cpuset_t *setp)

gcc [options] file -lccur_rt ...
```

Informational Commands

The following commands get or set information about the CPUs in the system. The *which* and *who* parameters are ignored.

MPA_CPU_PRESENT Returns a mask indicating which CPUs are physically present in the system. CPUs brought down with the **cpu(1)** command are still included.

MPA_CPU_ACTIVE Returns a mask indicating which CPUs are active, that is, initialized and accepting work, regardless of how many exist in the backplane. If a CPU has been brought down using the **cpu(1)** command, it is not included.

MPA_CPU_BOOT Returns a mask indicating the CPU that booted the system. The boot CPU has some responsibilities not shared with the other CPUs.

Control Commands

The following commands provide control over the use of CPUs by a process, a process group, or a user.

MPA_PRC_GETBIAS Return the CPU set for the CPU affinity of the specified process. Only **MPA_PID** is supported at this time.

MPA_PRC_SETBIAS Sets the CPU affinity of all the specified processes to the specified cpuset. To change the CPU affinity of a process, the real or effective user ID must match the real or the saved (from **exec(2)**) user ID of the process, unless the current user has the **CAP_SYS_NICE** capability.

MPA_PRC_GETRUN Return a CPU set with exactly one CPU in it that corresponds to the CPU where the specified process is currently running (or waiting to run). Only **MPA_PID** is

supported at this time. Note that it is possible that the CPU assignment may have already changed by the time the value is returned (if more than one CPU is specified in the process' affinity).

Using *which* and *who*

which — Used to specify the selection criteria. Can be one of the following:

MPA_PID (a specific process)
MPA_PGID (a process group)
MPA_UID (a user)

who — Interpreted relative to *which*:

a process identifier
a process group identifier
user identifier

A *who* value of 0 causes the process identifier, process group identifier, or user identifier of the caller to be used.

Assigning CPU Affinity to init

All general processes are a descendent from **init(8)**. By default, **init** has a mask that includes all CPUs in the system and only selected processes with appropriate capabilities can modify their CPU affinity. If it is desired that by default all processes are restricted to a subset of CPUs, a CPU affinity can be assigned by a privileged user to the **init** process. To achieve this goal, the **run(1)** command can be invoked early during the system initialization process.

For example, to bias **init** and all its descendants to CPUs 1, 2 and 3, the following command may be added at the end of the system's **/etc/rc.sysinit** script, which is called early during system initialization (see **inittab(5)**). The **init** process is specified in this command by its process ID which is always 1.

```
/usr/bin/run -b 1-3 -p 1
```

The same effect can be achieved by using the **shield(1)** command. The advantage of using this command is that it can be done from the command line at any run level. The **shield** command will take care of migrating processes already running in the CPU to be shielded. In addition, with the **shield** command you can also specify different levels of shielding. See the section "Shield Command" or the **shield(1)** man page for more information on this command.

For example, to shield CPU 0 from running processes, you would issue the following command.

```
$ shield -p 0
```

After shielding a CPU, you can always specify selected processes to run in the shielded CPU using the **run** command.

For example, to run **mycommand** on CPU 0 which was previously shielded from processes, you would issue the following command:

```
$ run -b 0 ./mycommand
```

Example of Setting Up a Shielded CPU

The following example shows how to use a shielded CPU to guarantee the best possible interrupt response to an edge-triggered interrupt from the RCIM. In other words, the intent is to optimize the time it takes to wake up a user-level process when the edge-triggered interrupt on an RCIM occurs and to provide a deterministic execution environment for that process when it is awakened. In this case the shielded CPU should be set up to handle just the RCIM interrupt and the program responding to that interrupt.

The first step is to direct interrupts away from the shielded processor through the **shield(1)** command. The local timer interrupt will also be disabled and background processes will be precluded to achieve the best possible interrupt response. The shield command that would accomplish these results for CPU 1 is:

```
$ shield -a 1
```

At this point, there are no interrupts and no processes that are allowed to execute on shielded CPU 1. The shielding status of the CPUs can be checked using any of the following methods:

via the **shield(1)** command:

```
$ shield -c
     CPUID        all      irqs     ltmrs     procs
-----------------------------------------------------
         0         no        no        no        no
         1         no        no        no        no
         2         no        no        no        no
         3         no        no        no        no
```

via the **cpu(1)** command:

```
$ cpu

log id
(phys id)  state  shielding
---------  -----  -------------
0 (0)       up     none
1 (0)       up     none
2 (1)       up     none
3 (1)       up     none
```

or via the **/proc** file system:

```
$ cat /proc/shield/all
00000002
```

To check only the status of interrupt shielding on CPU 1:

```
$ cat /proc/shield/irqs
00000002
```

This indicates that all interrupts are precluded from executing on CPU 1. In this example, the goal is to respond to a particular interrupt on the shielded CPU, so it is necessary to direct the RCIM interrupt to CPU 1 and to allow the program that will be responding to this interrupt to run on CPU 1.

The first step is to determine the IRQ to which the RCIM interrupt has been assigned. The assignment between interrupt and IRQ will be constant for devices on the motherboard and for a PCI device in a particular PCI slot. If a PCI board is moved to a new slot, its IRQ assignment may change. To find the IRQ for your device, perform the following command:

```
$ cat /proc/interrupts
   IRQs     CPU0      CPU1      CPU2      CPU3
     0:    41067         0         0         0    IO-APIC-edge       timer
     1:        6         0         0         0    IO-APIC-edge       keyboard
     2:        0         0         0         0         XT-PIC        cascade
     4:       11         0         0         0    IO-APIC-edge       serial
     7:        0         0         0         0    IO-APIC-level      usb-ohci
     8:        1         0         0         0    IO-APIC-edge       rtc
    12:        5         0         0         0    IO-APIC-edge       PS/2 Mouse
    14:        4         0         0         0    IO-APIC-edge       ide0
    16:     6831         0         0         0    IO-APIC-level      aic7xxx
    17:    13742         0         0         0    IO-APIC-level      eth0
    20:        0         0         0         0    IO-APIC-level      rcim
    21:       71         0         0         0    IO-APIC-level      megaraid
   NMI:        0         0         0         0
   LOC:    38462     38482     38484     38466
   ERR:        0
   MIS:        0
SOFTIRQs     CPU0      CPU1      CPU2      CPU3
     0:    39681         0         0         0    timer tasklets
     1:     9414         0         0         0    hi tasklets
     2:     3926         0         0         0    net xmit
     3:     7267       312         0       107    net recv
     4:   168382         0         0         2    lo tasklets
          240434       312         0       109    totals
    BHs      CPU0      CPU1      CPU2      CPU3
     0:        0         0         0         0    TIMER_BH
     1:     2716         0         0         0    TQUEUE_BH
     3:        9         0         0         0    SERIAL_BH
     8:     6948         0         0         0    SCSI_BH
     9:        6         0         0         0    IMMEDIATE_BH
          9679         0         0         0    totals
```

The first section of the **/proc/interrupts** file contains the IRQ to device assignments. The RCIM is assigned to IRQ 20 in the list above. Now that its IRQ number is known, the interrupt for the RCIM can be assigned to the shielded processor via the **/proc** file that represents the affinity mask for IRQ 20. The affinity mask for an IRQ is an 8 digit ASCII hexadecimal value. The value is a bit mask of CPUs. Each bit set in the mask represents a CPU where the interrupt routine for this interrupt may be handled. The

radix position of each set bit is the number of a logical CPU that can handle the interrupt. The following command sets the CPU affinity mask for IRQ 20 to CPU 1:

```
$ echo 2 > /proc/irq/20/smp_affinity
```

Note that the "**smp_affinity**" file for IRQs is installed by default with permissions such that only the root user can change the interrupt assignment of an IRQ. The **/proc** file for IRQ affinity can also be read to be sure that the change has taken effect:

```
$ cat /proc/irq/20/smp_affinity
00000002 user 00000002 actual
```

Note that the value returned for "user" is the bit mask that was specified by the user for the IRQ's CPU affinity. The value returned for "actual" will be the resulting affinity after any non-existent CPUs and shielded CPUs have been removed from the mask. Note that shielded CPUs will only be stripped from an IRQ's affinity mask if the user set an affinity mask that contained both shielded and non-shielded CPUs. This is because a CPU shielded from interrupts will only handle an interrupt if there are no unshielded CPUs in the IRQ's affinity mask that could handle the interrupt. In this example, CPU 1 is shielded from interrupts, but CPU 1 will handle IRQ 20 because its affinity mask specifies that only CPU 1 is allowed to handle the interrupt.

The next step is to be sure that the program responding to the RCIM edge-triggered interrupt will run on the shielded processor. Each process in the system has an assigned CPU affinity mask. For a CPU shielded from background processes, only a process that has a CPU affinity mask which specifies ONLY shielded CPUs will be allowed to run on a shielded processor. Note that if there are any non-shielded CPUs in a process' affinity mask, then the process will only execute on the non-shielded CPUs.

The following command will execute the user program "edge-handler" at a real-time priority and force it to run on CPU 1:

```
$ run -s fifo -P 50 -b 1 edge-handler
```

Note that the program could also set its own CPU affinity by calling the library routine **mpadvise(3)** as described in the section "Multiprocessor Control Using mpadvise."

The CPU affinity of the program edge-handler can be checked via the **/proc** file system as well. First the PID of this program must be attained through the **ps(1)** command:

```
$ ps
  PID TTY          TIME CMD
 5548 pts/2        00:00:00 ksh
 9326 pts/2        00:00:00 edge-handler
 9354 pts/2        00:00:00 ps
```

Then view the **/proc** file for this program's affinity mask:

```
$ cat /proc/9326/affinity
00000002 user 00000002 actual
```

Alternatively, the **run(1)** command can be used to check the program's affinity:

```
$ run -i -n edge-handler
Pid   Bias  Actual  Policy  Pri  Nice  Name
9326  0x2   0x2     fifo    50   0     edge-handler
```

Note that the value returned for "user"/"bias" is the bit mask that was specified by the user for the process' CPU affinity. The value returned for "actual" will be the resulting affinity after any non-existent CPUs and shielded CPUs have been removed from the mask. Note that shielded CPUs will only be stripped from a process' affinity mask if the user set an affinity mask that contained both shielded and non-shielded CPUs. This is because a CPU shielded from background processes will only handle a process if there are no unshielded CPUs in the process' affinity mask that could run the program. In this example, CPU 1 is shielded from background processes, but CPU 1 will run the "edge-handler" program because its affinity mask specifies that only CPU 1 is allowed to run this program.

Procedures for Increasing Determinism

The following sections explain various ways in which you can improve performance using the following techniques:

- locking a process' pages in memory
- using favorable static priority assignments
- removing non-critical processing from interrupt level
- speedy wakeup of processes
- judicious use of hyper-threading

Locking Pages in Memory

You can avoid the overhead associated with paging and swapping by using the **`mlockall(2)`**, **`munlockall(2)`**, **`mlock(2)`**, and **`munlock(2)`** system calls.

These system calls allow you to lock and unlock all or a portion of a process' virtual address space in physical memory. These interfaces are based on IEEE Standard 1003.1b-1993.

With each of these calls, pages that are not resident at the time of the call are faulted into memory and locked. To use the **`mlockall(2)`**, **`munlockall(2)`**, **`mlock(2)`**, and **`munlock(2)`** system calls you must have the **CAP_IPC_LOCK** capability (for additional information on capabilities, refer to Chapter 12 and the **`pam_capability(8)`** man page.

Procedures for using these system calls are fully explained in the corresponding man pages.

Setting the Program Priority

The RedHawk Linux kernel accommodates static priority scheduling—that is, processes scheduled under certain POSIX scheduling policies do not have their priorities changed by the operating system in response to their run-time behavior.

Processes that are scheduled under one of the POSIX real-time scheduling policies always have static priorities. (The real-time scheduling policies are **SCHED_RR** and **SCHED_FIFO**; they are explained Chapter 4.) To change a process' scheduling priority, you may use the **sched_setscheduler(2)** and the **sched_setparam(2)** system calls. Note that to use these system calls to change the priority of a process to a higher (more favorable) value, you must have the **CAP_SYS_NICE** capability (for complete information on capability requirements for using these routines, refer to the corresponding man pages).

The highest priority process running on a particular CPU will have the best process dispatch latency. If a process is not assigned a higher priority than other processes running on a CPU, its process dispatch latency will be affected by the time that the higher priority processes spend running. As a result, if you have more than one process that requires good process dispatch latency, it is recommended that you distribute those processes among several CPUs. Refer to the section "Assigning Processes to CPUs," for the procedures for assigning processes to particular CPUs.

Process scheduling is fully described in Chapter 4. Procedures for using the **sched_setscheduler** and **sched_setparam** system calls to change a process' priority are also explained.

Setting the Priority of Deferred Interrupt Processing

Linux supports several mechanisms that are used by interrupt routines in order to defer processing that would otherwise have been done at interrupt level. The processing required to handle a device interrupt is split into two parts. The first part executes at interrupt level and handles only the most critical aspects of interrupt completion processing. The second half of the interrupt routine is deferred to run at program level. By removing non-critical processing from interrupt level, the system can achieve better interrupt response time as described earlier in this chapter in the section "Effect of Interrupts."

The second half of an interrupt routine can be handled by several different kernel daemons, depending on which deferred interrupt technique is used by the device driver. There are kernel tunables that allow a system administrator to set the priority of the kernel daemons that handle deferred interrupt processing. When a real-time task executes on a CPU that is handling deferred interrupts, it is possible to set the priority of the deferred interrupt kernel daemon so that a high-priority user process has a more favorable priority than the deferred interrupt kernel daemon. This allows more deterministic response time for this real-time process.

For more information on deferred interrupt processing, including the daemons used and kernel tunables for setting their priorities, see the chapter "Device Drivers and Real Time."

Waking Another Process

In multiprocess applications, you often need to wake a process to perform a particular task. One measure of the system's responsiveness is the speed with which one process can wake another process. The fastest method you can use to perform this switch to another task is to use the **postwait(2)** system call. For compatibility with legacy code, the **server_block(2)** and **server_wake1(2)** functions are provided in RedHawk Linux.

Procedures for using these functions are explained in Chapter 5 of this guide.

Hyper-threading

Hyper-threading is a feature of the Intel Pentium Xeon processor that allows for a single physical processor to run multiple threads of software applications simultaneously. This is achieved by having two sets of architectural state on each processor while sharing one set of processor execution resources. The architectural state tracks the flow of a program or thread, and the execution resources are the units on the processor that do the work: add, multiply, load, etc. Each of the two sets of architectural state in a hyper-threaded physical CPU can be thought of as a "logical" CPU. The term "sibling CPU" will be used when referring to the other CPU in a pair of logical CPUs that reside on the same physical CPU.

When scheduling threads, the operating system treats the two logical CPUs on a physical CPU as if they were separate processors. Commands like **ps(1)** or **shield(1)** will identify each logical CPU. This allows multiprocessor capable software to run unmodified on twice as many logical processors. While hyper-threading technology will not provide the level of performance scaling achieved by adding a second physical processor, some benchmark tests show that parallel applications can experience as much as a 30 percent gain in performance. See the section "Recommended CPU Configurations" for ideas on how to best utilize hyper-threading for real-time applications.

The performance gain from hyper-threading comes from the fact that having one processor with two logical CPUs allows the processor to more efficiently utilize execution resources. During normal program operation on a non-hyper-threaded CPU there are often times when execution resources on the chip sit idle awaiting input. Because the two logical CPUs share one set of execution resources, the thread executing on the second logical CPU can use resources that would otherwise be idle if only one thread was executing. For example while one logical CPU is stalled waiting for a fetch from memory to complete, the other logical CPU can continue processing its instruction stream. Because the speeds of the processor and the memory bus are very unequal, a processor can spend a significant portion of its time waiting for data to be delivered from memory. Thus, for certain parallel applications hyper-threading provides a significant performance improvement. Another example of the parallelism that can be achieved through hyper-threading is that one logical processor can execute a floating-point operation while the other logical processor executes an addition and a load operation. These operations execute in parallel because they utilize different processor execution units on the chip.

While hyper-threading will generally provide better performance in terms of the execution time for a multi-thread workload, for real-time applications hyper-threading can be problematic. This is because of the impact on the determinism of execution of a thread. Because a hyper-threaded CPU shares the execution unit of the processor with another

thread, the execution unit itself becomes another level of resource contention when a thread executes on a hyper-threaded CPU. Because the execution unit will not always be available when a high priority process on a hyper-threaded CPU attempts to execute an instruction, the amount of time that it takes to execute a code segment on a hyper-threaded CPU is not as predictable as on a CPU that does not have hyper-threading enabled.

A designer of a parallel real-time application will have to make a decision about whether hyper-threading makes sense for his application. It should first be determined whether the application benefits from having its tasks run in parallel on a hyper-threaded CPU as compared to running those tasks sequentially. If hyper-threading does provide a benefit, then the developer can make measurements to determine how much execution time jitter is introduced into the execution speeds of important high-priority threads by running them on a hyper-threaded CPU.

The level of jitter that is acceptable is highly application dependent. If there is an unacceptable amount of jitter introduced into a real-time application because of hyper-threading, then the affected real-time task should be run on a shielded CPU with the sibling CPU marked down via the **`cpu(1)`** command. It should be noted that certain cross processor interrupts will still be handled on a CPU that is marked down (see the **`cpu(1)`** man page for more information). An example of a system with a CPU marked down is given later in this chapter. If desired, hyper-threading can be disabled on a system-wide basis. See the section “RedHawk and Hyper-threading” below for details.

Hyper-threading technology is complementary to MP-based systems because the operating system can not only schedule separate threads to execute on each physical processor simultaneously, but on each logical processor simultaneously as well. This improves overall performance and system response because many parallel threads can be dispatched sooner due to twice as many logical processors being available to the system. Even though there are twice as many logical processors available, they are still sharing the same amount of execution resources. So the performance benefit of another physical processor with its own set of dedicated execution resources will offer greater performance levels. In other words, hyper-threading technology is complementary to multiprocessing by offering greater parallelism within each processor in the system, but is not a replacement for dual or multiprocessing. This can be especially true for applications that are using shielded CPUs for obtaining a deterministic execution environment.

As mentioned above, each logical CPU maintains a complete set of the architecture state. The architecture state (which is *not* shared by the sibling CPUs) consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers and some machine state registers. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses. Each logical processor has its own interrupt controller or APIC. Assigning interrupts to logical CPUs works the same as it always has because interrupts sent to a specific logical CPU are handled only by that logical CPU.

RedHawk and Hyper-threading

Hyper-threading is enabled by default on all physical CPUs. This means that if a system was previously running an older version of RedHawk, commands like **top(1)** and **run(1)** will report twice as many CPUs as were previously present on the same system.

The system administrator can disable hyper-threading on a system-wide basis by turning off the CONFIG_HT kernel tunable accessible through the Processor Type and Features selection of the Linux Kernel Configuration menu or at boot time by specifying the "noht" option. Note that when hyper-threading is disabled on a system-wide basis, the logical CPU numbers are equivalent to the physical CPU numbers.

Hyper-threading can be disabled on a per-CPU basis using the **cpu(1)** command to mark one of the siblings down. Refer to the **cpu(1)** man page for more details.

Recommended CPU Configurations

Hyper-threading technology offers the possibility of better performance for parallel applications. However, because of the manner in which CPU resources are shared between the logical CPUs on a single physical CPU, different application mixes will have varied performance results. This is especially true when an application has real-time requirements requiring deterministic execution times for the application. Therefore, it is important to test the performance of the application under various CPU configurations to determine optimal performance. For example, if there are two tasks that could be run in parallel on a pair of sibling CPUs, be sure to compare the time it takes to execute these tasks in parallel using both siblings versus the time it takes to execute these tasks serially with one of the siblings down. This will determine whether these two tasks can take advantage of the unique kind of parallelism provided by hyper-threading.

Below are suggested ways of configuring an SMP system that contains hyper-threaded CPUs for real-time applications. These examples contain hints about configurations that might work best for applications with various performance characteristics.

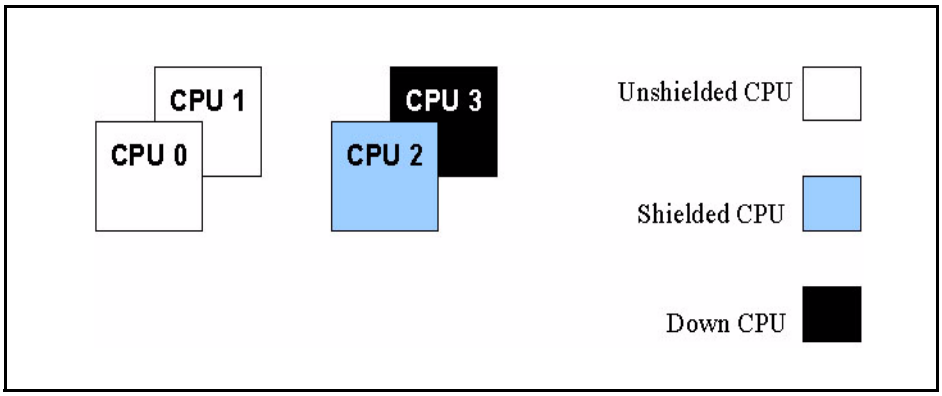
Standard Shielded CPU Model

This model would be used by applications having very strict requirements for determinism in program execution. A shielded CPU provides the most deterministic environment for these types of tasks (see the section "How Shielding Improves Real-Time Performance" for more information on shielded CPUs). In order to maximize the determinism of a shielded CPU, hyper-threading on that physical CPU is disabled. This is accomplished by marking down the shielded CPU's sibling logical CPU using the **cpu(1)** command.

In the Standard Shielded CPU Model, the non-shielded CPUs have hyper-threading enabled. These CPUs are used for a non-critical workload because in general hyper-threading allows more CPU resources to be applied.

Figure 2-7 illustrates the Standard Shielded CPU Model on a system that has two physical CPUs (four logical CPUs). In this example, CPU 3 has been taken down and CPU 2 is shielded from interrupts, processes and hyper-threading. A high priority interrupt and the program responding to that interrupt would be assigned to CPU 2 for the most deterministic response to that interrupt.

Figure 2-7. The Standard Shielded CPU Model



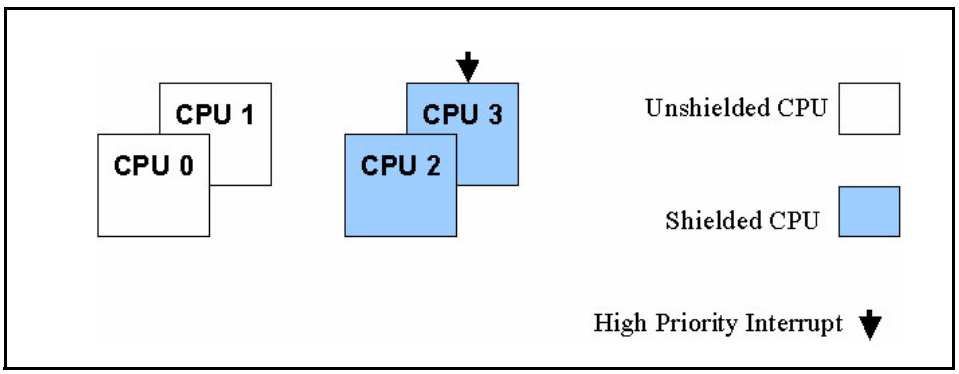
The commands to set up this configuration are:

```
$ shield -a 2
$ cpu -d 3
```

Shielding with Interrupt Isolation

This model is very similar to the Standard Shielded CPU Model. However, in this case all logical CPUs are used, none are taken down. Like the Standard Shielded CPU Model, a subset of the logical CPUs is shielded. But rather than taking down the siblings of the shielded CPUs, those CPUs are also shielded and are dedicated to handling high priority interrupts that require deterministic interrupt response. This is accomplished by shielding the sibling CPUs from processes and interrupts and then setting the CPU affinity of a particular interrupt to that sibling CPU. Shielding with interrupt isolation is illustrated in Figure 2-8.

Figure 2-8. Shielding with Interrupt Isolation



The benefit of this approach is that it provides a small amount of parallelism between the interrupt routine (which runs on CPU 3) and execution of high priority tasks on the sibling CPU (the program awaiting the interrupt runs on CPU 2). Because the interrupt routine is the only code executing on CPU 3, this interrupt routine will generally be held in the L1 cache in its entirety, and the code will stay in the cache, providing optimum execution

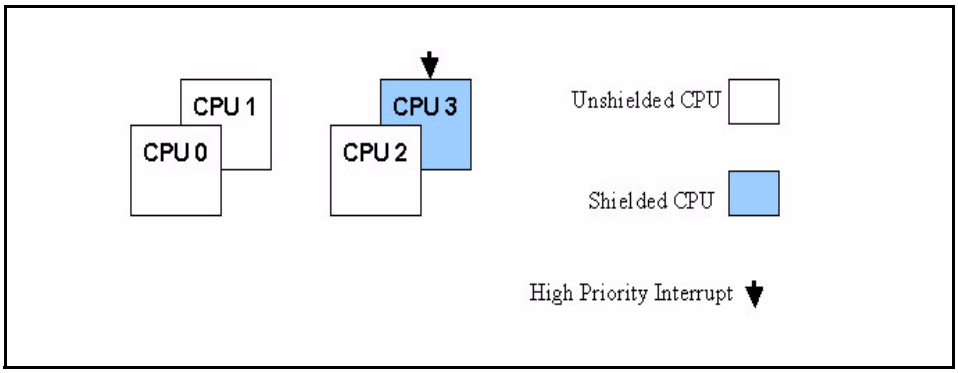
times for the interrupt routine. There is a small penalty to pay however, because the interrupt routine must send a cross processor interrupt in order to wake the task that is awaiting this interrupt on the sibling CPU. This additional overhead has been measured at less than two microseconds.

Another potential use of using shielding with interrupt isolation is to improve I/O throughput for a device. Because we are dedicating a CPU to handling a device interrupt, this interrupt will always complete as quickly as possible when an I/O operation has completed. This allows the interrupt routine to immediately initiate the next I/O operation, providing better I/O throughput.

Hyper-thread Shielding

This configuration is another variation of the Standard Shielded CPU Model. In this case, one sibling is shielded while the other sibling is allowed to run general tasks. The shielded CPU will have its determinism impacted by the activity on its sibling CPU. However, the advantage is that much more of the CPU power of this physical CPU can be utilized by the application. Figure 2-9 illustrates a Hyper-thread Shielding configuration.

Figure 2-9. Hyper-thread Shielding



In this example, CPU 3 is shielded and allowed to run only a high priority interrupt and the program that responds to that interrupt. CPU 2 is either not shielded and therefore available for general use or is set up to run a specific set of tasks. The tasks that run on CPU 2 will not directly impact interrupt response time, because when they disable preemption or block interrupts there is no effect on the high priority interrupt or task running on CPU 3. However, at the chip resource level there is contention that will impact the determinism of execution on CPU 3. The amount of impact is very application dependent.

Floating-point / Integer Sharing

This configuration can be used when the application has some programs that primarily perform floating-point operations and some programs that primarily perform integer arithmetic operations. Both siblings of a hyper-threaded CPU are used to run specific tasks. Programs that are floating-point intensive are assigned to one sibling CPU and programs that primarily execute integer operations are assigned to the other sibling CPU. The benefit of this configuration is that floating-point operations and integer operations use different chip resources. This allows the application to make good use of hyper-thread style parallelism because there is more parallelism that can be exploited at the chip level.

It should also be noted that applications on the CPU that are only performing integer operations would see faster context switch times because there won't be save/restore of the floating-point registers during the context switch.

Shared Data Cache

This configuration can be used when the application is a producer/consumer style of application. In other words, one process (the consumer) is operating on data that has been passed from another process (the producer). In this case, the producer and consumer threads should be assigned to the siblings of a hyper-threaded CPU. Because the two sibling CPUs share the data cache, it is likely that the data produced by the producer process is still in the data cache when the consumer process accesses the data that has been passed from the producer task. Using two sibling CPUs in this manner allows the producer and consumer tasks to operate in parallel, and the data passed between them is essentially passed via the high-speed cache memory. This offers significant opportunity for exploiting hyper-thread style parallelism.

Another potential use of this model is for the process on one sibling CPU to pre-fetch data into the data cache for a process running on the other sibling on a hyper-threaded CPU.

Shielded Uniprocessor

This configuration is a variation of the Hyper-thread Shielding configuration. The only difference is that we are applying this technique to a uniprocessor rather than to one physical CPU in an SMP system. Because a physical CPU now contains two logical CPUs, a uniprocessor can now be used to create a shielded CPU. In this case, one of the CPUs is marked shielded while the other CPU is used to run background activity. Determinism on this type of shielded CPU will not be as solid as using CPU shielding on a distinct physical CPU, but it will be significantly better than with no shielding at all.

Known Issues with Linux Determinism

The following are issues with standard Linux that are known to have a negative impact on real-time performance. These actions are generally administrative in nature and should not be performed while the system is executing a real-time application.

- The **`hdparm(1)`** utility is a command line interface for enabling special parameters for IDE and SCSI disks. This utility is known to disable interrupts for very lengthy periods of time.
- The **`blkdev_close(2)`** interface is used by BootLoaders to write to the raw block device. This is known to disable interrupts for very lengthy periods of time.
- Avoid scrolling the frame-buffer (fb) console. This is known to disable interrupts for very lengthy periods of time.
- When using virtual consoles, don't switch consoles. This is known to disable interrupts for very lengthy periods of time.

- Avoid mounting and unmounting CDs and unmounting file systems. These actions produce long latencies.
- Turn off auto-mount of CDs. This is a polling interface and the periodic poll introduces long latencies.
- By default the Linux kernel locks the Big Kernel Lock (BKL) before calling a device driver's **`ioctl()`** routine. This can cause delays when the **`ioctl()`** routine is called by a real-time process or is called on a shielded CPU. See the "Device Drivers and Real Time" chapter for more information on how to correct this problem.

3
Real-Time Interprocess Communication

3
Real-Time Interprocess Communication

This chapter describes RedHawk Linux support for real-time interprocess communication through POSIX and System V message passing facilities.

Appendix A contains an example program that illustrates the use of the System V message queue facilities.

Overview

Message queues allow one or more processes to write messages to be read by one or more reading processes.

Real-time interprocess communication support in RedHawk Linux includes POSIX message-passing facilities that are based on the IEEE 1003.1b-1993 Standard as well as the System V message type of interprocess communication (IPC).

POSIX message passing facilities provide the means for passing arbitrary amounts of data between cooperating processes. POSIX message queue library routines allow a process to create, open, query and destroy a message queue, send and receive messages from a message queue, associate a priority with a message to be sent, and request asynchronous notification when a message arrives.

System V messages are also supported. Information about this type of interprocess communication (IPC) is presented in “Understanding System V Messages.”

POSIX and System V messaging functionality operate independent of each other. The recommended message-passing mechanism is the POSIX message queue facility because of its efficiency and portability.

Understanding POSIX Message Queues

An application may consist of multiple cooperating processes, possibly running on separate processors. These processes may use system-wide POSIX message queues to efficiently communicate and coordinate their activities.

The primary use of POSIX message queues is for passing data between processes. In contrast, there is little need for functions that pass data between cooperating threads in the same process because threads within the same process already share the entire address space. However, nothing prevents an application from using message queues to pass data between threads in one or more processes.

“Understanding Basic Concepts” presents basic concepts related to the use of POSIX message queues. “Understanding Advanced Concepts” presents advanced concepts.

Understanding Basic Concepts

POSIX message queues are implemented as files in the **mqueue** file system. This file system must be mounted on the mountpoint **/dev/mqueue**. A message queue appears as a file in the directory **/dev/mqueue**; for example:

```
/dev/mqueue/my_queue
```

Cat(1), **stat(1)**, **ls(1)**, **chmod(1)**, **chown(1)**, **rm(1)**, **touch(1)**, and **umask(2)** all work on mqueue files as on other files. File permissions and modes also work the same as with other files.

Other common file manipulation commands like **copy** may work to some degree, but mqueue files, although reported as regular files by the VFS, are not designed to be used in this manner. One may find some utility with some commands, however; notably **cat** and **touch**:

- **touch** will create a queue with limits set to '0'.
- **cat** will produce some information in four fields:

QSIZE	number of bytes in memory occupied by the entire queue
NOTIFY	notification marker (see **mq_notify(2)**)
SIGNO	the signal to be generated during notification
NOTIFY_PID	which process should be notified

One cannot create a directory under **/dev/mqueue** because the mqueue file system does not support directories. One might use the system calls **close(2)**, **open(2)**, and **unlink(2)** for operating on POSIX message queues within source code, but full access to POSIX message queue features can only be achieved through library routines provided by the RedHawk Linux real-time library interface, which serves as a wrapper around kernel-level **ioctl**s. The **ioctl**s perform the actual operations. See "Understanding Message Queue Library Routines" for futher discussion on using the interface.

When the RedHawk Linux real-time library rpm is installed, the mountpoint **/dev/mqueue** is created and the following line is appended to **/etc/fstab**:

```
none     /dev/mqueue      mqueue   defaults     0 0
```

To manually mount the mqueue file system on **/dev/mqueue**, issue the command:

```
# mkdir -p /dev/mqueue        (if the mountpoint does not exist)
# mount -t mqueue none /dev/mqueue
```

This is how the mounted mqueue file system appears after issuing the **mount** command with no options (see the **mount(8)** man page).

```
none on /dev/mqueue type mqueue (rw)
```

It is important not to become confused about the nature of **`/dev/mqueue`**. It is a mountpoint only and ***not*** a device file. The 'none' argument in the mount command reflects this:

```
mount -t type device dir
```

type = mqueue
device = none
dir = /dev/mqueue

The major number 0 (reserved as null device number) is used by the kernel for unnamed devices (e.g. non-device mounts).

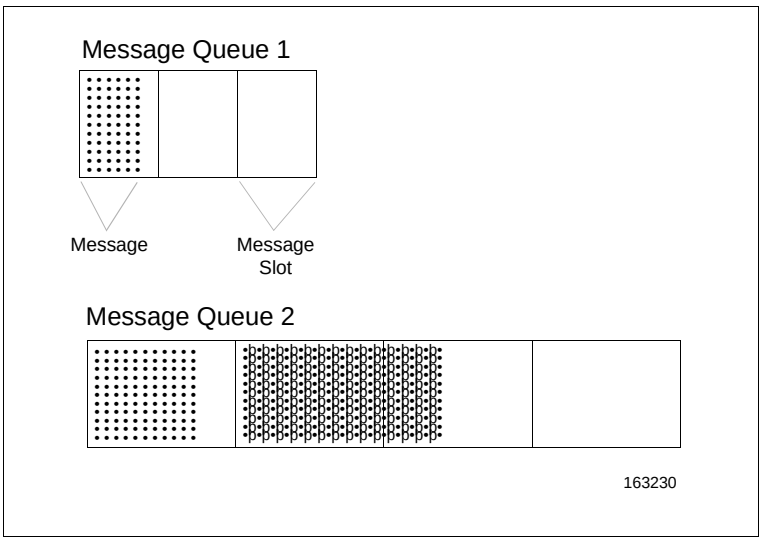
The act of mounting **`/dev/mqueue`** starts the file system driver. Nothing prevents mounting the mqueue file system on paths other than **`/dev/mqueue`**, but the interface and driver will reject references to POSIX message queues that aren't rooted under **`/dev/mqueue`**.

The following system limits apply to POSIX message queues:

MQ_MAXMSG	40	max number of message queues
MQ_MAX	4	max number of messages in each queue
MQ_MSGSIZE	16384	max message size
MQ_MAXSYSSIZE	1048576	max size that all queues can have together
MQ_PRIO_MAX	32768	max message priority

A message queue consists of message slots. To optimize message sending and receiving, all message slots within one message queue are the same size. A message slot can hold one message. The message size may differ from the message slot size, but it must not exceed the message slot size. Messages are not padded or null-terminated; message length is determined by byte count. Message queues may differ by their message slot sizes and the maximum number of messages they hold. Figure 3-1 illustrates some of these facts.

Figure 3-1. Example of Two Message Queues and Their Messages



POSIX message queue library routines allow a process to:

- create, open, query, close, and destroy a message queue
- send messages to and receive messages from a message queue (may be timed)
- associate a priority with a message to be sent
- request asynchronous notification via a user-specified signal when a message arrives at a specific empty message queue

Processes communicate with message queues via message queue descriptors. A child process created by a **fork(2)** system call inherits all of the parent process' open message queue descriptors. The **exec(2)** and **exit(2)** system calls close all open message queue descriptors.

When one thread within a process opens a message queue, all threads within that process can use the message queue if they have access to the message queue descriptor.

A process attempting to send messages to or receive messages from a message queue may have to wait. Waiting is also known as being blocked.

Two different types of priority play a role in message sending and receiving: message priority and process-scheduling priority. Every message has a message priority. The oldest, highest-priority message is received first by a process.

Every process has a scheduling priority. Assume that multiple processes are blocked to send a message to a full message queue. When space becomes free in that message queue, the system wakes the highest-priority process; this process sends the next message. When there are multiple processes having the highest priority, the one that has been blocked the longest is awakened. Assume that multiple processes are blocked to receive a message from an empty message queue. When a message arrives at that message queue, the same criteria is used to determine the process that receives the message.

Understanding Advanced Concepts

Spin locks synchronize access to the message queue, protecting message queue structures. While a spin lock is locked, most signals are blocked to prevent the application from aborting. However, certain signals cannot be blocked.

Assume that an application uses message queues and has a lock. If a signal aborts this application, the message queue becomes unusable by any process; all processes attempting to access the message queue hang attempting to gain access to the lock. For successful accesses to be possible again, a process must destroy the message queue via **mq_unlink(2)** and re-create the message queue via **mq_open(2)**. For more information on these routines, see "Using the mq_unlink Routine" and "Using the mq_open Routine," respectively.

Understanding Message Queue Library Routines

The POSIX library routines that support message queues depend on a message queue attribute structure. “Understanding the Message Queue Attribute Structure” describes this structure. “Using the Library Routines” presents the library routines.

All applications that call message queue library routines must link in the Concurrent real-time library. You may link this library either statically or dynamically. The following example shows the typical command-line format:

gcc [*options...*] *file* **-lccur_rt** ...

Understanding the Message Queue Attribute Structure

The message queue attribute structure `mq_attr` holds status and attribute information about a specific message queue. When a process creates a message queue, it automatically creates and initializes this structure. Every attempt to send messages to or receive messages from this message queue updates the information in this structure. Processes can query the values in this structure.

You supply a pointer to an `mq_attr` structure when you invoke **mq_getattr(2)** and optionally when you invoke **mq_open(2)**. Refer to “Using the mq_getattr Routine” and “Using the mq_open Routine,” respectively, for information about these routines.

The `mq_attr` structure is defined in **<mqueue.h>** as follows:

```
struct mq_attr {
    long mq_flags;
    long mq_maxmsg;
    long mq_msgsize;
    long mq_curmsgs;
};
```

The fields in the structure are described as follows.

`mq_flags`	a flag that indicates whether or not the operations associated with this message queue are in nonblocking mode
`mq_maxmsg`	the maximum number of messages this message queue can hold
`mq_msgsize`	the maximum size in bytes of a message in this message queue
`mq_curmsgs`	the number of messages currently in this message queue

Using the Library Routines

The POSIX library routines that support message queues are briefly described as follows:

mq_open	create and open a new message queue or open an existing message queue
mq_close	close an open message queue
mq_unlink	remove a message queue and any messages in it
mq_send	write a message to an open message queue
mq_timedsend	write a message to an open message queue with timeout value
mq_receive	read the oldest, highest-priority message from an open message queue
mq_timedreceive	read the oldest, highest-priority message from an open message queue with timeout value
mq_notify	register for notification of the arrival of a message at an empty message queue such that when a message arrives, the calling process is sent a user-specified signal
mq_setattr	set the attributes associated with a message queue
mq_getattr	obtain status and attribute information about an open message queue

Procedures for using each of the routines are presented in the sections that follow.

Using the mq_open Routine

The **mq_open(2)** library routine establishes a connection between a calling process and a message queue. Depending on flag settings, **mq_open** may create a message queue. The **mq_open** routine always creates and opens a new message queue descriptor. Most other library routines that support message queues use this message queue descriptor to refer to a message queue.

Synopsis

```
#include <mqueue.h>

mqd_t mq_open(const char name, int oflag, /*
mode_t mode, struct mq_attr *attr */ ...);
```

The arguments are defined as follows:

name is concatenated with the mountpoint **/dev/mqueue** to form an absolute path naming the mqueue file. For example, if name is **/my_queue**, the path becomes **/dev/mqueue/my_queue**. This path must be within the limits of PATH_MAX.

Processes calling **mq_open** with the same value of *name* refer to the same message queue. If the *name* argument is not the name of an existing message queue and you did not request creation, **mq_open** fails and returns an error.

oflag an integer value that shows whether the calling process has send and receive access to the message queue; this flag also shows whether the calling process is creating a message queue or establishing a connection to an existing one.

The *mode* a process supplies when it creates a message queue may limit the *oflag* settings for the same message queue. For example, assume that at creation, the message queue *mode* permits processes with the same effective group ID to read but not write to the message queue. If a process in this group attempts to open the message queue with *oflag* set to write access (**O_WRONLY**), **mq_open** returns an error.

The only way to change the *oflag* settings for a message queue is to call **mq_close** and **mq_open** to respectively close and reopen the message queue descriptor returned by **mq_open**.

Processes may have a message queue open multiple times for sending, receiving, or both. The value of *oflag* must include exactly one of the three following access modes:

O_RDONLY Open a message queue for receiving messages. The calling process can use the returned message queue descriptor with **mq_receive** but not **mq_send**.

O_WRONLY Open a message queue for sending messages. The calling process can use the returned message queue descriptor with **mq_send** but not **mq_receive**.

O_RDWR Open a message queue for both receiving and sending messages. The calling process can use the returned message queue descriptor with **mq_send** and **mq_receive**.

The value of *oflag* may also include any combination of the remaining flags:

O_CREAT Create and open an empty message queue if it does not already exist. If message queue *name* is not currently open, this flag causes **mq_open** to create an empty message queue. If message queue *name* is already open on the system, the effect of this flag is as noted under **O_EXCL**. When you set the **O_CREAT** flag, you must also specify the *mode* and *attr* arguments.

A newly-created message queue has its user ID set to the calling process' effective user ID and its group ID set to the calling process' effective group ID.

O_EXCL Return an error if the calling process attempts to create an existing message queue. The **mq_open** routine

fails if **O_EXCL** and **O_CREAT** are set and message queue *name* already exists. The **mq_open** routine succeeds if **O_EXCL** and **O_CREAT** are set and message queue *name* does not already exist. The **mq_open** routine ignores the setting of **O_EXCL** if **O_EXCL** is set but **O_CREAT** is not set.

O_NONBLOCK On an **mq_send**, return an error rather than wait for space to become free in a full message queue. On an **mq_receive**, return an error rather than wait for a message to arrive at an empty message queue.

mode an integer value that sets the read, write, and execute/search permission for a message queue if this is the **mq_open** call that creates the message queue. The **mq_open** routine ignores all other mode bits (for example, *setuid*). The setting of the file-creation mode mask, **umask**, modifies the value of *mode*. For more information on mode settings, see the **chmod(1)** and **umask(2)** man pages.

When you set the **O_CREAT** flag, you must specify the *mode* argument to **mq_open**.

attr the null pointer constant or a pointer to a structure that sets message queue attributes--for example, the maximum number of messages in a message queue and the maximum message size. For more information on the `mq_attr` structure, see "Understanding the Message Queue Attribute Structure."

If *attr* is NULL, the system creates a message queue with system limits. If *attr* is not NULL, the system creates the message queue with the attributes specified in this field. If *attr* is specified, it takes effect only when the message queue is actually created.

If *attr* is not NULL, the following attributes must be set to a value greater than zero:

```
attr.mq_maxmsg
attr.mq_msgsize
```

A return value of a message queue descriptor shows that the message queue has been successfully opened. A return value of ((mqd_t) **-1**) shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_open(2)** man page for a listing of the types of errors that may occur.

Using the mq_close Routine

The **mq_close(2)** library routine breaks a connection between a calling process and a message queue. The **mq_close** routine does this by removing the message queue descriptor that the calling process uses to access a message queue. The **mq_close** routine does not affect a message queue itself or the messages in a message queue.

Note

If a process requests notification about a message queue and later closes its connection to the message queue, this request is removed; the message queue is available for another process to request notification. For information on notification requests via **mq_notify**, see "Using the mq_notify Routine."

Synopsis

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

The argument is defined as follows:

mqdes a message queue descriptor obtained from an **mq_open**.

A return value of **0** shows that the message queue has been successfully closed. A return value of **-1** shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_close(2)** man page for a listing of the types of errors that may occur.

Using the mq_unlink Routine

The **mq_unlink(2)** library routine prevents further **mq_open** calls to a message queue. When there are no other connections to this message queue, **mq_unlink** removes the message queue and the messages in it.

Synopsis

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

The argument is defined as follows:

name is concatenated with the mountpoint **/dev/mqueue** to form an absolute path naming the mqueue file. For example, if name is **/my_queue**, the path becomes **/dev/mqueue/my_queue**. This path must be within the limits of PATH_MAX.

If a process has message queue *name* open when **mq_unlink** is called, **mq_unlink** immediately returns; destruction of message queue *name* is postponed until all references to the message queue have been closed. A process can successfully remove message queue *name* only if the **mq_open** that created this message queue had a *mode* argument that granted the process both read and write permission.

A return value of **0** shows that a message queue has been successfully removed. A return value of **-1** shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_unlink(2)** man page for a listing of the types of errors that may occur.

Using the mq_send and mq_timedsend Routines

The **mq_send(2)** library routine adds a message to the specified message queue. The **mq_send** routine is an *async-safe* operation; that is, you can call it within a signal-handling routine.

A successful **mq_send** to an empty message queue causes the system to wake the highest priority process that is blocked to receive from that message queue. If a message queue has a notification request attached and no processes blocked to receive, a successful **mq_send** to that message queue causes the system to send a signal to the process that attached the notification request. For more information, read about **mq_receive** in "Using the mq_receive and mq_timedreceive Routines" and **mq_notify** in "Using the mq_notify Routine."

The **mq_timedsend** library routine can be used to specify a timeout value so that if the specified message queue is full, the wait for sufficient room in the queue is terminated when the timeout expires.

Synopsis

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

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
unsigned int msg_prio);

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t
msg_len, unsigned int msg_prio, const struct timespec
*abs_timeout);
```

The arguments are defined as follows:

mqdes a message queue descriptor obtained from an **mq_open**. If the specified message queue is full and **O_NONBLOCK** is set in *mqdes*, the message is not queued, and **mq_send** returns an error. If the specified message queue is full and **O_NONBLOCK** is not set in *mqdes*, **mq_send** blocks until space becomes available to queue the message or until **mq_send** is interrupted by a signal.

Assume that multiple processes are blocked to send a message to a full message queue. When space becomes free in that message queue (because of an **mq_receive**), the system wakes the highest-priority process that has been blocked the longest. This process sends the next message.

For **mq_send** to succeed, the **mq_open** call for this message queue descriptor must have had **O_WRONLY** or **O_RDWR** set in *oflag*. For information on **mq_open**, see "Using the mq_open Routine."

msg_ptr a string that specifies the message to be sent to the message queue represented by *mqdes.*

msg_len an integer value that shows the size in bytes of the message pointed to by *msg_ptr*. The **mq_send** routine fails if *msg_len* exceeds the `mq_msgsize` message size attribute of the message queue set on the creating **mq_open**. Otherwise, the **mq_send** routine copies the message pointed to by the *msg_ptr* argument to the message queue.

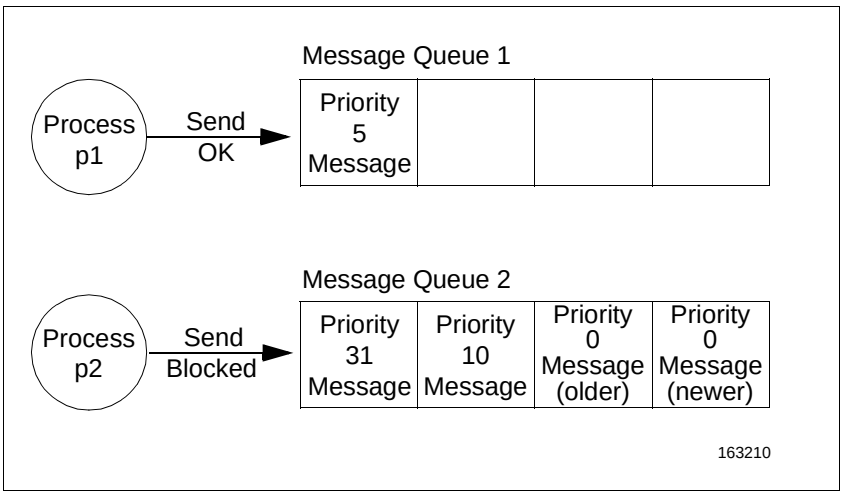
msg_prio an unsigned integer value that shows the message priority. The system keeps messages in a message queue in order by message priority. A newer message is queued before an older one only if the newer message has a higher message priority. The value for *msg_prio* ranges from 0 through **MQ_PRIO_MAX**, where 0 represents the least favorable priority. For correct usage, the message priority of an urgent message should exceed that of an ordinary message. Note that message priorities give you some ability to define the message receipt order but not the message recipient.

abs_timeout
a timeout value in nanoseconds. The range is 0 to 1000 million. If the specified message queue is full and **O_NONBLOCK** is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue is terminated when the timeout expires.

Figure 3-2 illustrates message priorities within a message queue and situations where processes are either blocked or are free to send a message to a message queue. Specifically, the following facts are depicted:

- The operating system keeps messages in each message queue in order by message priority.
- Several messages within the same message queue may have the same message priority.
- By default, a process trying to send a message to a full message queue is blocked.

Figure 3-2. The Result of Two mq_sends



A return value of **0** shows that the message has been successfully sent to the designated message queue. A return value of **-1** shows that an error has occurred; **`errno`** is set to show the error. Refer to the **`mq_send(2)`** man page for a listing of the types of errors that may occur.

Using the mq_receive and mq_timedreceive Routines

The **`mq_receive(2)`** library routine reads the oldest of the highest-priority messages from a specific message queue, thus freeing space in the message queue. The **`mq_receive`** routine is an *async-safe* operation; that is, you can call it within a signal-handling routine.

A successful **`mq_receive`** from a full message queue causes the system to wake the highest-priority process that is blocked to send to that message queue. For more information, read about **`mq_send`** in "Using the mq_send and mq_timedsend Routines."

The **`mq_timedreceive`** library routine can be used to specify a timeout value so that if no message exists on the queue to satisfy the receive, the wait for such a message is terminated when the timeout expires.

Synopsis

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

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t
msg_len, unsigned int msg_prio);

ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t
msg_len, unsigned int msg_prio, const struct timespec
*abs_timeout);
```

The arguments are defined as follows:

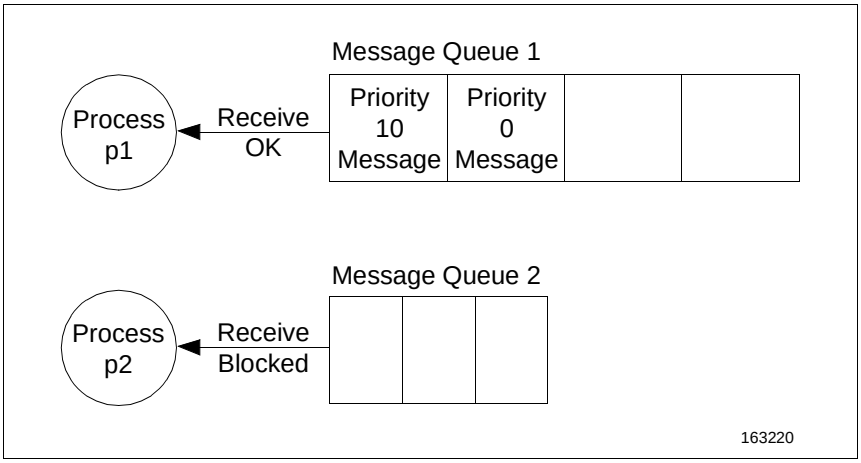
mqdes a message queue descriptor obtained from an **mq_open**. If **O_NONBLOCK** is set in *mqdes* and the referenced message queue is empty, nothing is read, and **mq_receive** returns an error. If **O_NONBLOCK** is not set in *mqdes* and the specified message queue is empty, **mq_receive** blocks until a message becomes available or until **mq_receive** is interrupted by a signal.

Assume that multiple processes are blocked to receive a message from an empty message queue. When a message arrives at that message queue (because of an **mq_send**), the system wakes the highest-priority process that has been blocked the longest. This process receives the message.

For **mq_receive** to succeed, the process' **mq_open** call for this message queue must have had **O_RDONLY** or **O_RDWR** set in *oflag*. For information on **mq_open**, see "Using the mq_open Routine."

Figure 3-3 shows two processes without **O_NONBLOCK** set in *mqdes*. Although both processes are attempting to receive messages, one process is blocked because it is accessing an empty message queue. In the figure, the arrows indicate the flow of data.

Figure 3-3. The Result of Two mq_receives



msg_ptr a pointer to a character array (message buffer) that will receive the message from the message queue represented by *mqdes*. The return value of a successful **mq_receive** is a byte count.

msg_len an integer value that shows the size in bytes of the array pointed to by *msg_ptr*. The **mq_receive** routine fails if *msg_len* is less than the `mq_msgsize` message-size attribute of the message queue set on the creating **mq_open**. Otherwise, the **mq_receive** routine removes the message from the message queue and copies it to the array pointed to by the *msg_ptr* argument.

msg_prio the null pointer constant or a pointer to an unsigned integer variable that will receive the priority of the received message. If *msg_prio* is NULL, the **mq_receive** routine discards the message priority. If *msg_prio* is not NULL, the **mq_receive** routine stores the priority of the received message in the location referenced by *msg_prio*. The received message is the oldest, highest-priority message in the message queue.

abs_timeout
a timeout value in nanoseconds. The range is 0 to 1000 million. If **O_NONBLOCK** is not specified when the message queue was opened via **mq_open(2)**, and no message exists on the queue to satisfy the receive, the wait for such a message is terminated when the timeout expires.

A return value of **-1** shows that an error has occurred; **errno** is set to show the error and the contents of the message queue are unchanged. A non-negative return value shows the length of the successfully-received message; the received message is removed from the message queue. Refer to the **mq_receive(2)** man page for a listing of the types of errors that may occur.

Using the mq_notify Routine

The **mq_notify(2)** library routine allows the calling process to register for notification of the arrival of a message at an empty message queue. This functionality permits a process to continue processing rather than blocking on a call to **mq_receive(2)** to receive a message from a message queue (see "Using the mq_receive and mq_timedreceive Routines" for an explanation of this routine). Note that for a multithreaded program, a more efficient means of attaining this functionality is to spawn a separate thread that issues an **mq_receive** call.

At any time, only one process can be registered for notification by a message queue. However, a process can register for notification by each *mqdes* it has open except an *mqdes* for which it or another process has already registered. Assume that a process has already registered for notification of the arrival of a message at a particular message queue. All future attempts to register for notification by that message queue will fail until notification is sent to the registered process or the registered process removes its registration. When notification is sent, the registration is removed for that process. The message queue is again available for registration by any process.

Assume that one process blocks on **mq_receive** and another process registers for notification of message arrival at the same message queue. When a message arrives at the message queue, the blocked process receives the message, and the other process' registration remains pending.

Synopsis

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent
*notification);
```

The arguments are defined as follows:

mqdes a message queue descriptor obtained from an **mq_open**.

notification

the null pointer constant or a pointer to a structure that specifies the way in which the calling process is to be notified of the arrival of a message at the specified message queue. If *notification* is not NULL and neither the calling process nor any other process has already registered for notification by the specified message queue, **mq_notify** registers the calling process to be notified of the arrival of a message at the message queue. When a message arrives at the empty message queue (because of an **mq_send**), the system sends the signal specified by the *notification* argument to the process that has registered for notification. Usually the calling process reacts to this signal by issuing an **mq_receive** on the message queue.

When notification is sent to the registered process, its registration is removed. The message queue is then available for registration by any process.

If *notification* is NULL and the calling process has previously registered for notification by the specified message queue, the existing registration is removed.

If the value of *notification* is not NULL, the only meaningful value that *notification->sigevent.sigev_notify* can specify is **SIGEV_SIGNAL**. With this value set, a process can specify a signal to be delivered upon the arrival of a message at an empty message queue.

If you specify **SIGEV_SIGNAL**, *notification->sigevent.sigev_signal* must specify the number of the signal that is to be generated, and *notification->sigevent.sigev_value* must specify an application-defined value that is to be passed to a signal-handling routine defined by the receiving process. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file **<signal.h>**. The application-defined value may be a pointer or an integer value. If the process catching the signal has invoked the **sigaction(2)** system call with the **SA_SIGINFO** flag set prior to the time that the signal is generated, the signal and the application-defined value are queued to the process when a message arrives at the message queue. The `siginfo_t` structure may be examined in the signal handler when the routine is entered. The following values should be expected (see **siginfo.h**):

`si_value` specified *sigevent.sigev_value* to be sent on notification
`si_code` SI_MESGQ (real time message queue state change value: -3)
`si_signo` specified *sigevent.sigev_signal* to be generated
`si_errno` associated `errno` value with this signal

A return value of **0** shows that the calling process has successfully registered for notification of the arrival of a message at the specified message queue. A return value of **-1** shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_notify(2)** man page for a listing of the types of errors that may occur.

Using the mq_setattr Routine

The **mq_setattr(2)** library routine allows the calling process to set the attributes associated with a specific message queue.

Synopsis

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat,
struct mq_attr *omqstat);
```

The arguments are defined as follows:

mqdes a message queue descriptor obtained from an **mq_open**. The **mq_setattr** routine sets the message queue attributes for the message queue associated with *mqdes*.

mqstat a pointer to a structure that specifies the flag attribute of the message queue referenced by *mqdes*. The value of this flag may be zero or **O_NONBLOCK**. **O_NONBLOCK** causes the **mq_send** and **mq_receive** operations associated with the message queue to operate in nonblocking mode.

The values of mq_maxmsg, mq_msgsize, and mq_curmsgs are ignored by **mq_setattr**.

For information on the mq_attr structure, see "Understanding the Message Queue Attribute Structure." For information on the **mq_send** and **mq_receive** routines, see "Using the mq_send and mq_timedsend Routines" and "Using the mq_receive and mq_timedreceive Routines," respectively.

omqstat the null pointer constant or a pointer to a structure to which information about the previous attributes and the current status of the message queue referenced by *mqdes* is returned. For information on the mq_attr structure, see "Understanding the Message Queue Attribute Structure."

A return value of **0** shows that the message queue attributes have been successfully set as specified. A return value of **-1** shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_setattr(2)** man page for a listing of the types of errors that may occur.

Using the mq_getattr Routine

The **mq_getattr(2)** library routine obtains status and attribute information associated with a specific message queue.

Synopsis

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

The arguments are defined as follows:

mqdes a message queue descriptor obtained from an **mq_open**. The **mq_getattr** routine provides information about the status and attributes of the message queue associated with *mqdes*.

mqstat a pointer to a structure that receives current information about the status and attributes of the message queue referenced by *mqdes*. For information on the `mq_attr` structure, see "Understanding the Message Queue Attribute Structure."

A return value of **0** shows that the message queue attributes have been successfully attained. A return value of **-1** shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_getattr(2)** man page for a listing of the types of errors that may occur.

Understanding System V Messages

The System V message type of interprocess communication (IPC) allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can send and receive messages.

Before a process can send or receive a message, it must have the operating system generate the necessary software mechanisms to handle these operations. A process does this using the **msgget(2)** system call. In doing this, the process becomes the owner/creator of a message queue and specifies the initial operation permissions for all processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl(2)** system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl** to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process which is attempting to send a message can wait until it becomes possible to post the message to the specified message queue; the receiving process isn't involved (except indirectly; for example, if the consumer isn't consuming, the queue space will eventually be exhausted) and vice versa. A process which specifies that execution is to be suspended is performing a *blocking message operation*. A process which does not allow its execution to be suspended is performing a *nonblocking message operation.*

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- the operation is successful
- the process receives a signal
- the message queue is removed from the system

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, **-1** is returned to the process, and an external error number variable, **errno**, is set accordingly.

Using Messages

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier is called the message queue identifier (`msqid`); it is used to identify or refer to the associated message queue and data structure. This identifier is accessible by any process in the system, subject to normal access restrictions.

A message queue's corresponding kernel data structures are used to maintain information about each message being sent or received. This information, which is used internally by the system, includes the following for each message:

- message type
- message text size
- message text address

There is one associated data structure for the uniquely identified message queue, `msqid_ds`. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

NOTE

All C header files discussed in this chapter are located in the **`/usr/include`** subdirectories.

The definition of the associated message queue data structure `msqid_ds` includes the members shown in Figure 3-4.

Figure 3-4. Definition of msqid_ds Structure

```
struct ipc_perm msg_perm;/* structure describing operation permission */
__time_t msg_stime; /* time of last msgsnd command */
__time_t msg_rtime; /* time of last msgrcv command */
__time_t msg_ctime; /* time of last change */
unsigned long int __msg_cbytes; /* current number of bytes on queue */
msgqnum_t msg_qnum; /* number of messages currently on queue */
msglen_t msg_qbytes;/* max number of bytes allowed on queue */
__pid_t msg_lspid;  /* pid of last msgsnd() */
__pid_t msg_lrpid;  /* pid of last msgrcv() */
```

The C programming language data structure definition for `msqid_ds` should be obtained by including the <**sys/msg.h**> header file, even though this structure is actually defined in <**bits/msq.h**>.

The definition of the interprocess communication permissions data structure, `ipc_perm`, includes the members shown in Figure 3-5:

Figure 3-5. Definition of ipc_perm Structure

```
__key_t __key;                      /* Key.  */
__uid_t uid;                        /* Owner's user ID.  */
__gid_t gid;                        /* Owner's group ID.  */
__uid_t cuid;                       /* Creator's user ID.  */
__gid_t cgid;                       /* Creator's group ID.  */
unsigned short int mode;            /* Read/write permission.  */
unsigned short int __seq;           /* Sequence number.  */
```

The C programming language data structure definition of `ipc_perm` should be obtained by including the <**sys/ipc.h**> header file, even though the actual definition for this structure is located in <**bits/ipc.h**>. Note that <**sys/ipc.h**> is commonly used for all IPC facilities.

The **msgget(2)** system call performs one of two tasks:

- creates a new message queue identifier and create an associated message queue and data structure for it
- locates an existing message queue identifier that already has an associated message queue and data structure

Both tasks require a *key* argument passed to the **msgget** system call. If *key* is not already in use for an existing message queue identifier, a new identifier is returned with an

associated message queue and data structure created for the key it, provided no system tunable parameter would be exceeded.

There is also a provision for specifying a *key* of value zero (0), known as the private key (**IPC_PRIVATE**). When this key is specified, a new identifier is always returned with an associated message queue and data structure created for it, unless a system limit for the maximum number of message queues (**MSGMNI**) would be exceeded. The **ipcs(8)** command will show the *key* field for the `msqid` as all zeros.

If a message queue identifier exists for the key specified, the value of the existing identifier is returned. If you do not want to have an existing message queue identifier returned, a control command (**IPC_EXCL**) can be specified (set) in the *msgflg* argument passed to the system call (see "Using msgget" for details of this system call).

When a message queue is created, the process that calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator. The message queue creator also determines the initial operation permissions for it.

Once a uniquely identified message queue has been created or an existing one is found, **msgop(2)** (message operation) and **msgctl(2)** (message control) can be used.

Message operations, as mentioned before, consist of sending and receiving messages. The **msgsnd** and **msgrcv** system calls are provided for each of these operations (see "Operations for Messages" for details of the **msgsnd** and **msgrcv** system calls).

The **msgctl** system call permits you to control the message facility in the following ways:

- by retrieving the data structure associated with a message queue identifier (**IPC_STAT**)
- by changing operation permissions for a message queue (**IPC_SET**)
- by changing the size (`msg_qbytes`) of the message queue for a particular message queue identifier (**IPC_SET**)
- by removing a particular message queue identifier from the RedHawk Linux operating system along with its associated message queue and data structure (**IPC_RMID**)

See the section "Using msgctl" for details of the **msgctl** system call.

Getting Message Queues

This section describes how to use the **msgget** system call. The accompanying program illustrates its use.

Using msgget

Synopsis

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

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

All of these `#include` files are located in the **/usr/include** subdirectories of the RedHawk Linux operating system.

`key_t` is defined by a `typedef` in the **<bits/types.h>** header file to be an integral type (this header file is included internally by **<sys/types.h>**).

The integer returned from this function upon successful completion is the message queue identifier, `msqid`. Upon failure, the external variable **errno** is set to indicate the reason for failure, and **-1** (which is not a valid `msqid`) is returned.

As declared, the process calling the **msgget** system call must supply two arguments to be passed to the formal *key* and *msgflg* arguments.

A new `msqid` with an associated message queue and data structure is provided if either

- *key* is equal to **IPC_PRIVATE**,

or

- *key* is a unique integer and the control command **IPC_CREAT** is specified in the *msgflg* argument.

The value passed to the *msgflg* argument must be an integer-type value that will specify the following:

- operation permissions
- control fields (commands)

Operation permissions determine the operations that processes are permitted to perform on the associated message queue. “Read” permission is necessary for receiving messages or for determining queue status by means of a **msgctl IPC_STAT** operation. “Write” permission is necessary for sending messages.

Table 3-1 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 3-1. Message Queue Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific value is derived by adding or bitwise OR'ing the octal values for the operation permissions wanted. That is, if "read by user" and "read/write by others" is desired, the code value would be 00406 (00400 plus 00006).

Control flags are predefined constants (represented by all upper-case letters). The flags which apply to the **msgget** system call are **IPC_CREAT** and **IPC_EXCL** and are defined in the **<bits/ipc.h>** header file, which is internally included by the **<sys/ipc.h>** header file.

The value for *msgflg* is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by adding or bitwise OR'ing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *msgflg* value can easily be set by using the flag names in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));
msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **msgget(2)** man page, success or failure of this system call depends upon the argument values for *key* and *msgflg* and upon system wide resource limits.

The system call will attempt to return a new message queue identifier if one of the following conditions is true:

- *key* is equal to **IPC_PRIVATE**
- *key* does not already have a message queue identifier associated with it and (*msgflg* & **IPC_CREAT**) is "true" (not zero).

The system call will always be attempted. Exceeding the **MSGMNI** limit always causes a failure. The **MSGMNI** limit value determines the system-wide number of unique message queues that may be in use at any given time. This limit value is a fixed define value located in **<linux/msg.h>**.

IPC_EXCL is another control command used in conjunction with **IPC_CREAT**. It causes the system call to return an error if a message queue identifier already exists for the specified key. This is necessary to prevent the process from thinking it has received a new identifier when it has not. In other words, when both **IPC_CREAT** and **IPC_EXCL** are specified, a new message queue identifier is returned if the system call is successful.

Refer to the **msgget(2)** man page for specific associated data structure initialization, as well as the specific failure conditions and their error names.

Example Program

The example program presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **msgget(2)** system call to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values.

This program begins (lines 5-9) by including the required header files as specified by the **msgget(2)** man page. Note that the **<errno.h>** header file is additionally included for referencing the **errno** variable.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are intended to be self-explanatory. These names make the program more readable and are valid because they are local to the program.

The variables declared for this program and their roles are:

`key`	passes the value for the desired key
`opperm`	stores the desired operation permissions
`flags`	stores the desired control commands (flags)
`opperm_flags`	stores the combination from the logical ORing of the `opperm` and `flags` variables; it is then used in the system call to pass the `msgflg` argument
`msqid`	stores the message queue identification number for a successful system call or the error code (`-1`) for an unsuccessful one.

The program begins by prompting for a hexadecimal `key`, an octal operation permissions code, and finally for the control command combinations (`flags`) which are selected from a menu (lines 16-33). All possible combinations are allowed even though they might not be valid, in order to allow observing errors for invalid combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the `opperm_flags` variable (lines 37-52).

The system call is made next, and the result is stored in the `msqid` variable (line 54).

Because the `msqid` variable now contains a valid message queue identifier or the error code (`-1`), it can be tested to see if an error occurred (line 56). If `msqid` equals `-1`, a message indicates that an error resulted, and the external `errno` variable is displayed (line 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the **msgget** system call follows. This file is provided as **/usr/share/doc/ccur/examples/msgget.c**.

```
1   /*
2    * Illustrates the message get, msgget(),
3    * system service capabilities.
4    */
5   #include <stdio.h>
6   #include <sys/types.h>
7   #include <sys/ipc.h>
8   #include <sys/msg.h>
9   #include <errno.h>
10  /* Start of main C language program */
11  main()
12  {
13      key_t key;
14      int opperm, flags;
15      int msqid, opperm_flags;
16      /* Enter the desired key */
17      printf("Enter the desired key in hex = ");
18      scanf("%x", &key);
19      /* Enter the desired octal operation
20         permissions. */
21      printf("\nEnter the operation\n");
22      printf("permissions in octal = ");
23      scanf("%o", &opperm);
24      /* Set the desired flags. */
25      printf("\nEnter corresponding number to\n");
26      printf("set the desired flags:\n");
27      printf("No flags                  = 0\n");
28      printf("IPC_CREAT                 = 1\n");
29      printf("IPC_EXCL                  = 2\n");
30      printf("IPC_CREAT and IPC_EXCL    = 3\n");
31      printf("           Flags          = ");
32      /* Get the flag(s) to be set. */
33      scanf("%d", &flags);
34      /* Check the values. */
35      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
36          key, opperm, flags);
37      /* Incorporate the control fields (flags) with
38         the operation permissions */
39      switch (flags)
40      {
41      case 0:    /* No flags are to be set. */
42          opperm_flags = (opperm | 0);
43          break;
44      case 1:    /* Set the IPC_CREAT flag. */
45          opperm_flags = (opperm | IPC_CREAT);
46          break;
47      case 2:    /* Set the IPC_EXCL flag. */
48          opperm_flags = (opperm | IPC_EXCL);
49          break;
50      case 3:    /* Set the IPC_CREAT and IPC_EXCL flags. */
51          opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52      }
53      /* Call the msgget system call.*/
54      msqid = msgget (key, opperm_flags);
55      /* Perform the following if the call is unsuccessful. */
56      if(msqid == -1)
57      {
58          printf ("\nThe msgget call failed, error number = %d\n",
59              errno);
60      }
61      /* Return the msqid upon successful completion. */
62      else
63          printf ("\nThe msqid = %d\n", msqid);
64      exit(0);
65  }
```

Controlling Message Queues

This section describes how to use the **`msgctl(2)`** system call. The accompanying program illustrates its use.

Using msgctl

Synopsis

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

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

The **`msgctl`** system call returns an integer value, which is zero for successful completion or **`-1`** otherwise.

The *msqid* variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **`msgget`** system call.

The *cmd* argument can be any one of the following values:

IPC_STAT returns the status information contained in the associated data structure for the specified message queue identifier, and places it in the data structure pointed to by the *buf* pointer in the user memory area

IPC_SET writes the effective user and group identification, operation permissions, and the number of bytes for the message queue to the values contained in the data structure pointed to by the *buf* pointer in the user memory area

IPC_RMID removes the specified message queue identifier along with its associated message queue and data structure

To perform an **IPC_SET** or **IPC_RMID** control command, a process must meet one or more of the following conditions:

- have an effective user id of OWNER
- have an effective user id of CREATOR
- be the super-user
- have the **CAP_SYS_ADMIN** capability

Additionally, when performing an **IPC_SET** control command that increases the size of the `msg_qbytes` value beyond the value of **MSGMNB** (defined in **`<linux/msg.h>`**), the process must have the **CAP_SYS_RESOURCE** capability.

Note that a message queue can also be removed by using the **`ipcrm(8)`** command by specifying the "**`msg`** *id*" option, where *id* specifies the *msqid* for the message queue. To use this command, the user must have the same effective user id or capability that is required for performing an **IPC_RMID** control command. See the **`ipcrm(8)`** man page for additional information on the use of this command.

Read permission is required to perform the **IPC_STAT** control command.

The details of this system call are discussed in the following example program. If you need more information on the logic manipulations in this program, read the **msgget(2)** man page; it goes into more detail than would be practical for this document.

Example Program

The example program presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **msgctl** system call to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values.

This program begins (lines 6-11) by including the required header files as specified by the **msgctl(2)** man page. Note that the **<errno.h>** header file is additionally added for referencing the external **errno** variable.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations should be self-explanatory. These names make the program more readable and are valid because they are local to the program.

The variables declared for this program and their roles are:

`uid`	stores the `IPC_SET` value for the effective user identification
`gid`	stores the `IPC_SET` value for the effective group identification
`mode`	stores the `IPC_SET` value for the operation permissions
`bytes`	stores the `IPC_SET` value for the number of bytes in the message queue (`msg_qbytes`)
`rtrn`	stores the return integer value from the system call
`msqid`	stores and pass the message queue identifier to the system call
`command`	stores the code for the desired control command so that subsequent processing can be performed on it
`choice`	determines which member is to be changed for the `IPC_SET` control command
`msqid_ds`	receives the specified message queue identifier's data structure when an `IPC_STAT` control command is performed
`buf`	a pointer passed to the system call which locates the data structure in the user memory area where the `IPC_STAT` control command is to place its return values or where the `IPC_SET` command gets the values to set

Although the `buf` pointer is declared to be a pointer to a data structure of the `msqid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 18).

The program first prompts for a valid message queue identifier which is stored in the `msqid` variable (lines 20, 21). This is required for every **`msgctl`** system call.

Next, the code for the desired control command must be entered (lines 22-28) and stored in the command variable. The code is tested to determine the control command for subsequent processing.

If the `IPC_STAT` control command is selected (code 1), the system call is performed (lines 38, 39) and the status information returned is printed out (lines 40-47); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 107), the status information of the last successful call is printed out. In addition, an error message is displayed and the `errno` variable is printed out (line 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

If the `IPC_SET` control command is selected (code 2), the current status information for the message queue identifier specified must be obtained (lines 51-53). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. Next, the program prompts for a code corresponding to the member to be changed (lines 54-60). This code is stored in the `choice` variable (line 61). Now, depending upon the member picked, the program prompts for the new value (lines 67-96). The value is placed into the appropriate member in the user memory area data structure, and the system call is made (lines 97-99). Depending upon success or failure, the program returns the same messages as for `IPC_STAT` above.

If the `IPC_RMID` control command (code 3) is selected, the system call is performed (lines 101-104), and the `msqid` along with its associated message queue and data structure are removed from the RedHawk Linux operating system. Note that the `buf` pointer is ignored in performing this control command, and its value can be zero or `NULL`. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **msgctl** system call follows. This file is provided as **/usr/share/doc/ccur/examples/msgctl.c**.

```
/*
 * Illustrates the message control,
 * msgctl(),system service capabilities
 */

/* Include necessary header files. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
/* Start of main C language program */
main()
{
     int uid, gid, mode, bytes;
     int rtrn, msqid, command, choice;
     struct msqid_ds msqid_ds, *buf;
     buf = &msqid_ds;
     /* Get the msqid, and command. */
     printf("Enter the msqid = ");
     scanf("%d", &msqid);
     printf("\nEnter the number for\n");
     printf("the desired command:\n");
     printf("IPC_STAT    =  1\n");
     printf("IPC_SET     =  2\n");
     printf("IPC_RMID    =  3\n");
     printf("Entry       =  ");
     scanf("%d", &command);
     /* Check the values. */
     printf ("\nmsqid =%d, command = %d\n",
         msqid, command);
     switch (command)
     {
     case 1: /* Use msgctl() to duplicate
              the data structure for
              msqid in the msqid_ds area pointed
              to by buf and then print it out. */
         rtrn = msgctl(msqid, IPC_STAT,
              buf);
         printf ("\nThe USER ID = %d\n",
              buf->msg_perm.uid);
         printf ("The GROUP ID = %d\n",
              buf->msg_perm.gid);
         printf ("The operation permissions = 0%o\n",
              buf->msg_perm.mode);
         printf ("The msg_qbytes = %d\n",
              buf->msg_qbytes);
         break;
     case 2: /* Select and change the desired
              member(s) of the data structure. */
         /* Get the original data for this msqid
         data structure first. */
         rtrn = msgctl(msqid, IPC_STAT, buf);
         printf("\nEnter the number for the\n");
         printf("member to be changed:\n");
         printf("msg_perm.uid   = 1\n");
         printf("msg_perm.gid   = 2\n");
         printf("msg_perm.mode  = 3\n");
         printf("msg_qbytes     = 4\n");
         printf("Entry          = ");
         scanf("%d", &choice);
         /* Only one choice is allowed per
              pass as an invalid entry will
              cause repetitive failures until
              msqid_ds is updated with
              IPC_STAT. */
         switch(choice){
         case 1:
```

```
69                printf("\nEnter USER ID = ");
70                scanf ("%ld", &uid);
71                buf->msg_perm.uid =(uid_t)uid;
72                printf("\nUSER ID = %d\n",
73                     buf->msg_perm.uid);
74                break;
75           case 2:
76                printf("\nEnter GROUP ID = ");
77                scanf("%d", &gid);
78                buf->msg_perm.gid = gid;
79                printf("\nGROUP ID = %d\n",
80                     buf->msg_perm.gid);
81                break;
82           case 3:
83                printf("\nEnter MODE = ");
84                scanf("%o", &mode);
85                buf->msg_perm.mode = mode;
86                printf("\nMODE = 0%o\n",
87                     buf->msg_perm.mode);
88                break;
89           case 4:
90                printf("\nEnter msq_bytes = ");
91                scanf("%d", &bytes);
92                buf->msg_qbytes = bytes;
93                printf("\nmsg_qbytes = %d\n",
94                     buf->msg_qbytes);
95                break;
96           }
97           /* Do the change. */
98           rtrn = msgctl(msqid, IPC_SET,
99                buf);
100          break;
101     case 3: /* Remove the msqid along with its
102               associated message queue
103               and data structure. */
104          rtrn = msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
105     }
106     /* Perform the following if the call is unsuccessful. */
107     if(rtrn == -1)
108     {
109          printf("\nThe msgctl call failed, error number = %d\n", errno);
110     }
111     /* Return the msqid upon successful completion. */
112     else
113          printf ("\nMsgctl was successful for msqid = %d\n",
114               msqid);
115     exit (0);
116 }
```

Operations for Messages

This section describes how to use the **msgsnd** and **msgrcv** system calls. The accompanying program illustrates their use.

Using Message Operations: msgsnd and msgrcv

Synopsis

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

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

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

Sending a Message

The **msgsnd** system call returns an integer value, which is zero for successful completion or **-1** otherwise.

The *msqid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget** system call.

The *msgp* argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The *msgsz* argument specifies the length of the character array in the data structure pointed to by the *msgp* argument. This is the length of the message. The maximum size of this array is determined by the **MSGMAX** define, which is located in **<linux/msg.h>**.

The *msgflg* argument allows the blocking message operation to be performed if the **IPC_NOWAIT** flag is not set ((*msgflg* & **IPC_NOWAIT**)= = 0); the operation blocks if the total number of bytes allowed on the specified message queue are in use (`msg_qbytes`). If the **IPC_NOWAIT** flag is set, the system call fails and returns **-1**.

Further details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, see the section "Using msgget." It goes into more detail than would be practical for every system call.

Receiving Messages

When the **msgrcv** system call is successful, it returns the number of bytes received; when unsuccessful it returns **-1**.

The *msqid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget** system call.

The *msgp* argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The *msgsz* argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired (see the *msgflg* argument below).

The *msgtyp* argument is used to pick the first message on the message queue of the particular type specified:

- If *msgtyp* is equal to zero, the first message on the queue is received.
- If *msgtyp* is greater than zero and the **MSG_EXCEPT** *msgflg* is **not set**, the first message of the same type is received.
- If *msgtyp* is greater than zero and the **MSG_EXCEPT** *msgflg* is **set**, the first message on the message queue that is **not equal to** *msgtyp* is received.
- If *msgtyp* is less than zero, the lowest message type that is less than or equal to the absolute value of *msgtyp* is received.

The *msgflg* argument allows the blocking message operation to be performed if the **IPC_NOWAIT** flag is not set ((*msgflg* & **IPC_NOWAIT**) == 0); the operation blocks if the total number of bytes allowed on the specified message queue are in use (`msg_qbytes`). If the **IPC_NOWAIT** flag is set, the system call fails and returns a -1. And, as mentioned in the previous paragraph, when the **MSG_EXCEPT** flag is set in the *msgflg* argument and the *msgtyp* argument is greater than 0, the first message in the queue that has a message type that is different from the *msgtyp* argument is received.

If the **IPC_NOWAIT** flag is set, the system call fails immediately when there is not a message of the desired type on the queue. *msgflg* can also specify that the system call fail if the message is longer than the size to be received; this is done by not setting the **MSG_NOERROR** flag in the *msgflg* argument ((*msgflg* & **MSG_NOERROR**)) == 0). If the **MSG_NOERROR** flag is set, the message is truncated to the length specified by the *msgsz* argument of **msgrcv**.

Further details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, see the section “Using msgget.” It goes into more detail than would be practical for every system call.

Example Program

The example program presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **msgsnd** and **msgrcv** system calls to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values.

This program begins (lines 5-10) by including the required header files as specified by the **msgop(2)** man page.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are intended to be self-explanatory. These names make the program more readable and are valid because they are local to the program.

The variables declared for this program and their roles are:

sndbuf	a buffer which contains a message to be sent (line 14); it uses the msgbuf1 data structure as a template (lines 11-14). The msgbuf1 structure is a duplicate of the msgbuf structure contained in the **<sys/msg.h>** header file, except that the size of the character array for mtext is tailored to fit this application. The msgbuf structure should not be used directly because mtext has only one element that would limit the size of each message to one character. Instead, declare your own structure. It should be identical to msgbuf except that the size of the mtext array should fit your application.
rcvbuf	a buffer used to receive a message (line 14); it uses the msgbuf1 data structure as a template (lines 11-14)
msgp	a pointer (line 14) to both the sndbuf and rcvbuf buffers
i	a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd** system call; it also serves as a counter to output the received message for the **msgrcv** system call
c	receives the input character from the getchar function (line 50)
flag	stores the code of IPC_NOWAIT for the **msgsnd** system call (line 61)
flags	stores the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv** system call (line 117)
choice	stores the code for sending or receiving (line 30)
rtrn	stores the return values from all system calls
msqid	stores and passes the desired message queue identifier for both system calls
msgsz	stores and passes the size of the message to be sent or received
msgflg	passes the value of flag for sending or the value of flags for receiving
msgtyp	specifies the message type for sending or for picking a message type for receiving

Note that a msqid_ds data structure is set up in the program (line 21) with a pointer initialized to point to it (line 22); this allows the data structure members affected by message operations to be observed. They are observed by using the **msgctl** (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 160-167).

The program first prompts for whether to send or receive a message. A corresponding code must be entered for the desired operation; it is stored in the `choice` variable (lines 23-30). Depending upon the code, the program proceeds as in the following **`msgsnd`** or **`msgrcv`** sections.

msgsnd

When the code is to send a message, the `msgp` pointer is initialized (line 33) to the address of the send data structure, `sndbuf`. Next, a message type must be entered for the message; it is stored in the variable `msgtyp` (line 42), and then (line 43) it is put into the `mtype` member of the data structure pointed to by `msgp`.

The program then prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the `mtext` array of the data structure (lines 48-51). This will continue until an end-of-file is recognized which, for the `getchar` function, is a CTRL-d immediately following a carriage return (RETURN).

The message is immediately echoed from the `mtext` array of the `sndbuf` data structure to provide feedback (lines 54-56).

The next and final decision is whether to set the `IPC_NOWAIT` flag. The program does this by requesting that a code of `1` be entered for yes or anything else for no (lines 57-65). It is stored in the flag variable. If `1` is entered, `IPC_NOWAIT` is logically ORed with `msgflg`; otherwise, `msgflg` is set to zero.

The **`msgsnd`** system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed and should be zero (lines 73-76).

Every time a message is successfully sent, three members of the associated data structure are updated. They are:

`msg_qnum`	represents the total number of messages on the message queue; it is incremented by one
`msg_lspid`	contains the process identification (PID) number of the last process sending a message; it is set accordingly
`msg_stime`	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly

These members are displayed after every successful message send operation (lines 79-92).

msgrcv

When the code is to receive a message, the program continues execution as in the following paragraphs.

The `msgp` pointer is initialized to the `rcvbuf` data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested; it is stored in `msqid` (lines 100-103).

The message type is requested; it is stored in `msgtyp` (lines 104-107).

The code for the desired combination of control flags is requested next; it is stored in flags (lines 108-117). Depending upon the selected combination, `msgflg` is set accordingly (lines 118-131).

Finally, the number of bytes to be received is requested; it is stored in `msgsz` (lines 132-135).

The **`msgrcv`** system call is performed (line 142). If it is unsuccessful, a message and error number is displayed (lines 143-145). If successful, a message indicates so, and the number of bytes returned and the `msg` type returned (because the value returned may be different from the value requested) is displayed followed by the received message (lines 150-156).

When a message is successfully received, three members of the associated data structure are updated. They are:

`msg_qnum`	contains the number of messages on the message queue; it is decremented by one
`msg_lrpid`	contains the PID of the last process receiving a message; it is set accordingly
`msg_rtime`	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly

The sample code for the **`msgop`** system call follows. This file is provided as **`/usr/share/doc/ccur/examples/msgop.c`**.

```
1   /*
2    * Illustrates the message operations,
3    * msgop(), system service capabilities.
4    */
5   /* Include necessary header files. */
6   #include <stdio.h>
7   #include <sys/types.h>
8   #include <sys/ipc.h>
9   #include <sys/msg.h>
10  #include <errno.h>
11  struct msgbuf1 {
12       long    mtype;
13       char    mtext[8192];
14  } sndbuf, rcvbuf, *msgp;
15  /* Start of main C language program */
16  main()
17  {
18       int i, c, flag, flags, choice;
19       int rtrn, msqid, msgsz, msgflg;
20       long mtype, msgtyp;
21       struct msqid_ds msqid_ds, *buf;
22       buf = &msqid_ds;
23       /* Select the desired operation. */
24       printf("Enter the corresponding\n");
25       printf("code to send or\n");
26       printf("receive a message:\n");
27       printf("Send            =  1\n");
28       printf("Receive         =  2\n");
29       printf("Entry           =  ");
30       scanf("%d", &choice);
31       if(choice == 1) /* Send a message. */
32       {
33           msgp = &sndbuf; /*Point to user send structure. */
34           printf("\nEnter the msqid of\n");
35           printf("the message queue to\n");
36           printf("handle the message = ");
```

```
37            scanf("%d", &msqid);
38            /* Set the message type. */
39            printf("\nEnter a positive integer\n");
40            printf("message type (long) for the\n");
41            printf("message = ");
42            scanf("%ld", &msgtyp);
43            msgp->mtype = msgtyp;
44            /* Enter the message to send. */
45            printf("\nEnter a message: \n");
46            /* A  control-d (^d) terminates as
47               EOF. */
48            /* Get each character of the message
49               and put it in the mtext array. */
50            for(i = 0; ((c = getchar()) != EOF); i++)
51                sndbuf.mtext[i] = c;
52            /* Determine the message size. */
53            msgsz = i;
54            /* Echo the message to send. */
55            for(i = 0; i < msgsz; i++)
56                putchar(sndbuf.mtext[i]);
57            /* Set the IPC_NOWAIT flag if
58               desired. */
59            printf("\nEnter a 1 if you want \n");
60            printf("the IPC_NOWAIT flag set:  ");
61            scanf("%d", &flag);
62            if(flag == 1)
63                msgflg = IPC_NOWAIT;
64            else
65                msgflg = 0;
66            /* Check the msgflg. */
67            printf("\nmsgflg = 0%o\n", msgflg);
68            /* Send the message. */
69            rtrn = msgsnd(msqid, (const void*) msgp, msgsz, msgflg);
70            if(rtrn == -1)
71                printf("\nMsgsnd failed.  Error = %d\n",
72                    errno);
73            else {
74                /* Print the value of test which
75                   should be zero for successful. */
76                printf("\nValue returned = %d\n", rtrn);
77                /* Print the size of the message
78                   sent. */
79                printf("\nMsgsz = %d\n", msgsz);
80                /* Check the data structure update. */
81                msgctl(msqid, IPC_STAT, buf);
82                /* Print out the affected members. */
83                /* Print the incremented number of
84                   messages on the queue. */
85                printf("\nThe msg_qnum = %d\n",
86                    buf->msg_qnum);
87                /* Print the process id of the last sender. */
88                printf("The msg_lspid = %d\n",
89                    buf->msg_lspid);
90                /* Print the last send time. */
91                printf("The msg_stime = %d\n",
92                    buf->msg_stime);
93            }
94        }
95        if(choice == 2)  /*Receive a message. */
96        {
97            /* Initialize the message pointer
98               to the receive buffer. */
99            msgp = &rcvbuf;
100           /* Specify the message queue which contains
101              the desired message. */
102           printf("\nEnter the msqid = ");
103           scanf("%d", &msqid);
104           /* Specify the specific message on the queue
105              by using its type. */
106           printf("\nEnter the msgtyp = ");
107           scanf("%ld", &msgtyp);
108           /* Configure the control flags for the
```

```
109             desired actions. */
110          printf("\nEnter the corresponding code\n");
111          printf("to select the desired flags: \n");
112          printf("No flags                    =  0\n");
113          printf("MSG_NOERROR                 =  1\n");
114          printf("IPC_NOWAIT                  =  2\n");
115          printf("MSG_NOERROR and IPC_NOWAIT  =  3\n");
116          printf("                Flags       =  ");
117          scanf("%d", &flags);
118          switch(flags) {
119          case 0:
120               msgflg = 0;
121               break;
122          case 1:
123               msgflg = MSG_NOERROR;
124               break;
125          case 2:
126               msgflg = IPC_NOWAIT;
127               break;
128          case 3:
129               msgflg = MSG_NOERROR | IPC_NOWAIT;
130               break;
131          }
132          /* Specify the number of bytes to receive. */
133          printf("\nEnter the number of bytes\n");
134          printf("to receive (msgsz) = ");
135          scanf("%d", &msgsz);
136          /* Check the values for the arguments. */
137          printf("\nmsqid =%d\n", msqid);
138          printf("\nmsgtyp = %ld\n", msgtyp);
139          printf("\nmsgsz = %d\n", msgsz);
140          printf("\nmsgflg = 0%o\n", msgflg);
141          /* Call msgrcv to receive the message. */
142          rtrn = msgrcv(msqid, (void*) msgp, msgsz, msgtyp,msgflg);
143          if(rtrn == -1)  {
144               printf("\nMsgrcv failed., Error = %d\n", errno);
145          }
146          else {
147               printf ("\nMsgctl was successful\n");
148               printf("for msqid = %d\n",
149                    msqid);
150               /* Print the number of bytes received,
151                  it is equal to the return
152                  value. */
153               printf("Bytes received = %d\n", rtrn);
154               /* Print the received message. */
155               for(i = 0; i<rtrn; i++)
156                    putchar(rcvbuf.mtext[i]);
157          }
158          /* Check the associated data structure. */
159          msgctl(msqid, IPC_STAT, buf);
160          /* Print the decremented number of messages. */
161          printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
162          /* Print the process id of the last receiver. */
163          printf("The msg_lrpid = %d\n", buf->msg_lrpid);
164          /* Print the last message receive time */
165          printf("The msg_rtime = %d\n", buf->msg_rtime);
166      }
167  }
```

4
Process Scheduling

4
Process Scheduling

This chapter provides an overview of process scheduling on RedHawk Linux systems. It explains how the process scheduler decides which process to execute next and describes POSIX scheduling policies and priorities. It explains the procedures for using the program interfaces and commands that support process scheduling and discusses performance issues.

Overview

In the RedHawk Linux OS, the schedulable entity is always a process. Scheduling priorities and scheduling policies are attributes of processes. The system scheduler determines when processes run. It maintains priorities based on configuration parameters, process behavior, and user requests; it uses these priorities as well as the CPU affinity to assign processes to a CPU.

The scheduler offers three different scheduling policies, one for normal non-critical processes (**SCHED_OTHER)**, and two fixed-priority policies for real-time applications (**SCHED_FIFO** and **SCHED_RR**). These policies are explained in detail in the section “Scheduling Policies” on page 4-3.

By default, the scheduler uses the **SCHED_OTHER** time-sharing scheduling policy. For processes in the **SCHED_OTHER** policy, the scheduler manipulates the priority of runnable processes dynamically in an attempt to provide good response time to interactive processes and good throughput to CPU-intensive processes.

Fixed-priority scheduling allows users to set static priorities on a per-process basis. The scheduler never modifies the priority of a process that uses one of the fixed priority scheduling policies. The highest fixed-priority process always gets the CPU as soon as it is runnable, even if other processes are runnable. An application can therefore specify the exact order in which processes run by setting process priority accordingly.

For system environments in which real-time performance is not required, the default scheduler configuration works well, and no fixed-priority processes are needed. However, for real-time applications or applications with strict timing constraints, fixed-priority processes are the only way to guarantee that the critical application's requirements are met. When certain programs require very deterministic response times, fixed priority scheduling policies should be used and tasks that require the most deterministic response should be assigned the most favorable priorities.

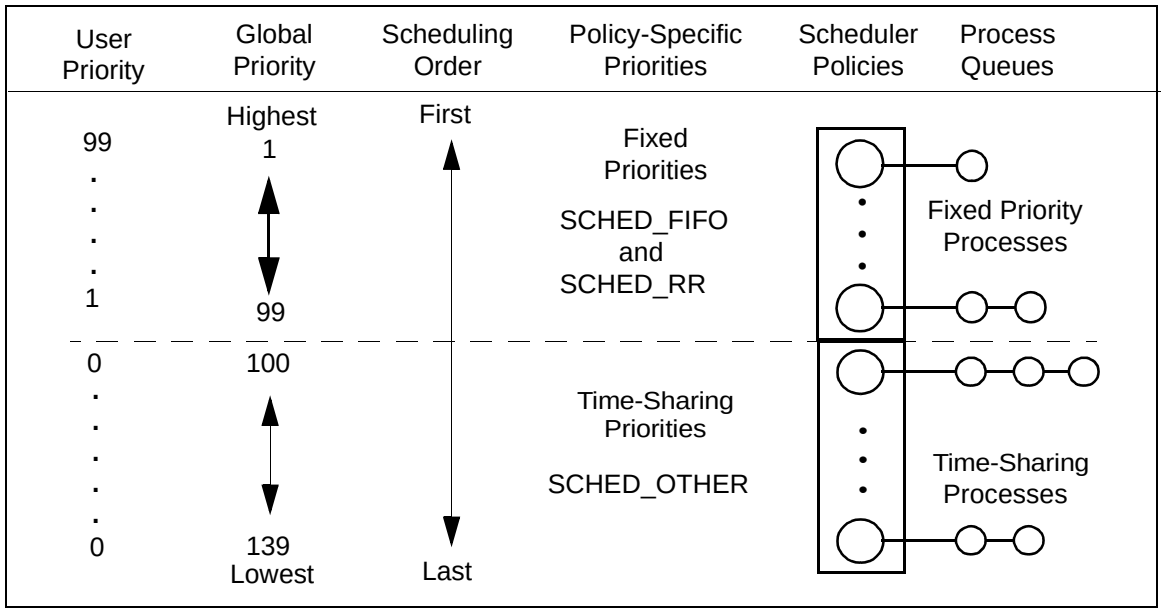
A set of system calls based on IEEE Standard 1003.1b provides direct access to a process’ scheduling policy and priority. Included in the set are system calls that allow processes to obtain or set a process’ scheduling policy and priority; obtain the minimum and maximum priorities associated with a particular scheduling policy; and obtain the time quantum associated with a process scheduled under the round robin (**SCHED_RR**) scheduling policy. You may alter the scheduling policy and priority for a process at the command level by

using the **run(1)** command. The system calls and the **run** command are detailed later in this chapter along with procedures and hints for effective use.

How the Process Scheduler Works

Figure 4-1 illustrates how the scheduler operates.

Figure 4-1. The RedHawk Linux Scheduler



When a process is created, it inherits its scheduling parameters, including scheduling policy and a priority within that policy. Under the default configuration, a process begins as a time-sharing process scheduled with the **SCHED_OTHER** policy. In order for a process to be scheduled under a fixed-priority policy, a user-request must be made via system calls or the **run(1)** command.

When the user sets the priority of a process, he is setting the "user priority." This is also the priority that will be reported by the **sched_getparam(2)** call when a user retrieves the current priority. A portable application should use the **sched_get_priority_min()** and **sched_get_priority_max()** interfaces to determine the range of valid priorities for a particular scheduling policy. A user priority value (`sched_priority`) is assigned to each process. **SCHED_OTHER** processes can only be assigned a user priority of 0. **SCHED_FIFO** and **SCHED_RR** processes have a user priority in the range 1 to 99 by default. This range can be modified with CONFIG_MAX_USER_RT_PRIO kernel tunable accessible through the General Setup selection of the Linux Kernel Configuration menu (see the "Configuring and Building the Kernel" chapter of this guide).

The scheduler converts scheduling policy-specific priorities into global priorities. The global priority is the scheduling policy value used internally by the RedHawk Linux kernel. The scheduler maintains a list of runnable processes for each possible global priority value. There are 40 global scheduling priorities associated with the **SCHED_OTHER**

scheduling policy; there are 99 global scheduling priorities associated with the fixed priority scheduling policies (**SCHED_RR** and **SCHED_FIFO**). The scheduler looks for the non-empty list with the highest global priority and selects the process at the head of this list for execution on the current CPU. The scheduling policy determines for each process where it will be inserted into the list of processes with equal user priority and the process' relative position in this list when processes in the list are blocked or made runnable.

As long as a fixed-priority process is ready-to-run for a particular CPU, no time-sharing process will run on that CPU.

Once the scheduler assigns a process to the CPU, the process runs until it uses up its time quantum, sleeps, blocks or is preempted by a higher-priority process.

Note that the priorities displayed by **`ps(1)`** and **`top(1)`** are internally computed values and only indirectly reflect the priority set by the user.

Scheduling Policies

POSIX defines three types of scheduling policies that control the way a process is scheduled:

SCHED_FIFO	first-in-first-out (FIFO) scheduling policy
SCHED_RR	round-robin (RR) scheduling policy
SCHED_OTHER	default time-sharing scheduling policy

First-In-First-Out Scheduling (SCHED_FIFO)

SCHED_FIFO can only be used with user priorities higher than 0. That means when a **SCHED_FIFO** process becomes runnable, it will always immediately preempt any currently running **SCHED_OTHER** process. **SCHED_FIFO** is a simple scheduling algorithm without time slicing. For processes scheduled under the **SCHED_FIFO** policy, the following rules are applied: A **SCHED_FIFO** process that has been preempted by another process of higher priority will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. When a **SCHED_FIFO** process becomes runnable, it will be inserted at the end of the list for its priority. A call to **`sched_setscheduler(2)`** or **`sched_setparam(2)`** will put the **SCHED_FIFO** process identified by pid at the end of the list if it was runnable. A process calling **`sched_yield(2)`** will be put at the end of its priority list. No other events will move a process scheduled under the **SCHED_FIFO** policy in the wait list of runnable processes with equal user priority. A **SCHED_FIFO** process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls **`sched_yield`**.

Round-Robin Scheduling (SCHED_RR)

SCHED_RR is a simple enhancement of **SCHED_FIFO**. Everything described above for **SCHED_FIFO** also applies to **SCHED_RR**, except that each process is only allowed to run for a maximum time quantum. If a **SCHED_RR** process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A **SCHED_RR** process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved by **sched_rr_get_interval(2)**. The length of the time quantum is affected by the nice value associated with a process scheduled under the **SCHED_RR** scheduling policy. Higher nice values are assigned larger time quantums.

Time-Sharing Scheduling (SCHED_OTHER)

SCHED_OTHER can only be used at user priority 0. **SCHED_OTHER** is the default universal time-sharing scheduler policy that is intended for all processes that do not require special user priority real-time mechanisms. The process to run is chosen from the user priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level (set by the **nice(2)** or **setpriority(2)** system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all **SCHED_OTHER** processes. Other factors, such as the number of times a process voluntarily blocks itself by performing an I/O operation, also come into consideration.

Procedures for Enhanced Performance

How to Set Priorities

The following code segment will place the current process into the **SCHED_RR** fixed-priority scheduling policy at a fixed priority of 60. See the section "Process Scheduling Interfaces" later in this chapter for information about the POSIX scheduling routines.

```
#include <sched.h>
...
struct sched_param sparms;

sparms.sched_priority = 60;
if (sched_setscheduler(0, SCHED_RR, &sparms) < 0)
{
    perror("sched_setsched");
        exit(1);
}
```

Bottom Half Interrupt Routines

Processes scheduled in one of the fixed-priority scheduling policies will be assigned a higher priority than the processing associated with bottom half interrupt routines (this includes softirqs and tasklets). These bottom half interrupt routines perform work on behalf of interrupt routines that have executed on a given CPU. The real interrupt routine runs at a hardware interrupt level and preempts all activity on a CPU (including processes scheduled under one of the fixed-priority scheduling policies). Device driver writers under Linux are encouraged to perform the minimum amount of work required to interact with a device to make the device believe that the interrupt has been handled. The device driver can then raise one of the bottom half interrupt mechanisms to handle the remainder of the work associated with the device interrupt routine. Because fixed-priority processes run at a priority above bottom half interrupt routines, this interrupt architecture allows fixed-priority processes to experience the minimum amount of jitter possible from interrupt routines. For more information about interrupt routines in device drivers, see the "Device Drivers and Real Time" chapter.

SCHED_FIFO vs SCHED_RR

The two fixed priority scheduling policies are very similar in their nature, and under most conditions they will behave in an identical manner. It is important to remember that while **SCHED_RR** has a time quantum associated with the process, when that time quantum expires the process will only yield the CPU if there currently is a ready-to-run process of equal priority in one of the fixed priority scheduling policies. If there is no ready-to-run process of equal priority, the scheduler will determine that the original **SCHED_RR** process is still the highest priority process ready to run on this CPU and the same process will again be selected for execution.

This means that the only time there is a difference between processes scheduled under **SCHED_FIFO** and **SCHED_RR** is when there are multiple processes running under one of the fixed-priority scheduling policies scheduled at the exact same scheduling priority. In this case, **SCHED_RR** will allow these processes to share a CPU according to the time quantum that has been assigned to the process. Note that a process' time quantum is affected by the **`nice(2)`** system call. Processes with higher nice values will be assigned a larger time quantum. A process' time quantum can also be changed via the **`run(1)`** command (see "The run Command" later in this chapter for details).

Access to Lower Priority Processes

A non-blocking endless loop in a process scheduled under the **SCHED_FIFO** and **SCHED_RR** scheduling policies will block all processes with lower priority indefinitely. As this scenario can starve the CPU of other processes completely, precautions should be taken to avoid this.

During software development, a programmer can break such an endless loop by keeping available on the console a shell scheduled under a higher user priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected. As **SCHED_FIFO** and **SCHED_RR** processes can preempt other processes forever, only root processes or processes with the **CAP_SYS_NICE** capability are allowed to activate these policies.

Memory Locking

Paging and swapping often add an unpredictable amount of system overhead time to application programs. To eliminate performance losses due to paging and swapping, use the **mlockall(2)**, **munlockall(2)**, **mlock(2)** and **munlock(2)** system calls to lock all or a portion of a process' virtual address space in physical memory.

CPU Affinity and Shielded Processors

Each process in the system has a CPU affinity mask. The CPU affinity mask determines on which CPUs the process is allowed to execute. When a CPU is shielded from processes, that CPU will only run processes that have explicitly set their CPU affinity to a set of CPUs that only includes shielded CPUs. Utilizing these techniques adds additional control to where and how a process executes. See the "Real-Time Performance" chapter of this guide for more information.

Process Scheduling Interfaces

A set of system calls based on IEEE Standard 1003.1b provides direct access to a process' scheduling policy and priority. You may alter the scheduling policy and priority for a process at the command level by using the **run(1)** command. The system calls are detailed below. The **run** command is detailed on page 4-13.

POSIX Scheduling Routines

The sections that follow explain the procedures for using the POSIX scheduling system calls. These system calls are briefly described as follows:

Scheduling Policy:

sched_setscheduler	set a process' scheduling policy and priority
sched_getscheduler	obtain a process' scheduling policy

Scheduling Priority:

sched_setparam	change a process' scheduling priority
sched_getparam	obtain a process' scheduling priority

Relinquish CPU:

sched_yield	relinquish the CPU

Low/High Priority:

sched_get_priority_min obtain the lowest priority associated with a scheduling policy

sched_get_priority_max obtain the highest priority associated with a scheduling policy

Round-Robin Policy:

sched_rr_get_interval obtain the time quantum associated with a process scheduled under the **SCHED_RR** scheduling policy

The sched_setscheduler Routine

The **sched_setscheduler(2)** system call allows you to set the scheduling policy and the associated parameters for the process.

It is important to note that to use the **sched_setscheduler** call to (1) change a process' scheduling policy to the **SCHED_FIFO** or the **SCHED_RR** policy or (2) change the priority of a process scheduled under the **SCHED_FIFO** or the **SCHED_RR** policy, one of the following conditions must be met:

- The calling process must have root capability.
- The effective user ID (uid) of the calling process must match the effective user ID of the target process (the process for which the scheduling policy and priority are being set), or the calling process must have superuser or **CAP_SYS_NICE** capability.

Synopsis

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);

    struct sched_param {
        ...
        int sched_priority;
        ...
};
```

The arguments are defined as follows:

pid the process identification number (PID) of the process for which the scheduling policy and priority are being set. To specify the current process, set the value of *pid* to zero.

policy a scheduling policy as defined in the file <**sched.h**>. The value of *policy* must be one of the following:

SCHED_FIFO	first-in-first-out (FIFO) scheduling policy
SCHED_RR	round-robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

p a pointer to a structure that specifies the scheduling priority of the process identified by *pid*. The priority is an integer value that lies in the range of priorities defined for the scheduler class associated with the specified policy. You can determine the range of priorities associated with that policy by invoking one of the following system calls: **sched_get_priority_min** or **sched_get_priority_max** (for an explanation of these system calls, see page 4-11).

If the scheduling policy and priority of the specified process are successfully set, the **sched_setscheduler** system call returns the process' previous scheduling policy. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sched_setscheduler(2)** man page for a listing of the types of errors that may occur. If an error occurs, the process' scheduling policy and priority are not changed.

It is important to note that when you change a process' scheduling policy, you also change its time quantum to the default time quantum that is defined for the scheduler associated with the new policy and the priority. You can change the time quantum for a process scheduled under the **SCHED_RR** scheduling policy at the command level by using the **run(1)** command (see p. 4-13 for information on this command).

The sched_getscheduler Routine

The **sched_getscheduler(2)** system call allows you to obtain the scheduling policy for a specified process. Scheduling policies are defined in the file <**sched.h**> as follows:

SCHED_FIFO	first-in-first-out (FIFO) scheduling policy
SCHED_RR	round-robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

Synopsis

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

The argument is defined as follows:

pid the process identification number (PID) of the process for which you wish to obtain the scheduling policy. To specify the current process, set the value of *pid* to zero.

If the call is successful, **sched_getscheduler** returns the scheduling policy of the specified process. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sched_getscheduler(2)** man page for a listing of the types of errors that may occur.

The sched_setparam Routine

The **sched_setparam(2)** system call allows you to set the scheduling parameters associated with the scheduling policy of a specified process.

It is important to note that to use the **sched_setparam** call to change the scheduling priority of a process scheduled under the **SCHED_FIFO** or the **SCHED_RR** policy, one of the following conditions must be met:

- The calling process must have the root capability.
- The effective user ID (euid) of the calling process must match the effective user ID of the target process (the process for which the scheduling policy and priority are being set), or the calling process must have superuser or **CAP_SYS_NICE** capability.

Synopsis

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *p);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

The arguments are defined as follows:

pid — the process identification number (PID) of the process for which the scheduling priority is being changed. To specify the current process, set the value of *pid* to zero.

p — a pointer to a structure that specifies the scheduling priority of the process identified by *pid*. The priority is an integer value that lies in the range of priorities associated with the process' current scheduling policy. High numbers represent more favorable priorities and scheduling.

You can obtain a process' scheduling policy by invoking the **sched_getscheduler(2)** system call (see p. 4-7 for an explanation of this system call). You can determine the range of priorities associated with that policy by invoking the **sched_get_priority_min(2)** and **sched_get_priority_max(2)** system calls (see page 4-11 for explanations of these system calls).

A return value of **0** indicates that the scheduling priority of the specified process has been successfully changed. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sched_setparam(2)** man page for a listing of the types of errors that may occur. If an error occurs, the process' scheduling priority is not changed.

The sched_getparam Routine

The **sched_getparam(2)** system call allows you to obtain the scheduling parameters of a specified process.

Synopsis

```
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *p);

struct sched_param {
    ...
    int sched_priority;
    ...
};
```

The arguments are defined as follows:

pid the process identification number (PID) of the process for which you wish to obtain the scheduling priority. To specify the current process, set the value of *pid* to zero.

p a pointer to a structure to which the scheduling priority of the process identified by *pid* will be returned.

A return value of **0** indicates that the call to **sched_getparam** has been successful. The scheduling priority of the specified process is returned in the structure to which *p* points. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sched_getparam(2)** man page for a listing of the types of errors that may occur.

The sched_yield Routine

The **sched_yield(2)** system call allows the calling process to relinquish the CPU until it again becomes the highest priority process that is ready to run. Note that a call to **sched_yield** is effective only if a process whose priority is equal to that of the calling process is ready to run. This system call cannot be used to allow a process whose priority is lower than that of the calling process to execute.

Synopsis

```
#include <sched.h>

int sched_yield(void);
```

A return value of **0** indicates that the call to **sched_yield** has been successful. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error.

The sched_get_priority_min Routine

The **sched_get_priority_min(2)** system call allows you to obtain the lowest (least favorable) priority associated with a specified scheduling policy.

Synopsis

```
#include <sched.h>

int sched_get_priority_min(int policy);
```

The argument is defined as follows:

policy a scheduling policy as defined in the file <**sched.h**>. The value of *policy* must be one of the following:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. The value returned by **sched_get_priority_max** will be greater than the value returned by **sched_get_priority_min**.

RedHawk Linux by default allows the user priority value range 1 to 99 for **SCHED_FIFO** and **SCHED_RR** and the priority 0 for **SCHED_OTHER**. Scheduling priority ranges for the **SCHED_FIFO** and **SCHED_RR** scheduling policies can be altered using the CONFIG_MAX_USER_RT_PRIO kernel parameter accessible through the General Setup selection of the Linux Kernel Configuration menu.

If the call is successful, **sched_get_priority_min** returns the lowest priority associated with the specified scheduling policy. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the man page for **sched_get_priority_max(2)** to obtain a listing of the errors that may occur.

The sched_get_priority_max Routine

The **sched_get_priority_max(2)** system call allows you to obtain the highest (most favorable) priority associated with a specified scheduling policy.

Synopsis

```
#include <sched.h>

int sched_get_priority_max(int policy);
```

The argument is defined as follows:

policy a scheduling policy as defined in the file <**sched.h**>. The value of *policy* must be one of the following:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. The value returned by **sched_get_priority_max** will be greater than the value returned by **sched_get_priority_min**.

RedHawk Linux by default allows the user priority value range 1 to 99 for **SCHED_FIFO** and **SCHED_RR** and the priority 0 for **SCHED_OTHER**. Scheduling priority ranges for the **SCHED_FIFO** and **SCHED_RR** scheduling policies can be altered using the CONFIG_MAX_USER_RT_PRIO kernel parameter accessible through the General Setup selection of the Linux Kernel Configuration menu.

If the call is successful, **sched_get_priority_max** returns the highest priority associated with the specified scheduling policy. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. For a listing of the types of errors that may occur, refer to the **sched_get_priority_max(2)** man page.

The sched_rr_get_interval Routine

The **sched_rr_get_interval(2)** system call allows you to obtain the time quantum for a process that is scheduled under the **SCHED_RR** scheduling policy. The time quantum is the fixed period of time for which the kernel allocates the CPU to a process. When the process to which the CPU has been allocated has been running for its time quantum, a scheduling decision is made. If another process of the same priority is ready to run, that process will be scheduled. If not, the other process will continue to run.

Synopsis

```
include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *tp);

struct timespec {
    time_t  tv_sec;     /* seconds */
    long    tv_nsec;    /* nanoseconds */
};
```

The arguments are defined as follows:

pid — the process identification number (PID) of the process for which you wish to obtain the time quantum. To specify the current process, set the value of *pid* to zero.

tp — a pointer to a timespec structure to which the round robin time quantum of the process identified by *pid* will be returned. The identified process should be running under the **SCHED_RR** scheduling policy.

A return value of **0** indicates that the call to **sched_rr_get_interval** has been successful. The time quantum of the specified process is returned in the structure to which *tp* points. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sched_rr_get_interval(2)** man page for a listing of the types of errors that may occur.

The run Command

The **`run(1)`** command can be used to control process scheduler attributes and CPU affinity. The command syntax is:

`run` [*options*] {*command* [*args*] | *process_specifier*}

The **`run`** command executes the specified command in the environment described by the list of options and exits with the command's exit value. If a process specifier is given, **`run`** modifies the environment of the set of processes selected by the specifier. The process specifiers are defined below. A command may not be combined with a process specifier on the same command line invocation.

The **`run`** command allows you to run a program under a specified POSIX scheduling policy and at a specified priority (see p. 4-3 for a complete explanation of POSIX scheduling policies). It also allows you to set the time quantum for a program scheduled under the **SCHED_RR** policy.

To set a program's scheduling policy and priority, invoke the **`run`** command from the shell, and specify either the **--policy=***policy* or **–s** *policy* option and the **--priority=***priority* or **-P** *priority* option. Valid keywords for *policy* are:

SCHED_FIFO or **fifo**	for first-in-first-out scheduling
SCHED_RR or **rr**	for round robin scheduling, and
SCHED_OTHER or **other**	for timeshare scheduling.

The value of *priority* must be an integer value that is valid for the specified scheduling policy (or the current scheduling policy if the **-s** option is not used). Higher numerical values represent more favorable scheduling priorities.

To set the time quantum for a program being scheduled under the **SCHED_RR** scheduling policy, also specify the **--quantum=***quantum* or **-q** *quantum* option. *quantum* is specified as a nice value between -20 and 19 inclusive, with -20 being the longest slice of time and 19 being the shortest, or as a millisecond value corresponding to a nice value. **--quantum=list** displays the nice values and their equivalent millisecond values.

You can set the CPU affinity using the **--bias=***list* or **-b** *list* option. *list* is a comma-separated list of logical CPU numbers or ranges, for example: "0,2-4,6". *list* may also be specified as the string **active** or **boot** to specify all active processors or the boot processor, respectively. The **CAP_SYS_NICE** capability is required to add additional CPUs to an affinity.

The **--negate** or **-N** option negates the CPU bias list. A bias list option must also be specified when the negate option is specified. The bias used will contain all CPUs on the system except those specified in the bias list.

The **--copies=***count* or **-c** *count* option enables the user to run the specified number of identical copies of the command.

Other options are available for displaying information and running the command in the background. See the **`run(1)`** man page for more information.

PROCESS SPECIFIER

Only one of the following process specifiers may be specified. Multiple comma separated values can be specified for all *list*s and ranges are allowed where appropriate.

--pid=*list*, **-p** *list*	Specify a list of existing PIDs to modify.
--group=*list*, **-g** *list*	Specify a list of process groups to modify; all existing processes in the process groups listed will be modified.
--user=*list*, **-u** *list*	Specify a list of users to modify; all existing processes owned by the users listed will be modified. Each user in the list may either be a valid numeric user ID or character login ID.
--name=*list*, **-n** *list*	Specify a list of existing process names to modify.

EXAMPLES

1. The following command runs **make(1)** in the background on any of CPUs 0-3 under the default **SCHED_OTHER** scheduling policy with default priority.

```
run --bias=0-3 make &
```

2. The following command runs **date(1)** with a priority of 10 in the **SCHED_RR** (i.e. Round Robin) scheduling policy.

```
run -s SCHED_RR -P 10 date
```

3. The following command changes the scheduling priority of process ID 987 to level 32.

```
run --priority=32 -p 987
```

4. The following command moves all processes in process group 1456 to CPU 3.

```
run -b 3 -g 1456
```

5. The following command sets all processes whose name is "pilot" to run in the **SCHED_FIFO** scheduling policy with a priority of 21.

```
run -s fifo -P 21 -n pilot
```

Refer to the **run(1)** man page for additional information.

5
Interprocess Synchronization

5
Interprocess Synchronization

This chapter describes the tools that RedHawk Linux provides to meet a variety of interprocess synchronization needs. All of the interfaces described here provide the means for cooperating processes to synchronize access to shared resources.

The most efficient mechanism for synchronizing access to shared data by multiple programs in a multiprocessor system is by using spin locks. However, it is not safe to use a spin lock from user level without also using a rescheduling variable to protect against preemption while holding a spin lock.

If portability is a larger concern than efficiency, then POSIX counting semaphores are the next best choice for synchronizing access to shared data. In addition, the System V semaphores are provided, which allow processes to communicate through the exchange of semaphore values. Since many applications require the use of more than one semaphore, this facility allows you to create sets or arrays of semaphores.

Problems associated with synchronizing cooperating processes' access to data in shared memory are discussed as well as the tools that have been developed by Concurrent to provide solutions to these problems.

Understanding Interprocess Synchronization

Multiprocess real-time applications require synchronization mechanisms that allow cooperating processes to coordinate access to the same set of resources—for example, a number of I/O buffers, units of a hardware device, or a critical section of code.

RedHawk Linux supplies a variety of interprocess synchronization tools. These include tools for controlling a process' vulnerability to rescheduling, serializing processes' access to critical sections with busy-wait mutual exclusion mechanisms, semaphores for mutual exclusion to critical sections and coordinating interaction among processes.

Application programs that consist of two or more processes sharing portions of their virtual address space through use of shared memory need to be able to coordinate their access to shared memory efficiently. Two fundamental forms of synchronization are used to coordinate processes' access to shared memory: *mutual exclusion* and *condition synchronization*. Mutual exclusion mechanisms serialize cooperating processes' access to shared resources. Condition synchronization mechanisms delay a process' progress until an application-defined condition is met.

Mutual exclusion mechanisms ensure that only one of the cooperating processes can be executing in a critical section at a time. Three types of mechanisms are typically used to provide mutual exclusion—those that involve busy waiting, those that involve sleepy waiting, and those that involve a combination of the two when a process attempts to enter a locked critical section. Busy-wait mechanisms, also known as *spin locks*, use a locking technique that obtains a lock using a hardware supported test and set operation. If a process attempts to obtain a busy-wait lock that is currently in a locked state, the locking process continues to retry the test and set operation until the process that currently holds the lock has cleared it and the test and set operation succeeds. In contrast, a sleepy-wait mechanism, such as a semaphore, will put a process to sleep if it attempts to obtain a lock that is currently in a locked state.

Busy-wait mechanisms are highly efficient when most attempts to obtain the lock will succeed. This is because a simple test and set operation is all that is required to obtain a busy-wait lock. Busy-wait mechanisms are appropriate for protecting resources when the amount of time that the lock is held is short. There are two reasons for this: 1) when lock hold times are short, it is likely that a locking process will find the lock in an unlocked state and therefore the overhead of the lock mechanism will also be minimal and 2) when the lock hold time is short, the delay in obtaining the lock is also expected to be short. It is important when using busy-wait mutual exclusion that delays in obtaining a lock be kept short, since the busy-wait mechanism is going to waste CPU resources while waiting for a lock to become unlocked. As a general rule, if the lock hold times are all less than the time it takes to execute two context switches, then a busy-wait mechanism is appropriate.

Critical sections are often very short. To keep the cost of synchronization comparatively small, synchronizing operations performed on entry to and exit from a critical section cannot enter the kernel. It is undesirable for the execution overhead associated with entering and leaving the critical section to be longer than the length of the critical section itself.

In order for spin locks to be used as an effective mutual exclusion tool, the expected time that a process will spin waiting for another process to release the lock must be not only brief but also predictable. Such unpredictable events as page faults, signals, and the preemption of a process holding the lock cause the real elapsed time in a critical section to significantly exceed the expected execution time. At best, these unexpected delays inside of a critical section may cause other CPUs to delay longer than anticipated; at worst, they may cause deadlock. Locking pages in memory can be accomplished during program initialization so as not to have an impact on the time to enter a critical section. The mechanisms for rescheduling control provide a low-overhead means of controlling signals and process preemption.

Semaphores are another mechanism for providing mutual exclusion. Semaphores are a form of sleepy-wait mutual exclusion because a process that attempts to lock a semaphore that is already locked will be blocked or put to sleep. POSIX counting semaphores provide a portable means of controlling access to shared resources. A counting semaphore is an object that has an integer value and a limited set of operations defined for it. Counting semaphores provide a simple interface that is implemented to achieve the fastest performance for lock and unlock operations. System V semaphores are a complex data type that allows many additional functions (for example the ability to find out how many waiters there are on a semaphore or the ability to operate on a set of semaphores).

Tools for providing rescheduling control are described in “Rescheduling Control.” Tools for implementing busy-wait mutual exclusion are explained in “Busy-Wait Mutual Exclusion.” Tools for coordinating interaction between processes are explained in “Condition Synchronization.”

POSIX counting semaphores are described in the section “POSIX Counting Semaphores.” System V semaphores are described in the section “System V Semaphores.”

Rescheduling Control

Multiprocess, real-time applications frequently wish to defer CPU rescheduling for brief periods of time. To use busy-wait mutual exclusion effectively, spinlock hold times must be small and predictable.

CPU rescheduling and signal handling are major sources of unpredictability. A process would like to make itself immune to rescheduling when it acquires a spinlock, and vulnerable again when it releases the lock. A system call could raise the caller’s priority to the highest in the system, but the overhead of doing so is prohibitive.

A rescheduling variable provides control for rescheduling and signal handling. You allocate the variable in your application, notify the kernel of its location, and manipulate it directly from your application to disable and re-enable rescheduling. While rescheduling is disabled, quantum expirations, preemptions, and certain types of signals are held.

A system call and a set of macros accommodate use of the rescheduling variable. In the sections that follow, the variable, the system call, and the macros are described, and the procedures for using them are explained.

The primitives described here provide low overhead control of CPU rescheduling and signal delivery.

Understanding Rescheduling Variables

A rescheduling variable is a data structure, defined in **<sys/rescntl.h>** that controls a single process’ vulnerability to rescheduling:

```
struct resched_var {
                        pid_t rv_pid;
                        ...
                        volatile int rv_nlocks;
                        ...
};
```

It is allocated on a per-process basis by the application, not by the kernel. The **resched_cntl(2)** system call informs the kernel of the location of the variable, and the kernel examines the variable before making rescheduling decisions. Direct manipulation of the variable from user mode with the **resched_lock** and **resched_unlock** macros disable and re-enable rescheduling.

Use of the **resched_cntl** system call is explained in "Using the resched_cntl System Call." A set of rescheduling control macros enables you to manipulate the variable from your application. Use of these macros is explained in "Using the Rescheduling Control Macros."

These interfaces should be used only by single-threaded processes. A rescheduling variable is valid only for the process or thread that registers the location of the rescheduling variable.

Using the resched_cntl System Call

The **resched_cntl** system call enables you to perform a variety of rescheduling control operations. These include initializing a rescheduling variable, informing the kernel of its location, obtaining its location, and setting a limit on the length of time that rescheduling can be deferred.

Synopsis

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

int resched_cntl(cmd, arg)

int cmd;
char *arg;

gcc [options] file -lccur_rt ...
```

Arguments are defined as follows:

cmd the operation to be performed

arg a pointer to an argument whose value depends upon the value of *cmd*

cmd can be one of the following. The values of *arg* that are associated with each command are indicated.

RESCHED_SET_VARIABLE

This command informs the kernel of the location of the caller's rescheduling variable. The rescheduling variable must be located in a process private page, which excludes pages in shared memory segments or files that have been mapped MAP_SHARED.

Two threads of the same process should not register the same address as their rescheduling variable. If *arg* is not NULL, the `struct resched_var` it points to is initialized and locked into physical memory. If *arg* is NULL, the caller is disassociated from any existing variable, and the appropriate pages are unlocked.

After a **fork(2)**, the child process inherits rescheduling variables from its parent. The `rv_pid` field of the child's rescheduling variable is updated to the process ID of the child.

If a child process has inherited a rescheduling variable and it, in turn, forks one or more child processes, those child processes inherit the rescheduling variable with the `rv_pid` field updated.

If a rescheduling variable is locked in the parent process at the time of the call to fork, the rescheduling variable is locked in the child process.

Use of this command requires root capability or CAP_IPC_LOCK and CAP_SYS_RAWIO privileges.

RESCHED_SET_LIMIT This command is a debugging tool. If *arg* is not NULL, it points to a `struct timeval` specifying the maximum length of time the caller expects to defer rescheduling. The **SIGABRT** signal is sent to the caller when this limit is exceeded if the local timer of the CPU is enabled. If *arg* is NULL, any previous limit is forgotten.

RESCHED_GET_VARIABLE
This command returns the location of the caller's rescheduling variable. *arg* must point to a rescheduling variable pointer. The pointer referenced by *arg* is set to NULL if the caller has no rescheduling variable, and is set to the location of the rescheduling variable otherwise.

RESCHED_ESIGNAL The process received one of the error signals that are not deferred while a rescheduling variable lock was held.

RESCHED_SERVE This command is used by **`resched_unlock`** to service pending signals and context switches. Applications should not need to use this command directly. *arg* must be 0.

Using the Rescheduling Control Macros

A set of rescheduling control macros enables you to disable and re-enable rescheduling and to determine the number of rescheduling locks in effect. These macros are briefly described as follows:

`resched_lock` increments the number of rescheduling locks held by the calling process

`resched_unlock` decrements the number of rescheduling locks held by the calling process

`resched_nlocks` returns the number of rescheduling locks currently in effect

resched_lock

Synopsis

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

void resched_lock(r);

struct resched_var *r;
```

The argument is defined as follows:

r a pointer to the calling process' rescheduling variable

Resched_lock does not return a value; it increments the number of rescheduling locks held by the calling process. As long as the process does not enter the kernel, quantum expirations, preemptions, and some signal deliveries are deferred until all rescheduling locks are released.

However, if the process generates an exception (e.g., a page fault) or makes a system call, it may receive signals or otherwise context switch regardless of the number of rescheduling locks it holds. The following signals represent error conditions and are NOT affected by rescheduling locks: SIGILL, SIGTRAP, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGABRT, SIGSYS, SIGPIPE, SIGXCPU, and SIGXFSZ.

Making system calls while a rescheduling variable is locked is possible but not recommended. However, it is not valid to make any system call that results in putting the calling process to sleep while a rescheduling variable is locked.

resched_unlock

Synopsis

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

void resched_unlock(r);

struct resched_var *r;
```

The argument is defined as follows:

r a pointer to the calling process' rescheduling variable

Resched_unlock does not return a value. If there are no outstanding locks after the decrement and a context switch or signal are pending, they are serviced immediately.

NOTE

The `rv_nlocks` field must be a positive integer for the lock to be considered active. Thus, if the field is zero or negative, it is considered to be unlocked.

resched_nlocks

Synopsis

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

int resched_nlocks(r);

struct resched_var *r;
```

The argument is defined as follows:

r a pointer to the calling process' rescheduling variable

Resched_nlocks returns the number of rescheduling locks currently in effect.

For additional information on the use of these macros, refer to the **resched_cntl(2)** man page.

Applying Rescheduling Control Tools

The following C program segment illustrates the procedures for controlling rescheduling by using the tools described in the preceding sections. This program segment defines a rescheduling variable (rv) as a global variable; initializes the variable with a call to **resched_cntl**; and disables and re-enables rescheduling with calls to **resched_lock** and **resched_unlock**, respectively.

```
01 struct resched_var rv;
02
03 void
04 main ()
05 {
06   resched_cntl(RESCHED_SET_VARIABLE, (char *)&rv);
07
08   resched_lock(&rv);
09
10   /* nonpreemptible code */
11   ...
12
13   resched_unlock(&rv);
14 }
```

Busy-Wait Mutual Exclusion

Busy-wait mutual exclusion is achieved by associating a synchronizing variable with a shared resource. When a process or thread wishes to gain access to the resource, it locks the synchronizing variable. When it completes its use of the resource, it unlocks the synchronizing variable. If another process or thread attempts to gain access to the resource while the first process or thread has the resource locked, that process or thread must delay by repeatedly testing the state of the lock. This form of synchronization requires that the synchronizing variable be accessible directly from user mode and that the lock and unlock operations have very low overhead.

RedHawk Linux busy-wait mutual exclusion tools include a low-overhead busy-wait mutual exclusion variable (a spin lock) and a corresponding set of macros. In the sections that follow, the variable and the macros are defined, and the procedures for using them are explained.

The threads library, **libpthread**, also provides a set of spin lock routines. These routines are described in the man page for **pthread_mutex_init(3)**. It is recommended that you use the macros described in this chapter instead of the pthread mutex routines because the macros are more efficient and they give you more flexibility; for example, the spin lock macros allow you to construct a synchronization primitive that is a combination of the busy-wait and sleepy-wait primitives. If you were to construct such a primitive, the primitive would gain access to the lock by spinning for some number of spins and then blocking if the lock were not available. The advantage that this type of lock offers is that you do not have to use rescheduling variables to prevent deadlock.

Understanding the Busy-Wait Mutual Exclusion Variable

The busy-wait mutual exclusion variable is a data structure known as a spin lock. This variable is defined in **<spin.h>** as follows:

```
struct spin_mutex {

      ...

};
```

The spin lock has two states: locked and unlocked. When initialized, the spin lock is in the unlocked state.

If you wish to use spin locks to coordinate access to shared resources, you must allocate them in your application program and locate them in memory that is shared by the processes or threads that you wish to synchronize. You can manipulate them by using the macros described in "Using the Busy-Wait Mutual Exclusion Macros."

Using the Busy-Wait Mutual Exclusion Macros

A set of busy-wait mutual exclusion macros allows you to initialize, lock, and unlock spin locks and determine whether or not a particular spin lock is locked. These macros are briefly described as follows:

spin_init	initialize a spin lock to the unlocked state
spin_trylock	attempt to lock a specified spin lock
spin_unlock	unlock a specified spin lock
spin_islock	determine whether or not a specified spin lock is locked

It is important to note that none of these macros enables you to lock a spin lock unconditionally. You can construct this capability by using the tools that are provided.

CAUTION

Operations on spin locks are not recursive; a process or thread can deadlock if it attempts to relock a spin lock that it has already locked.

You must initialize spin locks before you use them by calling the **spin_init** macro. You call **spin_init** only once for each spin lock. If the specified spin lock is locked, **spin_init** effectively unlocks it. The **spin_init** macro is specified as follows:

```
#include <spin.h>

void spin_init(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m	the starting address of the spin lock to be initialized

Spin_init does not return a value; it places the spin lock in the unlocked state.

The **spin_trylock** macro is specified as follows:

```
#include <spin.h>

int spin_trylock(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m	a pointer to the spin lock that you wish to try to lock

A return of TRUE indicates that the calling process or thread has succeeded in locking the spin lock. A return of FALSE indicates that it has not succeeded. **Spin_trylock** does not block the calling process or thread.

The **spin_unlock** macro is specified as follows:

```
#include <spin.h>

void spin_unlock(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m a pointer to the spin lock that you wish to unlock

Spin_unlock does not return a value.

The **spin_islock** macro is specified as follows:

```
#include <spin.h>

int spin_islock(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m a pointer to the spin lock whose state you wish to determine

A return of TRUE indicates that the specified spin lock is locked. A return of FALSE indicates that it is unlocked. **Spin_islock** does not attempt to lock the spin lock.

For additional information on the use of these macros, refer to the **spin_trylock(3)** man page.

Applying Busy-Wait Mutual Exclusion Tools

Procedures for using the tools for busy-wait mutual exclusion are illustrated by the following code segments. The first segment shows how to use these tools along with rescheduling control to acquire a spin lock; the second shows how to release a spin lock. Note that these segments contain no system calls or procedure calls.

The _m argument points to a spin lock, and the _r argument points to the calling process' or thread's rescheduling variable. It is assumed that the spin lock is located in shared memory. To avoid the overhead associated with paging and swapping, it is recommended that the pages that will be referenced inside the critical section be locked in physical memory (see the **mlock(2)** and **shmctl(2)** system calls).

```
#define spin_acquire(_m,_r) \
{ \
   resched_lock(_r); \
   while (!spin_trylock(_m)) { \
      resched_unlock(_r); \
      while (spin_islock(_m)); \
      resched_lock(_r); \
   } \
}

#define spin_release(_m,_r) \
{ \
   spin_unlock(_m); \
   resched_unlock(_r); \
}
```

In the first segment, note the use of the **spin_trylock** and **spin_islock** macros. If a process or thread attempting to lock the spin lock finds it locked, it waits for the lock to be released by calling **spin_islock**. This sequence is more efficient than polling directly with **spin_trylock**. The **spin_trylock** macro contains special instructions to perform test-and-set atomically on the spin lock. These instructions are less efficient than the simple memory read performed in **spin_islock**.

Note also the use of the rescheduling control macros. To prevent deadlock, a process or thread disables rescheduling prior to locking the spin lock and re-enables it after unlocking the spin lock. A process or thread also re-enables rescheduling just prior to the call to **spin_islock** so that rescheduling is not deferred any longer than necessary.

POSIX Counting Semaphores

Overview

Counting semaphores provide a simple interface that can be implemented to achieve the fastest performance for lock and unlock operations. A counting semaphore is an object that has an integer value and a limited set of operations defined for it. These operations and the corresponding POSIX interfaces include the following:

- An initialization operation that sets the semaphore to zero or a positive value—**sem_init**
- A lock operation that decrements the value of the semaphore—**sem_wait**. If the resulting value is negative, the process performing the operation blocks.
- An unlock operation that increments the value of the semaphore—**sem_post**. If the resulting value is less than or equal to zero, one of the processes blocked on the semaphore is awakened. If the resulting value is greater than zero, no processes were blocked on the semaphore.
- A conditional lock operation that decrements the value of the semaphore only if the value is positive—**sem_trywait**. If the value is zero or negative, the operation fails.
- A query operation that provides a snapshot of the value of the semaphore—**sem_getvalue**

The lock, unlock, and conditional lock operations are *atomic* (the set of operations are performed at the same time and only if they can all be performed simultaneously).

A counting semaphore may be used to control access to any resource that can be used by multiple cooperating threads. A process creates and initializes an unnamed semaphore through a call to the **sem_init(2)** routine. The semaphore is initialized to a value that is specified on the call. All threads within the application have access to the unnamed semaphore once it has been created with the **sem_init** routine call.

When the unnamed semaphore is initialized, its value should be set to the number of available resources. To use an unnamed counting semaphore to provide mutual exclusion, the semaphore value should be set to one.

A cooperating process that wants access to a critical resource must lock the semaphore that protects that resource. When the process locks the semaphore, it knows that it can use the resource without interference from any other cooperating process in the system. An application must be written so that the resource is accessed only after the semaphore that protects it has been acquired.

As long as the semaphore value is positive, resources are available for use; one of the resources is allocated to the next process that tries to acquire it. When the semaphore value is zero, then none of the resources are available; a process trying to acquire a resource must wait until one becomes available. If the semaphore value is negative, then there may be one or more processes that are blocked and waiting to acquire one of the resources. When a process completes use of a resource, it unlocks the semaphore, indicating that the resource is available for use by another process.

The concept of ownership does not apply to a counting semaphore. One process can lock a semaphore; another process can unlock it.

The semaphore unlock operation is *async-signal safe*; that is, a process can unlock a semaphore from a signal-handling routine without causing deadlock.

The absence of ownership precludes priority inheritance. Because a process does not become the owner of a semaphore when it locks the semaphore, it cannot temporarily inherit the priority of a higher-priority process that blocks trying to lock the same semaphore. As a result, unbounded priority inversion can occur.

NOTE

RedHawk Linux does not currently support named semaphores. As a result, the **sem_open**, **sem_close** and **sem_unlink** functions all return an **errno** value of **ENOSYS**, and should not be used at this time.

Interfaces

The sections that follow explain the procedures for using the POSIX counting semaphore interfaces. These interfaces are briefly described as follows:

sem_init	initializes an unnamed counting semaphore
sem_destroy	removes an unnamed counting semaphore
sem_wait	locks a counting semaphore
sem_trywait	locks a counting semaphore only if it is currently unlocked
sem_post	unlocks a counting semaphore
sem_getvalue	obtains the value of a counting semaphore

Note that to use these interfaces, you must link your application with the Linux Threads library. You may link with this library statically or dynamically (with shared libraries). The following example shows the command line invocation when using shared libraries:

gcc [*options*] *file*.**c -lpthread**

The same application could be built statically with the following invocation line:

gcc [*options*] **-static** *file*.**c -lpthread**

Using the sem_init Routine

The **sem_init(3)** library routine allows the calling process to initialize an unnamed counting semaphore by setting the semaphore value to the number of available resources being protected by the semaphore. To use a counting semaphore for mutual exclusion, the process sets the value to one.

Currently RedHawk Linux only supports counting semaphores that are local to the process (the *pshared* parameter must be set to 0). These semaphores may thus only be shared among threads within the same application.

After one thread in an application creates and initializes a semaphore, other cooperating threads within that same application can operate on the semaphore by using the **sem_wait**, **sem_trywait**, **sem_post** and **sem_getvalue** library routines. These routines are explained in the sections that follow.

A child process created by a **fork(2)** system call does *not* inherit access to a semaphore that has already been initialized by the parent. A process also loses access to a semaphore after invoking the **exec(3)** or **exit(2)** system calls.

CAUTION

The IEEE 1003.1b-1993 standard does not indicate what happens when multiple processes invoke **sem_init** for the same semaphore. Currently, the RedHawk Linux implementation simply reinitializes the semaphore to the value specified on **sem_init** calls that are made after the initial **sem_init** call.

To be certain that application code can be ported to any system that supports POSIX interfaces (including future Concurrent systems), cooperating processes that use **sem_init** should ensure that a single process initializes a particular semaphore and that it does so only once.

If **sem_init** is called after it has already been initialized with a prior **sem_init** call, and there are currently threads that are waiting on this same semaphore, then these threads will never return from their **sem_wait** calls, and they will need to be explicitly terminated.

An unnamed counting semaphore is removed by invoking the **sem_destroy** routine (see "Using the sem_destroy Routine" for an explanation of this routine).

Synopsis

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

The arguments are defined as follows:

sem a pointer to a `sem_t` structure that represents the unnamed counting semaphore to be initialized

pshared an integer value that indicates whether or not the semaphore is to be shared by other processes. If *pshared* is set to a non-zero value, then the semaphore is shared among processes. If *pshared* is set to zero, then the semaphore is shared only among threads within the same application.

NOTE

RedHawk Linux does not currently support process-shared semaphores. Therefore, **sem_init** always returns with **-1** and **errno** set to **ENOSYS** if *pshared* is not set to zero.

value zero or a positive integer value that initializes the semaphore value to the number of resources currently available. This number cannot exceed the value of **SEM_VALUE_MAX** (see the file <**semaphore.h**> to determine this value).

A return value of **0** indicates that the call to **sem_init** has been successful. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sem_init(3)** man page for a listing of the types of errors that may occur.

Using the sem_destroy Routine

CAUTION

An unnamed counting semaphore should not be removed until there is no longer a need for any process to operate on the semaphore and there are no processes currently blocked on the semaphore.

Synopsis

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

The argument is defined as follows:

sem a pointer to the unnamed counting semaphore to be removed. Only a counting semaphore created with a call to **sem_init(3)** may be removed by invoking **sem_destroy**.

A return value of **0** indicates that the call to **sem_destroy** has been successful. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sem_destroy(3)** man page for a listing of the types of errors that may occur.

Using the sem_wait Routine

The **sem_wait(3)** library routine allows the calling process to lock an unnamed counting semaphore. If the semaphore value is equal to zero, the semaphore is already locked. In this case, the process blocks until it is interrupted by a signal or the semaphore is unlocked. If the semaphore value is greater than zero, the process locks the semaphore and proceeds. In either case, the semaphore value is decremented.

Synopsis

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

The argument is defined as follows:

sem a pointer to the unnamed counting semaphore to be locked

A return value of **0** indicates that the process has succeeded in locking the specified semaphore. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sem_wait(3)** man page for a listing of the types of errors that may occur.

Using the sem_trywait Routine

The **sem_trywait(3)** library routine allows the calling process to lock a counting semaphore only if the semaphore value is greater than zero, indicating that the semaphore is unlocked. If the semaphore value is equal to zero, the semaphore is already locked, and the call to **sem_trywait** fails. If a process succeeds in locking the semaphore, the semaphore value is decremented; otherwise, it does not change.

Synopsis

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

The argument is defined as follows:

sem a pointer to the unnamed counting semaphore that the calling process is attempting to lock

A return value of **0** indicates that the calling process has succeeded in locking the specified semaphore. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sem_trywait(3)** man page for a listing of the types of errors that may occur.

Using the sem_post Routine

The **sem_post(3)** library routine allows the calling process to unlock a counting semaphore. If one or more processes are blocked waiting for the semaphore, the waiting process with the highest priority is awakened when the semaphore is unlocked.

Synopsis

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

The argument is defined as follows:

sem a pointer to the unnamed counting semaphore to be unlocked

A return value of **0** indicates that the call to **sem_post** has been successful. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sem_post(3)** man page for a listing of the types of errors that may occur.

Using the sem_getvalue Routine

The **sem_getvalue(3)** library routine allows the calling process to obtain the value of an unnamed counting semaphore.

Synopsis

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

The arguments are defined as follows:

sem — a pointer to the unnamed counting semaphore for which you wish to obtain the value

sval — a pointer to a location where the value of the specified unnamed counting semaphore is to be returned. The value that is returned represents the actual value of the semaphore at some unspecified time during the call. It is important to note, however, that this value may not be the actual value of the semaphore at the time of the return from the call.

A return value of **0** indicates that the call to **sem_getvalue** has been successful. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sem_getvalue(3)** man page for a listing of the types of errors that may occur.

System V Semaphores

Overview

The System V semaphore is an interprocess communication (IPC) mechanism that allows processes to synchronize via the exchange of semaphore values. Since many applications require the use of more than one semaphore, the operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores, up to a limit of **SEMMSL** (as defined in **<linux/sem.h>**). Semaphore sets are created using the **semget(2)** system call.

When only a simple semaphore is needed, a counting semaphore is more efficient (see the section "POSIX Counting Semaphores").

The process performing the **semget** system call becomes the owner/creator, determines how many semaphores are in the set, and sets the initial operation permissions for all processes, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl(2)** system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl** to perform other control functions.

Any process can manipulate the semaphore(s) if the owner of the semaphore grants permission. Each semaphore within a set can be incremented and decremented with the **semop(2)** system call (see the section "Using the semop System Call" later in this chapter).

To increment a semaphore, an integer value of the desired magnitude is passed to the **semop** system call. To decrement a semaphore, a minus (-) value of the desired magnitude is passed.

The operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (**IPC_NOWAIT** flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a *blocking semaphore operation*. This ability is also available for a process which is testing for a semaphore equal to zero; only read permission is required for this test; it is accomplished by passing a value of zero to the **semop** system call.

On the other hand, if the process is not successful and did not request to have its execution suspended, it is called a *nonblocking semaphore operation*. In this case, the process is returned **-1** and the external **errno** variable is set accordingly.

The blocking semaphore operation allows processes to synchronize via the values of semaphores at different points in time. Remember also that IPC facilities remain in the operating system until removed by a permitted process or until the system is reinitialized.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is numbered one less than the total in the set.

A single system call can be used to perform a sequence of these blocking/nonblocking operations on a set of semaphores. When performing a sequence of operations, the blocking/nonblocking operations can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. For example, if the first three of six operations on a set of ten semaphores could be completed successfully, but the fourth operation would be blocked, no changes are made to the set until all six operations can be performed without blocking. Either all of the operations are successful and the semaphores are changed, or one or more (nonblocking) operation is unsuccessful and none are changed. In short, the operations are performed atomically.

Remember, any unsuccessful nonblocking operation for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, **`-1`** is returned to the process, and the external variable **`errno`** is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, **`-1`** is returned to the process, and the external variable **`errno`** is set accordingly.

Using System V Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified data structure and semaphore set (array) must be created. The unique identifier is called the semaphore set identifier (`semid`); it is used to identify or refer to a particular data structure and semaphore set. This identifier is accessible by any process in the system, subject to normal access restrictions.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (*nsems*) in a semaphore set is user selectable.

The `sembuf` structure, which is used on **`semop(2)`** system calls, is shown in Figure 5-1.

Figure 5-1. Definition of sembuf Structure

```
struct sembuf {
   unsigned short int sem_num;  /* semaphore number */
   short int sem_op;            /* semaphore operation */
   short int sem_flg;           /* operation flag */
};
```

The `sembuf` structure is defined in the **`<sys/sem.h>`** header file.

The struct semid_ds structure, which is used on certain **semctl(2)** service calls, is shown in Figure 5-2.

Figure 5-2. Definition of semid_ds Structure

```
struct semid_ds {
    struct ipc_perm sem_perm;         /* operation permission struct */
    __time_t sem_otime;               /* last semop() time */
    unsigned long int __unused1;
    __time_t sem_ctime;               /* last time changed by semctl() */
    unsigned long int __unused2;
    unsigned long int sem_nsems;      /* number of semaphores in set */
    unsigned long int __unused3;
    unsigned long int __unused4;
};
```

Though the semid_ds data structure is located in **<bits/sem.h>**, user applications should include the **<sys/sem.h>** header file, which internally includes the **<bits/sem.h>** header file.

Note that the sem_perm member of this structure is of type ipc_perm.

The ipc_perm data structure is the same for all IPC facilities; it is located in the **<bits/ipc.h>** header file, but user applications should include the **<sys/ipc.h>** file, which internally includes the **<bits/ipc.h>** header file. The details of the ipc_perm data structure are given in the section entitled "Understanding System V Messages" in Chapter 3.

A **semget(2)** system call performs one of two tasks:

- creates a new semaphore set identifier and creates an associated data structure and semaphore set for it
- locates an existing semaphore set identifier that already has an associated data structure and semaphore set

The task performed is determined by the value of the *key* argument passed to the **semget** system call. If *key* is not already in use for an existing semid and the **IPC_CREAT** flag is set, a new semid is returned with an associated data structure and semaphore set created for it, provided no system tunable parameter would be exceeded.

There is also a provision for specifying a *key* of value zero (0), which is known as the private key (**IPC_PRIVATE**). When this key is specified, a new identifier is always returned with an associated data structure and semaphore set created for it, unless a system-tunable parameter would be exceeded. The **ipcs(8)** command will show the *key* field for the semid as all zeros.

When a semaphore set is created, the process which calls **semget** becomes the owner/creator and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator (see the "Controlling Semaphores" section). The creator of the semaphore set also determines the initial operation permissions for the facility.

If a semaphore set identifier exists for the key specified, the value of the existing identifier is returned. If you do not want to have an existing semaphore set identifier returned, a control command (**IPC_EXCL)** can be specified (set) in the *semflg* argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (*nsems*) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use `0` for *nsems* (see "Using the semget System Call" for more information).

Once a uniquely identified semaphore set and data structure are created or an existing one is found, **`semop(2)`** and **`semctl(2)`** can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. The **`semop`** system call is used to perform these operations (see "Operations On Semaphores" for details of the **`semop`** system call).

The **`semctl`** system call permits you to control the semaphore facility in the following ways:

- by returning the value of a semaphore (**GETVAL**)
- by setting the value of a semaphore (**SETVAL**)
- by returning the PID of the last process performing an operation on a semaphore set (**GETPID**)
- by returning the number of processes waiting for a semaphore value to become greater than its current value (**GETNCNT**)
- by returning the number of processes waiting for a semaphore value to equal zero (**GETZCNT**)
- by getting all semaphore values in a set and placing them in an array in user memory (**GETALL**)
- by setting all semaphore values in a semaphore set from an array of values in user memory (**SETALL**)
- by retrieving the data structure associated with a semaphore set (**IPC_STAT**)
- by changing operation permissions for a semaphore set (**IPC_SET**)
- by removing a particular semaphore set identifier from the operating system along with its associated data structure and semaphore set (**IPC_RMID**)

See the section "Controlling Semaphores" for details of the **`semctl`** system call.

Getting Semaphores

This section describes how to use the **`semget`** system call. The accompanying program illustrates its use.

Using the semget System Call

The **semget** system call creates a new semaphore set or identifies an existing one.

Synopsis

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

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

key_t is defined by a typedef in the **<bits/sys/types.h>** header file to be an integral type (this header file is included internally by **<sys/types.h>**). The integer returned from this system call upon successful completion is the semaphore set identifier (semid).

A new semid with an associated semaphore set and data structure is created if one of the following conditions is true:

- *key* is equal to **IPC_PRIVATE**,
- *key* is a unique integer and *semflg* ANDed with **IPC_CREAT** is "true."

The value passed to the *semflg* argument must be an integer that will specify the following:

- operation permissions
- control fields (commands)

Table 5-1 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 5-1. Semaphore Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

A specific value is derived by adding or bitwise ORing the values for the operation permissions wanted. That is, if "read by user" and "read/alter by others" is desired, the code value would be 00406 (00400 plus 00006).

Control flags are predefined constants. The flags that apply to the **semget** system call are **IPC_CREAT** and **IPC_EXCL**.

The value for *semflg* is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by adding or bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *semflg* value can easily be set by using the flag names in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL|0400));
```

As specified by the **semget(2)** man page, success or failure of this system call depends upon the actual argument values for *key*, *nsems*, and *semflg*, and system-tunable parameters. The system call will attempt to return a new semaphore set identifier if one of the following conditions is true:

- *key* is equal to **IPC_PRIVATE**
- *key* does not already have a semaphore set identifier associated with it and *semflg* and **IPC_CREAT** is “true” (not zero).

The following values are defined in **<linux/sem.h>**. Exceeding these values always cause a failure.

SEMMNI determines the maximum number of unique semaphore sets (`semid` values) that may be in use at any given time.

SEMMSL determines the maximum number of semaphores in each semaphore set.

SEMMNS determines the maximum number of semaphores in all semaphore sets system wide

A list of semaphore limit values may be obtained with the **ipcs(8)** command by using the following options:

```
ipcs -s -l
```

See the **ipcs(8)** man page for details.

IPC_EXCL can be used in conjunction with **IPC_CREAT**. This causes the system call to return an error if a semaphore set identifier already exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) identifier when it has not. In other words, when both **IPC_CREAT** and **IPC_EXCL** are specified, a new semaphore set identifier is returned only if a new semaphore is created. Any value for *semflg* returns a new identifier if *key* equals zero (**IPC_PRIVATE**), assuming no system-tunable parameters are exceeded.

Refer to the **semget(2)** man page for specific associated data structure initialization, as well as the specific failure conditions and their error names.

Example Program

The example program presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **semget** system call to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values.

This program begins (lines 5-9) by including the required header files as specified by the **semget(2)** man page.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are intended to be self-explanatory. These names make the program more readable and are valid because they are local to the program.

The variables declared for this program and their roles are:

`key`	passes the value for the desired key
`opperm`	stores the desired operation permissions
`flags`	stores the desired control commands (flags)
`opperm_flags`	stores the combination from the logical ORing of the `opperm` and `flags` variables; it is then used in the system call to pass the `semflg` argument
`semid`	returns the semaphore set identification number for a successful system call or the error code (**-1**) for an unsuccessful one.

The program begins by prompting for a hexadecimal `key`, an octal operation permissions code, and the control command combinations (`flags`) which are selected from a menu (lines 16-33). All possible combinations are allowed even though they might not be valid, in order to allow observing the errors for invalid combinations.

Next, the menu selection for the flags is combined with the operation permissions and the result is stored in `opperm_flags` (lines 37-53).

Then, the number of semaphores for the set is requested (lines 54-58) and its value is stored in `nsems`.

The system call is made next; the result is stored in the `semid` (lines 61, 62).

Because the `semid` variable now contains a valid semaphore set identifier or the error code (`-1`), it can be tested to see if an error occurred (line 64). If `semid` equals `-1`, a message indicates that an error resulted and the external `errno` variable is displayed (line 66). Remember that the external `errno` variable is only set when a system call fails; it should only be examined immediately following system calls.

If no error occurs, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget** system call follows. This file is provided as **/usr/share/doc/ccur/examples/semget.c**.

```
1   /*
2    * Illustrates the semaphore get, semget(),
3    * system call capabilities.
4    */
```

```
5   #include    <stdio.h>
6   #include    <sys/types.h>
7   #include    <sys/ipc.h>
8   #include    <sys/sem.h>
9   #include    <errno.h>
10  /* Start of main C language program */
11  main()
12  {
13     key_t key;
14     int opperm, flags, nsems;
15     int semid, opperm_flags;
16     /* Enter the desired key */
17     printf("\nEnter the desired key in hex = ");
18     scanf("%x", &key);
19     /* Enter the desired octal operation
20        permissions. */
21     printf("\nEnter the operation\n");
22     printf("permissions in octal = ");
23     scanf("%o", &opperm);
24     /* Set the desired flags. */
25     printf("\nEnter corresponding number to\n");
26     printf("set the desired flags:\n");
27     printf("No flags                  = 0\n");
28     printf("IPC_CREAT                 = 1\n");
29     printf("IPC_EXCL                  = 2\n");
30     printf("IPC_CREAT and IPC_EXCL    = 3\n");
31     printf("           Flags          = ");
32     /* Get the flags to be set. */
33     scanf("%d", &flags);
34     /* Error checking (debugging) */
35     printf ("\nkey =0x%x, opperm = 0%o, flags = %d\n",
36        key, opperm, flags);
37     /* Incorporate the control fields (flags) with
38        the operation permissions. */
39     switch (flags)
40     {
41     case 0:    /* No flags are to be set. */
42        opperm_flags = (opperm | 0);
43        break;
44     case 1:    /* Set the IPC_CREAT flag. */
45        opperm_flags = (opperm | IPC_CREAT);
46        break;
47     case 2:    /* Set the IPC_EXCL flag. */
48        opperm_flags = (opperm | IPC_EXCL);
49        break;
50     case 3: /* Set the IPC_CREAT and IPC_EXCL
51              flags. */
52        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
53     }
54     /* Get the number of semaphores for this set. */
55     printf("\nEnter the number of\n");
56     printf("desired semaphores for\n");
57     printf("this set (max is SEMMSL) = ");
58     scanf("%d", &nsems);
59     /* Check the entry.* /
60     printf("\nNsems = %d\n", nsems);
61     /* Call the semget system call.* /
62     semid = semget(key, nsems, opperm_flags);
63     /* Perform the following if the call is unsuccessful.* /
64     if(semid == -1)
65     {
66        printf("The semget call failed, error no. = %d\n", errno);
67     }
68     /* Return the semid upon successful completion.* /
69     else
70        printf("\nThe semid = %d\n", semid);
71     exit(0);
72  }
```

Controlling Semaphores

This section describes how to use the **semctl(2)** system call. The accompanying program illustrates its use.

Using the semctl System Call

Synopsis

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

int semctl (semid, semnum, cmd, arg)
int          semid, cmd;
int          semnum;
union semun
{
        int val;
        struct semid_ds *buf;
        ushort *array;
} arg;
```

The *semid* argument must be a valid, non-negative, integer value that has already been created using the **semget** system call.

The *semnum* argument is used to select a semaphore by its number. This relates to sequences of operations (atomically performed) on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore is numbered one less than the total in the set.

The *cmd* argument can be replaced by one of the following values:

GETVAL returns the value of a single semaphore within a semaphore set

SETVAL sets the value of a single semaphore within a semaphore set

GETPID returns the PID of the process that performed the last operation on the semaphore within a semaphore set

GETNCNT returns the number of processes waiting for the value of a particular semaphore to become greater than its current value

GETZCNT returns the number of processes waiting for the value of a particular semaphore to be equal to zero

GETALL returns the value for all semaphores in a semaphore set

SETALL sets all semaphore values in a semaphore set

IPC_STAT returns the status information contained in the associated data structure for the specified *semid*, and places it in the data structure pointed to by *arg*.buf

IPC_SET	sets the effective user/group identification and operation permissions for the specified semaphore set (*semid*)
IPC_RMID	removes the specified semaphore set (*semid*) along with its associated data structure

NOTE

The **`semctl(2)`** service also supports the **IPC_INFO**, **SEM_STAT** and **SEM_INFO** commands. However, since these commands are only intended for use by the **`ipcs(8)`** utility, these commands are not discussed.

To perform an **IPC_SET** or **IPC_RMID** control command, a process must meet one or more of the following conditions:

- have an effective user id of OWNER
- have an effective user id of CREATOR
- be the super-user
- have the **CAP_SYS_ADMIN** capability

Note that a semaphore set can also be removed by using the **`ipcrm(1)`** command and specifying the "**`sem`***id*" option, where *id* specifies the identifier for the semaphore set. To use this command, a process must have the same capabilities as those required for performing an **IPC_RMID** control command. See the **`ipcrm(1)`** man page for additional information on the use of this command.

The remaining control commands require either read or write permission, as appropriate.

The *arg* argument is used to pass the system call the appropriate union member for the control command to be performed. For some of the control commands, the *arg* argument is not required and is simply ignored.

- *arg*`.val` required: **SETVAL**
- *arg*`.buf` required: **IPC_STAT**, **IPC_SET**
- *arg*`.array` required: **GETALL**, **SETALL**
- *arg* ignored: **GETVAL, GETPID, GETNCNT, GETZCNT, IPC_RMID**

The details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, read "Using the semget System Call." It goes into more detail than would be practical for every system call.

Example Program

The example program presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **semctl** system call to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values.

This program begins (lines 5-10) by including the required header files as specified by the **semctl(2)** man page plus the **<errno.h>** header file, which is used for referencing errno.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are intended to be self-explanatory. These names make the program more readable and are valid because they are local to the program.

The variables declared for this program and their roles are:

semid_ds	receives the specified semaphore set identifier's data structure when an IPC_STAT control command is performed
c	receives the input values from the scanf function (line 119) when performing a SETALL control command
i	stores a counter value to increment through the union arg.array when displaying the semaphore values for a GETALL (lines 98-100) control command, and when initializing the arg.array when performing a SETALL (lines 117-121) control command
length	stores a variable value to test for the number of semaphores in a set against the i counter variable (lines 98, 117)
uid	stores the IPC_SET value for the user identification
gid	stores the IPC_SET value for the group identification
mode	stores the IPC_SET value for the operation permissions
retrn	stores the return value from the system call
semid	stores and passes the semaphore set identifier to the system call
semnum	stores and passes the semaphore number to the system call
cmd	stores the code for the desired control command so that subsequent processing can be performed on it
choice	used to determine which member (uid, gid, mode) for the IPC_SET control command is to be changed
semvals[]	stores the set of semaphore values when getting (GETALL) or initializing (SETALL)
arg.val	stores a value for the system call to set, or stores a value returned from the system call, for a single semaphore (union member)

`arg.buf` — a pointer passed to the system call which locates the data structure in the user memory area where the `IPC_STAT` control command is to place its return values, or where the `IPC_SET` command gets the values to set (union member)

`arg.array` — a pointer passed to the system call which locates the array in the user memory where the `GETALL` control command is to place its return values, or when the `SETALL` command gets the values to set (union member)

Note that the `semvals` array is declared to have 250 elements (0 through 249). This number corresponds to the usual maximum number of semaphores allowed per set (**SEMMSL**), a define located in **<linux/sem.h>**. The actual value of **SEMMSL** in use on the currently executing kernel may be viewed with the "**ipcs -s -l**" command.

First, the program prompts for a valid semaphore set identifier, which is stored in the `semid` variable (lines 24-26). This is required for all **semctl** system calls.

Next, the code for the desired control command must be entered (lines 17-42), and the code is stored in the `cmd` variable. The code is tested to determine the control command for subsequent processing.

If the `GETVAL` control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 48, 49). When it is entered, it is stored in the `semnum` variable (line 50). Then, the system call is performed and the semaphore value is displayed (lines 51-54). Note that the `arg` argument is not required in this case and the system call will ignore it. If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 197, 198); if the system call is unsuccessful, an error message is displayed along with the value of the external `errno` variable (lines 194, 195).

If the `SETVAL` control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 55, 56). When it is entered, it is stored in the `semnum` variable (line 57). Next, a message prompts for the value to which the semaphore is to be set; it is stored as the `arg.val` member of the union (lines 58, 59). Then, the system call is performed (lines 60, 62). Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETPID` control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 63-66), and the PID of the process performing the last operation is displayed. Note that the `arg` argument is not required in this case, and the system call will ignore it. Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETNCNT` control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 67-71). When entered, it is stored in the `semnum` variable (line 73). Then, the system call is performed and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 73-76). Note that the `arg` argument is not required in this case, and the system call will ignore it. Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETZCNT` control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 77-80). When it is entered, it is stored in the `semnum` variable (line 81). Then the system call is performed and the number of processes

waiting for the semaphore value to become equal to zero is displayed (lines 82-85). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 87-93). The length variable is set to the number of semaphores in the set (line 93). The arg.array union member is set to point to the semvals array where the system call is to store the values of the semaphore set (line 96). Then, a loop is entered which displays each element of the arg.array from zero to one less than the value of length (lines 98-104). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 107-110). The length variable is set to the number of semaphores in the set (line 113). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the semvals array to contain the desired values of the semaphore set (lines 115-121). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The arg.array union member is set to point to the semvals array from which the system call is to obtain the semaphore values. The system call is then made (lines 122-125). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 129), and the status information returned is printed out (lines 130-141); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the errno variable is printed out (line 194).

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 145-149). This is necessary because this example program provides for changing only one member at a time, and the **semctl** system call changes all of them. Also, if an invalid value is stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. Next, the program prompts for a code corresponding to the member to be changed (lines 150-156). This code is stored in the choice variable (line 157). Then, depending upon the member picked, the program prompts for the new value (lines 158-181). The value is placed into the appropriate member in the user memory area data structure and the system call is made (line 184). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 186-188). The semaphore set identifier, along with its associated data structure and semaphore set, is removed from the operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl** system call follows. This file is provided as **/usr/share/doc/ccur/examples/semctl.c**.

```
/*
 * Illustrates the semaphore control, semctl(),
 * system call capabilities
 */
/* Include necessary header files.* /
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include <errno.h>
/* Start of main C language program* /
main()
{
     struct semid_ds semid_ds;
     int c, i, length;
     int uid, gid, mode;
     int retrn, semid, semnum, cmd, choice;
     ushort semvals[250];
     union semun  {
         int val;
         struct semid_ds *buf;
         ushort *array;
     } arg;
     /* Enter the semaphore ID.* /
     printf("Enter the semid = ");
     scanf("%d", &semid);
     /* Choose the desired command.* /
     printf("\nEnter the number for\n");
     printf("the desired cmd:\n");
     printf("GETVAL      =  1\n");
     printf("SETVAL      =  2\n");
     printf("GETPID      =  3\n");
     printf("GETNCNT     =  4\n");
     printf("GETZCNT     =  5\n");
     printf("GETALL      =  6\n");
     printf("SETALL      =  7\n");
     printf("IPC_STAT    =  8\n");
     printf("IPC_SET     =  9\n");
     printf("IPC_RMID    =  10\n");
     printf("Entry       =  ");
     scanf("%d", &cmd);
     /* Check entries.* /
     printf ("\nsemid =%d, cmd = %d\n\n",
         semid, cmd);
     /* Set the command and do the call.* /
     switch (cmd)
     {
     case 1: /* Get a specified value.* /
         printf("\nEnter the semnum = ");
         scanf("%d", &semnum);
         /* Do the system call.* /
         retrn = semctl(semid, semnum, GETVAL, arg);
         printf("\nThe semval = %d", retrn);
         break;
     case 2: /* Set a specified value.* /
         printf("\nEnter the semnum = ");
         scanf("%d", &semnum);
         printf("\nEnter the value = ");
         scanf("%d", &arg.val);
         /* Do the system call.* /
         retrn = semctl(semid, semnum, SETVAL, arg);
         break;
     case 3: /* Get the process ID.* /
         retrn = semctl(semid, 0, GETPID, arg);
         printf("\nThe sempid = %d", retrn);
         break;
     case 4: /* Get the number of processes
                waiting for the semaphore to
```

```
 69                 become greater than its current
 70                 value.* /
 71             printf("\nEnter the semnum = ");
 72             scanf("%d", &semnum);
 73             /* Do the system call.* /
 74             retrn = semctl(semid, semnum, GETNCNT, arg);
 75             printf("\nThe semncnt = %d", retrn);
 76             break;
 77         case 5: /* Get the number of processes
 78                    waiting for the semaphore
 79                    value to become zero.* /
 80             printf("\nEnter the semnum = ");
 81             scanf("%d", &semnum);
 82             /* Do the system call.* /
 83             retrn = semctl(semid, semnum, GETZCNT, arg);
 84             printf("\nThe semzcnt = %d", retrn);
 85             break;
 86         case 6: /* Get all of the semaphores.* /
 87             /* Get the number of semaphores in
 88                the semaphore set.* /
 89             arg.buf = &semid_ds;
 90             retrn = semctl(semid, 0, IPC_STAT, arg);
 91             if(retrn == -1)
 92                 goto ERROR;
 93             length = arg.buf->sem_nsems;
 94             /* Get and print all semaphores in the
 95                specified set.* /
 96             arg.array = semvals;
 97             retrn = semctl(semid, 0, GETALL, arg);
 98             for (i = 0; i < length; i++)
 99             {
100                 printf("%d", semvals[i]);
101                 /* Separate each
102                    semaphore.* /
103                 printf(" ");
104             }
105             break;
106         case 7: /* Set all semaphores in the set.* /
107             /* Get the number of semaphores in
108                the set.* /
109             arg.buf = &semid_ds;
110             retrn = semctl(semid, 0, IPC_STAT, arg);
111             if(retrn == -1)
112                 goto ERROR;
113             length = arg.buf->sem_nsems;
114             printf("Length = %d\n", length);
115             /* Set the semaphore set values.* /
116             printf("\nEnter each value:\n");
117             for(i = 0; i < length ; i++)
118             {
119                 scanf("%d", &c);
120                 semvals[i] = c;
121             }
122             /* Do the system call.* /
123             arg.array = semvals;
124             retrn = semctl(semid, 0, SETALL, arg);
125             break;
126         case 8: /* Get the status for the semaphore set.* /
127             /* Get and print the current status values.* /
128             arg.buf = &semid_ds;
129             retrn = semctl(semid, 0, IPC_STAT, arg);
130             printf ("\nThe USER ID = %d\n",
131                 arg.buf->sem_perm.uid);
132             printf ("The GROUP ID = %d\n",
133                 arg.buf->sem_perm.gid);
134             printf ("The operation permissions = 0%o\n",
135                 arg.buf->sem_perm.mode);
136             printf ("The number of semaphores in set = %d\n",
137                 arg.buf->sem_nsems);
138             printf ("The last semop time = %d\n",
139                 arg.buf->sem_otime);
140             printf ("The last change time  = %d\n",
```

```
141                arg.buf->sem_ctime);
142             break;
143          case 9:    /* Select and change the desired
144                        member of the data structure.* /
145             /* Get the current status values.* /
146             arg.buf = &semid_ds;
147             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
148             if(retrn == -1)
149                goto ERROR;
150             /* Select the member to change.* /
151             printf("\nEnter the number for the\n");
152             printf("member to be changed:\n");
153             printf("sem_perm.uid   = 1\n");
154             printf("sem_perm.gid   = 2\n");
155             printf("sem_perm.mode  = 3\n");
156             printf("Entry          = ");
157             scanf("%d", &choice);
158             switch(choice){
159             case 1: /* Change the user ID.* /
160                printf("\nEnter USER ID = ");
161                scanf ("%d", &uid);
162                arg.buf->sem_perm.uid = uid;
163                printf("\nUSER ID = %d\n",
164                   arg.buf->sem_perm.uid);
165                break;
166             case 2: /* Change the group ID.* /
167                printf("\nEnter GROUP ID = ");
168                scanf("%d", &gid);
169                arg.buf->sem_perm.gid = gid;
170                printf("\nGROUP ID = %d\n",
171                   arg.buf->sem_perm.gid);
172                break;
173             case 3: /* Change the mode portion of
174                        the operation
175                        permissions.* /
176                printf("\nEnter MODE in octal = ");
177                scanf("%o", &mode);
178                arg.buf->sem_perm.mode = mode;
179                printf("\nMODE = 0%o\n",
180                   arg.buf->sem_perm.mode);
181                break;
182             }
183             /* Do the change.* /
184             retrn = semctl(semid, 0, IPC_SET, arg);
185             break;
186          case 10:    /* Remove the semid along with its
187                         data structure.* /
188             retrn = semctl(semid, 0, IPC_RMID, arg);
189          }
190          /* Perform the following if unsuccessful.* /
191          if(retrn == -1)
192          {
193 ERROR:
194          printf("\nsemctl failed!,error no.= %d\n", errno);
195          exit(0);
196       }
197       printf ("\n\nsemctl successful\n");
198       printf ("for semid = %d\n", semid);
199       exit (0);
200 }
```

Operations On Semaphores

This section describes how to use the **semop(2)** system call. The accompanying program illustrates its use.

Using the semop System Call

Synopsis

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

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

The **semop** system call returns an integer value, which is zero for successful completion or **-1** otherwise.

The *semid* argument must be a valid, non-negative, integer value. In other words, it must have already been returned from a prior **semget(2)** system call.

The *sops* argument points to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number (*sem_num*)
- the operation to be performed (*sem_op*)
- the control flags (*sem_flg*)

The *sops declaration means that either an array name (which is the address of the first element of the array) or a pointer to the array can be used. sembuf is the tag name of the data structure used as the template for the structure members in the array; it is located in the **<sys/sem.h>** header file.

The *nsops* argument specifies the length of the array (the number of structures in the array). The maximum size of this array is determined by the **SEMOPM** system-tunable parameter. Therefore, a maximum of **SEMOPM** operations can be performed for each **semop** system call.

The semaphore number (*sem_num*) determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- If *sem_op* is positive, the semaphore value is incremented by the value of *sem_op*.
- If *sem_op* is negative, the semaphore value is decremented by the absolute value of *sem_op*.
- If *sem_op* is zero, the semaphore value is tested for equality to zero.

The following operation commands (flags) can be used:

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

SEM_UNDO tells the system to undo the process' semaphore changes automatically when the process exits; it allows processes to avoid deadlock problems. To implement this feature, the system maintains a table with an entry for every process in the system. Each entry points to a set of undo structures, one for each semaphore used by the process. The system records the net change.

Example Program

The example program presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`semop`** system call to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values.

This program begins (lines 5-10) by including the required header files as specified by the **`semop(2)`** man page plus the **`<errno.h>`** header file, which is used for referencing `errno`.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are intended to be self-explanatory. These names make the program more readable and are valid because they are local to the program.

The variables declared for this program and their roles are:

`sembuf[10]` an array buffer (line 14) to contain a maximum of ten `sembuf` type structures; ten is the standard value of the tunable parameter **SEMOPM**, the maximum number of operations on a semaphore set for each **`semop`** system call

`sops` a pointer (line 14) to the `sembuf` array for the system call and for accessing the structure members within the array

`string[8]` a character buffer to hold a number entered by the user

`rtrn` stores the return value from the system call

`flags`	stores the code of the `IPC_NOWAIT` or `SEM_UNDO` flags for the **`semop`** system call (line 59)
`sem_num`	stores the semaphore number entered by the user for each semaphore operation in the array
`i`	a counter (line 31) for initializing the structure members in the array, and used to print out each structure in the array (line 78)
`semid`	stores the desired semaphore set identifier for the system call
`nsops`	specifies the number of semaphore operations for the system call; must be less than or equal to **SEMOPM**

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 18-21). `semid` is stored in the `semid` variable (line 22).

A message is displayed requesting the number of operations to be performed on this set (lines 24-26). The number of operations is stored in the `nsops` variable (line 27).

Next, a loop is entered to initialize the array of structures (lines 29-76). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (`nsops`) to be performed for the system call, so `nsops` is tested against the `i` counter for loop control. Note that `sops` is used as a pointer to each element (structure) in the array, and `sops` is incremented just like `i.` `sops` is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 77-84).

The `sops` pointer is set to the address of the array (lines 85, 86). `sembuf` could be used directly, if desired, instead of `sops` in the system call.

The system call is made (line 88), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **`semctl`** **GETALL** control command.

The example program for the **`semop`** system call follows. This file is provided as **`/usr/share/doc/ccur/examples/semop.c`**.

```
 1    /*
 2     * Illustrates the semaphore operations, semop(),
 3     * system call capabilities.
 4     */
 5    /* Include necessary header files. */
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/sem.h>
10    #include <errno.h>
11    /* Start of main C language program */
12    main()
13    {
14         struct sembuf sembuf[10], *sops;
15         char string[8];
16         int retrn, flags, sem_num, i, semid;
17         unsigned nsops;
18         /* Enter the semaphore ID. */
19         printf("\nEnter the semid of\n");
```

```
20          printf("the semaphore set to\n");
21          printf("be operated on = ");
22          scanf("%d", &semid);
23          printf("\nsemid = %d", semid);
24          /* Enter the number of operations. */
25          printf("\nEnter the number of semaphore\n");
26          printf("operations for this set = ");
27          scanf("%d", &nsops);
28          printf("\nsops = %d", nsops);
29          /* Initialize the array for the
30             number of operations to be performed. */
31          for(i = 0, sops = sembuf; i < nsops; i++, sops++)
32          {
33               /* This determines the semaphore in
34                  the semaphore set. */
35               printf("\nEnter the semaphore\n");
36               printf("number (sem_num) = ");
37               scanf("%d", &sem_num);
38               sops->sem_num = sem_num;
39               printf("\nThe sem_num = %d", sops->sem_num);
40               /* Enter a (-)number to decrement,
41                  an unsigned number (no +) to increment,
42                  or zero to test for zero.  These values
43                  are entered into a string and converted
44                  to integer values. */
45               printf("\nEnter the operation for\n");
46               printf("the semaphore (sem_op) = ");
47               scanf("%s", string);
48               sops->sem_op = atoi(string);
49               printf("\nsem_op = %d\n", sops->sem_op);
50               /* Specify the desired flags. */
51               printf("\nEnter the corresponding\n");
52               printf("number for the desired\n");
53               printf("flags:\n");
54               printf("No flags                  = 0\n");
55               printf("IPC_NOWAIT                = 1\n");
56               printf("SEM_UNDO                  = 2\n");
57               printf("IPC_NOWAIT and SEM_UNDO   = 3\n");
58               printf("            Flags         = ");
59               scanf("%d", &flags);
60               switch(flags)
61               {
62               case 0:
63                    sops->sem_flg = 0;
64                    break;
65               case 1:
66                    sops->sem_flg = IPC_NOWAIT;
67                    break;
68               case 2:
69                    sops->sem_flg = SEM_UNDO;
70                    break;
71               case 3:
72                    sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
73                    break;
74               }
75               printf("\nFlags = 0%o\n", sops->sem_flg);
76          }
77          /* Print out each structure in the array. */
78          for(i = 0; i < nsops; i++)
79          {
80               printf("\nsem_num = %d\n", sembuf[i].sem_num);
81               printf("sem_op = %d\n", sembuf[i].sem_op);
82               printf("sem_flg = 0%o\n", sembuf[i].sem_flg);
83               printf(" ");
84          }
```

```
85          sops = sembuf; /* Reset the pointer to
86                            sembuf[0]. */
87          /* Do the semop system call. */
88          retrn = semop(semid, sops, nsops);
89          if(retrn == -1)  {
90               printf("\nSemop failed, error = %d\n", errno);
91          }
92          else {
93               printf ("\nSemop was successful\n");
94               printf("for semid = %d\n", semid);
95               printf("Value returned = %d\n", retrn);
96          }
97     }
```

Condition Synchronization

The following sections describe the **postwait(2)** and **server_block**/**server_wake(2)** system calls that can be used to manipulate cooperating processes.

Using the postwait System Call

The **postwait(2)** function is a fast, efficient, sleep/wakeup/timer mechanism used between a cooperating group of processes.

To go to sleep, a process calls **pw_wait()**. The process will wake up when:

- the timer expires
- the process is posted to by another process in the cooperating group calling **pw_post()**
- the call is interrupted

Synopsis

```
#include <sys/time.h>
#include <sys/rescntl.h>
#include <sys/pw.h>

int pw_getukid(ukid_t *ukid);
int pw_wait(struct timespec *t, struct resched_var  *r);
int pw_post(ukid_t ukid, struct resched_var *r);
int pw_postv(int count, ukid_t targets[], int errors[], struct
resched_var *r );
int pw_getvmax(void);
```

gcc [*options*] *file* **-lccur_rt** ...

Processes using **postwait(2)** are identified by their ukid. A process should call **pw_getukid()** to obtain its ukid. The ukid maps to a pid. This value should then be shared with the other cooperating processes that may wish to post to this process.

pw_wait() returns a value of 1 if posted or 0 if timed-out. **pw_wait()** is called with a timeout value and a rescheduling value. If the caller specifies a rescheduling value (that is, the value is not NULL), its lock-count is decremented.

If the timeout value is NULL, the process will not timeout. If the timeout value is 0, **pw_wait()** immediately returns with a 1 if posted, or returns an EAGAIN in all other cases.

If the time specified for the timeout value is greater than 0, the process sleeps for that amount of time unless it is posted to before it times out. The timeout value is updated to reflect the amount of time remaining if the process is posted to or interrupted.

If one or more **pw_post** or **pw_postv** operations have occurred prior to a **pw_wait**, the **pw_wait** call does not block the calling process. A subsequent **pw_wait** without any intervening **pw_post** or **pw_postv** calls will block if the time specified for the timeout is greater than zero.

pw_postv() can be used to post to multiple processes at once. It attempts to post to all processes specified in *targets*. The default (maximum) number of targets to be posted by **pw_postv()** is 64 as determined by the CONFIG_PW_VMAX kernel tunable accessible through the General Setup selection of the Linux Kernel Configuration menu (refer to the "Configuring and Building the Kernel" chapter). Errors for respective targets are returned in the *errors* array. **pw_postv()** returns a 0 if all succeed, or the error value of the last target to cause an error if there are any errors.

If the caller specifies a rescheduling variable (that is, the rescheduling variable is not NULL), its lock-count is decremented.

pw_getvmax() returns the maximum number of targets that can be posted to with one **pw_postv()** call.

Refer to the **postwait(2)** man page for a listing of the types of errors that may occur.

Using the Server System Calls

This set of system calls enables you to manipulate processes acting as servers using an interface compatible with the PowerMAX operating system. These system calls are briefly described as follows:

server_block blocks the calling process only if no wake-up request has occurred since the last return from **server_block**. If a wake-up has occurred, **server_block** returns immediately.

server_wake1 wakes server if it is blocked in the **server_block** system call; if the specified server is not blocked in this call, the wake-up request is applied to the server's next call to **server_block**.

server_wakevec serves the same purpose as **server_wake1**, except that a vector of processes may be specified rather than one process.

CAUTION

These system calls should be used only by single-threaded processes. The global process ID of a multiplexed thread changes according to the process on which the thread is currently scheduled. Therefore, it is possible that the wrong thread will be awakened or blocked when these interfaces are used by multiplexed threads.

server_block

server_block blocks the calling process only if no wake-up request has occurred since the last return from **server_block**.

Synopsis

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/pw.h>

int server_block(options, r, timeout)
int options;
struct resched_var *r;
struct timeval *timeout;

gcc [options] file -lccur_rt ...
```

Arguments are defined as follows:

options the value of this argument must be zero

r a pointer to the calling process' rescheduling variable. This argument is optional: its value can be NULL.

timeout a pointer to a `timeval` structure that contains the maximum length of time the calling process will be blocked. This argument is optional: its value can be NULL. If its value is NULL, there is no time out.

The **server_block** system call returns immediately if the calling process has a pending wake-up request; otherwise, it returns when the calling process receives the next wake-up request. A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Note that upon return, the calling process should retest the condition that caused it to block; there is no guarantee that the condition has changed because the process could have been prematurely awakened by a signal.

server_wake1

Server_wake1 is invoked to wake a server that is blocked in the **server_block** call.

Synopsis

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/pw.h>

int server_wake1(server, r)
global_lwpid_t server;
struct resched_var *r;
```

gcc [*options*] *file* **-lccur_rt ...**

Arguments are defined as follows:

server	the global process ID of the server process to be awakened
r	a pointer to the calling process' rescheduling variable. This argument is optional; its value can be NULL.

It is important to note that to use the **server_wake1** call, the real or effective user ID of the calling process must match the real or saved [from **exec**] user ID of the process specified by *server.*

Server_wake1 wakes the specified server if it is blocked in the **server_block** call. If the server is not blocked in this call, the wake-up request is held for the server's next call to **server_block**. **Server_wake1** also decrements the number of rescheduling locks associated with the rescheduling variable specified by *r.*

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **errno** is set to indicate the error.

server_wakevec

The **server_wakevec** system call is invoked to wake a group of servers blocked in the **server_block** call.

Synopsis

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/pw.h>

int server_wakevec(servers, nservers, r)
global_lwpid_t *servers;
int nservers;
struct resched_var *r;

gcc [options] file -lccur_rt ...
```

Arguments are defined as follows:

servers	a pointer to an array of the global process IDs of the server processes to be awakened
nservers	an integer value specifying the number of elements in the array
r	a pointer to the calling process' rescheduling variable. This argument is optional; its value can be NULL.

It is important to note that to use the **server_wakevec** call, the real or effective user ID of the calling process must match the real or saved [from **exec**] user IDs of the processes specified by *servers.*

Server_wakevec wakes the specified servers if they are blocked in the **server_block** call. If a server is not blocked in this call, the wake-up request is applied to the server's next call to **server_block**. **Server_wakevec** also decrements the number of rescheduling locks associated with a rescheduling variable specified by *r.*

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **errno** is set to indicate the error.

For additional information on the use of these calls, refer to the **server_block(2)** man page.

Applying Condition Synchronization Tools

The rescheduling variable, spin lock, and server system calls can be used to design functions that enable a producer and a consumer process to exchange data through use of a mailbox in a shared memory region. When the consumer finds the mailbox empty, it blocks until new data arrives. After the producer deposits new data in the mailbox, it wakes the waiting consumer. An analogous situation occurs when the producer generates data faster than the consumer can process it. When the producer finds the mailbox full, it blocks until the data is removed. After the consumer removes the data, it wakes the waiting producer.

A mailbox can be represented as follows:

```
struct mailbox {
    struct spin_mutex mx;/* serializes access to mailbox */
    queue_of consumers:   /* waiting consumers */
    queue_of data;        /* the data, type varies */
};
```

The `mx` field is used to serialize access to the mailbox. The `data` field represents the information that is being passed from the producer to the consumer. The `full` field is used to indicate whether the mailbox is full or empty. The `producer` field identifies the process that is waiting for the mailbox to be empty. The `consumer` field identifies the process that is waiting for the arrival of data.

Using the **`spin_acquire`** and the **`spin_release`** functions, a function to enable the consumer to extract data from the mailbox can be defined as follows:

```
void
consume (box, data)
    struct mailbox *box;
    any_t *data;
{
    spin_acquire (&box->mx, &rv);
    while (box->data == empty) {
        enqueue (box->consumers, rv.rv_glwpid);
        spin_unlock (&box->mx);
        server_block (0, &rv, 0);
        spin_acquire (&box->mx, &rv);
    }
    *data = dequeue (box->data;
    spin_release (&box->mx, &rv);
}
```

Note that in this function, the consumer process locks the mailbox prior to checking for and removing data. If it finds the mailbox empty, it unlocks the mailbox to permit the producer to deposit data, and it calls **`server_block`** to wait for the arrival of data. When the consumer is awakened, it must again lock the mailbox and check for data; there is no guarantee that the mailbox will contain data—the consumer may have been awakened prematurely by a signal.

A similar function that will enable the producer to place data in the mailbox can be defined as follows:

```
void
produce (box, data)
    struct mailbox *box;
    any_t data;
{
    spin_acquire (&box->mx, &rv);
    enqueue (box->data, data);
    if (box->consumer == empty)
            spin_release (&box->mx, &rv);
    else {
            global_lwpid_t id = dequeue (box->consumers);
            spin_unlock (&box->mx);
            server_wake1 (id, &rv);
    }
}
```

In this function, the producer process waits for the mailbox to empty before depositing new data. The producer signals the arrival of data only when the consumer is waiting; note that it does so after unlocking the mailbox. The producer must unlock the mailbox first so that the awakened consumer can lock it to check for and remove data. Unlocking the mailbox prior to the call to **`server_wake1`** also ensures that the mutex is held for a short time. To prevent unnecessary context switching, rescheduling is disabled until the consumer is awakened.

6
Programmable Clocks and Timers

6
Programmable Clocks and Timers

This chapter provides an overview of some of the facilities that can be used for timing. The POSIX clocks and timers interfaces are based on IEEE Standard 1003.1b-1993. The clock interfaces provide a high-resolution clock, which can be used for such purposes as time stamping or measuring the length of code segments. The timer interfaces provide a means of receiving a signal or process wakeup asynchronously at some future time. In addition, high-resolution system calls are provided which can be used to put a process to sleep for a very short time quantum and specify which clock should be used for measuring the duration of the sleep. Additional clocks and timers are provided by the RCIM PCI card.

Understanding Clocks and Timers

Real-time applications must be able to operate on data within strict timing constraints in order to schedule application or system events. High resolution clocks and timers allow applications to use relative or absolute time based on a high resolution clock and to schedule events on a one-shot or periodic basis. Applications can create multiple timers for each process.

Several timing facilities are available on the iHawk system. These include POSIX clocks and timers under RedHawk Linux as well as non-interrupting clocks and real-time clock timers provided by the Real-Time Clock and Interrupt Module (RCIM) PCI card. These clocks and timers and their interfaces are explained in the sections that follow.

See Chapter 7 for information about system clocks and timers.

RCIM Clocks and Timers

The Real-Time Clock and Interrupt Module (RCIM) provides two non-interrupting clocks. These clocks can be synchronized with other RCIMs when the RCIMs are chained together. The RCIM clocks are:

primary (tick) clock — a 64-bit non-interrupting clock that increments by one on each tick of the common 400ns clock signal. This clock can be reset to zero and synchronized across the RCIM chain providing a common time stamp.

The primary clock can be read on any system, master or slave, using direct reads when the device file **`/dev/rcim/sclk`** is mapped into the address space of a program.

secondary (POSIX) — a 64-bit non-interrupting counter encoded in POSIX 1003.1 format. The upper 32 bits contain seconds and the lower 32 bits contain nanoseconds. This clock is incremented by 400 on each

tick of the common 400ns clock signal. Primarily used as a high-resolution local clock.

The secondary clock is accessed in a manner similar to the primary clock in that the same utilities and device files are used. The secondary clock can be loaded with any desired time; however, the value loaded is not synchronized with other clocks in an RCIM chain. Only the secondary clock of the RCIM attached to the host is updated.

The RCIM also provides four real-time clock (RTC) timers. Each of these counters is accessible using a special device file and each can be used for almost any timing/frequency control function. They are programmable to several different resolutions which, when combined with a clock count value, provide a variety of timing intervals. This makes them ideal for running processes at a given frequency (e.g., 100Hz) or for timing code segments. In addition to being able to generate an interrupt on the host system, the output of an RTC can be distributed to other RCIM boards for delivery to their corresponding host systems, or delivered to external equipment attached to one of the RCIMs external output interrupt lines. The RTC timers are controlled by **open(2)**, **close(2)** and **ioctl(2)** system calls.

For complete information about the RCIM clocks and timers, refer to the *Real-Time Clock and Interrupt Module (RCIM) User's Guide* manual.

POSIX Clocks and Timers

The POSIX clocks provide a high-resolution mechanism for measuring and indicating time. The following system-wide POSIX clocks are available:

CLOCK_REALTIME
CLOCK_REALTIME_HR
the system time-of-day clock, defined in the file <**time.h**>. This clock supplies the time used for file system creation and modification, accounting and auditing records, and IPC message queues and semaphores. **CLOCK_REALTIME_HR** is obsoleted by the fact that both clocks have 1 microsecond resolution, share the same characteristics and are operated on simultaneously. In addition to the POSIX clock routines described in this chapter, the following commands and system calls read and set this clock: **date(1)**, **gettimeofday(2)**, **settimeofday(2)**, **stime(2)**, **time(1)** and **adjtimex(2**).

CLOCK_MONOTONIC
CLOCK_MONOTONIC_HR
the system uptime clock measuring the time in seconds and nanoseconds since the system was booted. The monotonic clocks cannot be set. **CLOCK_MONOTONIC_HR** is obsoleted by the fact that both clocks have 1 microsecond resolution, share the same characteristics and are operated on simultaneously.

There are two types of timers: one-shot and periodic. They are defined in terms of an initial expiration time and a repetition interval. The initial expiration time indicates when the timer will first expire. It may be absolute (for example, at 8:30 a.m.) or relative to the current time (for example, in 30 seconds). The repetition interval indicates the amount of time that will elapse between one expiration of the timer and the next. The clock to be used for timing is specified when the timer is created.

A one-shot timer is armed with either an absolute or a relative initial expiration time and a repetition interval of zero. It expires only once--at the initial expiration time--and then is disarmed.

A periodic timer is armed with either an absolute or a relative initial expiration time and a repetition interval that is greater than zero. The repetition interval is always relative to the time at the point of the last timer expiration. When the initial expiration time occurs, the timer is reloaded with the value of the repetition interval and continues counting. The timer may be disarmed by setting its initial expiration time to zero.

The local timer is used as the interrupt source for scheduling POSIX timer expiries. See Chapter 7 for information about the local timer.

NOTE

Access to high resolution clocks and timers is provided by a set of related POSIX system calls located within **`/usr/lib/libccur_rt.a`** . Some of the timer functions are also provided as low-resolution by the standard gnu libc **`librt.so`** library.

Understanding the POSIX Time Structures

The POSIX routines related to clocks and timers use two structures for time specifications: the `timespec` structure and the `itimerspec` structure. These structures are defined in the file **`<time.h>`**.

The `timespec` structure specifies a single time value in seconds and nanoseconds. You supply a pointer to a `timespec` structure when you invoke routines to set the time of a clock or obtain the time or resolution of a clock (for information on these routines, see "Using the POSIX Clock Routines"). The structure is defined as follows:

```
struct timespec {
       time_t  tv_sec;
       long    tv_nsec;
};
```

The fields in the structure are described as follows:

`tv_sec`	specifies the number of seconds in the time value
`tv_nsec`	specifies the number of additional nanoseconds in the time value. The value of this field must be in the range zero to 999,999,999.

The `itimerspec` structure specifies the initial expiration time and the repetition interval for a timer. You supply a pointer to an `itimerspec` structure when you invoke routines to set the time at which a timer expires or obtain information about a timer's expiration time (for information on these routines, see "Using the POSIX Timer Routines"). The structure is defined as follows:

```
struct itimerspec {
       struct timespec it_interval;
       struct timespec it_value;
};
```

The fields in the structure are described as follows.

`it_interval`	specifies the repetition interval of a timer
`it_value`	specifies the timer's initial expiration

Using the POSIX Clock Routines

The POSIX routines that allow you to perform a variety of functions related to clocks are briefly described as follows:

`clock_settime`	sets the time of a specified clock
`clock_gettime`	obtains the time from a specified clock
`clock_getres`	obtains the resolution in nanoseconds of a specified clock

Procedures for using each of these routines are explained in the sections that follow.

Using the clock_settime Routine

The **`clock_settime(2)`** system call allows you to set the time of the system time-of-day clock, **CLOCK_REALTIME**. The calling process must have root or the **CAP_SYS_NICE** capability. By definition, the **CLOCK_MONOTONIC** clocks cannot be set.

It should be noted that if you set **CLOCK_REALTIME** after system start-up, the following times may not be accurate:

- file system creation and modification times
- times in accounting and auditing records
- the expiration times for kernel timer queue entries

Setting the system clock does not affect queued POSIX timers.

Synopsis

```
#include <time.h>

int clock_settime(clockid_t which_clock,
const struct timespec *setting);
```

`gcc` [*options*] *file* **`-lccur_rt`** ...

The arguments are defined as follows:

which_clock the identifier for the clock for which the time will be set. The value of *which_clock* cannot be either **CLOCK_MONOTONIC** or **CLOCK_MONOTONIC_HR**. Note that setting either **CLOCK_REALTIME** or **CLOCK_REALTIME_HR** changes both.

setting a pointer to a structure that specifies the time to which *which_clock* is to be set. When *which_clock* is **CLOCK_REALTIME**, the time-of-day clock is set to a new value. Time values that are not integer multiples of the clock resolution are truncated down.

A return value of **0** indicates that the specified clock has been successfully set. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **clock_settime(2)** man page for a listing of the types of errors that may occur.

Using the clock_gettime Routine

The **clock_gettime(2)** system call allows you to obtain the time from a specified clock.

Synopsis

```
#include <time.h>

int clock_gettime(clockid_t *which_clock, struct timespec
*setting);

gcc [options] file -lccur_rt ...
```

The arguments are defined as follows:

which_clock the identifier for the clock from which to obtain the time. The value of *which_clock* may be any of the following: **CLOCK_REALTIME, CLOCK_REALTIME_HR, CLOCK_MONOTONIC** or **CLOCK_MONOTONIC_HR**.

setting a pointer to a structure where the time of *which_clock* is returned.

A return value of **0** indicates that the call to **clock_gettime** has been successful. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **clock_gettime(2)** man page for a listing of the types of errors that may occur.

Using the clock_getres Routine

The **clock_getres(2)** system call allows you to obtain the resolution in nanoseconds of a specified clock.

The clock resolutions are system dependent and cannot be set by the user.

Synopsis

```
#include <time.h>

int clock_getres(clockid_t *which_clock, struct timespec
*resolution);

gcc [options] file -lccur_rt ...
```

The arguments are defined as follows:

which_clock the identifier for the clock for which you wish to obtain the resolution. *which_clock* may be any of the following: **CLOCK_REALTIME**, **CLOCK_REALTIME_HR, CLOCK_MONOTONIC** or **CLOCK_MONOTONIC_HR.**

resolution a pointer to a structure where the resolution of *which_clock* is returned

A return value of **0** indicates that the call to **clock_getres** has been successful. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **clock_getres(2)** man page for a listing of the types of errors that may occur.

Using the POSIX Timer Routines

Processes can create, remove, set, and query timers and may receive notification when a timer expires.

The POSIX system calls that allow you to perform a variety of functions related to timers are briefly described as follows:

timer_create creates a timer using a specified clock

timer_delete removes a specified timer

timer_settime arms or disarms a specified timer by setting the expiration time

timer_gettime obtains the repetition interval for a specified timer and the time remaining until the timer expires

timer_getoverrun	obtains the overrun count for a specified periodic timer
nanosleep	pauses execution for a specified time
clock_nanosleep	provides a higher resolution pause based on a specified clock

Procedures for using each of these system calls are explained in the sections that follow.

Using the timer_create Routine

The **timer_create(2)** system call allows the calling process to create a timer using a specified clock as the timing source.

A timer is disarmed when it is created. It is armed when the process invokes the **timer_settime(2)** system call (see "Using the timer_settime Routine" for an explanation of this system call).

It is important to note the following:

- When a process invokes the **fork** system call, the timers that it has created are not inherited by the child process.
- When a process invokes the **exec** system call, the timers that it has created are disarmed and deleted.

Linux threads in the same thread group can share timers. The thread which calls **timer_create** will receive all of the signals, but other threads in the same threads group can manipulate the timer through calls to **timer_settime(2)**.

Synopsis

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

int timer_create(clockid_t which_clock, struct sigevent
*timer_event_spec, timer_t created_timer_id);

gcc [options] file -lccur_rt ...
```

The arguments are defined as follows:

which_clock the identifier for the clock to be used for the timer. The value of *which_clock* must be **CLOCK_REALTIME** or **CLOCK_REALTIME_HR**.

timer_event_spec
the null pointer constant or a pointer to a structure that specifies the way in which the calling process is to be asynchronously notified of the expiration of the timer:

NULL **SIGALRM** is sent to the process when the timer expires.

sigev_notify=**SIGEV_SIGNAL**
: a signal specified by *sigev_signo* is sent to the process when the timer expires.

sigev_notify=**SIGEV_THREAD**
: the specified *sigev_notify* function is called in a new thread with *sigev_value* as the argument when the timer expires.

sigev_notify=**SIGEV_THREAD_ID**
: the *sigev_notify_thread_id* number should contain the `pthread_t` id of the thread that is to receive the signal *sigev_signo* when the timer expires.

sigev_notify=**SIGEV_NONE**
: no notification is delivered when the timer expires

NOTE

The signal denoting expiration of the timer may cause the process to terminate unless it has specified a signal-handling system call. To determine the default action for a particular signal, refer to the **`signal(2)`** man page.

created_timer_id
: a pointer to the location where the timer ID is stored. This identifier is required by the other POSIX timer system calls and is unique within the calling process until the timer is deleted by the **`timer_delete(2)`** system call.

A return value of **`0`** indicates that the call to **`timer_create`** has been successful. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`timer_create(2)`** man page for a listing of the types of errors that may occur.

Using the timer_delete Routine

The **`timer_delete(2)`** system call allows the calling process to remove a specified timer. If the selected timer is already started, it will be disabled and no signals or actions assigned to the timer will be delivered or executed. A pending signal from an expired timer, however, will not be removed.

Synopsis

```
#include <time.h>

int timer_delete(timer_t timer_id);

gcc [options] file -lccur_rt ...
```

The argument is defined as follows:

timer_id the identifier for the timer to be removed. This identifier comes from a previous call to **`timer_create(2)`** (see "Using the timer_create Routine" for an explanation of this system call).

A return value of **0** indicates that the specified timer has been successfully removed. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **timer_delete(2)** man page for a listing of the types of errors that may occur.

Using the timer_settime Routine

The **timer_settime(2)** system call allows the calling process to arm a specified timer by setting the time at which it will expire. The time to expire is defined as absolute or relative. A calling process can use this system call on an armed timer to (1) disarm the timer or (2) reset the time until the next expiration of the timer.

Synopsis

```
#include <time.h>

int timer_settime(timer_t timer_id, int flags, const struct
itimerspec *new_setting, const struct itimerspec *old_setting);
```

gcc [*options*] *file* **-lccur_rt** ...

The arguments are defined as follows:

timer_id — the identifier for the timer to be set. This identifier comes from a previous call to **timer_create(2)** (see "Using the timer_create Routine" for an explanation of this system call).

flags — an integer value that specifies one of the following:

- **TIMER_ABSTIME** — causes the selected timer to be armed with an absolute expiration time. The timer will expire when the clock associated with the timer reaches the value specified by *it_value*. If this time has already passed, **timer_settime** succeeds, and the timer-expiration notification is made.
- 0 — causes the selected timer to be armed with a relative expiration time. The timer will expire when the clock associated with the timer reaches the value specified by *it_value*.

new_setting — a pointer to a structure that contains the repetition interval and the initial expiration time of the timer.

If you wish to have a one-shot timer, specify a repetition interval (*it_interval*) of zero. In this case, the timer expires once, when the initial expiration time occurs, and then is disarmed.

If you wish to have a periodic timer, specify a repetition interval (*it_interval*) that is not equal to zero. In this case, when the initial expiration time occurs, the timer is reloaded with the value of the repetition interval and continues to count.

In either case, you may set the initial expiration time to a value that is absolute (for example, at 3:00 p.m.) or relative to the current time (for example, in 30 seconds). To set the initial expiration time to an absolute time, you must have set the **TIMER_ABSTIME** bit in the *flags* argument. Any signal that is already pending due to a previous timer expiration for the specified timer will still be delivered to the process.

To disarm the timer, set the initial expiration time to zero. Any signal that is already pending due to a previous timer expiration for this timer will still be delivered to the process.

old_setting the null pointer constant or a pointer to a structure to which the previous repetition interval and initial expiration time of the timer are returned. If the timer has been disarmed, the value of the initial expiration time is zero. The members of *old_setting* are subject to the resolution of the timer and are the same values that would be returned by a **timer_gettime(2)** call at that point in time.

A return value of **0** indicates that the specified timer has been successfully set. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **timer_settime(2)** man page for a listing of the types of errors that may occur.

Using the timer_gettime Routine

The **timer_gettime(2)** system call allows the calling process to obtain the repetition interval for a specified timer and the amount of time remaining until the timer expires.

Synopsis

```
#include <time.h>

int timer_gettime(timer_t timer_id, struct itimerspec
*setting);
```

gcc [*options*] *file* **-lccur_rt** ...

The arguments are defined as follows:

timer_id the identifier for the timer whose repetition interval and time remaining are requested. This identifier comes from a previous call to **timer_create(2)** (see "Using the timer_create Routine" for an explanation of this system call).

setting a pointer to a structure to which the repetition interval and the amount of time remaining on the timer are returned. The amount of time remaining is relative to the current time. If the timer is disarmed, the value is zero.

A return value of **0** indicates that the call to **timer_gettime** has been successful. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **timer_gettime(2)** man page for a listing of the types of errors that may occur.

Using the timer_getoverrun Routine

The **timer_getoverrun(2)** system call allows the calling process to obtain the overrun count for a particular periodic timer. A timer may expire faster than the system can deliver signals to the application. If a signal is still pending from a previous timer expiration rather than queuing another signal, a count of missed expirations is maintained with the pending signal. This is the overrun count.

Timers may overrun because the signal was blocked by the application or because the timer was over-committed.

Assume that a signal is already queued or pending for a process with a timer using timer-expiration notification **SIGEV_SIGNAL**. If this timer expires while the signal is queued or pending, a timer overrun occurs, and no additional signal is sent.

NOTE

You must invoke this system call from the timer-expiration signal-handling. If you invoke it outside this system call, the overrun count that is returned is not valid for the timer-expiration signal last taken.

Synopsis

```
#include <time.h>

int timer_getoverrun(timer_t timer_id);

gcc [options] file -lccur_rt ...
```

The argument is defined as follows:

timer_id — the identifier for the periodic timer for which you wish to obtain the overrun count. This identifier comes from a previous call to **timer_create(2)** (see "Using the timer_create Routine" for an explanation of this system call).

If the call is successful, **timer_getoverrun** returns the overrun count for the specified timer. This count cannot exceed **DELAYTIMER_MAX** in the file **<limits.h>**. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **timer_getoverrun(2)** man page for a listing of the types of errors that may occur.

Using the POSIX Sleep Routines

The **nanosleep(2)** and the **clock_nanosleep(2)** POSIX system calls provide a high-resolution sleep mechanism that causes execution of the calling process or thread to be suspended until (1) a specified period of time elapses or (2) a signal is received and the associated action is to execute a signal-handling system call or terminate the process.

The **clock_nanosleep(2)** system call provides a high-resolution sleep with a specified clock. It suspends execution of the currently running thread until the time specified by *rqtp* has elapsed or until the thread receives a signal.

The use of these system calls has no effect on the action or blockage of any signal.

Using the nanosleep Routine

Synopsis

```
#include <time.h>

int nanosleep(const struct timespec *req, struct timespec
*rem);

gcc [options] file -lccur_rt ...
```

Arguments are defined as follows:

req — a pointer to a `timespec` structure that contains the length of time that the process is to sleep. The suspension time may be longer than requested because the *req* value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. Except for the case of being interrupted by a signal, the suspension time will not be less than the time specified by *req*, as measured by **CLOCK_REALTIME**. You will obtain a resolution of one microsecond on the blocking request.

rem — the null pointer constant or a pointer to a `timespec` structure to which the amount of time remaining in the sleep interval is returned if **nanosleep** is interrupted by a signal. If *rem* is **NULL** and **nanosleep** is interrupted by a signal, the time remaining is not returned.

A return value of **0** indicates that the requested period of time has elapsed. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **nanosleep(2)** man page for a listing of the types of errors that may occur.

Using the clock_nanosleep Routine

Synopsis

```
#include <time.h>

int clock_nanosleep(clockid_t *which_clock, int flags,
const struct timespec *rqtp, struct timespec *rmtp);

gcc [options] file -lccur_rt ...
```

The arguments are defined as follows:

which_clock the identifier for the clock to be used. The value of *which_clock* may be **CLOCK_REALTIME**, **CLOCK_REALTIME_HR, CLOCK_MONOTONIC,** or **CLOCK_MONOTONIC_HR**.

flags an integer value that specifies one of the following:

- **TIMER_ABSTIME** interprets the time specified by *rqtp* to be absolute with respect to the clock value specified by *which_clock*.
- 0 interprets the time specified by *rqtp* to be relative to the current time.

rqtp a pointer to a `timespec` structure that contains the length of time that the process is to sleep. If the **TIMER_ABSTIME** flag is specified and the time value specified by *rqtp* is less than or equal to the current time value of the specified clock (or the clock's value is changed to such a time), the function will return immediately. Further, the time slept is affected by any changes to the clock after the call to **clock_nanosleep(2)**. That is, the call will complete when the actual time is equal to or greater than the requested time no matter how the clock reaches that time, via setting or actual passage of time or some combination of these.

The time slept may be longer than requested as the specified time value is rounded up to an integer multiple of the clock resolution, or due to scheduling and other system activity. Except for the case of interruption by a signal, the suspension time is never less than requested.

rmtp If **TIMER_ABSTIME** is not specified, the `timespec` structure pointed to by *rmtp* is updated to contain the amount of time remaining in the interval (i.e., the requested time minus the time actually slept). If *rmtp* is NULL, the remaining time is not set. The *rmtp* value is not set in the case of an absolute time value.

On success, **clock_nanosleep** returns a value of **0** after at least the specified time has elapsed. On failure, **clock_nanosleep** returns the value **-1** and **errno** is set to indicate the error. Refer to the **clock_nanosleep(2)** man page for a listing of the types of errors that may occur.

/proc Interface to POSIX Timers

For most applications, the default resolution for the POSIX timers and nanosleep functionality should be acceptable. If an application has problems with the way the timing occurs and it is prohibitive to change the application, or when it is desirable to group expiries together, adjustments may be appropriate. The kernel interface to POSIX timers is through the **/proc** file system. The files listed below control the resolution of POSIX timers and nanosleep functionality and can be used to limit the rate at which timers expire. The files are in the directory **/proc/sys/kernel/posix-timers**:

max_expiries The maximum number of expiries to process from a single interrupt. The default is 20.

recovery_time The time in nanoseconds to delay before processing more timer expiries if the **max_expiries** limit is hit. The default is 100000.

min_delay The minimum time between timer interrupts in nanoseconds. This ensures that the timer interrupts do not consume all of the CPU time. The default is 10000.

nanosleep_res The resolution of **nanosleep(2)** in nanoseconds. The default is 1000.

resolution The resolution of other POSIX timer functions including **clock_nanosleep(2)**. The default is 1000.

7
System Clocks and Timers

7
System Clocks and Timers

This chapter describes the local timer and global timer. It also discusses the effect of disabling the local timer on system functions.

Local Timer

On Concurrent's iHawk systems, each CPU has a local (private) timer which is used as a source of periodic interrupts local to that CPU. By default these interrupts occur 100 times per second and are staggered in time so that only one CPU is processing a local timer interrupt at a time.

The local timer interrupt routine performs the following local timing functions, which are explained in more detail in the sections that follow:

- gathers CPU utilization statistics, used by `top(1)` and other utilities
- causes the process running on the CPU to periodically consume its time quantum
- causes the running process to release the CPU in favor of another running process when its time quantum is used up
- periodically balances the load of runnable processes across CPUs
- implements process and system profiling
- implements system time-of-day (wall) clock and execution time quota limits for those processes that have this feature enabled
- provides the interrupt source for POSIX timers

The local timer interrupt can be disabled on a per CPU basis. This improves both the worst-case interrupt response time and the determinism of program execution on the CPU as described in the "Real-Time Performance" chapter. However, disabling the local timer interrupt has an effect on some functionality normally provided by RedHawk Linux. These effects are described below.

Functionality

The local timer performs the functions described in the sections below. The effect of disabling the local timer is discussed as well as viable alternatives for some of the features.

CPU Accounting

The local timer is used to accumulate the user and system execution time accounted for on a per process basis. This execution time accounting is reported by various system features such as **times(2)**, **wait4(2)**, **sigaction(2)**, **uptime(1)**, **w(1)**, **getrusage(2)**, **mpstat(1)**, **clock(3)**, **acct(2)**, **ps(1)**, **top(1)** and **/proc/***pid***/stat**. When the local timer is disabled, user and system times are no longer accumulated.

An alternative to using the local timer for CPU accounting is using the High Resolution Process Timing Facility. This facility samples the time stamp counter (TSC) register at appropriate times. Because the TSC sampling rate is the clock-speed of the processor, this accounting method yields very high resolution, providing accurate process execution user and system times with minimal overhead. The High Resolution Process Timing Facility is enabled through the CONFIG_HR_PROC_ACCT kernel tunable accessible through the General Setup selection on the Linux Kernel Configuration menu (refer to the "Configuring and Building the Kernel" chapter of this guide). The Concurrent real-time library "libccur_rt" provides many "hrtime" library routines for obtaining process execution times, converting times, and managing the High Resolution Process Timing Facility.

Process Execution Time Quanta and Limits

The local timer is used to expire the quantum of processes scheduled in the **SCHED_OTHER** and **SCHED_RR** scheduling policies. This allows processes of equal scheduling priority to share the CPU in a round-robin fashion. If the local timer is disabled on a CPU, processes on that CPU will no longer have their quantum expired. This means that a process executing on this CPU will run until it either blocks, or until a higher priority process becomes ready to run. In other words, on a CPU where the local timer interrupt is disabled, a process scheduled in the **SCHED_RR** scheduling policy will behave as if it were scheduled in the **SCHED_FIFO** scheduling policy. Note that processes scheduled on CPUs where the local timer is still enabled are unaffected. For more information about process scheduling policies, see Chapter 4, "Process Scheduling".

The **setrlimit(2)** and **getrlimit(2)** system calls allow a process to set and get a limit on the amount of CPU time that a process can consume. When this time period has expired, the process is sent the signal SIGXCPU. The accumulation of CPU time is done in the local timer interrupt routine. Therefore if the local timer is disabled on a CPU, the time that a process executes on the CPU will not be accounted for. If this is the only CPU where the process executes, it will never receive a **SIGXCPU** signal.

Interval Timer Decrementing

The **setitimer(2)** and **getitimer(2)** system calls allow a process to set up a "virtual timer" and obtain the value of the timer, respectively. A virtual timer is decremented only when the process is executing. There are two types of virtual timers: one that decrements only when the process is executing at user level, and one that is decremented when the process is executing at either user level or kernel level. When a virtual timer expires, a signal is sent to the process. Decrementing virtual timers is done in the local timer routine. Therefore when the local timer is disabled on a CPU, none of the time used will be decremented from the virtual timer. If this is the only CPU where the process executes, then its virtual timer will never expire.

System Profiling

The local timer drives system profiling. The sample that the profiler records is triggered by the firing of the local timer interrupt. If the local timer is disabled on a given CPU, the **gprof(1)** command and **profil(2)** system service will not function correctly for processes that run on that CPU.

CPU Load Balancing

The local timer interrupt routine will periodically call the load balancer to be sure that the number of runnable processes on this CPU is not significantly lower than the number of runnable processes on other CPUs in the system. If this is the case, the load balancer will steal processes from other CPUs to balance the load across all CPUs. On a CPU where the local timer interrupt has been disabled, the load balancer will only be called when the CPU has no processes to execute. The loss of this functionality is generally not a problem for a shielded CPU because it is generally not desirable to run background processes on a shielded CPU.

CPU Rescheduling

The **RESCHED_SET_LIMIT** function of the **resched_cntl(2)** system call allows a user to set an upper limit on the amount of time that a rescheduling variable can remain locked. The **SIGABRT** signal is sent to the process when the time limit is exceeded. This feature is provided to debug problems during application development. When a process with a locked rescheduling variable is run on a CPU on which the local timer is disabled, the time limit is not decremented and therefore the signal may not be sent when the process overruns the specified time limit.

POSIX Timers

The local timer interrupt provides the timing source for POSIX timers. If a CPU is shielded from local timer interrupts, the local timer interrupts will still occur on the shielded CPU if a process on that CPU has an active POSIX timer or **nanosleep(2)** function. If a process is not allowed to run on the shielded CPU, its timers will be migrated to a CPU where the process is allowed to run.

Miscellaneous

In addition to the functionality listed above, some of the functions provided by some standard Linux commands and utilities may not function correctly on a CPU if its local timer is disabled. These include:

bash(1)
sh(1)
strace(1)

For more information about these commands and utilities, refer to the corresponding man pages.

Disabling the Local Timer

The local timer can be disabled for any mix of CPUs via the **shield(1)** command or by assigning a hexadecimal value to **/proc/shield/ltmrs**. This hexadecimal value is a bitmask of CPUs; the radix position of each bit identifies one CPU and the value of that bit specifies whether or not that CPU's local timer is to be disabled (=**1**) or enabled (=**0**). See Chapter 2, "Real-Time Performance" and the **shield(1)** man page for more information.

Global Timer

The Programmable Interrupt Timer (PIT) functions as a global system-wide timer on the iHawk system. This interrupt is called IRQ 0 and by default each occurrence of the interrupt will be delivered to any CPU not currently processing an interrupt.

This global timer is used to perform the following system-wide timer functions:

- updates the system time-of-day (wall) clock and ticks-since-boot times
- dispatches events off the system timer list. This includes driver watchdog timers and process timer functions such as **alarm(2)**.

The global timer interrupt cannot be disabled. However, it can be directed to some desired subset of CPUs via the **shield(1)** command or via assignment of a bitmask of allowed CPUs, in hexadecimal form, to **/proc/irq/0/smp_affinity**. See Chapter 2, "Real-Time Performance" for more information about CPU shielding.

8
File Systems and Disk I/O

8
File Systems and Disk I/O

This chapter explains the **xfs** journaling file system and the procedures for performing direct disk I/O on the RedHawk Linux operating system.

Journaling File System

Traditional file systems must perform special file system checks after an interruption, which can take many hours to complete depending upon how large the file system is. A journaling file system is a fault-resilient file system, ensuring data integrity by maintaining a special log file called a *journal*. When a file is updated, the file's metadata are written to the journal on disk before the original disk blocks are updated. If a system crash occurs before the journal entry is committed, the original data is still on the disk and only new changes are lost. If the crash occurs during the disk update, the journal entry shows what was supposed to have happened. On reboot, the journal entries are replayed and the update that was interrupted is completed. This drastically cuts the complexity of a file system check, reducing recovery time.

Support for the XFS journaling file system from SGI is enabled by default in RedHawk Linux. XFS is a multithreaded, 64-bit file system capable of handling files as large as a million terabytes. In addition to large files and large file systems, XFS can support extended attributes, variable block sizes, is extent based and makes extensive use of Btrees (directories, extents, free space) to aid both performance and scalability. Both user and group quotas are supported.

The journaling structures and algorithms log read and write data transactions rapidly, minimizing the performance impact of journaling. XFS is capable of delivering near-raw I/O performance.

Extended attributes are name/value pairs associated with a file. Attributes can be attached to regular files, directories, symbolic links, device nodes and all other types of inodes. Attribute values can contain up to 64KB of arbitrary binary data. Two attribute namespaces are available: a user namespace available to all users protected by the normal file permissions, and a system namespace accessible only to privileged users. The system namespace can be used for protected file system metadata such as access control lists (ACLs) and hierarchical storage manage (HSM) file migration status.

NFS Version 3 can be used to export 64-bit file systems to other systems that support that protocol. NFS V2 systems have a 32-bit limit imposed by the protocol.

Backup and restore of XFS file systems to local and remote SCSI tapes or files is done using **`xfsdump`** and **`xfsrestore`**. Dumping of extended attributes and quota information is supported.

The Data Management API (DMAPI/XDSM) allows implementation of hierarchical storage management software as well as high-performance dump programs without requiring raw access to the disk and knowledge of file system structures.

A full set of tools is provided with XFS. Extensive documentation for the XFS file system can be found at:

http://oss.sgi.com/projects/xfs/

Creating an XFS File System

To create an XFS file system, the following is required:

- Identify a partition on which to create the XFS file system. It may be from a new disk, unpartitioned space on an existing disk, or by overwriting an existing partition. Refer to the **fdisk(1)** man page if creating a new partition.
- Use **mkfs.xfs(8)** to create the XFS file system on the partition. If the target disk partition is currently formatted for a file system, use the **-f** (force) option.

  mkfs.xfs [-f] /dev/*devfile*

  where *devfile* is the partition where you wish to create the file system; e.g., **sdb3**. Note that this will destroy any data currently on that partition.

Mounting an XFS File System

Use the **mount(8)** command to mount an XFS file system:

mount -t xfs /dev/*devfile* **/***mountpoint*

Refer to the **mount(8)** man page for options available when mounting an XFS file system.

Because XFS is a journaling file system, before it mounts the file system it will check the transaction log for any unfinished transactions and bring the file system up to date.

Data Management API (DMAPI)

DMAPI is the mechanism within the XFS file system for passing file management requests between the kernel and a hierarchical storage management system (HSM).

To build DMAPI, set the CONFIG_XFS_DMAPI system parameter accessible through the File Systems selection of the Linux Kernel Configuration menu as part of your build.

For more information about building DMAPI, refer to

http://oss.sgi.com/projects/xfs/dmapi.html

Direct Disk I/O

Normally, all reads and writes to a file pass through a file system cache buffer. Some applications, such as database programs, may need to do their own caching. Direct I/O is an unbuffered form of I/O that bypasses the kernel's buffering of data. With direct I/O, the file system transfers data directly between the disk and the user-supplied buffer.

RedHawk Linux enables a user process to both read directly from--and write directly to--disk into its virtual address space, bypassing intermediate operating system buffering and increasing disk I/O speed. Direct disk I/O also reduces system overhead by eliminating copying of the transferred data.

To set up a disk file for direct I/O use the **`open(2)`** or **`fcntl(2)`** system call. Use one of the following procedures:

- Invoke the **`open`** system call from a program; specify the path name of a disk file; and set the O_DIRECT bit in the *oflag* argument.
- For an open file, invoke the **`fcntl`** system call; specify an open file descriptor; specify the **`F_SETFL`** command, and set the O_DIRECT bit in the *arg* argument.

Direct disk I/O transfers must meet all of the following requirements:

- The user buffer must be aligned on a byte boundary that is an integral multiple of the _PC_REC_XFER_ALIGN **`pathconf(2)`** variable.
- The current setting of the file pointer locates the offset in the file at which to start the next I/O operation. This setting must be an integral multiple of the value returned for the _PC_REC_XFER_ALIGN **`pathconf(2)`** variable.
- The number of bytes transferred in an I/O operation must be an integral multiple of the value returned for the _PC_REC_XFER_ALIGN **`pathconf(2)`** variable.

Enabling direct I/O for files on file systems not supporting direct I/O returns an error. Trying to enable direct disk I/O on a file in a file system mounted with the file system-specific **`soft`** option also causes an error. The **`soft`** option specifies that the file system need not write data from cache to the physical disk until just before unmounting.

Although not recommended, you can open a file in both direct and cached (nondirect) modes simultaneously, at the cost of degrading the performance of both modes.

Using direct I/O does not ensure that a file can be recovered after a system failure. You must set the POSIX synchronized I/O flags to do so.

You cannot open a file in direct mode if a process currently maps any part of it with the **`mmap(2)`** system call. Similarly, a call to **`mmap`** fails if the file descriptor used in the call is for a file opened in direct mode.

Whether direct I/O provides better I/O throughput for a task depends on the application:

- All direct I/O requests are synchronous, so I/O and processing by the application cannot overlap.
- Since the operating system cannot cache direct I/O, no read-ahead or write-behind algorithm improves throughput.

However, direct I/O always reduces system-wide overhead because data moves directly from user memory to the device with no other copying of the data. Savings in system overhead is especially pronounced when doing direct disk I/O between an embedded SCSI disk controller (a disk controller on the processor board) and local memory on the same processor board.

9
Memory Mapping

9
Memory Mapping

This chapter describes the methods provided by RedHawk Linux for a process to access the contents of another process' address space.

Establishing Mappings to a Target Process' Address Space

For each running process, the **/proc** file system provides a file that represents the address space of the process. The name of this file is **/proc/***pid***/mem**, where *pid* denotes the ID of the process whose address space is represented. A process can **open(2)** a **/proc/***pid***/mem** file and use the **read(2)** and **write(2)** system calls to read and modify the contents of another process' address space.

The **usermap(3)** library routine, which resides in the **libccur_rt** library, provides applications with a way to efficiently monitor and modify locations in currently executing programs through the use of simple CPU reads and writes.

The underlying kernel support for this routine is the **/proc** file system **mmap(2)** system service call, which lets a process map portions of another process' address space into its own address space. Thus, monitoring and modifying other executing programs becomes simple CPU reads and writes within the application's own address space, without incurring the overhead of **/proc** file system **read(2)** and **write(2)** calls.

The sections below describe these interfaces and lists considerations when deciding whether to use **mmap(2)** or **usermap(3)** within your application.

Using mmap(2)

A process can use **mmap(2)** to map a portion of its address space to a **/proc/***pid***/mem** file, and thus directly access the contents of another process' address space. A process that establishes a mapping to a **/proc/***pid***/mem** file is hereinafter referred to as a monitoring process. A process whose address space is being mapped is referred to as a target process.

To establish a mapping to a **/proc/***pid***/mem** file, the following requirements must be met:

- The file must be opened with at least read permission. If you intend to modify the target process' address space, then the file must also be opened with write permission.
- On the call to **mmap** to establish the mapping, the flags argument should specify the MAP_SHARED option, so that reads and writes to the target process' address space are shared between the target process and the monitoring process.

- The target mappings must be to real memory pages. The current implementation does not support the creation of mappings to I/O space pages.

It is important to note that a monitoring process' resulting **mmap** mapping is to the target process' physical memory pages that are currently mapped in the range [*offset*, *offset* + *length*). As a result, a monitoring process' mapping to a target process' address space can become invalid if the target's mapping changes after the **mmap** call is made. In such circumstances, the monitoring process retains a mapping to the underlying physical pages, but the mapping is no longer shared with the target process. Because a monitoring process cannot detect that a mapping is no longer valid, you must make provisions in your application for controlling the relationship between the monitoring process and the target. (The notation [*start*, *end*) denotes the interval from *start* to *end*, including *start* but excluding *end*.)

Circumstances in which a monitoring process' mapping to a target process' address space becomes invalid are:

- The target process terminates.
- The target process unmaps a page in the range [*offset*, *offset* + *length*) with either **munmap(2)** or **mremap(2)**.
- The target process maps a page in the range [*offset*, *offset* + *length*) to a different object with **mmap(2)**.
- The target process invokes **fork(2)** and writes into an unlocked, private, writable page in the range [*offset*, *offset* + *length*) before the child process does. In this case, the target process receives a private copy of the page, and its mapping and write operation are redirected to the copied page. The monitoring process retains a mapping to the original page.
- The target process invokes **fork(2)** and then locks into memory a private, writable page in the range [*offset*, *offset* + *length*), where this page is still being shared with the child process (the page is marked copy-on-write). In this case, the process that performs the lock operation receives a private copy of the page (as though it performed the first write to the page). If it is the target (parent) process that locks the page, then the monitoring process' mapping is no longer valid.
- The target process invokes **mprotect(2)** to enable write permission on a locked, private, read-only page in the range [*offset*, *offset* + *length*) that is still being shared with the child process (the page is marked copy-on-write). In this case, the target process receives a private copy of the page. The monitoring process retains a mapping to the original memory object.

If your application is expected to be the target of a monitoring process' address space mapping, you are advised to:

- Perform memory-locking operations in the target process before its address space is mapped by the monitoring process.
- Prior to invoking **fork(2)**, lock into memory any pages for which mappings by the parent and the monitoring process need to be retained.

If your application is not expected to be the target of address space mapping, you may wish to postpone locking pages in memory until after invoking **fork**.

Please refer to the **mmap(2)** man page for additional details.

Using usermap(3)

In addition to the **/proc** file system **mmap(2)** system service call support, RedHawk Linux also provides the **usermap(3)** library routine as an alternative method for mapping portions of a target process' address space into the virtual address space of the monitoring process. This routine resides in the **libccur_rt** library.

While the **usermap** library routine internally uses the underlying **/proc mmap** system service call interface to create the target address space mappings, **usermap** does provide the following additional features:

- The caller only has to specify the virtual address and length of the virtual area of interest in the target process' address space. The **usermap** routine will deal with the details of converting this request into a page aligned starting address and a length value that is a multiple of the page size before calling **mmap**.
- The **usermap** routine is intended to be used for mapping multiple target process data items, and therefore it has been written to avoid the creation of redundant **mmap** mappings. **usermap** maintains internal **mmap** information about all existing mappings, and when a requested data item mapping falls within the range of an already existing mapping, then this existing mapping is re-used, instead of creating a redundant, new mapping.
- When invoking **mmap**, you must supply an already opened file descriptor. It is your responsibility to **open(2)** and **close(2)** the target process' file descriptor at the appropriate times.

  When using **usermap**, the caller only needs to specify the process ID (pid_t) of the target process. The **usermap** routine will deal with opening the correct **/proc/***pid***/mem** file. It will also keep this file descriptor open, so that additional **usermap(3)** calls for this same target process ID will not require re-opening this **/proc** file descriptor.

  Note that leaving the file descriptor open may not be appropriate in all cases. However, it is possible to explicitly close the file descriptor(s) and flush the internal mapping information that **usermap** is using by calling the routine with a "*len*" parameter value of 0. It is recommended that the monitoring process use this close-and-flush feature only after all target mappings have been created, so that callers may still take advantage of the optimizations that are built into **usermap**. Please see the **usermap(3)** man page for more details on this feature.

Note that the same limitations discussed under "Using mmap(2)" about a monitoring process' mappings becoming no longer valid also apply to **usermap** mappings, since the **usermap** library routine also internally uses the same underlying **/proc/***pid***/mem** **mmap(2)** system call support.

For more information on the use of the **usermap(3)** routine, refer to the **usermap(3)** man page.

Considerations

In addition to the previously mentioned **usermap** features, it is recommended that you also consider the following remaining points when deciding whether to use the **usermap(3)** library routine or the **mmap(2)** system service call within your application:

- The **mmap(2)** system call is a standard System V interface, although the capability of using it to establish mappings to **/proc/***pid***/mem** files is a Concurrent RedHawk Linux extension. The **usermap(3)** routine is entirely a Concurrent RedHawk Linux extension.
- **Mmap(2)** provides direct control over the page protections and the location of mappings within the monitoring process. The **usermap(3)** routine does not.

Kernel Configuration Parameters

There are two Concurrent RedHawk Linux kernel configuration parameters that directly affect the behavior of the **/proc** file system **mmap(2)** calls. Because **usermap(3)** also uses the **/proc** file system **mmap(2)** support, **usermap(3)** is equally affected by these two configuration parameters.

The two kernel configuration parameters are accessible through the File Systems selection of the Linux Kernel Configuration menu:

CONFIG_PROC_MMAP

If this kernel configuration parameter is enabled, the **/proc** file system **mmap(2)** support will be built into the kernel.

If this kernel configuration parameter is disabled, no **/proc** file system **mmap(2)** support is built into the kernel. In this case, **usermap(3)** and **/proc mmap(2)** calls will return an `errno` value of ENODEV.

This kernel configuration parameter is enabled by default in all Concurrent RedHawk Linux kernel configuration files.

CONFIG_MEMIO_ANYONE

If this kernel configuration parameter is enabled, any **/proc/***pid***/mem** file that the monitoring process is able to successfully **open(2)** with read or read/write access may be used as the target process for a **/proc mmap(2)** or **usermap(3)** call.

If this kernel configuration parameter is disabled, the monitoring process may only **/proc mmap(2)** or **usermap(3)** a target process that is currently being ptraced by the monitoring process. Furthermore, the ptraced target process must also be in a stopped state at the time the **/proc mmap(2)** system service call is made. (See the **ptrace(2)** man page for more information on ptracing other processes.)

This kernel configuration parameter is enabled by default in all Concurrent RedHawk Linux kernel configuration files.

10
Configuring and Building the Kernel

10
Configuring and Building the Kernel

Introduction

The RedHawk kernels are located in the **/boot** directory. The actual kernel file names change from release to release, however, they generally have the following form:

vmlinuz-*kernelversion***-RedHawk-***x.x*[**-***flavor*]

kernelversion	is the official version of Linux kernel source code upon which the RedHawk kernel is based
x.x	is the version number of the RedHawk kernel release
flavor	is an optional keyword that specifies an additional kernel feature that is provided by the specific kernel

The kernel is loaded into memory each time the system is booted. It is a nucleus of essential code that carries out the basic functions of the system. The kernel remains in physical memory during the entire time that the system is running (it is not swapped in and out like most user programs).

The exact configuration of the kernel depends upon:

- a large number of tunable parameters that define the run-time behavior of the system
- a number of optional device drivers and loadable modules

Kernel configuration, or reconfiguration, is the process of redefining one or more of these kernel variables and then creating a new kernel according to the new definition.

In general, the supplied kernels are created with tunable parameters and device drivers that are suitable for most systems. However, you may choose to reconfigure the kernel if you want to alter any of the tunable parameters to optimize kernel performance for your specific needs.

After you change a tunable parameter or modify the hardware configuration, the kernel will need to be rebuilt, installed and rebooted.

Configuring a Kernel Using ccur-config

The RedHawk Linux product includes three pre-built kernels. The kernels are distinguished from each other by their "**-***flavor*" suffix. The following flavors are defined:

(no suffix)	The generic kernel. This kernel is the most optimized and will provide the best overall performance, however it lacks certain features required to take full advantage of the NightStar tools.
trace	The trace kernel. This kernel is recommended for most users as it supports all of the features of the generic kernel and in addition provides support for the kernel tracing feature of the NightTrace performance analysis tool.
debug	The debug kernel. This kernel supports all of the features of the trace kernel and in addition provides support for kernel-level debugging. This kernel is recommended for users who are developing drivers or trying to debug system problems.

Each pre-built RedHawk kernel has an associated configuration file that captures all of the details of the kernel's configuration. These files are located in the "**configs**" directory of the kernel source tree. For the three pre-built kernels, the configuration files are named as follows:

generic	**static.config**
trace	**trace-static.config**
debug	**debug-static.config**

In order to configure and build a kernel that matches one of the three pre-built kernels, you must **cd** to the top of the kernel source tree and run the **ccur-config** tool.

NOTE

The **ccur-config** script must be run as root and with the system in graphical mode (i.e. run-level 5) or with a valid **DISPLAY** variable set.

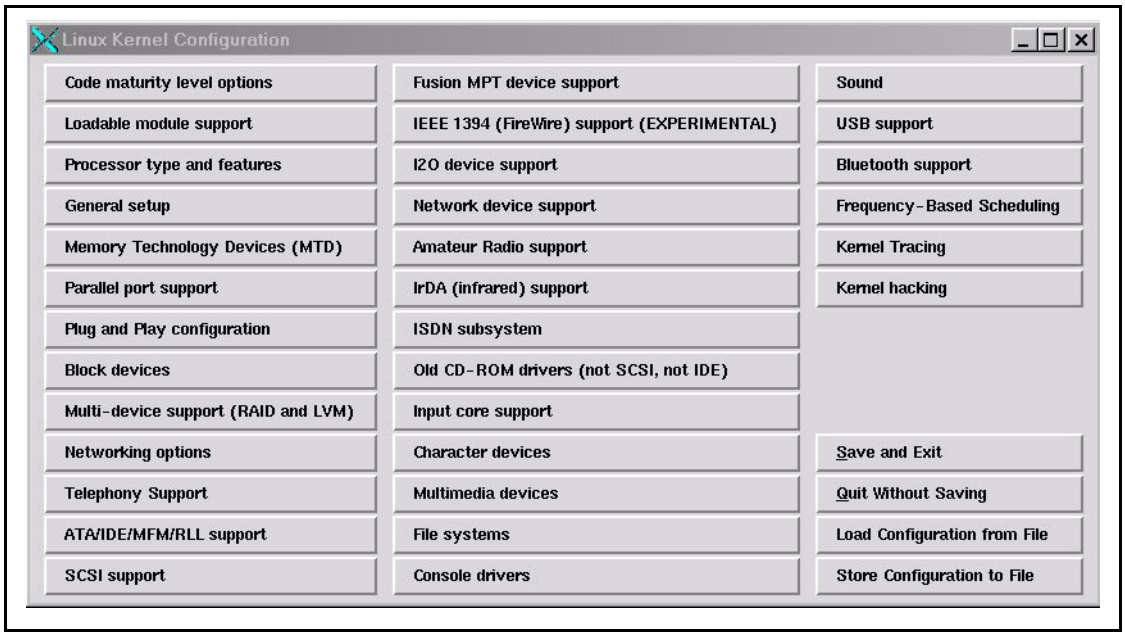
The following example configures the kernel source tree for building a new kernel based on the trace kernel's configuration. Note that it is not necessary to specify the "**.config**" suffix of the configuration file as that is automatically appended.

```
# cd /usr/src/linux-2.4.21-pre4RedHawk1.3
#./ccur-config trace-static
```

During the execution of **`ccur-config`** you will be presented with a graphical configuration window in which you can customize many different aspects of the RedHawk Linux kernel. See Screen 10-1 for an example of the graphical configuration main menu. Note that even if you do not change any configuration parameters it is still necessary to choose the "Save and Exit" button in order to properly update the kernel's configuration files.

An exhaustive list of the settings and configuration options that are available via the graphical configuration window is beyond the scope of this document, however many parameters related to real-time performance are discussed throughout this manual. In addition, online help is available for every parameter that can be modified; simply click the Help button located to the right of each parameter for more information about the parameter.

Screen 10-1. Linux Kernel Configuration, Main Menu



Building a Kernel

Regardless of which kernel configuration is used, the resulting kernel will be named with a "**vmlinuz**" prefix followed by the current kernel version string as it is defined in the top-level **Makefile**, followed with a "**-custom**" suffix added. For example:

```
vmlinuz-2.4.21-pre4-RedHawk-1.3-custom
```

The final suffix can be changed by editing the **REDHAWKFLAVOR** variable in the top-level **Makefile** *before* running **ccur-config**. When building multiple kernels from the same kernel source tree, it is important to change the suffix to avoid overwriting existing kernels accidentally.

NOTE

The pre-built kernels supplied by Concurrent have suffixes that are reserved for use by Concurrent. Therefore, you should ***not*** set the suffix to "-trace", "-debug" or " " (empty string). Use the **ccur-config -c** option if you need to build driver modules for one of the pre-built kernels (see the section "Building Driver Modules" later in this chapter).

Once kernel configuration has completed, a kernel can be built by issuing the appropriate **make(1)** commands. There are many targets in the top-level **Makefile**, however the following are of special interest:

make install Build a standalone kernel and install it into the **/boot** directory along with an associated **System.map** file. Note that when performing a "**make install**" of a RedHawk kernel it is normal for the make to terminate with errors *after* installing the kernel and map file into the **/boot** directory.

make modules Build any kernel modules that are specified in the kernel configuration.

make modules_install Install modules into the module directory associated with the currently configured kernel. Note that the name of this directory is derived from the kernel version string as defined in the top-level **Makefile**. For example, if the **REDHAWKFLAVOR** is defined as "**-custom**" then the resulting modules directory will be "**/lib/modules/2.4.21-pre4-RedHawk-1.3-custom**".

NOTE

To completely build and install a new kernel, all three of these **Makefile** targets must be issued in order.

For an example of a complete kernel configuration and build session, refer to Figure 10-1.

Figure 10-1. Example of Complete Kernel Configuration and Build Session

```
# cd /usr/src/linux-2.4.21-pre4RedHawk1.3
[ edit Makefile and change REDHAWKFLAVOR to "-test" ]
# ./ccur-config debug-static
Configuring version: 2.4.21-pre4-RedHawk-1.3-test
Cleaning source tree...
Starting graphical configuration tool...
[ configure kernel parameters as desired ]
Making kernel dependencies...

Configuration complete.

# make install
[ ignore error about missing module directory ]
# make modules
# make modules_install
[ edit /etc/grub.conf to reference new kernel and reboot ]
```

Building Driver Modules

It is often necessary to build driver modules for use with one of the pre-existing kernels supplied by Concurrent. To build driver modules for one of the pre-existing kernels, the following conditions must be met:

- The desired pre-built kernel must be the currently running kernel.
- The kernel source directory must be configured properly for the currently running kernel.

The **-c** option to **ccur-config** can be used to ensure that the kernel source directory is properly configured. For example:

```
# cd /usr/src/linux-2.4.21-pre4RedHawk1.3
# ./ccur-config -c -n
```

This automatically detects the running kernel and configures the source tree to properly match the running kernel. Driver modules can then be properly compiled for use with the running kernel.

NOTE

The **-c** option to **ccur_config** is only intended for configuring the kernel source tree to build driver modules and should not be used when building a new kernel.

Additional Information

There are many resources available that provide information to help understand and demystify Linux kernel configuration and building. A good first step is to read the **`README`** file located in the top-level of the installed RedHawk kernel source tree. In addition, the following HOWTO document is available via The Linux Documentation Project web site: http://www.tldp.org/HOWTO/Kernel-HOWTO.html

11
Linux Kernel Crash Dump (LKCD)

11
Linux Kernel Crash Dump (LKCD)

This chapter discusses the Linux Kernel Crash Dump facility, how it is configured and some examples of its use.

Introduction

The Linux Kernel Crash Dump (LKCD) facility contains kernel and user level code designed to:

- save the kernel memory image when the system dies due to a software failure
- recover the kernel memory image when the system is rebooted
- analyze the memory image to determine what happened when the failure occurred

When a crash dump is requested (a kernel Oops or panic occurs or a user forces a crash dump), the memory image is stored into a dump device, which is represented by one of the disk partitions on the system. After the operating system is rebooted, the memory image is moved to **/var/log/dump/***n*, where *n* is a number that increments with each successive crash dump. The files within that directory are used when analyzing the crash dump using **lcrash(1)**.

Installation/Configuration Details

The **lkcdutils** rpm is automatically installed as part of the RedHawk Linux installation. The default RedHawk Linux kernel configurations include the lkcd kernel patch. Concurrent has added scripts to automatically patch the **/etc/rc.d/rc.sysinit** file to configure lkcd to take dumps and to save dumps at boot time. LKCD will automatically self-configure to use the swap partition as the dump device. This is done by creating the file **/dev/vmdump** as a symbolic link to the swap partition.

The dump is saved to a disk partition and then copied to the dump directory on reboot. The system administrator must set up the disk partitions so there is enough space in **/var/log/dump** and the swap partition.

The **/etc/sysconfig/dump** configuration file contains configuration details. This file may be edited by the user if needed to make certain specifications; for example, to modify the method of compressing dumps or to change the directory where dumps are saved.

Documentation

With LKCD installed on the system, the documents listed in Table 11-1 are available at **/usr/share/doc/lkcd**. These documents contain all the information needed to understand and use LKCD, which is beyond the scope of this chapter. The user needs to be familiar with the contents of these documents.

Table 11-1. LKCD and lcrash Documents

File Name	Title and Contents of Document
lkcd_tutorial.pdf	**LKCD Installation and Configuration** This document explains the process used by LKCD to create a crash dump as well as important installation and configuration information.
lcrash.pdf and lcrash.htm	**Lcrash HOWTO** This document contains all the commands and features of the **lcrash** utility used to analyze a crash dump.

In addition to the Lcrash HOWTO document, there is a man page for **lcrash(1)**.

Additional information about LKCD and the LKCD project can be found at:

http://lkcd.sourceforge.net/

Forcing a Crash Dump on a Hung System

LKCD contains a Magic System Request (SysRq) option to force a dump. By default, Concurrent has configured the kernel with this option. To use SysRq, the user must enable it through the **/proc** file system as follows:

```
$echo 1 > /proc/sys/kernel/sysrq
```

The configuration file **/etc/sysctl.conf** sets the default value.

Two methods to force a dump are described below. Use the method applicable to your configuration:

Using PC keyboard:	Ctrl+Alt+SysRq+C
Using serial console:	Break followed by C

An example of how you can send a Break using a serial console is illustrated below:

Using minicom as terminal emulator: Ctrl+A+F to send break, followed by C

Using lcrash to Analyze a Crash Dump

A dump is a set of files contained in a directory structure that increments with each successive crash dump. For example:

```
# cd /var/log/dump
# ls
0   1   bounds

# cd 0
# ls
analysis.0  dump.0  kerntypes.0 lcrash.0    map.0
```

The example above shows navigating to **/var/log/dump/0** which contains the files for the initial crash dump. Note that the names of the files created by LKCD within that directory reflect that crash dump operation number. A copy of the **lcrash** utility found in **/sbin/lcrash** at the time of the crash is one of those files.

To invoke **lcrash** to operate on this initial crash dump, you would enter the following:

```
# lcrash -n 0

Please wait...
    Initializing vmdump access ... Done.
    Loading system map ... Done.
    Loading type info (Kerntypes) ... Done.
    Initializing arch specific data ... Done.
    Loading ksyms from dump ..... Done.

>>
```

Refer to the **lcrash(1)** man page and the lcrash documentation provided in **/usr/share/doc/lkcd** for complete instructions for using **lcrash**.

Examples of crash dump operations are given in the next section.

Crash Dump Examples

Example #1

This example shows a kernel Oops message and the progress message that are displayed as **lkcd** saves the memory image to the dump device.

```
Oops: 0002
CPU:    1
EIP:    0010:[<c03fa906>]    Not tainted
EFLAGS: 00010246
eax: 00000000   ebx: 00000000   ecx: 00000001   edx: 00000001
esi: 00000000   edi: 00000000   ebp: c127bfb8   esp: c127bfac
ds: 0018   es: 0018   ss: 0018
Process swapper (pid: 0, stackpage=c127b000)
Stack: 00000000 00000000 00000000 c127bfc0 c03faa7d c127b3f8 ffffffff 00000cf9
       c03f2000 c03f3f7c c011cf0b 00000ccc 00000000 00000292 00000001 c046b0ca
       00000246 0000002a c03f3f90 00000000 00000000
Call Trace: [<c011cf0b>]

Code: 00 00 8e 4d d3 3c 54 b9 84 3c 54 b9 84 3c 00 00 00 00 f4 01

dump: Dumping to device 0x305 [ide0(3,5)] on CPU 1 ...
dump: Compression value is 0x0, Writing dump header

dump: Pass 1: Saving Reserved Pages:
dump: Memory Bank[0]: 0 ... 7ffffff: .

dump: Pass 2: Saving Remaining Referenced Pages:
dump: Memory Bank[0]: 0 ... 7ffffff: ........................

dump: Pass 3: Saving Remaining Unreferenced Pages:
dump: Memory Bank[0]: 0 ... 7ffffff: ....

dump: Dump Complete; 32672 dump pages saved.
dump: Dump: Rebooting in 5 seconds
```

Example #2

This example illustrates the message received after using the Ctrl+Alt+SysRq+C key combination to force a dump.

```
SysRq : Start a Crash Dump (If Configured)
Dumping from interrupt handler !
Uncertain scenario - but will try my best

dump: Dumping to device 0x305 [ide0(3,5)] on CPU 0 ...
dump: Compression value is 0x0, Writing dump header
```

Example #3

When saving a crash dump, **lcrash** relies on the files **/boot/Kerntypes** and **/boot/System.map** matching the running kernel. Normally, several kernels are installed with version extended names, and the **Kerntypes** and **System.map** files are symbolic links to files that match the default kernel. If you boot a different kernel and the system takes a crash dump, it may copy the wrong **System.map** and **Kerntypes** files. In general, the core file will be saved successfully and the correct **System.map** and **Kerntypes** files can be copied manually or specified on the **lcrash** command line when examining the dump.

```
[jim@dual 0]$ /sbin/lcrash -n 0
map = map.0, dump = dump.0, outfile = stdout, kerntypes = kerntypes.0

Please wait...
        Initializing vmdump access ... Done.
        Loading system map ..................................... Done.
        Loading type info (Kerntypes) ... Done.
        Initializing arch specific data ... Done.
        Loading ksyms from dump ....... Done.
>>
```

Example #4

This example illustrates the message received when issuing the **stat** command. The **stat** command displays the log buffer which may contain clues as to what went wrong. It's also a way to determine if you have a good dump.

```
>> stat

   sysname : Linux
  nodename : dual
   release : 2.4.18-RedHawk-1.0
   version : #6 SMP Wed Apr 10 18:01:43 EDT 2002
   machine : i686
domainname : (none)

LOG_BUF:

    <1>Unable to handle kernel NULL pointer dereference at virtual address
000000a0
    <4> printing eip:
    <4>c012886b
    <1>*pde = 00000000
    <4>Oops: 0000
    <4>CPU:    1
    <4>EIP:    0010:[<c012886b>]    Not tainted
    <4>EFLAGS: 00210186
    <4>eax: 00000000   ebx: c5204000   ecx: 00000001   edx: 00000000
    <4>esi: 01800001   edi: 080a6038   ebp: bffff7b8   esp: c5205fc0
    <4>ds: 0018   es: 0018   ss: 0018
```

```
    <4>Process xscreensaver (pid: 1035, stackpage=c5205000)
    <4>Stack: c01075db 0808cff8 00000001 00000000 01800001 080a6038 bffff7b8
00000014
    <4>       4030002b 0000002b 00000014 40296167 00000023 00200282 bffff5dc
0000002b
    <4>Call Trace: [<c01075db>]
    <4>
    <4>Code: 8b 80 a0 00 00 00 c3 8d b4 26 00 00 00 00 8d bc 27 00 00 00
```

Example #5

This example illustrates the message received when issuing the **trace** command. The **trace** command without arguments displays the stack trace back for the processor that triggered the dump.

```
>> trace
================================================================
STACK TRACE FOR TASK: 0xc5204000(xscreensaver)

 0 save_other_cpu_states+72 [0xc026ccc8]
 1 __dump_configure_header+42 [0xc026ceaa]
 2 dump_configure_header+348 [0xc026b84c]
 3 dump_execute+31 [0xc026bf7f]
 4 die+121 [0xc0107e39]
 5 do_page_fault+390 [0xc01168d6]
 6 do_int3+38 [0xc01088f6]
 7 error_code+50 [0xc0107722]
   ebx: c5204000   ecx: 00000001   edx: 00000000   esi: 01800001
   edi: 080a6038   ebp: bffff7b8   eax: 00000000   ds:  0018
   es:  0018       eip: c012886b   cs:  0010       eflags: 00210186
 8 sys_getpid+11 [0xc012886b]
 9 system_call+84 [0xc01075d4]
   ebx: 0808cff8   ecx: 00000001   edx: 00000000   esi: 01800001
   edi: 080a6038   ebp: bffff7b8   eax: 00000014   ds:  002b
   es:  002b       eip: 40296167   cs:  0023       eflags: 00200282
   esp: bffff5dc   ss:  002b
================================================================
>>
```

Example #6

This example illustrates the message received when issuing the **ps** command followed by the **trace** command with a stack address (from the first column of the **ps** output) to trace an arbitrary stack.

```
>> ps
Address     Uid     Pid    PPid Stat      Flags SIZE:RSS   Command
------------------------------------------------------------------------------
c03f8000       0       0       0 0x00 0x00000100     0:0     swapper
c126a000       0       1       0 0x00 0x00000100   353:130   init
c1266000       0       2       0 0x01 0x00008140     0:0     migration_CPU0
c1264000       0       3       0 0x01 0x00008140     0:0     migration_CPU1
c12f6000       0       4       1 0x00 0x00000040     0:0     keventd
c12f2000       0       5       0 0x01 0x00008040     0:0     ksoftirqd_CPU0
c12f0000       0       6       0 0x01 0x00008040     0:0     ksoftirqd_CPU1
c12e0000       0       7       0 0x01 0x00000840     0:0     kswapd

   o
   o
   o
c2f84000       0    1334    1330 0x01 0x00000100   635:342   bash
c2de0000       0    1381    1334 0x01 0x00000100  1496:995   lcrash
c2422000       0    1415     752 0x01 0x00000040   399:176   crond
c2572000      41    1416    1415 0x00 0x00100104     0:0     python
------------------------------------------------------------------------------
57 processes found
>> trace c2572000
==============================================================
STACK TRACE FOR TASK: 0xc2572000(python)

 0 smp_call_function+144 [0xc0114280]
 1 flush_tlb_all+15 [0xc01140bf]
 2 vmfree_area_pages+213 [0xc01391b5]
 3 vfree+180 [0xc0139444]
 4 exit_mmap+13 [0xc0132d8d]
 5 mmput+79 [0xc011a61f]
 6 do_exit+197 [0xc0120135]
 7 sys_exit+11 [0xc012042b]
 8 system_call+84 [0xc01075d4]
   ebx: 00000000   ecx: 00000000   edx: 401a5154   esi: 401a42e0
   edi: 00000000   ebp: bffffdd8   eax: 00000001   ds:  002b
   es:  002b       eip: 40129afd   cs:  0023       eflags: 00000246
   esp: bffffdac   ss:  002b
==============================================================
>>
```

12
Pluggable Authentication Modules (PAM)

12
Pluggable Authentication Modules (PAM)

This chapter discusses the PAM facility that provides a secure and appropriate authentication scheme accomplished through a library of functions that an application may use to request that a user be authenticated.

Introduction

PAM, which stands for **P**luggable **A**uthentication **M**odules, is a way of allowing the system administrator to set authentication policy without having to recompile authentication programs. With **PAM**, you control how the modules are plugged into the programs by editing a configuration file.

Most users will never need to touch this configuration file. When you use **`rpm(8)`** to install programs that require authentication, they automatically make the changes that are needed to do normal password authentication. However, you may want to customize your configuration, in which case you must understand the configuration file.

PAM Modules

There are four types of modules defined by the **PAM** standard. These are:

auth	provides the actual authentication, perhaps asking for and checking a password, and they set “credentials” such as group membership
account	checks to make sure that the authentication is allowed (the account has not expired, the user is allowed to log in at this time of day, and so on)
password	used to set passwords
session	used once a user has been authenticated to allow them to use their account, perhaps mounting the user's home directory or making their mailbox available

These modules may be stacked, so that multiple modules are used. For instance, *rlogin* normally makes use of at least two authentication methods: if *rhosts* authentication succeeds, it is sufficient to allow the connection; if it fails, then standard password authentication is done.

New modules can be added at any time, and **PAM**-aware applications can then be made to use them.

Services

Each program using **PAM** defines its own "service" name. The **login** program defines the service type *login*, **ftpd** defines the service type *ftp*, and so on. In general, the service type is the name of the program used to access the service, not (if there is a difference) the program used to provide the service.

Role-Based Access Control

Role-Based Access Control for RedHawk Linux is implemented using **PAM**. In the Role-Based Access Control scheme, you set up a series of roles in the **capability.conf(5)** file. A role is defined as a set of valid Linux capabilities. The current set of all valid Linux capabilities can be found in the **/usr/include/linux/capability.h** kernel header file or by using the **_cap_names[]** string array.

Roles can act as building blocks in that once you have defined a role, it can be used as one of the capabilities of a subsequent role. In this way the newly defined role inherits the capabilities of the previously defined role. Examples of this feature are given below. See the **capability.conf(5)** man page for more information.

Once you have defined a role, it can be assigned to a user or a group in the **capability.conf(5)** file. A user is a standard Linux user login name that corresponds to a valid user with a login on the current system. A group is a standard Linux group name that corresponds to a valid group defined on the current system.

Files in **/etc/pam.d** correspond to a service that a user can use to log into the system. These files may be modified to include a **pam_capability** session line (examples of adding **pam_capability** session lines to service files are given in the "Examples" section below). For example: the **/etc/pam.d/login** file is a good candidate as it covers login via telnet. If a user logs into the system using a service that has not been modified, no special capability assignment takes place.

The following options can be specified when supplying a **pam_capability** session line to a file in **/etc/pam.d**:

conf=*conf_file* — specify the location of the configuration file. If this option is not specified then the default location will be **/etc/security/capability.conf**.

debug — Log debug information via **syslog**. The debug information is logged in the **syslog** *authpriv* class. Generally, this log information is collected in the **/var/log/secure** file.

Examples

The following examples illustrate adding session lines to **/etc/pam.d/login**:

1. To allow the roles defined in the **/etc/security/capability.conf** file to be assigned to users who login to the system via **telnet(1)** append the following line to **/etc/pam.d/login**:

   ```
   session required /lib/security/pam_capability.so
   ```

2. To allow the roles defined in the **/etc/security/capability.conf** file to be assigned to users who login to the system via **ssh(1)** append the following line to **/etc/pam.d/sshd**:

   ```
   session required /lib/security/pam_capability.so
   ```

3. To have **ssh** users get their role definitions from a different **capability.conf** file than the one located in **/etc/security** append the following lines to **/etc/pam.d/sshd**:

   ```
   session required /lib/security/pam_capability.so \
   conf=/root/ssh-capability.conf
   ```

   Thus, the roles defined in the **/root/ssh-capability.conf** file will be applied to users logging in via **ssh**.

Defining Capabilities

The **capability.conf** file provides information about the roles that can be defined and assigned to users and groups. The file has three types of entries: Roles, Users and Groups.

Roles

A role is a defined set of valid Linux capabilities. The current set of all valid Linux capabilities can be found in the **/usr/include/linux/capability.h** kernel header file or by using the **_cap_names[]** string array. This array is described in the **cap_from_text(3)** man page. In addition, the following capability keywords are pre-defined:

`all`	all capabilities (except `cap_setcap`)
`cap_fs_mask`	all file system-related capabilities
`none`	no capabilities whatsoever

As the name implies, it is expected that different roles will be defined, based on the duties that various system users and groups need to perform.

The format of a role entry in the **capability.conf** file is:

role *rolename* *capability_list*

Entries in the capability list can reference previously defined roles. For example, you can define a role called *basic* in the file and then add this role as one of your capabilities in the capability list of a subsequent role. Note that the capability list is a whitespace or comma separated list of capabilities that will be turned on in the user's inheritable set.

Users

A user is a standard Linux user login name that corresponds to a valid user with a login on the current system. User entries that do not correspond to valid users on the current system (verified by **`getpwnam(3)`**) are ignored.

The format of a user entry in the **`capability.conf`** file is:

`user` *username* *rolename*

The special username '*' can be used to assign a default role for users that do not match any listed users or have membership in a listed group:

`user` **`*`** *default_rolename*

Groups

A group is a standard Linux group name that corresponds to a valid group defined on the current system. Group entries that do not correspond to valid groups on the current system (verified by **`getgrnam(3)`**) are ignored.

The format of a group entry in the **`capability.conf`** file is:

`group` *groupname* *rolename*

Examples

1. The following example sets up an administrative role (`admin`) that is roughly equivalent to root:

```
role    admin           all
```

2. The following example sets up a desktop user role that adds sys_boot and sys_time to the inheritable capability set:

```
role    desktopuser     cap_sys_boot \
                        cap_sys_time
```

3. The following example sets up a poweruser user role, using the desktop user role created previously:

```
role     poweruser      desktopuser\
                        cap_sys_ptrace\
                        cap_sys_nice\
                        cap_net_admin
```

4. To assign the `desktopuser` role to a user, enter the following in the USERS section of the **`capability.conf`** file:

```
user     joe          desktopuser
```

5. To assign the `poweruser` role to a group, enter the following in the GROUPS section of the **`capability.conf`** file:

```
group    hackers      poweruser
```

Implementation Details

The following items address requirements for full implementation of the PAM functionality:

- **`Pam_capability`** requires that the running kernel be modified to inherit capabilities across the **`exec()`** system call. Kernels that have been patched with the kernel patch shipped with this module can enable capability inheritance using the CONFIG_INHERIT_CAPS_ACROSS_EXEC configuration option accessible through the General Setup selection of the Linux Kernel Configuration menu (refer to the “Configuring and Building the Kernel” chapter of this guide). All RedHawk Linux kernels have this option enabled by default.
- In order to use the **`pam_capability`** feature with **`ssh`**, the **`/etc/ssh/sshd_config`** file must have the following option set:

  `UsePrivilegeSeparation no`

13
Device Drivers and Real Time

13
Device Drivers and Real Time

A device driver runs in kernel mode and is an extension of the kernel itself. Device drivers therefore have the ability to influence the real-time performance of the system in the same way that any kernel code can affect real-time performance. This chapter provides a high-level overview of some of the issues related to device drivers and real-time.

It should be noted that while there are many open source device drivers that are available for Linux, these drivers have a wide range of quality associated with them, especially in regards to their suitability for a real-time system.

Interrupt Routines

The duration of an interrupt routine is very important in a real-time system because an interrupt routine cannot be preempted to execute a high-priority task. Lengthy interrupt routines directly affect the process dispatch latency of the processes running on the CPU to which the interrupt is assigned. The term *process dispatch latency* denotes the time that elapses from the occurrence of an external event, which is signified by an interrupt, until the process that is waiting for that external event executes its first instruction in user mode. For more information on how interrupts affect process dispatch latency, see the “Real-Time Performance” chapter.

If you are using a device driver in a real-time production environment, you should minimize the amount of work performed at interrupt level. Linux supports several different mechanisms for deferring processing that should not be performed at interrupt level. These mechanisms allow an interrupt routine to trigger processing that will be performed in the context of a kernel daemon at program level. Because the priority of these kernel daemons is configurable, it is possible to run high-priority real-time processes at a priority level higher than the deferred interrupt processing. This allows a real-time process to have higher priority than some activity that might normally be run at interrupt level. Using this mechanism, the execution of real-time tasks is not delayed by any deferred interrupt activity. See the “Deferred Interrupt Functions” section for more information about deferring interrupts.

Generally, a device’s interrupt routine can interact with the device to perform the following types of tasks:

- acknowledge the interrupt
- save data received from the device for subsequent transfer to a user
- initiate a device operation that was waiting for completion of the previous operation

A device's interrupt routine should *not* perform the following types of tasks:

- copy data from one internal buffer to another
- allocate or replenish internal buffers for the device
- replenish other resources used by the device

These types of tasks should be performed at program level via one of the deferred interrupt mechanisms. You can, for example, design a device driver so that buffers for the device are allocated at program level and maintained on a free list that is internal to the driver. When a process performs read or write operations, the driver checks the free list to determine whether or not the number of buffers available is sufficient for incoming interrupt traffic. The interrupt routine can thus avoid making calls to kernel buffer allocation routines, which are very expensive in terms of execution time. Should a device run out of resources and only notice this at interrupt level, new resources should be allocated as part of the deferred interrupt routine rather than at interrupt level.

Deferred Interrupt Functions

Linux supports several methods by which the execution of a function can be deferred. Instead of invoking the function directly, a "trigger" is set that causes the function to be invoked at a later time. These mechanisms are used by interrupt routines under Linux in order to defer processing that would otherwise have been done at interrupt level. By removing this processing from interrupt level, the system can achieve better interrupt response time as described above.

There are three different mechanisms for deferring interrupt processing. Each of these mechanisms has different requirements in terms of whether or not the code that is deferred must be reentrant or not. The three types of deferrable functions are softirqs, tasklets and bottom halves. A *softirq* must be completely reentrant because a single instance of a softirq can execute on multiple CPUs at the same time. *Tasklets* are implemented as a special type of softirq and are handled by the same kernel daemon. The difference is that a given tasklet function will always be serialized with respect to itself. In other words, no two CPUs will ever execute the same tasklet code at the same time. This property allows a simpler coding style in a device driver, since the code in a tasklet does not have to be reentrant with respect to itself. *Bottom halves* are globally serialized. In other words, only one bottom half of any type will ever execute in the system at the same time. Because of the performance implications of this global serialization, bottom halves are slowly being phased out of the Linux kernel.

By default under RedHawk, all deferred interrupt functions will only execute in the context of a kernel daemon. It should be noted that this is different than the way standard Linux operates. The priority and scheduling policy of these kernel daemons can be set via kernel configuration parameters. This allows the system to be configured such that a high-priority real-time task can preempt the activity of deferred interrupt functions.

Softirqs and tasklets are both run by the **ksoftirqd** daemon. There is one **ksoftirqd** daemon per logical CPU. A softirq or tasklet will run on the CPU that triggered its execution. Therefore, if a hard interrupt has its affinity set to a specific CPU, the corresponding softirq or tasklet will also run on that CPU. The priority of the **ksoftirqd** is determined by the CONFIG_SOFTIRQ_PRI kernel tunable, which is located under the

Processor Types and Features selection of the Linux Kernel Configuration menu. By default the value of this tunable is set to zero, which indicates that the **ksoftirqd** daemon will run as under the SCHED_FIFO scheduling policy at a priority of one less than the highest real-time priority. Setting this tunable to a positive value specifies the real-time priority value that will be assigned to all **ksoftirqd** daemons.

Bottom halves can be run by two different kernel daemons – **keventd** and **kbottomd**. The difference between these types of bottom halves is historical and has to do with whether or not hardware interrupts are blocked while the bottom half is executed. Unlike the **ksoftirqd**, there is only one instance of **keventd** and one instance of **kbottomd**. These daemons would therefore generally be run on a non-shielded CPU. The priority of these kernel daemons can be set via the kernel tunables CONFIG_EVENTD_PRI and CONFIG_BH_PRI respectively, which are located under the Processor Types and Features selection of the Linux Kernel Configuration menu. By default the **keventd** kernel daemon runs under the SCHED_OTHER scheduling policy, at the highest SCHED_OTHER priority. By default, the **kbottomd** kernel daemon is set to the same priority and scheduling policy as the **ksoftirqd** kernel daemon. By setting the value of the appropriate kernel configuration parameter to a positive value, the affected kernel daemon will run as a SCHED_FIFO process at the specified priority.

Table 13-1 provides a summary of the types of deferred interrupts and their corresponding values.

Table 13-1. Deferred Interrupt Types and Characteristics

Deferred Interrupt Type	Kernel Daemon	Daemon Tunable	Default Value
softirq	ksoftirqd	CONFIG_SOFTIRQ_PRI	Maximum SCHED_FIFO priority - 1
tasklet	ksoftirqd	CONFIG_SOFTIRQ_PRI	Maximum SCHED_FIFO priority - 1
bottom half (events)	keventd	CONFIG_EVENTD_PRI	Maximum SCHED_OTHER priority
bottom half (standard)	kbottomd	CONFIG_BH_PRI	Follows the setting for CONFIG_SOFTIRQ_PRI

When configuring a system where real-time processes can run at a higher priority than the deferred interrupt daemons, it is important to understand whether those real-time processes depend upon the services offered by the daemon. If a high-priority real-time task is CPU bound at a level higher than a deferred interrupt daemon, it is possible to starve the daemon so it is not receiving any CPU execution time. If the real-time process also depends upon the deferred interrupt daemon, a deadlock can result.

Multi-threading Issues

RedHawk Linux is built to support multiple CPUs in a single system. This means that all kernel code and device drivers must be written to protect their data structures so that the data structures cannot be modified simultaneously on more than one CPU. The process of multi-threading a device driver involves protecting accesses to data structures so that all modifications to a data structure are serialized. In general this is accomplished in Linux by using spin locks to protect these kinds of data structure accesses.

Locking a spin lock will cause preemption to be disabled and/or interrupts to be disabled. In either case, the worst case process dispatch latency for a process executing on the CPU where preemption or interrupts are disabled is directly impacted by the length of time that these features are disabled. It is therefore important when writing a device driver to minimize the length of time that spin locks are held, which will affect the amount of time that preemption and/or interrupts are disabled. Remember that locking a spin lock will implicitly cause preemption or interrupts to be disabled (depending upon which spin lock interface is used). For more information about this topic, see the "Real-Time Performance" chapter.

The Big Kernel Lock (BKL) and ioctl

The Big Kernel Lock (BKL) is a spin lock in the Linux kernel, which is used when a piece of kernel source code has not been fine-grain multi-threaded. While much use of the BKL has been removed by systematically multi-threading the Linux kernel, the BKL is still the most highly contended and longest held lock in the Linux kernel.

By default, the Linux kernel will lock the BKL before calling the **`ioctl(2)`** function that is associated with a device driver. If a device driver is multi-threaded, then it is not necessary to lock the BKL before calling the **`ioctl`** routine. RedHawk Linux supports a mechanism for a device driver to specify that the BKL should not be locked before calling the **`ioctl`** routine. When a device is used to support real-time functions or when an application makes calls to a device's **`ioctl`** routine on a shielded CPU, it is very important that the device driver be modified so that the BKL is not locked. Without this modification, a process could stall spinning on the BKL spin lock for an extended period of time causing jitter to the programs and interrupts that are assigned to the same CPU.

The mechanism for specifying that the BKL should not be locked on entry to a device's **`ioctl`** routine is to set the FOPS_IOCTL_NOBKL flag in the `file_operations` structure in the device driver source code. Below is an example of how the RCIM device sets this flag:

```
static struct file_operations rcim_fops = {
    owner:          THIS_MODULE,
    open:           rcim_master_open,
    release:        rcim_master_release,
    ioctl:          rcim_master_ioctl,
    mmap:           rcim_master_mmap,
    flags:          FOPS_IOCTL_NOBKL,
};
```

After making this change, the device driver must be rebuilt. For a static driver this means rebuilding the entire kernel. For a dynamically loadable module, only that module must be rebuilt. See the "Configuring and Building the Kernel" chapter for more information.

A
Example Program - Message Queues

This appendix contains an example program that illustrates the use of the System V message queue facilities. The program is written in C. In this program, a parent process spawns a child process to off load some of its work. The parent process also creates a message queue for itself and the child process to use.

When the child process completes its work, it sends the results to the parent process via the message queue and then sends the parent a signal. When the parent process receives the signal, it reads the message from the message queue.

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

#define MSGSIZE 40/* maximum message size */
#define MSGTYPE 10/* message type to be sent and received */

/* Use a signal value between SIGRTMIN and SIGRTMAX */
#define SIGRT1(SIGRTMIN+1)

/* The message buffer structure */
struct my_msgbuf {
        long mtype;
        char mtext[MSGSIZE];
};
struct my_msgbuf msg_buffer;

/* The message queue id */
int msqid;

/* SA_SIGINFO signal handler */
void sighandler(int, siginfo_t *, void *);

/* Set after SIGRT1 signal is received */
volatile int done = 0;

pid_t parent_pid;
pid_t child_pid;


main()
{
     int retval;
     sigset_t set;
     struct sigaction sa;


     /* Save off the parent PID for the child process to use. */
     parent_pid = getpid();
```

```
/* Create a private message queue. */
msqid = msgget(IPC_PRIVATE, IPC_CREAT | 0600);
if (msqid == -1) {
     perror("msgget");
     exit(-1);
}

/* Create a child process. */
child_pid = fork();

if (child_pid == (pid_t)-1) {
     /* The fork(2) call returned an error. */
     perror("fork");

     /* Remove the message queue. */
     (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);

     exit(-1);
}

if (child_pid == 0) {
     /* Child process */

     /* Set the message type. */
     msg_buffer.mtype = MSGTYPE;

     /* Perform some work for parent. */
     sleep(1);

     /* ... */

     /* Copy a message into the message buffer structure. */
     strcpy(msg_buffer.mtext, "Results of work");

     /* Send the message to the parent using the message
      * queue that was inherited at fork(2) time.
      */
     retval = msgsnd(msqid, (const void *)&msg_buffer,
               strlen(msg_buffer.mtext) + 1, 0);

     if (retval) {
          perror("msgsnd(child)");

          /* Remove the message queue. */
          (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);

          exit(-1);
     }

     /* Send the parent a SIGRT signal. */
     retval = kill(parent_pid, SIGRT1);
     if (retval) {
          perror("kill SIGRT");

          /* Remove the message queue. */
          (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);
          exit(-1);
     }
     exit(0);
}
```

```
     /* Parent */

     /* Setup to catch the SIGRT signal.  The child process
      * will send a SIGRT signal to the parent after sending
      * the parent the message.
      */
     sigemptyset(&set);
     sa.sa_mask = set;
     sa.sa_sigaction = sighandler;
     sa.sa_flags = SA_SIGINFO;
     sigaction(SIGRT1, &sa, NULL);

     /* Do not attempt to receive a message from the child
      * process until the SIGRT signal arrives. Perform parent
      * workload while waiting for results.
      */
     while (!done) {
          /* ... */
     }

     /* Remove the message queue.
     (void) msgctl(msqid, IPC_RMID, (struct msqid_ds *)NULL);
      */

     /* All done.
      */
     exit(0);
}


/*
* This routine reacts to a SIGRT1 user-selected notification
* signal by receiving the child process' message.
*/
void
sighandler(int sig, siginfo_t *sip, void *arg)
{
     int retval;
     struct ucontext *ucp = (struct ucontext *)arg;


     /* Check that the sender of this signal was the child process.
      */
     if (sip->si_pid != child_pid) {
          /* Ignore SIGRT from other processes.
           */
          printf("ERROR: signal received from pid %d\n", sip->si_pid);
          return;
     }

     /* Read the message that was sent to us.
      */
     retval = msgrcv(msqid, (void*)&msg_buffer,
               MSGSIZE, MSGTYPE, IPC_NOWAIT);

     done++;

     if (retval == -1) {
          perror("mq_receive (parent)");
          return;
```

```
        }

        if (msg_buffer.mtype != MSGTYPE) {
             printf("ERROR: unexpected message type %d received.\n",
                  msg_buffer.mtype);
             return;
        }

        printf("message type %d received: %s\n",
             msg_buffer.mtype, msg_buffer.mtext);
}
```

Index

D

E

F

G

H

I

J

K

R

S

T

U

W

X

Spine for 1” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up