RedHawk Linux® User's Guide

0898004-440
June 2006

Revision History:

Date	Level	Effective With
August 2002	000	RedHawk Linux Release 1.1
September 2002	100	RedHawk Linux Release 1.1
December 2002	200	RedHawk Linux Release 1.2
April 2003	300	RedHawk Linux Release 1.3, 1.4
December 2003	400	RedHawk Linux Release 2.0
March 2004	410	RedHawk Linux Release 2.1
July 2004	420	RedHawk Linux Release 2.2
May 2005	430	RedHawk Linux Release 2.3
June 2006	440	RedHawk Linux Release 2.3

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: appendixes, glossary 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 and shared memory 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.
- Chapter 10, *Non-Uniform Memory Access (NUMA)*, describes the NUMA support available on certain systems.

Part 2 - Administrator

- Chapter 11, *Configuring and Building the Kernel*, provides information on how to configure and build a RedHawk Linux kernel.
- Chapter 12, *Linux Kernel Crash Dump (LKCD)*, provides guidelines for saving, restoring and analyzing the kernel memory image using LKCD.
- Chapter 13, *Pluggable Authentication Modules (PAM)*, describes the PAM authentication capabilities of RedHawk Linux.
- Chapter 14, *Device Drivers*, describes RedHawk functionality and real-time issues involved with writing device drivers.
- Chapter 15, *PCI-to-VME Support*, describes RedHawk's support for a PCI-to-VME bridge.

Part 3 - Common Material

- Appendix A, *Example Message Queue Programs*, contains example programs illustrating the POSIX and System V message queue facilities.
- Appendix B, *Kernel Tunables for RedHawk Linux Features*, contains a listing of the kernel tunables that control unique features in RedHawk Linux and their default values in pre-built RedHawk kernels.
- Appendix C, *Capabilities in RedHawk Linux*, lists the capabilities included in RedHawk Linux and the permissions provided by each.
- Appendix D, *Kernel Trace Events*, lists pre-defined kernel trace points and methods for defining and logging custom events within kernel modules.
- Appendix E, *Migrating 32-bit Code to 64-bit Code*, provides information needed to migrate 32-bit code to 64-bit processing on the AMD Opteron processor.
- Appendix F, *Kernel-level Daemons on Shielded CPUs*, describes how kernel-level daemons execute on shielded CPUs and provides methods for improving performance.
- Appendix G, *Cross Processor Interrupts on Shielded CPUs*, describes how cross-processor interrupts execute on shielded CPUs and provides methods for improving performance.
- Appendix H, *Serial Console Setup*, provides instructions for configuring a serial console.
- The *Glossary* provides definitions for terms used throughout this Guide.
- 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 *italic*.
`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, messages and listings of files and programs appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify these options or arguments.
hypertext links	When viewing this document online, clicking on chapter, section, figure, table and page number references will display the corresponding text. Clicking on Internet URLs provided in blue type will launch your web browser and display the web site. Clicking on publication names and numbers in red type will display the corresponding manual PDF, if accessible.

Related Publications

RedHawk Linux Operating System Documentation	**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) PCI Form Factor User's Guide	0898007
iHawk Optimization Guide	0898011
RedHawk Linux FAQ	N/A
Partner Documentation	
SBS Technologies 1003 Linux Support Software	85221990
SBS Technologies Model 618-3 Adapter Hardware	85851150
Guide to SNARE for Linux	N/A

where *x.x* = release version

Contents

Screens

Illustrations

Tables

1
Introduction to RedHawk Linux

1
Introduction to RedHawk Linux

This chapter provides an introduction to the RedHawk Linux operating system and gives an overview of the real-time features included.

Overview

Concurrent Computer Corporation's RedHawk™ Linux® is a real-time version of the open source Linux operating system. Modifications are made to standard Linux version 2.6 to support the functionality and the performance required by complex time-critical applications. RedHawk uses a single kernel design to support a single programming environment that directly controls all system operation. This design allows deterministic program execution and response to interrupts while simultaneously providing high I/O throughput and deterministic file, networking, and graphics I/O operations. RedHawk is the ideal Linux environment for the deterministic applications found in simulation, data acquisition, industrial control and medical imaging systems.

Included with RedHawk is the popular Red Hat® Enterprise Linux distribution. The RedHawk installation CDs provide additional real-time kernels and libraries for accessing RedHawk specific kernel features. 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. Optionally, the NightStar™ development tool set is available for developing time-critical applications, and the Frequency-Based Scheduler and Performance Monitor can be used to schedule processes in cyclical execution patterns and monitor performance.

The RedHawk kernel integrates both open source patches and Concurrent developed features to provide a state of the art real-time kernel. Many of these features have been derived from the real-time UNIX® implementations that Concurrent has supported in its over 35 years experience developing real-time operating systems. These features are described briefly in the section “Real-Time Features in RedHawk Linux” later in this chapter with references to more detailed information.

RedHawk is included with each Concurrent iHawk system. iHawks are symmetric multi-processor (SMP) systems available in a variety of architectures and configurations. Either 32-bit or 64-bit versions of RedHawk and its supporting software products are installed, depending upon the iHawk architecture type and the support included in RedHawk for that architecture.

Support for SMPs is highly optimized. A unique concept known as *shielded CPUs* allows a subset of processors to be dedicated to tasks that require the most deterministic performance. Individual CPUs can be shielded from interrupt processing, kernel daemons, interrupt bottom halves, and other Linux tasks. Processor shielding provides a highly deterministic execution environment where interrupt response of less than 30 microseconds is guaranteed.

RedHawk Linux has at least the same level of POSIX conformance as other Linux distributions based on the 2.6 series of kernels. Concurrent has added additional POSIX compliance by adding some of the POSIX real-time extensions that are not present in standard Linux. 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.

NightStar is Concurrent's powerful tool set that provides a robust graphic interface for non-intrusive control, monitoring, analysis, and debugging of time-critical multiprocessing applications. The RedHawk kernel contains enhancements that allow these tools to efficiently perform their operations with a minimum of interference to the application's execution. All tools can be run natively on the same system as the application or remotely for less intrusive application control.

The NightStar tools include the following. Refer to the individual User's Guides for complete information.

- NightView™ source-level debugger – allows multi-language, multi-processor, multi-program and multi-thread monitoring and debugging from a single graphical interface. NightView has the capability to hot patch running programs to modify execution, retrieve or modify data and insert conditional breakpoints, monitor points and watch points that execute at full application speed.
- NightTrace™ run-time analyzer – used to analyze the dynamic behavior of a running application. User and system activities are logged and marked with high-resolution time stamps. These events are then graphically displayed to provide a detailed view of system activity that occurs while the application is running. NightTrace is ideal for viewing interactions between multiple processes, activity on multiple processors, applications that execute on distributed systems and user/kernel interactions. Its powerful capabilities allow searching for specific events or summarizing kernel or user states.
- NightSim™ periodic scheduler – allows the user to easily schedule applications that require periodic execution. A developer can dynamically control the execution of multiple coordinated processes, their priorities and CPU assignments. NightSim provides detailed, highly accurate performance statistics and allows various actions when frame overruns occur.
- NightProbe™ data monitor – used to sample, record or modify program data in multiple running programs. Program data is located with a symbol table browser. Application pages are shared at the physical page level to minimize the impact on the application's execution. NightProbe can be used for debugging, analysis, fault injection or in a production environment to create a GUI control panel for program input and output.
- NightTune™ performance tuner – a graphical tool for analyzing system and application performance including CPU usage, context switches, interrupts, virtual memory usage, network activity, process attributes, and CPU shielding. NightTune allows you to change the priority, scheduling policy, and CPU affinity of individual or groups of processes using pop-up dialogs or drag-and-drop actions. It also allows you to set the shielding and hyper-threading attributes of CPUs and change the CPU assignment of individual interrupts.

RedHawk Linux Kernels

There are three flavors of RedHawk Linux kernels. 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-*kernelversion*-RedHawk-*x.x*-trace	Default kernel with trace points but no debug checks
vmlinuz-*kernelversion*-RedHawk-*x.x*-debug	Kernel with both debug checks and kernel trace points
vmlinuz-*kernelversion*-RedHawk-*x.x*	Optimized kernel with no trace points and no debug checks
kernelversion is the official version of Linux kernel source code upon which the RedHawk kernel is based. *x.x* indicates the RedHawk Linux version number; for example, “2.0”.	

The default RedHawk Linux trace kernel 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, 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 Updates website. Refer to the *RedHawk Linux Release Notes* for details.

NOTE

Concurrent does not recommend downloading Red Hat updates.

The RedHawk Linux kernel replaces the standard Red Hat kernel and is likely to work with any version of the Red Hat distribution. However, installing upgrades, especially to **gcc** and **glibc**, from sources other than Concurrent may destabilize the system and is not recommended. Security updates from outside sources may be installed if desired.

Real-Time Features in RedHawk Linux

This section provides a brief description of the features included in the RedHawk Linux operating system for real-time processing and performance. 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 chapter references.

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 up to twelve external device interrupts
- up to eight real time clocks that can interrupt the system
- up to twelve 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 twelve 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 User's Guide* 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. See Appendix D for information about kernel tracing.

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 that is currently executing inside the kernel is provided in 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. Data structure protection mechanisms are built into the kernel to support symmetric multiprocessing.

Real-Time Scheduler

The real-time scheduler 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 Enhancements

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 enhancements applied to RedHawk Linux modify the algorithms in the identified worst-case preemption off scenarios to provide better interrupt response times.

Priority Inheritance

Semaphores used as sleepy-wait mutual exclusion mechanisms can introduce the problem of priority inversion. Priority inversion occurs when one or more low-priority processes executing in a critical section prevent the progress of one or more high-priority processes. Priority inheritance involves temporarily raising the priority of the low priority processes executing in the critical section to that of the highest priority waiting process. This ensures that the processes executing in the critical section have sufficient priority to continue execution until they leave the critical section.

Semaphore support for kernel services is configured in each of the pre-built RedHawk Linux kernels. It can be disabled if an individual system is set up to handle only non-critical tasks and any overhead associated with priority inheritance is unnecessary. Refer to Appendix B for the kernel tunables used with priority inheritance. Support for pthread mutexes that provide priority inheritance within a multithreaded application is included in an alternate **`glibc`**. Refer to Chapter 5 for details.

High Resolution Process Accounting

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 process accounting facility provides a mechanism for very accurate CPU execution time accounting, allowing better performance monitoring of applications. This facility is incorporated in the "debug" and "trace" kernels supplied by Concurrent and used by standard Linux CPU accounting services and the Performance Monitor on those kernels. See Chapter 7 for information about CPU accounting methods.

Capabilities Support

The Pluggable Authentication Module (PAM) 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 13 for information about the PAM functionality.

Kernel Debuggers

Two open source kernel debuggers, **`kdb`** and **`kgdb`**, are supported on RedHawk Linux "debug" kernels.

`kdb` is the default debugger. Information about using **`kdb`** can be found in the kernel source directory under **`Documentation/kdb`**. Note that to use **`kdb`** on a system with a USB keyboard, a serial console must be configured. Refer to Appendix H for instructions for setting up a serial console.

`kgdb` allows the kernel to be debugged with **`gdb`** as if it were a user application. **`gdb`** runs on a separate system containing a copy of the **`vmlinux`** file and the matching source, and communicates with the kernel being debugged through the console's serial port. More information about **`kgdb`** can be found on the web at kgdb.sourceforge.net.

Methods for specifying **kgdb** instead of **kdb** as the active debugger on these systems include:

- specifying the `gdb` option at boot
- specifying the `gdb vmlinux` command to connect to the **kgdb** serial port
- using the Alt+SysRq+G key combination

Note that some modifications have been made to **gdb** for use with RedHawk Linux on Opteron systems:

- A modification has been made to avoid an error check in stack backtrace, which expects that the stack grew in a consistent direction. Because the x86_64 kernel uses an interrupt stack in addition to the normal process stack, setting this option triggers this error check when tracing back from the interrupt to the process stack. Set this option with the command:

  ```
  gdb> set backtrace switch-stacks on
  ```

- Another modification allows a list of functions to be specified which should be skipped over when picking a frame to display in an “info thread” command. Set this option with the command:

  ```
  gdb> set skip-frame thread_return,schedule_timeout
  ```

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 postmortem analysis of a kernel core dump. See Chapter 12 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 **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 **/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. It 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. 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
- 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. See the man pages for details.

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 threads in a multithreaded process 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, which support interprocess semaphores, 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. The "aio" functions are documented in info pages ('info libc") on the system.

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. Refer to the man pages for more information.

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. Refer to the man pages for more information.

Clocks and Timers

Support for high-resolution POSIX clocks and timers is included in RedHawk Linux. System-wide POSIX clocks 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.

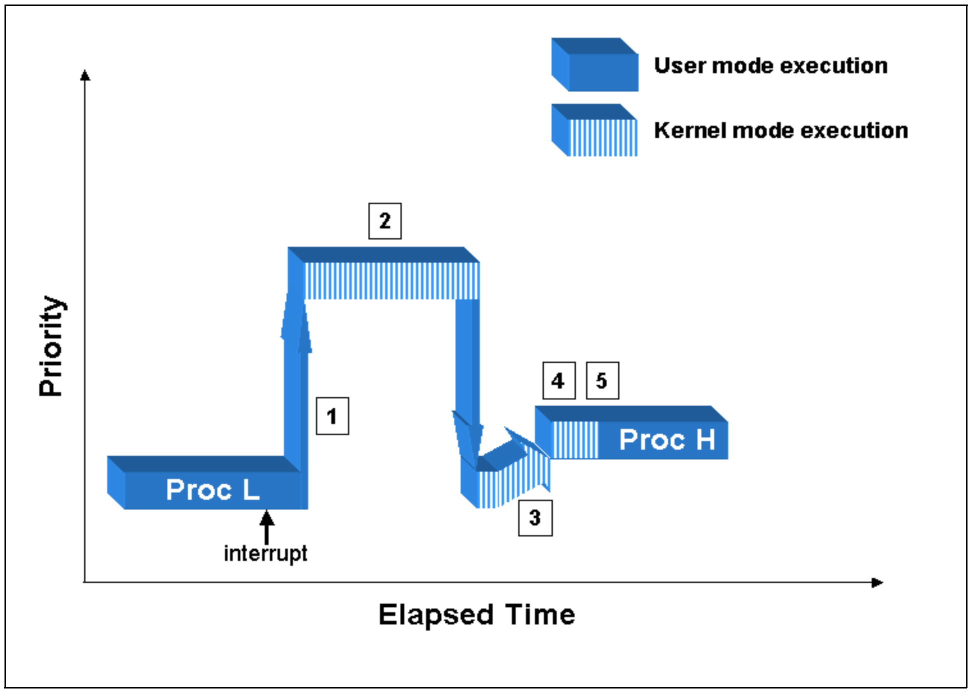
The term *process dispatch latency* denotes the time that elapses from the occurrence of an external event, which is signified by an interrupt, until the process waiting for that external event executes its first instruction in user mode. For real-time applications, the worst-case process dispatch latency 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.

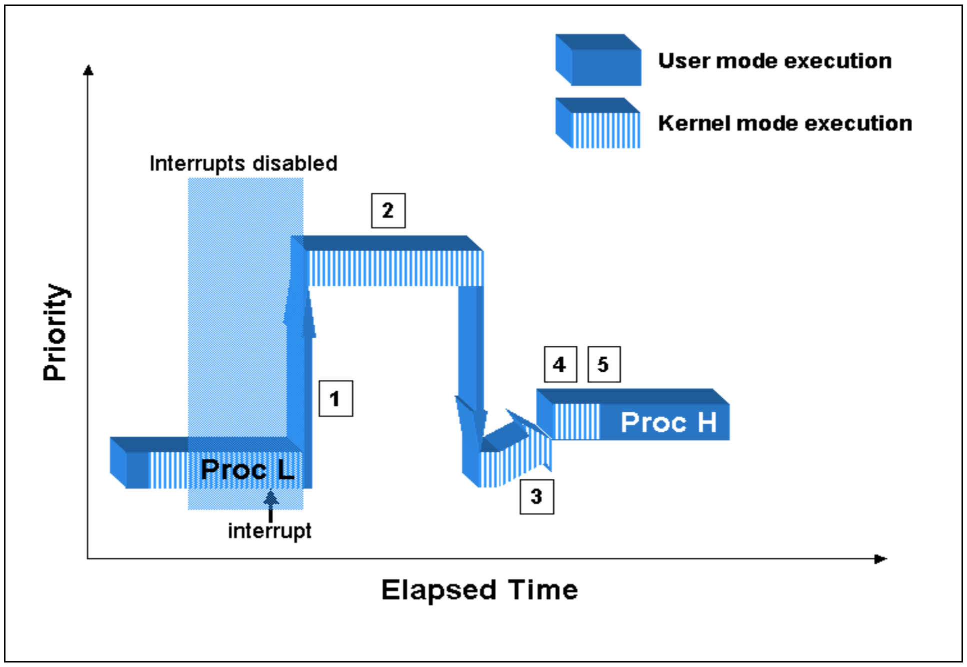
Effect of Disabling Interrupts

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.

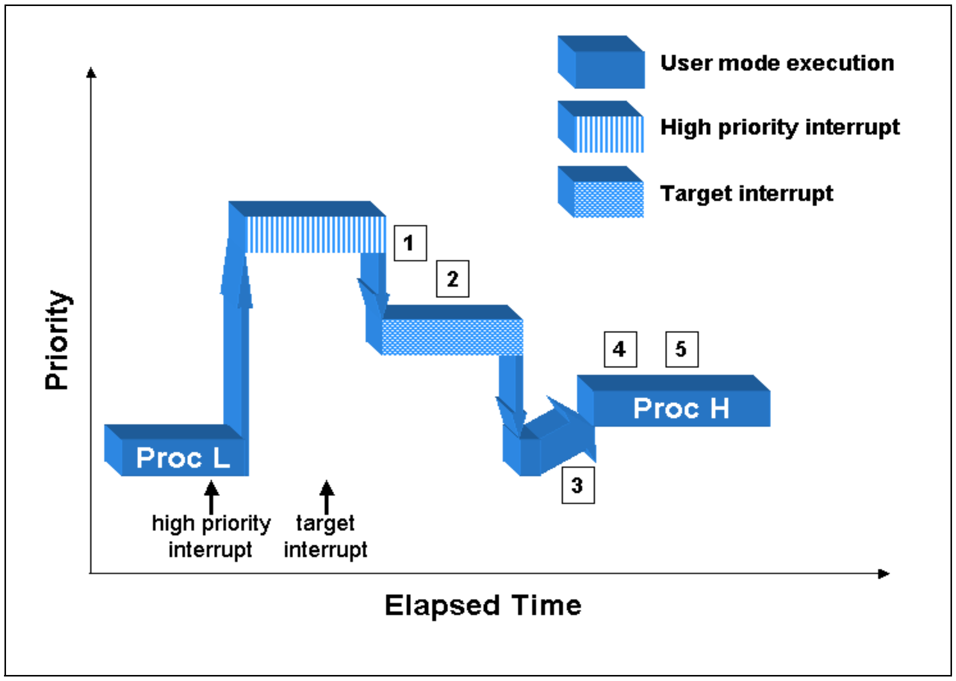
Figure 2-2 Effect of Disabling Interrupts on Process Dispatch Latency



Effect of Interrupts

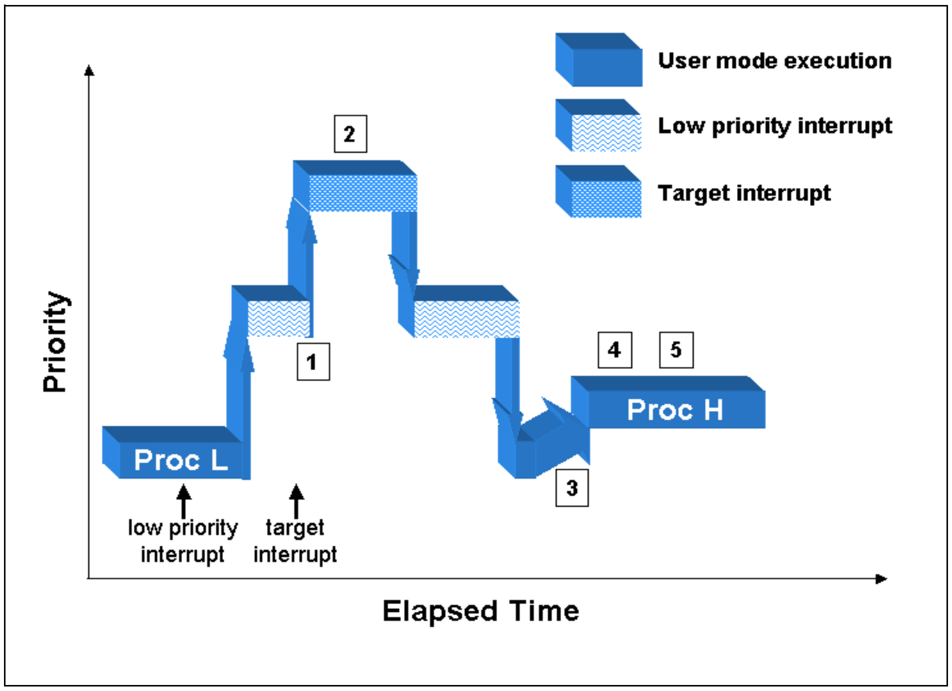
The receipt of an interrupt affects process dispatch latency in much the same way that disabling interrupts does. When a hardware interrupt is received, the system will block interrupts of the same or lesser priority than the current interrupt. The simple case is illustrated in Figure 2-3, where a higher 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

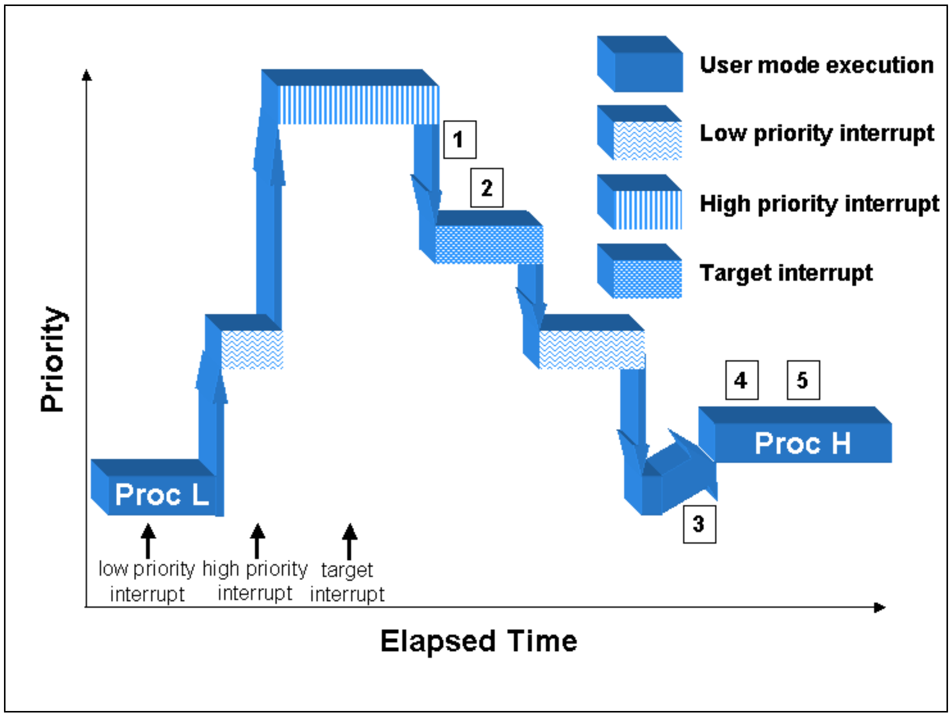


One of the biggest differences between the effect of disabling interrupts and receipt of an interrupt in terms of the impact on process dispatch latency is the fact that interrupts occur asynchronously to the execution of an application and at unpredictable times. This is important to understanding the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

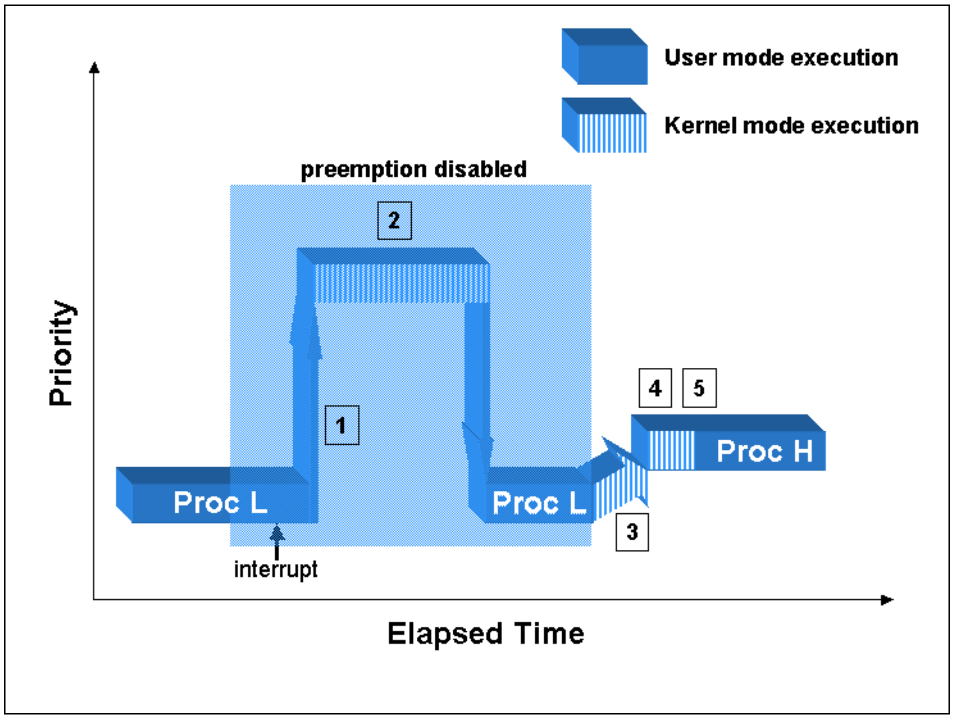
Figure 2-5 Effect of Multiple Interrupts on Process Dispatch Latency



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” 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 or thread in the system has a CPU affinity mask. The CPU affinity mask determines on which CPUs the process or thread is allowed to execute. The CPU affinity mask is inherited from the parent 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 and threads 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 or thread on a CPU shielded from background processes, it must have a CPU affinity mask that specifies ONLY shielded CPUs.

Certain kernel daemons created by Linux are replicated on every CPU in the system. Shielding a CPU from processes will not remove one of these "per-CPU" daemons from the shielded CPU. The impact of these daemons can be avoided through kernel configuration or careful control of application behavior. The kernel daemons, their functionality and methods to avoid jitter from per-CPU kernel daemons are described in Appendix F.

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**.

On the i386 architecture, the **kirqd** daemon periodically adjusts IRQ affinities in an attempt to balance interrupt load across CPUs. This daemon conflicts with interrupt-shielding and has been disabled by default in all RedHawk Linux kernel configurations through the IRQBALANCE kernel configuration option. It can be enabled with the kernel boot parameter "noirqbalance" or by enabling the IRQBALANCE kernel parameter.

Note that if it is desirable to disable all interrupts on all CPUs, the recommended procedure is to shield all CPUs from interrupts except one, then make a call to **`local_irq_disable(2)`** on the unshielded CPU. See the man page for details.

Certain activities can cause interrupts to be sent to shielded CPUs. These cross processor interrupts are used as a method for forcing another CPU to handle some per-CPU specific task. Cross processor interrupts can potentially cause noticeable jitter for shielded CPUs. Refer to Appendix G for a full discussion.

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.

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 timer, the local interrupt is effectively disabled and 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 timer 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 lost when the local interrupt is disabled on a particular CPU is the low resolution mechanism for CPU execution time accounting. This is the mechanism that measures how much CPU time is used by each process that executes on this CPU. Whenever the local interrupt fires, the last clock tick’s worth of time is charged to the process that was interrupted. If high resolution process accounting is configured, then CPU time will be accurately accounted for regardless of whether or not the local interrupt is enabled. High resolution process accounting is discussed in Chapter 7, “System Clocks and Timers.”

When a CPU is shielded from the local timer, 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 and alternatives that are available for some of the features.

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.

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

Option	Description
`--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

All users can read these files, but only root or users with the **CAP_SYS_NICE** capability and file permissions 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 or thread to a set of CPUs. The set of CPUs where a process is allowed to run is known as its CPU affinity.

By default, a process or thread can execute on any CPU in the system. Every process or thread has a bit mask, or CPU affinity, that determines the CPU or CPUs on which it can be scheduled. A process or thread 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 or threads by specifying the MPA_PRC_SETBIAS command on a call to **mpadvise(3)**, or the **-b** *bias* option to the **run(1)** command. **sched_setaffinity(2)** can also be used to set CPU affinity.

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' or thread's 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.
MPA_CPU_LMEM	Returns a mask indicating which CPUs have local memory on a system with NUMA support. CPUs brought down with the **cpu(1)** command are still included.

Control Commands

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

MPA_PRC_GETBIAS	Returns the CPU set for the CPU affinity of all threads in the specified process (MPA_PID) or the exact unique bias for the specified thread (MPA_TID).
MPA_PRC_SETBIAS	Sets the CPU affinity of all threads in the specified processes (MPA_PID) or the unique CPU affinity for the specified thread (MPA_TID) 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	Returns a CPU set with exactly one CPU in it that corresponds to the CPU where the specified thread is currently running (or waiting to run) (MPA_TID). When MPA_PID is specified, returns one CPU for non-threaded programs and the set of CPUs in use by all threads of a multi-threaded program. Note that it is possible that the CPU assignment may have already changed by the time the value is returned.

Using *which* and *who*

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

MPA_PID	a specific process and all its threads
MPA_TID	a specific thread
MPA_PGID	a process group
MPA_UID	a user
MPA_LWPID	same as MPA_TID (compatible with PowerMAX)

who — Interpreted relative to *which*:

a process identifier
a thread 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.

Using MPA_PID with a reference to a single subordinate (non-primordial) thread applies to the containing process as it does when a primordial thread is supplied.

When using MPA_TID, who must be the numeric thread ID (as returned by gettid), not a pthread identifier associated with the POSIX Threads library.

Assigning CPU Affinity to init

All general processes are a descendant of **`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 the following methods:

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

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

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

```
$ cpu
log id
(phys id)  state  shielding
---------  -----  -------------
0 (0)        up     none
1 (0)        up     proc irq ltmr
2 (1)        up     none
3 (1)        up     none
```

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

```
$ 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
           CPU0       CPU1       CPU2       CPU3
  0:   665386907          0          0          0   IO-APIC-edge  timer
  4:        2720          0          0          0   IO-APIC-edge  serial
  8:           1          0          0          0   IO-APIC-edge  rtc
  9:           0          0          0          0   IO-APIC-level  acpi
 14:     9649783          1          2          3   IO-APIC-edge  ide0
 15:          31          0          0          0   IO-APIC-edge  ide1
 16:   384130515          0          0          0   IO-APIC-level  eth0
 17:           0          0          0          0   IO-APIC-level  rcim,Intel..
 18:    11152391          0          0          0   IO-APIC-level  aic7xxx,...
 19:           0          0          0          0   IO-APIC-level  uhci_hcd
 23:           0          0          0          0   IO-APIC-level  uhci_hcd
NMI:   102723410  116948412          0          0   Non-maskable interrupts
LOC:   665262103  665259524 665264914 665262848   Local interrupts
RES:    36855410   86489991   94417799   80848546   Rescheduling interrupts
CAL:        2072       2074       2186       2119   function call interrupts
TLB:       32804      28195      21833      37493   TLB shootdowns
TRM:           0          0          0          0   Thermal event interrupts
SPU:           0          0          0          0   Spurious interrupts
ERR:           0          0          0          0   Error interrupts
MIS:           0          0          0          0   APIC errata fixups
```

The RCIM is assigned to IRQ 17 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 17. 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 17 to CPU 1:

```
$ echo 2 > /proc/irq/17/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/17/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 17 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 **run(1)** command can be used to check the program's affinity:

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

Note that the value returned for "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
- controlling cache access
- in a NUMA system, binding a program to local memory
- judicious use of hyper-threading
- avoiding a low memory state

Locking Pages in Memory

You can avoid the overhead associated with paging and swapping by using **`mlockall(2)`**, **`munlockall(2)`**, **`mlock(2)`**, and **`munlock(2)`**. 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** and **CAP_SYS_RAWIO** capabilities (for additional information on capabilities, refer to Chapter 13 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 part 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 part of an interrupt routine can be handled by 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.”

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.

Avoiding Cache Thrashing

Application writers trying to approach ‘constant runtime determinism’ must pay attention to how their program’s variables will be mapped into the CPU’s caches. For example, if a program has a variable i and a variable j and both are heavily used, and the memory locations assigned to them result in both i and j being in the same cache line, every reference to i will eject j from the cache and vice versa. This is called thrashing the cache and its occurrence is devastating to performance.

To avoid this, place all heavily accessed data close to one another in memory. If that range is smaller than the cache size, all the data will be guaranteed different cache lines. To view how large the CPU’s cache is, execute ‘‘grep cache /proc/cpuinfo’’.

Make sure that your system has cache that is larger than the data set of heavily accessed variables. For example, if you have 600,000 bytes of variables and you have a system with only .5 MB of cache, cache thrashing will be unavoidable.

If you have less than one page (4096 bytes) of critical variables, then you can force them to be in separate cache lines by forcing all the data to be in one physical page:

```
struct mydata{
    int i;
    int j;
    float fp[200];
    floag fq[200]; } __attribute__((__aligned__(4096)));

struct mydata mydata;
```

Binding to NUMA Nodes

On a system with non-uniform memory access (NUMA), such as an iHawk Opteron system, it takes longer to access some regions of memory than others. The memory on a NUMA system is divided into nodes, where a node is defined to be a region of memory and all CPUs that reside on the same physical bus as the memory region of the NUMA node. If a program running on this type of system is not NUMA-aware, it can perform poorly.

By default, pages are allocated from the node where the local CPU (from which the program is executed) resides, but the task or virtual areas within the task can be specified to allocate pages from certain nodes for better determinism and control. Refer to Chapter 10 for information about NUMA.

Understanding Hyper-threading

Hyper-threading is a feature of the Intel Pentium Xeon processor in iHawk 860 systems. It 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" refers 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)`** identify each logical CPU. This allows multiprocessor-capable software to run unmodified on twice as many logical processors. While hyper-threading technology does 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 occurs because one processor with two logical CPUs can more efficiently utilize execution resources. During normal program operation on a non-hyper-threaded CPU, execution resources on the chip often 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 are otherwise idle with only one thread executing. For example while one logical CPU is stalled waiting for a fetch from memory to complete, its sibling 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 parallelism is one logical processor executing a floating-point operation while the other 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 faster execution for a multi-thread workload, it can be problematic for real-time applications. 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 it takes to execute a code segment on a hyper-threaded CPU is not as predictable as on a non-hyper-threaded CPU.

The designer of a parallel real-time application should decide whether hyper-threading makes sense for his application. Will the application benefit from its tasks running in parallel on a hyper-threaded CPU as compared to running sequentially? If so, the developer can make measurements to determine how much 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 an unacceptable amount of jitter is introduced into a real-time application because of hyper-threading, then the affected task should be run on a shielded CPU with its sibling CPU marked down via the **`cpu(1)`** command. An example of a system with a CPU marked down is given later in this chapter. It should be noted that certain cross processor interrupts will still be handled on a downed CPU (see the **`cpu(1)`** man page for more information). If desired, hyper-threading can be disabled on a system-wide basis. See the section “System Configuration” below for details.

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. There are twice as many logical processors available to the system, however, 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. 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 general-purpose registers, control registers, 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.

Interrupts sent to a specific logical CPU are handled only by that logical CPU, regardless of whether hyper-threading is enabled or disabled.

System Configuration

The following items affect system-wide hyper-thread availability:

- The system must contain Intel Pentium Xeon processors.
- The kernel must be configured with hyper-threading enabled through the X86_HT kernel tunable accessible under Processor Type and Features on the Kernel Configuration GUI. Hyper-threading is enabled by default on all RedHawk i386 pre-defined kernels.
- Hyper-threading must be enabled in the BIOS to be available for use. Refer to your hardware documentation to determine which BIOS setting is involved, if needed.

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.

Note that with hyper-threading enabled, commands like **`top(1)`** and **`run(1)`** report twice as many CPUs as were previously present on systems running a version of RedHawk Linux prior to release 1.3 that did not have hyper-threading support. When hyper-threading is disabled on a system-wide basis, the logical CPU numbers are equivalent to the physical CPU numbers.

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.

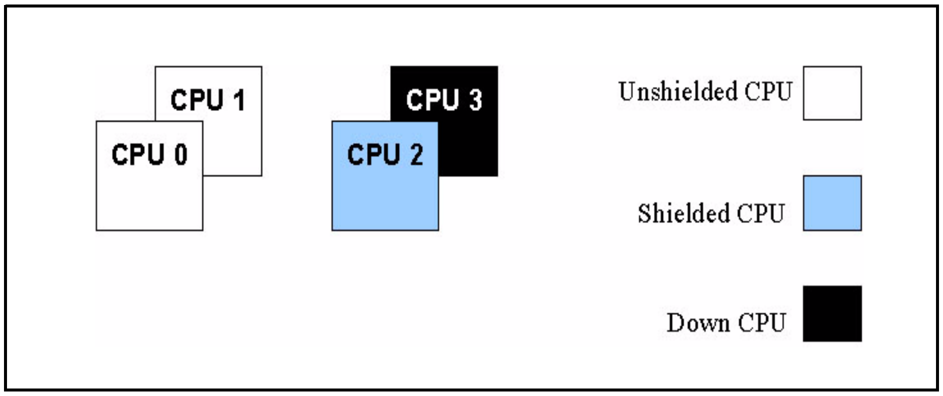
Standard Shielded CPU Model

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



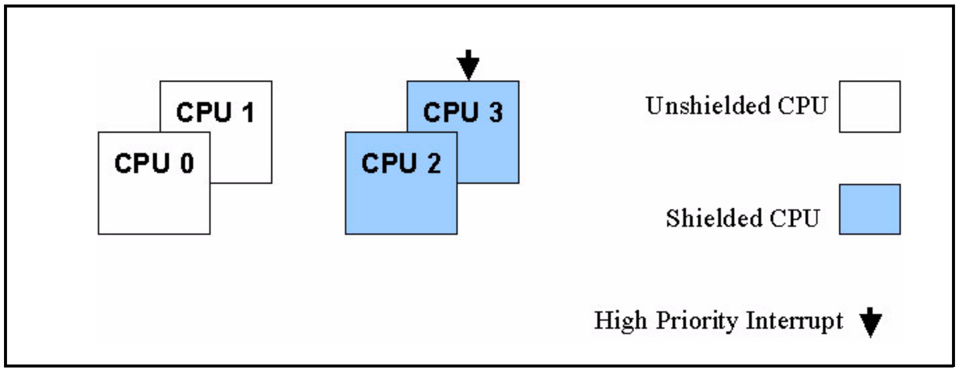
The commands to set up this configuration are:

```
$ 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



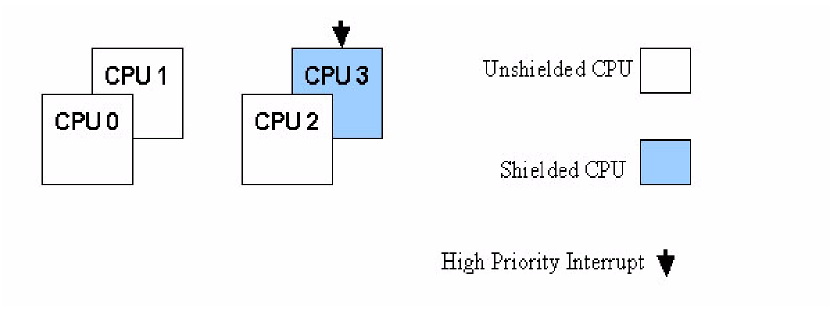
The benefit of this approach is that it provides a small amount of parallelism between the interrupt routine (which runs on CPU 3) and execution of high priority tasks on the sibling CPU (the program awaiting the interrupt runs on CPU 2). Because the interrupt routine is the only code executing on CPU 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.

Avoiding a Low Memory State

Ensure that your system has adequate physical RAM. Concurrent's real-time guarantees assume a properly configured system with adequate RAM for real-time application usage. In low-memory situations, real-time deadlines may be sacrificed to better ensure system integrity and maintain proper system behavior. When Linux runs out of memory, it randomly selects processes to kill in an attempt to free up memory so that other processes can proceed.

Memory usage can be monitored using a number of tools including **`/proc/meminfo`**, **`free(1)`** and **`vmstat(8)`**.

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” chapter for more information on how to correct this problem.
- Avoid unloading kernel modules. This action creates and destroys a number of per-CPU **`kmodule`** daemons that can add unnecessary jitter on the CPU.
- There may be possible real-time issues when starting and stopping the X server while running time-critical applications on shielded CPU(s). Depending upon the type of graphics cards your system uses, this may result in numerous cross-processor interrupts that slow performance. If you are experiencing this, refer to Appendix G for methods to reduce these interrupts.

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 and shared memory facilities.

Appendix A contains example programs that illustrate the use of the POSIX and System V message queue facilities.

Overview

RedHawk Linux provides several mechanisms that allow processes to exchange data. These mechanisms include message queues, shared memory and semaphores based on the IEEE Standard 1003.1b-1993 as well as those included in the System V Interprocess Communication (IPC) package. Message queues and shared memory are discussed in this chapter; semaphores are discussed in Chapter 5, Interprocess Synchronization.

Message queues allow one or more processes to write messages to be read by one or more reading processes. Facilities are provided 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 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. The sections “POSIX Message Queues” and “System V Messages” in this chapter describe these facilities.

Shared memory allows cooperating processes to share data through a common area of memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there.

As with messaging, POSIX and System V shared memory functionality operate independent of each other. It is recommended that you use a System V shared memory area in an application in which data placed in shared memory are temporary and do not need to exist following a reboot of the system. Data in a System V shared memory area are kept only in memory. No disk file is associated with that memory and therefore no disk traffic is generated by the **`sync(2)`** system call. Also, System V shared memory allows you to bind a shared memory segment to a section of physical I/O memory. Refer to the section “System V Shared Memory” for information about this facility.

An alternative to using System V shared memory is to use the **`mmap(2)`** system call to map a portion of the **`/dev/mem`** file. For information on the **`mmap`** system call, refer to Chapter 9, “Memory Mapping.” For information on the **`/dev/mem`** file, refer to the **`mem(4)`** man page.

POSIX shared memory interfaces are mapped to a disk file in the **`/var/tmp`** directory. If this directory is mounted on a **`memfs`** file system, then no extra disk traffic is generated to flush the shared data during the **`sync`** system call. If this directory is mounted on a regular disk partition, then disk traffic will be generated during the **`sync`** system call to keep the shared data updated in the mapped disk file. Whether the data that are written to POSIX shared memory are saved in a file or not, those data do not persist following a reboot of the system. The POSIX shared memory functionality is described in the "POSIX Shared Memory" section of this chapter.

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.

"Basic Concepts" presents basic concepts related to the use of POSIX message queues. "Advanced Concepts" presents advanced concepts. A sample program is provided in Appendix A.

Basic Concepts

POSIX message queues are implemented as files in the **mqueue** file system.

RedHawk mounts the mqueue file system on **`/dev/mqueue`**, although it can be mounted on any mount point. If an entry for **`/dev/mqueue`** is added to **`/etc/fstab`** by the system administrator, it is mounted at boot time per the entry; otherwise, the **`/etc/init.d/ccur`** script performs the mount.

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

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

The 'none' argument reflects that this is a mount point and not a device file. The major number 0 (reserved as null device number) is used by the kernel for unnamed devices (e.g. non-device mounts).

When mounted, a message queue appears as a file in the directory **`/dev/mqueue`**; for example:

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

The current implementation does not require that the **mqueue** file system be mounted; programs will run whether it is mounted or not. However when not mounted, mqueues are not visible and cannot be manipulated from the command line.

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. See "Message Queue Library Routines" for further discussion on using the interface.

It is possible to have multiple instances of mount points active simultaneously. These will be mirror images; what happens under one path is seen under all paths. This means that mqueues with the same name seen under different paths refer to the same mqueue; i.e., **`/dev/mqueue/my_queue`** and **`/mnt/mqueue/my_queue`** refer to the same mqueue. This is not the intended method of operation but it illustrates that an mqueue is a system-wide global data structure.

`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:

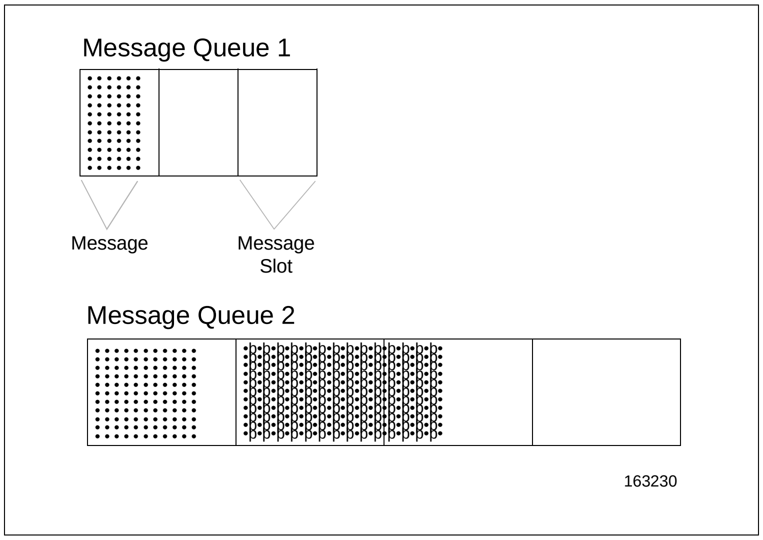
QSIZE	number of bytes in memory occupied by the entire queue
NOTIFY	notification marker (see **`mq_notify(2)`**)
SIGNO	the signal to be generated during notification
NOTIFY_PID	which process should be notified

The following system limits apply to POSIX message queues:

DFLT_QUEUESMAX	64	max number of message queues
DFLT_MSGMAX	40	max number of messages in each queue
DFLT_MSGSIZEMAX	16384	max message size
HARD_MSGMAX	(131072/sizeof(void*))	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.

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 "The mq_unlink Routine" and "The mq_open Routine," respectively.

Message Queue Library Routines

The POSIX library routines that support message queues depend on a message queue attribute structure. "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`** ...

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 "The mq_getattr Routine" and "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.

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 <u>not</u> 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 "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.

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 “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.

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.

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 "The mq_receive and mq_timedreceive Routines" and **`mq_notify`** in "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 “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.

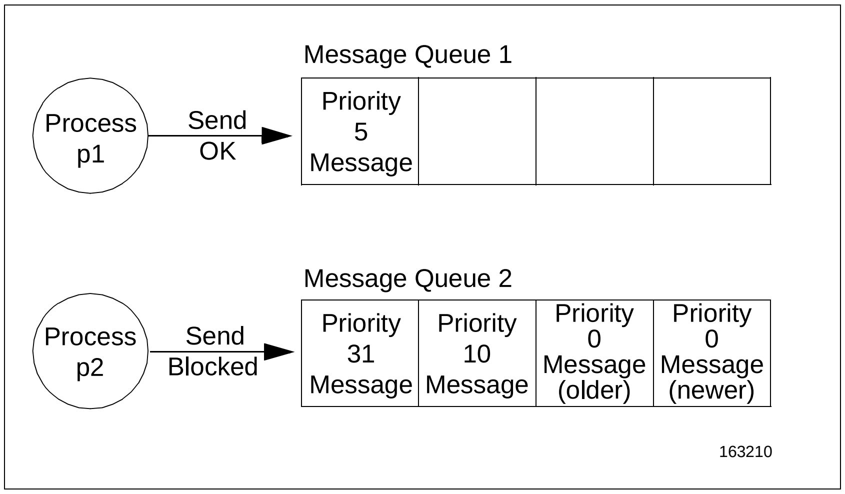
abs_timeout
a timeout value in nanoseconds. The range is 0 to 1000 million. If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue is terminated when the timeout expires.

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.

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 "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:

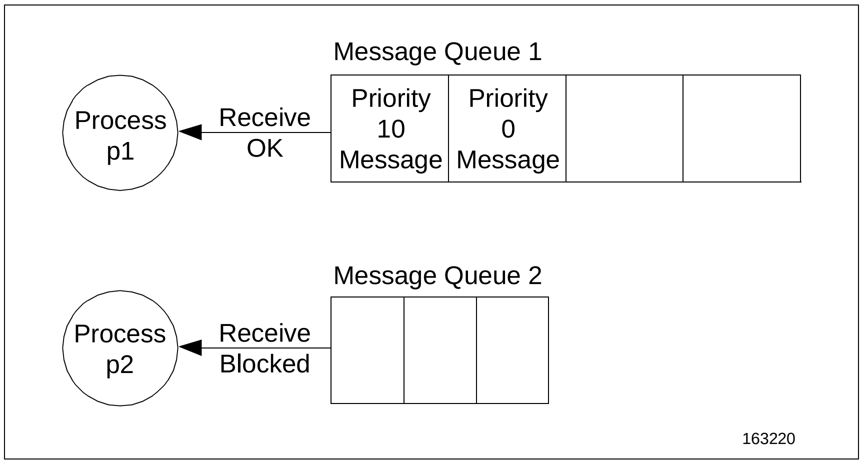
mqdes a message queue descriptor obtained from an **`mq_open`**. If O_NONBLOCK is set in *mqdes* and the referenced message queue is empty, nothing is read, and **`mq_receive`** returns an error. If O_NONBLOCK is not set in *mqdes* and the specified message queue is empty, **`mq_receive`** blocks until 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 "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.

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 "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.

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 “The Message Queue Attribute Structure.” For information on the **`mq_send`** and **`mq_receive`** routines, see “The mq_send and mq_timedsend Routines” and “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 “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.

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 "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.

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 `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 creates 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, 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 "The msgget System Call" 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 operations) 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 "The msgsnd and msgrcv System Calls" for details of these 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 "The msgctl System Call" for details of the **`msgctl`** system call.

Refer to Appendix A for a sample program using System V message queues. Additional sample programs can be found online that illustrate in depth use of each of the System V system calls. These are referenced within the section in this chapter that explains the system call.

The msgget System Call

`msgget(2)` creates a new message queue or identifies an existing one.

This section describes the **`msgget`** system call. For more detailed information, see the **`msgget(2)`** man page. A program illustrating use of this call can be found at **`/usr/share/doc/ccur/examples/msgget.c`** with extensive comments provided in **`README.msgget.txt`**.

Synopsis

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

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

All of the `#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 unique message queue identifier, *msqid*. (The *msqid* is discussed in the “Using Messages” section earlier in this chapter.) Upon failure, the external variable **`errno`** is set to indicate the reason for failure and **`-1`** is returned.

A new *msqid* with an associated message queue and data structure is created if one of the following conditions is true:

- *key* is equal to IPC_PRIVATE
- *key* does not already have a *msqid* associated with it and (*msgflg* and IPC_CREAT) is “true” (not zero).

The value of *msgflg* is a combination of:

- control commands (flags)
- operation permissions

Control commands are predefined constants. The following control commands apply to the **`msgget`** system call and are defined in the **`<bits/ipc.h>`** header file, which is internally included by the **`<sys/ipc.h>`** header file:

IPC_CREAT	used to create a new segment. If not used, **`msgget`** will find the message queue associated with *key*, verify access permissions and ensure the segment is not marked for destruction.
IPC_EXCL	used with IPC_CREAT to cause 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 (unique) identifier when it has not.

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 shows 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 ORing the octal values for the operation permissions desired. That is, if "read by user" and "read/write by others" is desired, the code value would be 00406 (00400 plus 00006).

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

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

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>`**.

A list of message queue limit values may be obtained with the **`ipcs(8)`** command by using the following options. See the man page for further details.

```
ipcs -q -l
```

Refer to the **`msgget(2)`** man page for specific associated data structure initialization as well as the specific error conditions.

The msgctl System Call

`msgctl(2)` is used to perform control operations on message queues.

This section describes the **`msgctl(2)`** system call. For more detailed information, see the **`msgctl(2)`** man page. A program illustrating use of this call can be found at **`/usr/share/doc/ccur/examples/msgctl.c`** with extensive comments provided in **`README.msgctl.txt`**.

Synopsis

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

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

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

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 that has already been created 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. Read permission is required.
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 along with its associated data structure

NOTE

The **`msgctl(2)`** service also supports the IPC_INFO, MSG_STAT and MSG_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

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 **`-q`** *msgid* or the **`-Q`** *msgkey* option, where *msgid* specifies the identifier for the message queue and *msgkey* specifies the key associated with 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.

The msgsnd and msgrcv System Calls

The message operations system calls, **`msgsnd`** and **`msgrcv`**, are used to send and receive messages.

This section describes the **`msgsnd`** and **`msgrcv`** system calls. For more detailed information, see the **`msgop(2)`** man page. A program illustrating use of these calls can be found at **`/usr/share/doc/ccur/examples/msgop.c`** with extensive comments provided in **`README.msgop.txt`**.

Synopsis

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

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

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

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

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 that has already been created 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.

Receiving a Message

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`**.

POSIX Shared Memory

The POSIX shared memory interfaces allow cooperating processes to share data and more efficiently communicate through the use of a shared memory object. A *shared memory object* is defined as a named region of storage that is independent of the file system and can be mapped to the address space of one or more processes to allow them to share the associated memory.

The interfaces are briefly described as follows:

`shm_open`	create a shared memory object and establish a connection between the shared memory object and a file descriptor
`shm_unlink`	remove the name of a shared memory object

Procedures for using the **`shm_open`** routine are presented in "Using the shm_open Routine." Procedures for using the **`shm_unlink`** routine are presented in "Using the shm_unlink Routine."

In order for cooperating processes to use these interfaces to share data, one process completes the following steps. Note that the order in which the steps are presented is typical, but it is not the only order that you can use.

STEP 1:	Create a shared memory object and establish a connection between that object and a file descriptor by invoking the **`shm_open`** library routine, specifying a unique name, and setting the O_CREAT and the O_RDWR bit to open the shared memory object for reading and writing.
STEP 2:	Set the size of the shared memory object by invoking the **`ftruncate(2)`** system call and specifying the file descriptor obtained in Step 1. This system call requires that the memory object be open for writing. For additional information on **`ftruncate(2)`**, refer to the corresponding man page.
STEP 3:	Map a portion of the process's virtual address space to the shared memory object by invoking the **`mmap(2)`** system call and specifying the file descriptor obtained in Step 1 (see the "Memory Mapping" chapter for an explanation of this system call).

To use the shared memory object, any other cooperating process completes the following steps. Note that the order in which the steps are presented is typical, but it is not the only order that you can use.

STEP 1:	Establish a connection between the shared memory object created by the first process and a file descriptor by invoking the **`shm_open`** library routine and specifying the same name that was used to create the object.
STEP 2:	If the size of the shared memory object is not known, obtain the size of the shared memory object by invoking the **`fstat(2)`** system call and specifying the file descriptor

obtained in Step 1 and a pointer to a `stat` structure (this structure is defined in <**`sys/stat.h`**>). The size of the object is returned in the `st_size` field of the `stat` structure. Access permissions associated with the object are returned in the `st_modes` field. For additional information on **`fstat(2)`**, refer to the corresponding system manual page.

STEP 3: Map a portion of the process's virtual address space to the shared memory object by invoking **`mmap`** and specifying the file descriptor obtained in Step 1 (see the "Memory Mapping" chapter for an explanation of this system call).

Using the shm_open Routine

The **`shm_open(3)`** routine allows the calling process to create a POSIX shared memory object and establish a connection between that object and a file descriptor. A process subsequently uses the file descriptor that is returned by **`shm_open`** to refer to the shared memory object on calls to**`ftruncate(2)`**, **`fstat(2)`**, and **`mmap(2)`**. After a process creates a shared memory object, other processes can establish a connection between the shared memory object and a file descriptor by invoking **`shm_open`** and specifying the same name.

After a shared memory object is created, all data in the shared memory object remain until every process removes the mapping between its address space and the shared memory object by invoking **`munmap(2)`**, **`exec(2)`**, or **`exit(2)`** and one process removes the name of the shared memory object by invoking **`shm_unlink(3)`**. Neither the shared memory object nor its name is valid after your system is rebooted.

Synopsis

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

int shm_open(const char *name, int oflag, mode_t mode);
```

The arguments are defined as follows:

name a pointer to a null–terminated string that specifies the name of the shared memory object. Note that this string may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretation of it (**`/shared_obj`** and **`shared_obj`** are interpreted as the same name). If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that *name* begin with a slash character.

oflag an integer value that sets one or more of the following bits:

Note that O_RDONLY and O_RDWR are mutually exclusive bits; one of them must be set.

O_RDONLY	causes the shared memory object to be opened for reading only
O_RDWR	causes the shared memory object to be opened for reading and writing. Note that the process that creates the shared memory object must open it for writing in order to be able to set its size by invoking **`ftruncate(2)`**.
O_CREAT	causes the shared memory object specified by *name* to be created if it does not exist. The memory object's user ID is set to the effective user ID of the calling process; its group ID is set to the effective group ID of the calling process; and its permission bits are set as specified by the *mode* argument. If the shared memory object specified by *name* exists, setting O_CREAT has no effect except as noted for O_EXCL.
O_EXCL	causes **`shm_open`** to fail if O_CREAT is set and the shared memory object specified by *name* exists. If O_CREAT is not set, this bit is ignored.
O_TRUNC	causes the length of the shared memory object specified by *name* to be truncated to zero if the object exists and has been opened for reading and writing. The owner and the mode of the specified shared memory object are unchanged.

mode an integer value that sets the permission bits of the shared memory object specified by *name* with the following exception: bits set in the process's file mode creation mask are cleared in the shared memory object's mode (refer to the **`umask(2)`** and **`chmod(2)`** man pages for additional information). If bits other than the permission bits are set in *mode*, they are ignored. A process specifies the *mode* argument only when it is creating a shared memory object.

If the call is successful, **`shm_open`** creates a shared memory object of size zero and returns a file descriptor that is the lowest file descriptor not open for the calling process. The FD_CLOEXEC file descriptor flag is set for the new file descriptor; this flag indicates that the file descriptor identifying the shared memory object will be closed upon execution of the **`exec(2)`** system call (refer to the **`fcntl(2)`** system manual page for additional information).

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

Using the shm_unlink Routine

The **shm_unlink(3)** routine allows the calling process to remove the name of a shared memory object. If one or more processes have a portion of their address space mapped to the shared memory object at the time of the call, the name is removed before **shm_unlink** returns, but data in the shared memory object are not removed until the last process removes its mapping to the object. The mapping is removed if a process invokes **munmap(2)**, **exec(2)**, or **exit(2)**.

Synopsis

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

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

The argument is defined as follows:

name	a pointer to a null–terminated string that specifies the shared memory object name that is to be removed. Note that this string may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretation of it (**/shared_obj** and **shared_obj** are interpreted as the same name). If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that *name* begin with a slash character.

A return value of 0 indicates that the call to **shm_unlink** has been successful. A return value of –1 indicates that an error has occurred; `errno` is set to indicate the error. Refer to the **shm_unlink(3)** man page for a listing of the types of errors that may occur. If an error occurs, the call to **shm_unlink** does not change the named shared memory object.

System V Shared Memory

Shared memory allows two or more processes to share memory and, consequently, the data contained therein. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware-dependent.

A process initially creates a shared memory segment using the **shmget(2)** system call. Upon creation, this process sets the overall operation permissions for the shared memory segment, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment.

If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

The shared memory segments on the system are visible via the **/proc/sysvipc/shm** file and **ipcs(8)** using the **-m** option.

Shared memory operations, **shmat(2)** (shared memory attach) and **shmdt(2)** (shared memory detach), can be performed on a shared memory segment. **shmat** allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed. **shmdt** allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl(2)** system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **shmctl** system call.

A process can bind a shared memory segment to a section of I/O memory by using the **shmbind(2)** system call. See the section "Binding a Shared Memory Segment to I/O Space" for details of the **shmbind** system call.

To facilitate use of shared memory by cooperating programs, a utility called **shmdefine(1)** is provided. Procedures for using this utility are explained in "The shmdefine Utility". To assist you in creating a shared memory segment and binding it to a section of physical memory, a utility called **shmconfig(1)** is also provided. Procedures for using this utility are explained in "The shmconfig Command".

Using Shared Memory

Sharing memory between processes occurs on a virtual segment basis. There is only one copy of each individual shared memory segment existing in the operating system at any time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (*shmid*); it is used to identify or refer to the associated data structure. This identifier is available to any process in the system, subject to normal access restrictions.

The data structure includes the following for each shared memory segment:

- Operation permissions
- Segment size
- Segment descriptor (for internal system use only)
- PID performing last operation
- PID of creator
- Current number of processes attached
- Last attach time
- Last detach time
- Last change time

The definition of the associated shared memory segment data structure shmid_ds includes the members shown in Figure 3-6.

Figure 3-6 Definition of shmid_ds Structure

```
struct shmid_ds {
     struct    ipc_perm shm_perm;  /* operation perms */
     int  shm_segsz;          /* size of segment (bytes) */
     time_t    shm_atime;          /* last attach time */
     time_t    shm_dtime;          /* last detach time */
     time_t    shm_ctime;          /* last change time */
     unsigned short shm_cpid; /* pid of creator */
     unsigned short shm_lpid; /* pid of last operator */
     short     shm_nattch;         /* no. of current attaches */
};
```

The C programming language data structure definition for the shared memory segment data structure shmid_ds is located in the **<sys/shm.h>** header file.

Note that the shm_perm member of this structure uses ipc_perm as a template. The ipc_perm data structure is the same for all IPC facilities; it is located in the **<sys/ipc.h>** header file.

The **shmget(2)** system call performs two tasks:

- It gets a new shared memory identifier and creates an associated shared memory segment data structure.
- It returns an existing shared memory identifier that already has an associated shared memory segment data structure.

The task performed is determined by the value of the *key* argument passed to the **shmget** system call.

The *key* can be an integer that you select, or it can be an integer that you have generated by using the **ftok** subroutine. The **ftok** subroutine generates a key that is based upon a path name and identifier that you supply. By using **ftok**, you can obtain a unique key and control users' access to the key by limiting access to the file associated with the path name. If you wish to ensure that a key can be used only by cooperating processes, it is recommended that you use **ftok**. This subroutine is specified as follows:

key_t ftok(*path_name*, *id*)

The *path_name* argument specifies a pointer to the path name of an existing file that should be accessible to the calling process. The *id* argument specifies a character that uniquely identifies a group of cooperating processes. **Ftok** returns a key that is based on the specified *path_name* and *id*. Additional information on the use of **ftok** is provided in the **ftok(3)** man page.

If the *key* is not already in use for an existing shared memory identifier and the IPC_CREAT flag is set in *shmflg*, a new identifier is returned with an associated shared memory segment data structure created for it provided no system-tunable parameters would be exceeded.

There is also a provision for specifying a *key* of value zero which is known as the private key (IPC_PRIVATE); when specified, a new *shmid* is always returned with an associated shared memory segment data structure created for it unless a system-tunable parameter would be exceeded. The **ipcs(8)** command will show the *key* field for the *shmid* as all zeros.

If a *shmid* exists for the *key* specified, the value of the existing *shmid* is returned. If it is not desired to have an existing *shmid* returned, a control command (IPC_EXCL) can be specified (set) in the *shmflg* argument passed to the system call.

When a new shared memory segment is created, the process that calls **shmget** 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 shmctl System Call"). The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, **shmbind**, **shmctl**, and shared memory operations (**shmop**) can be used.

The **shmbind** system call allows you to bind a shared memory segment to a section of I/O memory. See the section "Binding a Shared Memory Segment to I/O Space" for details of the **shmbind** system call.

The **shmctl(2)** system call permits you to control the shared memory facility in the following ways:

- by retrieving the data structure associated with a shared memory segment (IPC_STAT)
- by changing operation permissions for a shared memory segment (IPC_SET)
- by removing a particular shared memory segment from the operating system along with its associated shared memory segment data structure (IPC_RMID)
- by locking a shared memory segment in memory (SHM_LOCK)
- by unlocking a shared memory segment (SHM_UNLOCK)

See the section "The shmctl System Call" for details of the **shmctl** system call.

Shared memory segment operations (**shmop**) consist of attaching and detaching shared memory segments. **shmat** and **shmdt** are provided for each of these operations (see "The shmat and shmdt System Calls" for details of the **shmat** and **shmdt** system calls).

It is important to note that the **shmdefine(1)** and **shmconfig(1)** utilities also allow you to create shared memory segments. See the section "Shared Memory Utilities" for information about these utilities.

The shmget System Call

`shmget(2)` creates a new shared memory segment or identifies an existing one.

This section describes the **`shmget`** system call. For more detailed information, see the **`shmget(2)`** man page. A program illustrating use of this call can be found at **`/usr/share/doc/ccur/examples/shmget.c`** with extensive comments provided in **`README.shmget.txt`**.

Synopsis

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

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

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/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 shared memory segment identifier (*shmid*) associated to the value of *key*. (The *shmid* is discussed in the section "Using Shared Memory" earlier in this chapter.) Upon failure, the external variable `errno` is set to indicate the reason for failure, and -1 is returned.

A new *shmid* with an associated shared memory data structure is created if one of the following conditions is true:

- *key* is equal to IPC_PRIVATE.
- *key* does not already have a *shmid* associated with it and (*shmflg* and IPC_CREAT) is "true" (not zero).

The value of *shmflg* is a combination of:

- control commands (flags)
- operation permissions

Control commands are predefined constants. The following control commands apply to the **`shmget`** system call and are defined in the **`<bits/ipc.h>`** header file, which is internally included by the **`<sys/ipc.h>`** header file:

IPC_CREAT	used to create a new segment. If not used, **`shmget`** will find the segment associated with *key*, verify access permissions and ensure the segment is not marked for destruction.
IPC_EXCL	used with IPC_CREAT to cause the system call to return an error if a shared memory identifier already exists for the specified *key*. This is necessary to prevent the process from thinking it has received a new (unique) identifier when it has not.

Operation permissions define the read/write attributes for users, groups, and others. Table 3-2 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 3-2 Shared Memory 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 ORing the octal values for the operation permissions desired. That is, if "read by user" and "read/write by others" is desired, the code value would be 00406 (00400 plus 00006). The SHM_R and SHM_W constants located in **<sys/shm.h>** can be used to define read and write permission for the owner.

The *shmflg* value can easily be set by using the flag names in conjunction with the octal operation permissions value; for example:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));
shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

The following values are defined in **<sys/shm.h>**. Exceeding these values always causes a failure.

SHMMNI	determines the maximum number of unique shared memory segments (*shmid*s) that can be in use at any given time
SHMMIN	determines the minimum shared memory segment size
SHMMAX	determines the maximum shared memory segment size
SHMALL	determines the maximum shared memory pages

A list of shared memory limit values can be obtained with the **ipcs(8)** command by using the following options. See the man page for further details.

```
ipcs -m -l
```

Refer to the **shmget(2)** man page for specific associated data structure initialization as well as specific error conditions.

The shmctl System Call

shmctl(2) is used to perform control operations on shared memory segments.

This section describes the **shmctl** system call. For more detailed information, see the **shmctl(2)** man page. A program illustrating use of this call can be found at **/usr/share/doc/ccur/examples/shmctl.c** with extensive comments provided in **README.shmctl.txt**.

Synopsis

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

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

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

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

The *shmid* variable must be a valid, non-negative integer value that has already been created using the **shmget** system call.

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

IPC_STAT	returns the status information contained in the associated data structure for the specified *shmid* and places it in the data structure pointed to by the *buf* pointer in the user memory area. Read permission is required.
IPC_SET	sets the effective user and group identification and operation permissions for the specified *shmid*
IPC_RMID	removes the specified *shmid* along with its associated data structure
SHM_LOCK	prevents swapping of a shared memory segment. The user must fault in any pages that are required to be present after locking is enabled. The process must have superuser or CAP_IPC_LOCK privileges to perform this operation.
SHM_UNLOCK	unlocks the shared memory segment from memory. The process must have superuser or CAP_IPC_LOCK privileges to perform this operation.

NOTE

The **shmctl(2)** service also supports the IPC_INFO, SHM_STAT and SHM_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 shared memory segment can also be removed by using the **ipcrm(1)** command and specifying the **-m** *shmid* or the **-M** *shmkey* option, where *shmid* specifies the identifier for the shared memory segment and *shmkey* specifies the key associated with the segment. To use this command, a process must have the same privileges 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.

Binding a Shared Memory Segment to I/O Space

RedHawk Linux allows you to bind a shared memory segment to a region of I/O space. The procedures for doing so are as follows.

1. Create a shared memory segment (**shmget(2)**).
2. Obtain the physical address of the I/O region using the PCI BAR scan routines.
3. Bind the segment to I/O memory (**shmbind(2)**).
4. Attach the segment to the user's virtual address space (**shmat(2)**).

At command level, the **shmconfig(1)** utility can be used to create a shared memory segment and bind it to a physical memory region. Refer to the section "Shared Memory Utilities" for details.

You can attach a shared memory segment to and detach it from the user's virtual address space by using the **shmat** and **shmdt** system calls. Procedures for using these system calls are explained in "The shmat and shmdt System Calls."

Using shmget

The **shmget(2)** system call is invoked first to create a shared memory segment. Upon successful completion of the call, a shared memory segment of *size* bytes is created, and an identifier for the segment is returned.

When binding to I/O space, the size of the region can be obtained using the PCI BAR scan routines (see **bar_scan_open(3)**).

Complete information on the use of **shmget** is provided in "The shmget System Call."

Using shmbind

After you have created a shared memory segment, you can bind it to a region of I/O space by using the **`shmbind(2)`** system call. Note that to use this call, you must be root or have the CAP_SYS_RAWIO privilege.

`shmbind` must be called before the first process attaches to the segment. Thereafter, attaching to the segment via **`shmat()`** effectively creates a mapping in the calling process' virtual address space to the region of the physical address space.

The region of I/O space is defined by its starting address and the size of the shared memory segment to which it is being bound. The starting address must be aligned with a page boundary. The size of the shared memory segment has been established by specifying the *size* argument on the call to **`shmget`**. If you have created a shared memory segment of 1024 bytes, for example, and you wish to bind it to a section of physical memory that starts at location 0x2000000 (hexadecimal representation), the bound section of physical memory will include memory locations 0x2000000 through 0x2000BFF.

Be aware that the physical address for a device may change due to hardware changes in the system. To reliably reference a device, the physical address should be obtained using the PCI BAR scan routines; refer to the **`bar_scan_open(3)`** man page.

The specifications required for making the call to **`shmbind`** are as follows:

```
int shmbind(int shmid, unsigned long paddr)
```

Arguments are defined as follows:

shmid	the identifier for the shared memory segment that you wish to bind to a section of physical memory
paddr	the starting physical address of the section of memory to which you wish to bind the specified shared memory segment

The shmat and shmdt System Calls

The shared memory operations system calls, **`shmat`** and **`shmdt`**, are used to attach and detach shared memory segments to/from the address space of the calling process.

This section describes the **`shmat`** and **`shmdt`** system calls. For more detailed information, see the **`shmop(2)`** man page. A program illustrating use of these calls can be found at **`/usr/share/doc/ccur/examples/shmop.c`** with extensive comments provided in **`README.shmop.txt`**.

Synopsis

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

void *shmat (int shmid, const void *shmaddr, int shmflg);
int shmdt (const void *shmaddr);
```

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

Attaching a Shared Memory Segment

The **`shmat`** system call attaches the shared memory segment identified by *shmid* to the address space of the calling process. It returns a character pointer value. Upon successful completion, this value will be the address in memory where the process is attached to the shared memory segment; when unsuccessful, the value will be -1.

The *shmid* argument must be a valid, non-negative, integer value. It must have been created previously using the **`shmget`** system call.

The *shmaddr* argument can be zero or user supplied when passed to the **`shmat`** system call. If it is zero, the operating system selects the address where the shared memory segment will be attached. If it is user-supplied, the address must be a valid page-aligned address within the program's address space. The following illustrates some typical address ranges:

```
0xc00c0000
0xc00e0000
0xc0100000
0xc0120000
```

Allowing the operating system to select addresses improves portability.

The *shmflg* argument is used to pass the SHM_RND (round down) and SHM_RDONLY (read only) flags to the **`shmat`** system call.

Detaching Shared Memory Segments

The **`shmdt`** system call detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process. It returns an integer value, which is zero for successful completion or -1 otherwise.

Shared Memory Utilities

Redhawk Linux provides two utilities that facilitate use of shared memory segments. The **shmdefine(1)** utility allows you to create one or more shared memory segments that are to be used by cooperating programs. The **shmconfig(1)** command allows you to create a shared memory segment and bind it to a section of physical memory. These utilities are discussed in the sections that follow.

The shmdefine Utility

The **shmdefine** utility is designed to facilitate the use of shared memory by a set of cooperating programs. Although you may have a number of programs that will cooperate in using one or more shared memory segments, it is necessary to invoke the utility only once. Because **shmdefine** produces object files that must be linked to the source object file, you must invoke it prior to linking.

shmdefine currently operates with the GNU C, Fortran and Ada compilers (`gcc`, `g77` `GNAT`) for programs that execute on RedHawk Linux systems.

Refer to the *Quick Reference for shmdefine* (publication number 0898010) and the **shmdefine(1)** man page for details on using this utility.

The shmconfig Command

The **shmconfig(1)** command assists in creating a shared memory segment associated with a certain key and optionally binding it to a particular section of I/O memory.

The command syntax is:

/usr/bin/shmconfig -i *DEVSTR*
/usr/bin/shmconfig -b *BARSTR* [**-s** *SIZE*] [**-g** *GROUP*] [**-m** *MODE*] [**-u** *USER*]
{*key* | **-t** *FNAME*}
/usr/bin/shmconfig -s *SIZE* [**-p** *ADDR*] [**-g** *GROUP*] [**-m** *MODE*] [**-u** *USER*]
{*key* | **-t** *FNAME*}

For information about assigning NUMA memory policies to shared memory areas, refer to Chapter 10 or the **shmconfig(1)** man page.

Options are described in Table 3-3.

Table 3-3 Options to the shmconfig(1) Command

Option	Description
--info=*DEVSTR*, **-i** *DEVSTR*	Prints information about each memory region on each device matching *DEVSTR*, which consists of: *vendor_id***:***device_id* Helpful when using **--bind**. See **--bind** for information on *DEVSTR*.

Table 3-3 Options to the shmconfig(1) Command (Continued)

Option	Description
`--bind=`*BARSTR*, **`-b`** *BARSTR*	Identifies an I/O region in memory to be bound to the shared segment. *BARSTR* consists of: *vendor_id*:*device_id*:*bar_no*[:*dev_no*] *vendor_id* and *device_id* identify the hardware device; usually expressed as two hex values separated by a colon (e.g., 8086:100f). Can be obtained from the vendor's manual, **`/usr/share/hwdata/pci.ids`** or **`lspci -ns`**. Requires a "0x" base prefix when specifying these IDs; e.g., 0x8086:0x100f. See "Examples" below. *bar_no* identifies the memory region to be bound. Use **`-i`** option to obtain this value (output displays "Region *bar_no*: Memory at ..."). Only the memory regions can be bound. *dev_no* is optional and needed only to differentiate between multiple boards with matching vendor and device IDs. Use **`-i`** option to obtain this value (output displays "Logical device: *dev_no:*"). The user must have the CAP_SYS_RAWIO privilege to use this option.
`--size=`*SIZE*, **`-s`** *SIZE*	Specifies the size of the segment in bytes. Not required for **`--bind`**, where the default is the complete memory region.
`--physical=`*ADDR*, **`-p`** *ADDR*	Specifies *ADDR* as the starting address of the section of physical I/O memory to which the segment is to be bound. This option is being deprecated; use **`--bind`**. The user must have the CAP_SYS_RAWIO privilege to use this option.
`--user=`*USER*, **`-u`** *USER*	Specifies the login name of the owner of the shared memory segment.
`--group=`*GROUP*, **`-g`** *GROUP*	Specifies the name of the group to which group access to the segment is applicable.
`--mode=`*MODE*, **`-m`** *MODE*	Specifies *mode* as the set of permissions governing access to the shared memory segment. You must use the octal method to specify the permissions.
`--help, -h`	Describes available options and usage.
`--version, -v`	Prints out current version of the command.

The **`/proc`** and **`/sys`** file systems must be mounted in order to use this command.

It is important to note that the size of a segment as specified by the **`-s`** argument must match the size of the data that will be placed there. If **`shmdefine`** is being used, the size of the segment must match the size of the variables that are declared to be a part of the shared segment. Specifying a larger size will work. (For information on **shmdefine**, see "The shmdefine Utility.")

It is recommended that you specify the **-u**, **-g**, and **-m** options to identify the user and group associated with the segment and to set the permissions controlling access to it. If not specified, the default user ID and group ID of the segment are those of the owner; the default mode is 0644.

The *key* argument represents a user-chosen identifier for a shared memory segment. This identifier can be either an integer or a standard path name that refers to an existing file. When a pathname is supplied, an `ftok(key,0)` will be used as the key parameter for the **shmget(2)** call.

--tmpfs=*FNAME* / **-t** *FNAME* can be used to specify a tmpfs filesystem filename instead of a key. The **-u, -g** and **-m** options can be used to set or change the file attributes of this segment.

When **shmconfig** is executed, an internal data structure and shared memory segment are created for the specified key; if the **-p** option is used, the shared memory segment is bound to a contiguous section of I/O memory.

To access the shared memory segment that has been created by **shmconfig**, processes must first call **shmget(2)** to obtain the identifier for the segment. This identifier is required by other system calls for manipulating shared memory segments. The specification for **shmget** is:

int shmget(*key*, *size*, **0)**

The value of *key* is determined by the value of *key* specified with **shmconfig**. If the value of *key* was an integer, that integer must be specified as *key* on the call to **shmget**. If the value of *key* was a path name, you must first call the **ftok** subroutine to obtain an integer value based on the path name to specify as *key* on the call to **shmget**. It is important to note that the value of the *id* argument on the call to **ftok** must be zero because **shmconfig** calls **ftok** with an *id* of zero when it converts the path name to a key. The value of *size* must be equal to the number of bytes specified by the **-s** argument to **shmconfig**. A value of **0** is specified as the *flag* argument because the shared memory segment has already been created.

For complete information about **shmget**, see "The shmget System Call." For assistance in using **ftok**, see "Using Shared Memory" and the **ftok(3)** man page. When creating areas of mapped memory to be treated as global system resources, you may find it helpful to invoke **shmconfig** by adding a line to the **shmconfig** script in the **/etc/init.d** directory. Doing so allows you to reserve the IPC key before noncooperating processes have an opportunity to use it, and it enables you to establish the binding between the shared memory segment and physical memory before cooperating processes need to use the segment. Add a line similar to the following example:

```
/usr/bin/shmconfig -p 0xf00000 -s 0x10000 -u root -g sys -m 0666 key
```

Examples

In this example, a physical memory region on the RCIM device is identified using **lspci(8)** and bound to a shared memory region. Note that you must be root to use **lspci**. If you don't have root privileges you can view **/usr/share/hwdata/pci.ids** and search for the device name (RCIM); id values are listed to the left of the vendor/device description. When two or more device ids are listed for the same device, use **shmconfig -i** on each *device_id* listed to determine which one to use.

1. Find the *bus:slot.func* identifier for the RCIM board:

   ```
   # lspci -v  | grep -i rcim
   0d:06.0 System peripheral: Concurrent Computer Corp RCIM II
   Realtime Clock ...
   ```

2. Use the rcim identifier to get the *vendor_id:device_id* numbers:

   ```
   # lspci -ns 0d:06.0
   0d:06.0 Class 0880: 1542:9260 (rev 01)
   ```

3. Find the memory regions for this device. Note that **lspci** prints the *vendor_id:device_id* values in hex format but without a 0x prefix (1542:9260), however **shmconfig** requires a base identifier (0x1542:0x9260).

   ```
   # shmconfig -i 0x1542:0x9260
   Region 0:  Memory at f8d04000 (non-prefetchable) [size=256]
              /proc/bus/pci0/bus13/dev6/fn0/bar0
   Region 1:  I/O ports at 7c00 [size=256]
              /proc/bus/pci0/bus13/dev6/fn0/bar1
   Region 2:  Memory at f8d00000 (non-prefetchable) [size=16384]
              /proc/bus/pci0/bus13/dev6/fn0/bar2
   ```

4. Bind to rcim memory region #2:

   ```
   # shmconfig -b 0x1542:0x9260:2 -m 0644 -u me -g mygroup 42
   ```

5. Verify the IPC shared memory regions on the system. Note that `physaddr` represents the physical address we have bound and matches the address reported by the **shmconfig -i** command in step 3 above.

```
# cat /proc/sysvipc/shm
    key      shmid perms       size  cpid  lpid nattch   uid
gid  cuid  cgid      atime      dtime      ctime physaddr
     42          0   644      16384  1734     0      0  5388
100     0     0          0          0 1087227538  f8d00000
```

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.

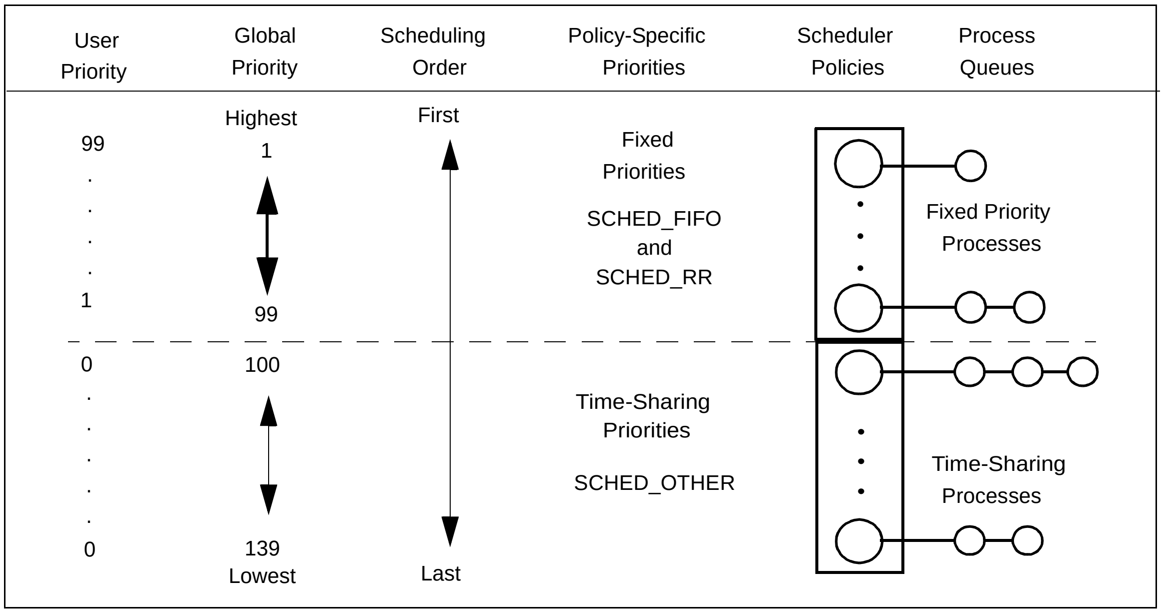
For system environments in which real-time performance is not required, the default scheduler configuration works well, and no fixed-priority processes are needed. However, for real-time applications or applications with strict timing constraints, fixed-priority processes are the only way to guarantee 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.

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);
}
```

Interrupt Routines

Processes scheduled in one of the fixed-priority scheduling policies will be assigned a higher priority than the processing associated with softirqs and tasklets. These 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 interrupt mechanisms to handle the remainder of the work associated with the device interrupt routine. Because fixed-priority processes run at a priority above these 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” 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).

Fixed Priority Processes Locking Up a CPU

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 allows the user priority value range 1 to 99 for SCHED_FIFO and SCHED_RR and the priority 0 for SCHED_OTHER.

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 allows the user priority value range 1 to 99 for SCHED_FIFO and SCHED_RR and the priority 0 for SCHED_OTHER.

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/THREAD_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 specifier is given, **`run`** modifies the environment of the set of processes/threads selected by the specifier. The specifiers are defined below. A command may not be combined with a 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. Options for setting NUMA memory policies are documented in Chapter 10. See the **`run(1)`** man page for more information.

PROCESS/THREAD_SPECIFIER

This parameter is used to specify the processes or threads to be acted upon. Only one of the following may be specified. Multiple comma separated values can be specified for all *list*s and ranges are allowed where appropriate.

`--all`, **`-a`**	Specify all existing processes and threads.
`--pid=`*list*, **`-p`** *list*	Specify a list of existing PIDs to modify. All scheduler operations are specific to the complete set of processes listed, including all sub-threads.
`--tid=`*list*, **`-t`** *list*	Specify a list of existing TIDs to modify. All scheduler operations are specific to only the listed threads and not unspecified sibling threads in the process. **-l** *list* can be used for PowerMAX compatibility.
`--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 and mutexes are the next best choice for synchronizing access to shared data. System V semaphores are also 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. Tools for implementing busy-wait mutual exclusion are explained in the section "Busy-Wait Mutual Exclusion."

Critical sections are often very short. To keep the cost of synchronization comparatively small, synchronizing operations performed on entry/exit to/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 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. Tools for providing rescheduling control are described in "Rescheduling Control."

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. POSIX counting semaphores are described in the section "POSIX Counting Semaphores." 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). System V semaphores are described in the section "System V Semaphores."

Mutexes allow multiple threads in a program to share the same resource but not simultaneously. A mutex is created and any thread that needs the resource must lock the mutex from other threads while using the resource and unlock it when it is no longer needed. In addition to the standard POSIX pthread mutexes, RedHawk includes extensions through an alternative **`glibc`** that provides *robust mutexes* and *priority inheritance*. Robustness gives applications a chance to recover if one of the application's threads dies while holding a mutex. Applications using a priority inheritance mutex can find the priority of the mutex's owner boosted from time to time. These are explained in the section "Extensions to POSIX Mutexes."

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 register the variable in your application and manipulate it directly from your application. 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 or per-thread basis by the application, not by the kernel. The **`resched_cntl(2)`** system call registers the variable, and the kernel examines the variable before making rescheduling decisions.

Use of the **`resched_cntl`** system call is explained in "Using 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."

Each thread must register its own rescheduling variable. A rescheduling variable is valid only for the process or thread that registers the location of the rescheduling variable. Under the current implementation, it is recommended that rescheduling variables be used only by single-threaded processes. Forking in a multi-threaded program that uses rescheduling variables should be avoided.

Using resched_cntl System Call

The **resched_cntl** system call enables you to perform a variety of rescheduling control operations. These include registering and initializing a rescheduling variable, 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 registers 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**, **vfork(2)** or **clone(2)**, the rescheduling variable aborts.

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_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 lock and unlock rescheduling variables 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; registers and initializes the variable with a call to **`resched_cntl`**; and locks and unlocks the rescheduling variables with calls to **`resched_lock`** and **`resched_unlock`**, respectively.

```
static struct resched_var rv;

int main (int argc, char *argv[])
{
  resched_cntl (RESCHED_SET_VARIABLE, (char *)&rv);

  resched_lock (&rv);

  /* nonpreemptible code */
  ...

  resched_unlock (&rv);
  return 0;
}
```

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 provides two types of low-overhead busy-wait mutual exclusion variables: **`spin_mutex`** and **`nopreempt_spin_mutex`**. A **`nopreempt_spin_mutex`** automatically uses rescheduling variables to make threads or processes non-preemptible while holding the mutex; a **`spin_mutex`** does not.

In the sections that follow, the variables and interfaces are defined, and the procedures for using them are explained.

Understanding the spin_mutex Variable

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

```
typedef struct spin_mutex {
    volatile int count;
} spin_mutex_t;
```

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 interfaces described in "Using the spin_mutex Interfaces."

Using the spin_mutex Interfaces

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

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

It is important to note that none of these interfaces 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 **`spin_init`**. You call **`spin_init`** only once for each spin lock. If the specified spin lock is locked, **`spin_init`** effectively unlocks it; it does not return a value. The **`spin_init`** interface is specified as follows:

```
#include <spin.h>
void spin_init(spin_mutex_t *m);
```

The argument is defined as follows:

m	the starting address of the spin lock

`spin_lock` spins until the spin lock can be locked. It does not return a value. The interface is specified as follows:

```
#include <spin.h>
void spin_lock(spin_mutex_t *m);
```

spin_trylock returns true if the calling process or thread has succeeded in locking the spin lock; false if it has not succeeded. **spin_trylock** does not block the calling process or thread. The interface is specified as follows:

```
#include <spin.h>
int spin_trylock(spin_mutex_t *m);
```

spin_islock returns true if the specified spin lock is locked; false if it is unlocked. It does not attempt to lock the spin lock. The interface is specified as follows:

```
#include <spin.h>
int spin_islock(spin_mutex_t *m);
```

spin_unlock unlocks the spin lock. It does not return a value. The interface is specified as follows:

```
#include <spin.h>
void spin_unlock(spin_mutex_t *m);
```

Note that **spin_lock**, **spin_trylock** and **spin_unlock** can log trace events to be monitored by NightTrace. An application can enable these trace events by defining SPIN_TRACE prior to including **<spin.h>**. For example:

```
#define SPIN_TRACE
#include <spin.h>
```

The application must also be linked with **-lntrace**, or **-lntrace_thr** if also linked with **-lpthread**.

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

Applying spin_mutex Tools

Procedures for using the spin_mutex 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** interfaces. 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** interface 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 interfaces. 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.

Understanding the nopreempt_spin_mutex Variable

The nopreempt_spin_mutex is a busy-wait mutex that allows multiple threads or processes to synchronize access to a shared resource. A rescheduling variable is used to make threads or processes non-preemptible while holding the mutex locked. A thread or process may safely nest the locking of multiple mutexes. The nopreempt_spin_mutex is defined in **<nopreempt_spin.h>** as follows:

```
typedef struct nopreempt_spin_mutex {
    spin_mutex_t spr_mux;
} nopreempt_spin_mutex_t;
```

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

If you wish to use non-preemptible 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 interfaces described in "Using the nopreempt_spin_mutex Interfaces."

Using the nopreempt_spin_mutex Interfaces

This set of busy-wait mutual exclusion interfaces allows you to initialize, lock, and unlock non-preemptible spin locks. A rescheduling variable is used to make threads or processes non-preemptible while holding the mutex locked. These are briefly described as follows:

nopreempt_spin_init	initialize a spin lock to the unlocked state
nopreempt_spin_init_thread	guarantee that preemption can be blocked
nopreempt_spin_lock	spin until the spin lock can be locked

nopreempt_spin_trylock	attempt to lock a specified spin lock
nopreempt_spin_islock	determine whether or not a specified spin lock is locked
nopreempt_spin_unlock	unlock a specified spin lock

You must initialize spin locks before you use them by calling **nopreempt_spin_init**. You call this interface only once for each spin lock. If the specified spin lock is locked, **nopreempt_spin_init** effectively unlocks it; it does not return a value. The interface is specified as follows:

```
#include <nopreempt_spin.h>
void nopreempt_spin_init(nopreempt_spin_mutex_t *m);
```

The argument is defined as follows:

m	the starting address of the spin lock

nopreempt_spin_init_thread guarantees that preemption can be blocked when **nopreempt_spin_lock** and **nopreempt_spin_trylock** are called. When a nopreempt_spin_mutex is used in a multi-threaded process, the process must be linked with **-lpthread** and each thread must call **nopreempt_spin_init_thread** at least once. If a process is not multi-threaded, it must call this routine at least once. This routine need only be called once regardless of how many mutexes the process or thread uses. It returns zero (0) if preemption blocking can be guaranteed; otherwise it returns -1 with **errno** set. The interface is specified as follows:

```
#include <nopreempt_spin.h>
int nopreempt_spin_init_thread(void)
```

nopreempt_spin_lock spins until the spin lock can be locked. It does not return a value. It is specified as follows:

```
#include <nopreempt_spin.h>
void nopreempt_spin_lock(nopreempt_spin_mutex_t *m);
```

nopreempt_spin_trylock returns true if the calling process or thread has succeeded in locking the spin lock; false if it has not succeeded. **nopreempt_spin_trylock** does not block the calling process or thread. The interface is specified as follows:

```
#include <nopreempt_spin.h>
int nopreempt_spin_trylock(nopreempt_spin_mutex_t *m);
```

nopreempt_spin_islock returns true if the specified spin lock is locked; false if it is unlocked. It does not attempt to lock the spin lock. The interface is specified as follows:

```
#include <nopreempt_spin.h>
int nopreempt_spin_islock(nopreempt_spin_mutex_t *m);
```

nopreempt_spin_unlock unlocks the spin lock. It does not return a value. The interface is specified as follows:

```
#include <nopreempt_spin.h>
void nopreempt_spin_unlock(nopreempt_spin_mutex_t *m);
```

Note that **nopreempt_spin_lock**, **nopreempt_spin_trylock** and **nopreempt_spin_unlock** can log trace events to be monitored by NightTrace. An application can enable these trace events by defining SPIN_TRACE prior to including **<nopreempt_spin.h>**. For example:

```
#define SPIN_TRACE
#include <nopreempt_spin.h>
```

The application must also be linked with **-lntrace**, or **-lntrace_thr** if also linked with **-lpthread**.

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

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** or **sem_open**
- A lock operation that decrements the value of the semaphore—**sem_wait** or **sem_timedwait**. If the resulting value is negative, the task 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 tasks blocked on the semaphore is awakened. If the resulting value is greater than zero, no tasks 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 counting semaphore can be named or unnamed.

A thread creates and initializes an *unnamed semaphore* through a call to the **sem_init(3)** 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.

A thread creates a *named semaphore* by invoking the **`sem_open`** routine and specifying a unique name that is simply a null-terminated string. The semaphore is initialized to a value that is supplied on the call to **`sem_open`** to create the semaphore. No space is allocated by the process for a named semaphore because the **`sem_open`** routine will include the semaphore in the process's virtual address space. Other processes can gain access to the named semaphore by invoking **`sem_open`** and specifying the same name.

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

A cooperating task that wants access to a critical resource must lock the semaphore that protects that resource. When the task locks the semaphore, it knows that it can use the resource without interference from any other cooperating task 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 task that tries to acquire it. When the semaphore value is zero, then none of the resources are available; a task trying to acquire a resource must wait until one becomes available. If the semaphore value is negative, then there may be one or more tasks that are blocked and waiting to acquire one of the resources. When a task completes use of a resource, it unlocks the semaphore, indicating that the resource is available for use by another task.

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

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

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

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_open`	creates, initializes and establishes a connection to a named counting semaphore
`sem_close`	relinquishes access to a named counting semaphore
`sem_unlink`	removes the name of a named counting semaphore
`sem_wait`	locks a counting semaphore
`sem_trywait`	locks a counting semaphore only if it is currently unlocked
`sem_timedwait`	locks a counting semaphore with timeout
`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 pthreads library. The following example shows the command line invocation when linking dynamically with shared libraries. The Native POSIX Threads Library (NPTL) is used by default.

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

The same application can be built statically with the following invocation line. This uses the LinuxThreads library.

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

Note that there is no support for process shared semaphores in the LinuxThreads library.

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.

Dynamically linked programs, which use the NPTL threads library, can share a semaphore across processes when the *pshared* parameter is set to a non-zero value. If *pshared* is set to zero, the semaphore is shared only among threads within the same process.

Statically linked programs, which use the LinuxThreads library, can only have counting semaphores shared among threads within the same process (*pshared* must be set to 0). After one thread in a process creates and initializes a semaphore, other cooperating threads within that same process can operate on the semaphore. 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.

The **sem_wait**, **sem_timedwait**, **sem_trywait**, **sem_post** and **sem_getvalue** library routines are used to operate on the semaphores. An unnamed counting semaphore is removed by invoking the **sem_destroy** routine. These routines are described in the sections that follow.

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.

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 process. Statically linked programs, which use the LinuxThreads library, cannot use semaphores shared between processes and must have *pshared* set to zero; if not set to zero, **`sem_init`** returns with **`-1`** and **`errno`** is set to **ENOSYS**.

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.

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.

The sem_open Routine

The **`sem_open(3)`** library routine allows the calling process to create, initialize, and establish a connection to a named counting semaphore. When a process creates a named counting semaphore, it associates a unique name with the semaphore. It also sets the semaphore value to the number of available resources being protected by the semaphore. To use a named counting semaphore for mutual exclusion, the process sets the value to one.

After a process creates a named semaphore, other processes can establish a connection to that semaphore by invoking **`sem_open`** and specifying the same name. Upon successful completion, the **`sem_open`** routine returns the address of the named counting semaphore. A process subsequently uses that address to refer to the semaphore on calls to **`sem_wait`**, **`sem_trywait`**, and **`sem_post`**. A process may continue to operate on the named semaphore until it invokes the **`sem_close`** routine or the **`exec(2)`** or **`_exit(2)`** system calls. On a call to **`exec`** or **`exit`**, a named semaphore is closed as if by a call to **`sem_close`**. A child process created by a **`fork(2)`** system call inherits access to a named semaphore to which the parent process has established a connection.

If a single process makes multiple calls to **`sem_open`** and specifies the same name, the same address will be returned on each call unless (1) the process itself has closed the semaphore through intervening calls to **`sem_close`** or (2) some process has removed the name through intervening calls to **`sem_unlink`**.

If multiple processes make calls to **`sem_open`** and specify the same name, the address of the same semaphore object will be returned on each call unless some process has removed the name through intervening calls to **`sem_unlink`**. (Note that the same address will not necessarily be returned on each call.) If a process has removed the name through an intervening call to **`sem_unlink`**, the address of a new instance of the semaphore object will be returned.

Synopsis

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag[, mode_t mode,
unsigned int value ]);
```

The arguments are defined as follows:

name a null-terminated string that specifies the name of a semaphore. The prefix "sem." is prepended to *name* and the semaphore will appear as a data file in **`/dev/shm`**. A leading slash (/) character is allowed (recommended for portable applications) but no embedded slashes. Neither a leading slash character nor the current working directory affects interpretations of it; e.g., **`/mysem`** and **`mysem`** are both interpreted as **`/dev/shm/sem.mysem`**. Note that this string, including the 4-character prefix, must consist of less than {NAME_MAX}, defined in **`/usr/include/limits.h`**.

oflag an integer value that indicates whether the calling process is creating a named counting semaphore or establishing a connection to an existing one. The following bits may be set:

O_CREAT causes the counting semaphore specified by *name* to be created if it does not exist. The semaphore's user ID is set to the effective user ID of the calling process; its group ID is set to the effective group ID of the calling process; and its permission bits are set as specified by the *mode* argument. The semaphore's initial value is set as specified by the *value* argument. Note that you must specify both the *mode* and the *value* arguments when you set this bit.

If the counting semaphore specified by *name* exists, setting O_CREAT has no effect except as noted for O_EXCL.

O_EXCL causes **sem_open** to fail if O_CREAT is set and the counting semaphore specified by *name* exists. If O_CREAT is not set, this bit is ignored.

Note that the **sem_open** routine returns an error if flag bits other than O_CREAT and O_EXCL are set in the *oflag* argument.

mode an integer value that sets the permission bits of the counting semaphore specified by *name* with the following exception: bits set in the process's file mode creation mask are cleared in the counting semaphore's mode (refer to the **umask(2)** and **chmod(2)** man pages for additional information). If bits other than the permission bits are set in *mode*, they are ignored. A process specifies the *mode* argument only when it is creating a named counting semaphore.

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 defined in the file <**limits.h**>. A process specifies the *value* argument only when it is creating a named counting semaphore.

If the call is successful, **sem_open** returns the address of the named counting semaphore. A return value of SEM_FAILED indicates that an error has occurred; errno is set to indicate the error. Refer to the **sem_open(3)** man page for a listing of the types of errors that may occur.

The sem_close Routine

The **sem_close(3)** library routine allows the calling process to relinquish access to a named counting semaphore. The **sem_close** routine frees the system resources that have been allocated for the process' use of the semaphore. Subsequent attempts by the process to operate on the semaphore may result in delivery of a SIGSEGV signal.

The count associated with the semaphore is not affected by a process' call to **sem_close**.

Synopsis

```
#include <semaphore.h>

int sem_close(sem_t  *sem);
```

The argument is defined as follows:

sem a pointer to the named counting semaphore to which access is to be relinquished. Only a counting semaphore to which a connection has been established through a call to **`sem_open(3)`** may be specified.

A return value of 0 indicates that the call to **`sem_close`** 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_close(3)`** man page for a listing of the types of errors that may occur.

The sem_unlink Routine

The **`sem_unlink(3)`** library routine allows the calling process to remove the name of a counting semaphore. A process that subsequently attempts to establish a connection to the semaphore by using the same name will establish a connection to a different instance of the semaphore. A process that has a reference to the semaphore at the time of the call may continue to use the semaphore until it invokes **`sem_close(3)`** or the **`exec(2)`** or **`exit(2)`** system call.

Synopsis

```
#include <semaphore.h>

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

The argument is defined as follows:

name a null-terminated string that specifies the name of a semaphore. The prefix "sem." is prepended to *name* and the semaphore will appear as a data file in **`/dev/shm`**. A leading slash (/) character is allowed (recommended for portable applications) but no embedded slashes. Neither a leading slash character nor the current working directory affects interpretations of it; e.g., **`/mysem`** and **`mysem`** are both interpreted as **`/dev/shm/sem.mysem`**. Note that this string, including the 4-character prefix, must consist of less than {NAME_MAX}, defined in **`/usr/include/limits.h`**.

A return value of 0 indicates that the call to **`sem_unlink`** 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_unlink(3)`** man page for a listing of the types of errors that may occur.

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.

The sem_timedwait Routine

The **`sem_timedwait(3)`** library routine allows the calling process to lock an unnamed counting semaphore; however, if the semaphore cannot be locked without waiting for another process or thread to unlock it via **`sem_post`**, the wait is terminated when the specified timeout expires.

When the program is linked to the **`libccur_rt`** library, the timeout is based on the CLOCK_REALTIME clock (see “POSIX Clocks and Timers” in Chapter 6 for more information).

Synopsis

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

int sem_timedwait(sem_t *sem, const struct timespec *ts);
```

The arguments are defined as follows:

sem	a pointer to the unnamed counting semaphore to be locked
ts	a pointer to a timespec structure defined in **`<time.h>`** which specifies a single time value in seconds and nanoseconds when the wait is terminated. For example: `ts.tv_sec = (NULL)+2` `ts.tv_nsec = 0` establishes a two second timeout. For more information on POSIX time structures, see “Understanding the POSIX Time Structures” in Chapter 6.

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.

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.

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. If a bad semaphore descriptor has been supplied, a segmentation fault results. 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.

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.

Extensions to POSIX Mutexes

A mutex is a mutual exclusion device useful for protecting shared data structures from concurrent modifications and implementing critical sections. A mutex has two possible states: unlocked (not owned by any thread) and locked (owned by one thread). A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

The standard POSIX pthread mutex functionality available in RedHawk includes the following services. For full information about these services refer to the man pages.

`pthread_mutex_init(3)`	initializes the mutex
`pthread_mutex_lock(3)`	locks the mutex
`pthread_mutex_trylock(3)`	tries to lock the mutex
`pthread_mutex_unlock(3)`	unlocks the mutex
`pthread_mutex_destroy(3)`	destroys the mutex
`pthread_mutexattr_init(3)`	initializes the mutex attribute object
`pthread_mutexattr_destroy(3)`	destroys the mutex attribute object
`pthread_mutexattr_settype(3)`	sets the mutex attribute type
`pthread_mutexattr_gettype(3)`	retrieves the mutex attribute type

In addition to those services, RedHawk includes the following POSIX pthread extensions that provide robustness and priority inheritance. *Robustness* gives applications a chance to recover if one of the application's threads dies while holding a mutex. *Priority inheritance* is the automatic boosting of the scheduling priority of a thread to the priority of the highest priority thread that is sleeping, directly or indirectly, on any of the mutexes owned by that thread. These conditions are discussed in more detail below.

These pthread extensions are accessible through an alternative **`glibc`**, which is discussed in the section "Alternative glibc" below. The services are described in the sections that follow and in the man pages.

`pthread_mutex_consistent_np(3)`	makes an inconsistent mutex consistent
`pthread_mutex_getunlock_np(3)`	returns the unlocking policy of the mutex
`pthread_mutex_setconsistency_np(3)`	sets the consistency state of the mutex
`pthread_mutex_setunlock_np(3)`	sets the unlocking policy of the mutex
`pthread_mutexattr_getfast_np(3)`	returns the operating mode
`pthread_mutexattr_getprotocol(3)`	returns the protocol
`pthread_mutexattr_getrobust_np(3)`	returns the robust level
`pthread_mutexattr_getunlock_np(3)`	returns the unlocking policy
`pthread_mutexattr_setfast_np(3)`	sets the operating mode
`pthread_mutexattr_setprotocol(3)`	sets the protocol
`pthread_mutexattr_setrobust_np(3)`	sets the robust level
`pthread_mutexattr_setunlock_np(3)`	sets the unlocking policy

Robust Mutexes

Applications using a robust mutex can detect whether the previous owner of the mutex terminated while holding the mutex. The new owner can then attempt to clean up the state protected by the mutex, and if able to do so, mark the mutex as again healthy. If cleanup of the state can't be done, the mutex can be marked unrecoverable so that any future attempts to lock it will get a status indicating that it is unrecoverable.

To implement this, two new `errno` codes, EOWNERDEAD and ENOTRECOVERABLE, are available. When a thread dies while holding a mutex, the mutex is automatically unlocked and marked dead. A dead lock operates like a normal lock except that each successful lock on a dead mutex returns an EOWNERDEAD error rather than success.

Therefore an application that is interested in robustness must examine the return status of every lock request. When EOWNERDEAD is seen, the application can ignore it, repair whatever is wrong in the application due to the death of the owner and mark it consistent (healthy), or if it cannot be repaired, mark it unrecoverable.

A mutex marked unrecoverable rejects all future operations on that mutex with an ENOTRECOVERABLE error. The only exception is the service which re-initializes the mutex and the services that inquire about the mutex state. Threads that were sleeping on a mutex that becomes unrecoverable wake up immediately with an ENOTRECOVERABLE error.

Priority Inheritance

An application using a priority inheritance mutex can find its priority temporarily boosted from time to time. The boosting happens to those threads that have acquired a mutex and other higher priority threads go to sleep waiting for that mutex. In this case, the priority of the sleeper is temporarily transferred to the lock owner for as long as that owner holds the lock.

As these sleeping threads in turn could own other mutexes, and thus themselves have boosted priorities, the max function takes care to use the sleeper's boosted, not base, priorities in making its decision on what priority to boost to.

User Interface

The services listed in this section are available by compiling and linking applications with **`ccur-gcc`**, which is described in the section "Alternative glibc" below. Full descriptions of these services are provided in the sections that follow and on the corresponding online man page.

The following services operate on the state of the mutex:

`pthread_mutex_consistent_np(3)`	makes an inconsistent mutex consistent
`pthread_mutex_getunlock_np(3)`	returns the unlocking policy of the mutex
`pthread_mutex_setconsistency_np(3)`	sets the consistency state of the mutex
`pthread_mutex_setunlock_np(3)`	sets the unlocking policy of the mutex

The services listed below modify or make inquires about attributes stored in mutex attribute objects. A *mutex attribute object* is a data structure that defines which mutex features are to be available in mutexes created with that attribute object. Since mutexes have a lot of features, a mutex attribute object makes it convenient for an application to define all the desired attributes in one mutex attribute object, then create all the mutexes that are to have that set of attributes with that object.

In addition, those attributes which must be fixed for the life of the mutex are definable only through a mutex attribute object. Likewise, attributes which can be changed during the life of a mutex can be given an initial definition through the mutex attribute object, then can be changed later via an equivalent attribute operation on the mutex itself.

To return an attribute:

`pthread_mutexattr_getfast_np(3)`	returns the operating mode
`pthread_mutexattr_getprotocol(3)`	returns the protocol
`pthread_mutexattr_getrobust_np(3)`	returns the robust level
`pthread_mutexattr_getunlock_np(3)`	returns the unlocking policy

To set an attribute:

`pthread_mutexattr_setfast_np(3)`	sets the operating mode
`pthread_mutexattr_setprotocol(3)`	sets the protocol
`pthread_mutexattr_setrobust_np(3)`	sets the robust level
`pthread_mutexattr_setunlock_np(3)`	sets the unlocking policy

pthread_mutex_consistent_np

This service makes an inconsistent mutex consistent.

Synopsis

```
int pthread_mutex_consistent_np (pthread_mutex_t *mutex)
```

A consistent mutex becomes inconsistent if its owner dies while holding it. In addition, on detection of the death of the owner, the mutex becomes unlocked, much as if a **`pthread_mutex_unlock`** was executed on it. The lock continues to operate as normal, except that subsequent owners receive an EOWNERDEAD error return from the **`pthread_mutex_lock`** that gave it ownership. This indicates to the new owner that the acquired mutex is inconsistent.

This service can only be called by the owner of the inconsistent mutex.

pthread_mutex_getunlock_np

This service returns the unlocking policy of this mutex.

```
int pthread_mutex_getunlock_np(const pthread_mutex_t *mutex,
     int *policy)
```

The unlocking policy is returned in **policy*, which may be set to:

PTHREAD_MUTEX_UNLOCK_SERIAL_NP
: **`pthread_mutex_unlock`** is to pass the lock directly from the owner to the highest priority thread waiting for the lock.

PTHREAD_MUTEX_UNLOCK_PARALLEL_NP
: The lock is unlocked and, if there are waiters, the most important of them is awakened. The awakened thread contends for the lock as it would if trying to acquire the lock for the first time.

PTHREAD_MUTEX_UNLOCK_AUTO_NP
: Select between the above two policies based on the POSIX scheduling policy of the to-be-awakened thread. If the thread is SCHED_OTHER, use the parallel policy; otherwise use the serial policy.

pthread_mutex_setconsistency_np

This service sets the consistency state of the given mutex.

```
int pthread_mutex_setconsistency_np(pthread_mutex_t *mutex,
     int state)
```

state may be any one of the following:

PTHREAD_MUTEX_ROBUST_CONSISTENT_NP
: Make an inconsistent mutex consistent. An application should do this only if it has been able to fix the problems that caused the mutex to be marked inconsistent.

PTHREAD_MUTEX_ROBUST_NOTRECOVERABLE_NP
: Mark an inconsistent mutex as unrecoverable. An application should do this if it is not able to fix the problems that caused the mutex to be marked inconsistent.

The mutex must originally be in an inconsistent state or this service returns an error. Only the owner of the mutex can change its consistency state.

pthread_mutex_setunlock_np

This service sets the unlocking policy of this mutex.

Synopsis

```
int pthread_mutex_setunlock_np(pthread_mutex_t *mutex, int policy)
```

policy may be PTHREAD_MUTEX_UNLOCK_SERIAL_NP, PTHREAD_MUTEX_UNLOCK_PARALLEL_NP or PTHREAD_MUTEX_UNLOCK_AUTO_NP. Refer to the section “pthread_mutex_getunlock_np” above for definitions.

pthread_mutexattr_getfast_np

This service returns whether mutexes initialized with the set of attributes in *attr* will run in fast or in slow mode.

Synopsis

```
int pthread_mutexattr_getfast_np(const pthread_mutexattr_t *attr,
     int *mode)
```

The answer is returned in **mode*, which will be set to:

PTHREAD_MUTEX_FASTPATH_NP
: Mutexes initialized with *attr* will run in fast mode. In this mode, uncontended locks and unlocks do not enter the kernel.

PTHREAD_MUTEX_SLOWPATH_NP
: Mutexes initialized with *attr* will run in slow mode. In this mode, the kernel is entered for every **pthread_mutex_lock** and **pthread_mutex_unlock**.

pthread_mutexattr_getprotocol

This services returns the protocol for mutexes initialized with this set of attributes.

Synopsis

```
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr,
     int *protocol)
```

The available protocols are:

PTHREAD_PRIO_NONE	A thread’s scheduling priority is not affected by operations on this mutex.

PTHREAD_PRIO_INHERIT	A thread's scheduling priority is changed according to the rules of the priority inheritance protocol: as long as the thread is the owner of the mutex, it will inherit the priority of the highest priority waiter that is directly or indirectly waiting to acquire the mutex.

pthread_mutexattr_getrobust_np

This service returns the robust level for mutexes initialized with this set of attributes.

Synopsis

```
int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t
    *attr, int *robustness)
```

The available levels are:

PTHREAD_MUTEX_ROBUST_NP	Mutexes created with this attribute object will be robust.
PTHREAD_MUTEX_STALLED_NP	Mutexes created with this attribute object will not be robust.

A robust mutex is one that detects when its owner dies and transitions to the inconsistent state. See "pthread_mutex_consistent_np" for the definition of the inconsistent state.

A nonrobust mutex does not detect when its owner dies and so remains locked indefinitely (that is, until it is interrupted by a signal or some other thread unlocks the mutex on behalf of the dead process).

pthread_mutexattr_getunlock_np

This service returns the unlocking policy for mutexes initialized with this set of attributes.

```
int pthread_mutexattr_getunlock_np(const phtread_mutexattr_t
    *attr, int *mode)
```

The available policies are PTHREAD_MUTEX_UNLOCK_SERIAL_NP, PTHREAD_MUTEX_UNLOCK_PARALLEL_NP and PTHREAD_MUTEX_UNLOCK_AUTO_NP. See the section "pthread_mutex_getunlock_np" for their definitions.

pthread_mutexattr_setfast_np

This service sets the operating mode for mutexes created with this set of attributes.

Synopsis

```
int pthread_mutexattr_setfast_np(pthread_mutexattr_t *attr,
    int mode)
```

mode may be PTHREAD_MUTEX_FASTPATH_NP or PTHREAD_MUTEX_SLOWPATH_NP. See the section "pthread_mutexattr_getfast_np" for their definitions.

pthread_mutexattr_setprotocol

This service sets the protocol of any mutex that is created from this set of mutex attributes.

Synopsis

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
     int protocol)
```

protocol may be PTHREAD_PRIO_NONE or PTHREAD_PRIO_INHERIT. See the section "pthread_mutexattr_getprotocol" for their definitions.

pthread_mutexattr_setrobust_np

This service sets the robust level for mutexes that are created with this mutex attribute object.

Synopsis

```
int pthread_mutexattr_setrobust_np(pthread_mutexattr_t *attr,
     int robustness)
```

robustness may be PTHREAD_MUTEX_ROBUST_NP or PTHREAD_MUTEX_STALLED_NP. See "pthread_mutexattr_getrobust_np" for definitions.

pthread_mutexattr_setunlock_np

This service sets the unlocking mode for mutexes that are created with this mutex attribute object.

```
int pthread_mutexattr_setunlock_np(pthread_mutexattr_t *attr,
     int mode)
```

mode may be PTHREAD_MUTEX_UNLOCK_SERIAL_NP, PTHREAD_MUTEX_UNLOCK_PARALLEL_NP, or PTHREAD_MUTEX_UNLOCK_AUTO_NP. See the section "pthread_mutex_getunlock_np" for their definitions.

Alternative glibc

The additional pthread mutex services described above are available to applications that use Concurrent's alternative **`glibc`**. In general, these mutexes support the additional features of robustness and priority inheritance that extend above and beyond the POSIX 1003.1-2001 standard.

The alternative **`glibc`** is based upon the fusyn-2.3.1 and rtnptl-2.3 open source patches. It is accessed by compiling and linking applications with **`ccur-gcc`**.

Due to the experimental nature of this library, applications are not allowed to statically link against the pthreads portion of this library.

Special System Administration Considerations

The alternative library requires that any time the system administrator makes a new library known to standard **`glibc`** via a call to **`/sbin/ldconfig`**, an equivalent call must be made to **/lib/ccur/sbin/ldconfig**, with the same arguments but with a **/usr/lib** appended to the end. This makes the same information available to the alternative library.

For example, if the administrator needs to execute the following:

```
$ /sbin/ldconfig
```

then he/she should also execute:

```
$ /lib/ccur/sbin/ldconfig /usr/lib
```

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 "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`. This 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 "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 "The semctl System Call" 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 "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 "The semop System Call" 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 "The semctl System Call" for details of the **`semctl`** system call.

The semget System Call

`semget(2)` creates a new semaphore set or identifies an existing one.

This section describes how to use the **`semget`** system call. For more detailed information, see the **`semget(2)`** man page. A program illustrating use of this call can be found at **`/usr/share/doc/ccur/examples/semget.c`** with extensive comments provided in **`README.semget.txt`**.

Synopsis

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

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

All of the `#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/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*). The *semid* is discussed in the section "Using System V Semaphores" earlier in this chapter.

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* does not already have a *semid* associated with it and (*semflg* and IPC_CREAT) is "true" (not zero).

The value of *semflg* is a combination of:

- control commands (flags)
- operation permissions

Control commands are predefined constants. The following control commands apply to the **`semget`** system call and are defined in the **`<bits/ipc.h>`** header file, which is internally included by the **`<sys/ipc.h>`** header file:

IPC_CREAT	used to create a new semaphore set. If not used, **`semget`** will find the semaphore set associated with *key* and verify access permissions.
IPC_EXCL	used with IPC_CREAT to cause the system call to return an error if a semaphore set identifier already exists for the specified *key*. This is necessary to prevent the process from thinking it has received a new (unique) identifier when it has not.

Operation permissions define the read/alter attributes for users, groups and others. 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 octal values for the operation permissions desired. That is, if “read by user” and “read/alter by others” is desired, the code value would be 00406 (00400 plus 00006).

The *semflg* value can easily be set by using the flag names in conjunction with the octal operation permissions value; for example:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

The following values are defined in **`<linux/sem.h>`**. Exceeding these values always causes a failure.

SEMMNI	determines the maximum number of unique semaphore sets (*semid*s) that can 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. See the man page for further details.

```
ipcs -s -l
```

Refer to the **`semget(2)`** man page for specific associated data structure initialization as well as the specific error conditions.

The semctl System Call

semctl(2) is used to perform control operations on semaphore sets.

This section describes the **semctl** system call. For more detailed information, see the **semctl(2)** man page. A program illustrating use of this call can be found at **/usr/share/doc/ccur/examples/semctl.c** with extensive comments provided in **README.semctl.txt**.

Synopsis

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

int semctl (int semid, int semnum, int cmd, int arg);

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

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

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 any 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 **`-s`** *semid* or the **`-S`** *semkey* option, where *semid* specifies the identifier for the semaphore set and *semkey* specifies the key associated with 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 semop System Call

semop(2) is used to perform operations on selected members of the semaphore set.

This section describes the **semop** system call. For more detailed information, see the **semop(2)** man page. A program illustrating use of this call can be found at **/usr/share/doc/ccur/examples/semop.c** with extensive comments provided in **README.semop.txt**.

Synopsis

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

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

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

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.

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.

The postwait System Call

The **`postwait(2)`** function is a fast, efficient, sleep/wakeup/timer mechanism used between a cooperating group of threads. The threads need not be members of the same process.

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 ...
```

To go to sleep, a thread calls **`pw_wait()`**. The thread will wake up when:

- the timer expires
- the thread is posted to by another thread by calling **`pw_post()`** or **`pw_postv()`** with the *ukid*(s) of the **`pw_wait`**ing thread(s)
- the call is interrupted

Threads using **postwait(2)** services are identified by their *ukid*. A thread should call **pw_getukid()** to obtain its *ukid*. The *ukid* maps to the caller's unique, global thread id. This value can be shared with the other cooperating threads that may wish to post to this thread.

For each thread, **postwait(2)** remembers at most one unconsumed post. Posting to a thread that has an unconsumed post has no effect.

For all **postwait(2)** services that have a rescheduling variable argument pointer, if that pointer is non-NULL, the lock-count of the associated rescheduling variable is decremented.

pw_wait() is used to consume a post. It is called with an optional timeout value and an optional rescheduling variable. It returns a value of 1 if it consumes a post or 0 if timed-out waiting for a post to consume.

If the time specified for the timeout value is greater than 0, the thread sleeps at most for that amount of time waiting for a post to consume. 0 is returned if this period expires without encountering a post. If the call is interrupted, EINTR is returned and the timeout value is updated to reflect the amount of time remaining. If posted to during this interval, or a previous unconsumed post is encountered, the post is consumed and 1 is returned.

If the timeout value is 0, **pw_wait()** will return immediately. It returns a 1 if it consumes a previously unconsumed post or it returns EAGAIN if there was no post available to consume.

If the pointer to the timeout value is NULL, the behavior is the same except that the thread will never timeout. If interrupted, EINTR is returned but the timeout value, which by definition is not specified, is not updated.

pw_post() sends a post to the thread identified by *ukid*. If that thread is waiting for a post, the thread wakes up and consumes the post. If that thread was not waiting for a post, the unconsumed post is remembered so that the next time that thread tries to wait for a post, it will consume the saved post and return without warning. At most, one unconsumed post can be remembered per thread.

pw_postv() can be used to post to multiple threads at once. These postings will be atomic in the sense that none will be allowed to preempt the thread doing the posting until all the postings are complete.

The *ukid*s of the target threads must be put into the *targets* array. Errors for respective targets are returned in the *errors* array. The number of entries used in the *targets* and *errors* arrays must be passed in through the *count* argument.

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.

pw_getvmax() returns the maximum number of targets that can be posted to with one **pw_postv()** call. This value is determined by the PW_VMAX kernel tunable.

Refer to the **postwait(2)** man page for a listing of the types of errors that may occur.

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:

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 tick 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.
POSIX clock	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 RCIM POSIX clock is accessed in a manner similar to the tick clock in that the same utilities and device files are used. The POSIX clock can be loaded with any desired time; however, the value loaded is not synchronized with other clocks in an RCIM chain. Only the POSIX clock of the RCIM attached to the host is updated.

The RCIM also provides up to eight 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 or 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 RCIM's 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) PCI Form Factor User's Guide*.

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`** with the HR_POSIX_TIMERS kernel parameter enabled (it is enabled by default in all RedHawk Linux kernels). If this feature is disabled, low resolution POSIX timers are used. Some of the timer functions are also provided as low-resolution by the standard gnu libc **`librt`** 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. Only CLOCK_REALTIME can be set.
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. This call always returns the best available resolution for the clock, usually better than one microsecond.

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 CLOCK_REALTIME or CLOCK_ MONOTONIC.
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. This resolution determines the rounding accuracy of timing expiries set with **`clock_settime(2)`** and the precision used by **`clock_nanosleep(2)`** and **`nanosleep(2)`** calls using the same 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 CLOCK_REALTIME or CLOCK_MONOTONIC.
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.

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 or CLOCK_MONOTONIC.
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 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 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

Per process user and system execution times are reported by the following system features: **ps(1)**, **top(1)**, **times(2)**, **wait4(2)**, **sigaction(2)**, **uptime(1)**, **w(1)**, **getrusage(2)**, **mpstat(1)**, **clock(3)**, **acct(2)** and **/proc/***pid***/stat**. On the generic pre-built RedHawk Linux kernel, the local timer is used to obtain these values. When the CPU is shielded from the local timer using this kernel, no CPU accounting is performed. However, on kernels where the high resolution process accounting facility is configured, it is used for CPU accounting instead of the local timer, and shielding the CPU from the local timer does not affect CPU accounting.

High resolution process accounting samples the time stamp counter (TSC) register during context switching, system calls and interrupt processing. Because the TSC sampling rate is the clock-speed of the processor, this accounting method yields very high resolution with minimal overhead. It maintains system, user, interrupted system and interrupted user time for each process in the system. Support for high resolution process accounting is provided by the **/proc** file system and several "hracct" library routines. The Performance Monitor uses this timing facility for its measurements when used to analyze system performance. The pre-built "debug" and "trace" kernels have this facility configured; it can be configured on other kernels if desired via the HRACCT kernel tunable accessible under General Setup on the Kernel Configuration GUI. Refer to the **hracct(3)** and **hracct(7)** man pages for complete information about this 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 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 describes 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 XFS_DMAPI system parameter accessible under File Systems on the Kernel Configuration GUI 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.

User Space Address Layout

The traditional organization of the virtual address space on i386 systems has the lower 3 GB reserved for the user application and the upper 1 GB for the kernel.

The 3GB user application portion is itself broken down into various sections: the stack starts at 0xc0000000 and grows down, unpinned mmaps start finding space at 0x40000000 and grow up, (by link edit convention), the application text, data and bss regions are made contiguous and are loaded in near address zero (0x08000000). The **`sbrk(2)`** area, used by **`malloc(3)`** to acquire the memory it needs to hand out, starts growing up where the text/data/bss region leaves off.

The problem with this approach is that unpinned mmaps (which most are) are restricted only to the 2GB range 0x40000000 - 0xc00000000. If this proves insufficient then **`mmap(2)`** calls will start to fail when in fact there is enough address space available if the kernel searches for it outside of the traditional range.

The i386 version of RedHawk introduces a new configurable, LARGE_MMAP_SPACE. When it is off, unpinned mmaps behave traditionally as described above. When it is on, unpinned mmaps will allocate their space from the traditional range for as long as possible. At the mmap call where the first allocation failure would have occurred, however, rather than failing, the mmap will search for sufficient space outside of the traditional range.

This feature is enabled by default in all the pre-built i386 RedHawk kernels. The LARGE_MMAP_SPACE kernel tunable is accessible under the Processor Type and Features option of the Kernel Configuration GUI.

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 and not within a HUGETLB area. The current implementation does not support the creation of mappings to HUGETLB areas.

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 location of mappings within the monitoring process. **usermap(3)** 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 configuration parameters.

The kernel configuration parameters are accessible under `Pseudo File Systems` on the Kernel Configuration GUI:

PROCMEM_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.

PROCMEM_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
Non-Uniform Memory Access (NUMA)

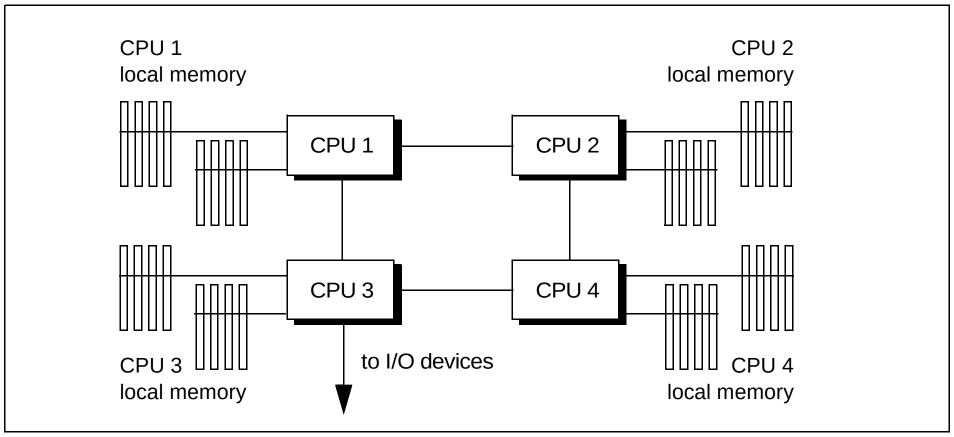
10
Non-Uniform Memory Access (NUMA)

NUMA support, available on Opteron systems, allows you to influence the memory location from which a program's pages are to be allocated.

Overview

On a system with non-uniform memory access (NUMA), it takes longer to access some regions of memory than others. A multiprocessor Opteron system is a NUMA architecture. This is because each CPU chip is associated with its own memory resources. The CPU and its associated memory are located on a unique physical bus. A CPU may quickly access the memory region that is on its local memory bus, but other CPUs must traverse one or more additional physical bus connections to access memory which is not local to that CPU. The relationship between CPUs and buses is shown in Figure 10-1.

Figure 10-1 CPU/Bus Relationship on a NUMA System



This means that the time to access memory on an Opteron system is going to be dependent upon the CPU where a program runs and the memory region where the program's pages are allocated.

A NUMA node is defined to be one region of memory and all CPUs that reside on the same physical bus as the memory region of the NUMA node. During system boot the kernel determines the NUMA memory-to-CPU layout, creating structures that define the association of CPUs and NUMA nodes. On current Opteron systems, the physical bus where a memory region resides is directly connected to only one CPU.

To get optimal performance, a program must run on a CPU that is local to the memory pages being utilized by that program. The NUMA interfaces described in this chapter allow a program to specify the node from which a program's pages are allocated. When coupled with the mechanisms for setting a process' CPU affinity, these interfaces allow a program to obtain very deterministic memory access times.

NUMA support is available only on iHawk systems with Opteron processors. It is possible to configure an Opteron system so that some CPUs do not have any memory that is local. In this case the CPUs with no memory will be assigned to a NUMA node, but all of the memory accesses from this CPU will be remote memory accesses. This affects the memory performance of processes that run on both the CPU with no memory and processes on the CPU that is in the same NUMA node which does have memory. This is not an optimal configuration for deterministic program execution.

Refer to the section “Configuration” later in this chapter for configuration details. Refer to the section “Performance Guidelines” for more information on how to optimize memory performance and to obtain deterministic memory access time. Note that deterministic memory access is crucial for obtaining deterministic program execution times.

Memory Policies

NUMA support implements the concept of memory policies. These memory policies are applied task-wide on a per-user-task basis. Ranges of virtual address space within a given task may also have their own separate memory policy, which takes precedence over the task-wide memory policy for those pages. Memory policies, both task-wide and for virtual address areas, are inherited by the child task during a fork/clone operation.

The NUMA memory policies are:

MPOL_DEFAULT	This is the default where memory pages are allocated from memory local to the current CPU, provided that memory is available. This is the policy that is used whenever a task or its children have not specified a specific memory policy. You can explicitly set the MPOL_DEFAULT policy as the task-wide memory policy or for a virtual memory area within a task that is set to a different task-wide memory policy.
MPOL_BIND	This is a strict policy that restricts memory allocation to only the nodes specified in a nodemask at the time this policy is set. Pages are allocated only from the specified node(s) and page allocations can fail even when memory is available in other nodes not in the bind nodemask. This policy provides more certainty as to which node(s) pages are allocated from than the other memory policies.
	Note that the only way to guarantee that all future memory allocations for a process will be to local memory is to set both the CPU affinity and MPOL_BIND policy to a single CPU.
MPOL_PREFERRED	This policy sets a preferred (single) node for allocation. The kernel will try to allocate pages from this node first and use other nodes when the preferred node is low on free memory.
MPOL_INTERLEAVE	This policy interleaves (in a round-robin fashion) allocations to the nodes specified in the nodemask. This optimizes for bandwidth instead of latency. To be effective, the memory area should be fairly large.

In addition to user-space page allocations, many of the kernel memory allocation requests are also determined by the currently executing task's task-wide memory policy. However, not all kernel page allocations are controlled by the current task's memory policy. For example, most device drivers that allocate memory for DMA purposes will instead allocate memory from the node where the device's I/O bus resides, or the from the node that is closest to that I/O bus.

Page allocations that have already been made are not affected by changes to a task's memory policies. As an example, assume that there is a 1-to-1 CPU to node correspondence on a system with two CPUs:

> If a task has been executing for a while on CPU 0 with a CPU affinity of 0x1 and a memory policy of MPOL_DEFAULT, and it then changes its CPU affinity to 0x2 and its memory policy to MPOL_BIND with a nodemask value of 0x2, there will most likely be pages in its address space that will be non-local to the task once that task begins execution on CPU 1.

The following sections describe the system services, library functions and utilities available for NUMA management.

NUMA User Interface

The **run(1)** command can be used to establish or change memory policies for a task at run time. **shmconfig(1)** can be used for shared memory areas.

Library functions, system services and other utilities and files are also available for NUMA control.

Details of this support are given in the sections below.

NUMA Support for Processes using run(1)

The "mempolicy" option to **run(1)** can be used to establish a task-wide NUMA memory policy for the process about to be executed as well as display related information.

The synopsis is:

```
run [OPTIONS] COMMAND [ARGS]
```

"mempolicy" is one of the available *OPTIONS* and has the following forms:

```
--mempolicy=MEMPOLICY_SPECIFIER
-M MEMPOLICY_SPECIFIER
```

Note that a *PROCESS/THREAD_SPECIFIER*, which identifies the existing process or thread that **run** acts upon, cannot be used with the mempolicy option, which affects only the process(es) about to be created.

MEMPOLICY_SPECIFIER includes only one of the following. Each can be abbreviated to its initial unique character. *list* is a comma-separated list or range of CPUs; e.g., "0,2-4,6". "active" or "boot" can be used to specify all active processors or the boot processor, respectively. An optional tilde [~] negates the list, although "active" cannot be negated.

[**~**] *list*
b[**ind**]**=***list*
: Executes the specified program using the MPOL_BIND memory policy using the memory local to the CPUs in *list*.

b[**ind**]
: Executes the specified program using the MPOL_BIND memory policy using memory local to the CPUs specified with the **--bias** option. The **--bias** option defines the CPUs on which the program is to run and must also be specified with this choice.

i[**nterleave**]**=** [**~**] *list*
: Executes the specified program using the MPOL_INTERLEAVE memory policy using the memory local to the CPUs in *list*.

p[**referred**]**=***cpu*
: Executes the specified program using the MPOL_PREFERRED memory policy, preferring to use memory local to the single specified CPU.

d[**efault**]
: Executes the specified program using the MPOL_DEFAULT memory policy. This is the default memory policy.

n[**odes**]
: Displays the CPUs included in each NUMA node along with total memory and currently free memory on each node. No other options or programs are specified with this invocation of **run**.

v[**iew**]
: Displays the memory policy setting of the current process. No other options or programs are specified with this invocation of **run**.

When a system contains one or more CPUs without local memory, these CPUs are assigned to a node in round-robin fashion during system initialization. Although assigned to a node, they do not actually have local memory and will always make non-local memory accesses, including memory accesses to their own assigned node. Under this type of configuration, **v**[**iew**] output will include an additional "NoMemCpus" column which will indicate the CPUs on each NUMA node that contain no local memory. It is recommended that hardware be configured so that each CPU has a memory module installed when using a NUMA-enabled kernel.

Refer to the **run(1)** man page or the section "The run Command" in Chapter 4 for other options to **run**.

If **numactl(8)** is available on your system, it can also be used to set NUMA memory policies.

NUMA Support for Shared Memory Areas using shmconfig(1)

NUMA policies can be assigned to new shared memory areas or modified for existing shared memory areas using **`shmconfig(1)`** with the “mempolicy” option.

The synopsis is:

`/usr/bin/shmconfig -M` *MEMPOLICY* [**`-s`** *SIZE*] [**`-g`** *GROUP*] [**`-m`** *MODE*] [**`-u`** *USER*]
[**`-o`** *offset*] [**`-S`**] [**`-T`**] {*key* | **`-t`** *FNAME*}

The “mempolicy” option has the following forms:

`--mempolicy`=*MEMPOLICY*
`-M` *MEMPOLICY*

MEMPOLICY includes only one of the following. Each can be abbreviated to its initial unique character. *LIST* is a comma-separated list or range of CPUs; e.g., “0,2-4,6”. “active” or “boot” can be used to specify all active processors or the boot processor, respectively. An optional tilde [~] negates the list, although “active” cannot be negated.

To view the CPUs that are included in each node, and total and available free memory for each node, use **`run -M nodes`**.

[**`~`**] *LIST*
`b`[**`ind`**]**`=`***LIST*

Sets the specified segment to the MPOL_BIND memory policy using the memory local to the CPUs in *LIST*.

`i`[**`nterleave`**]**`=`** [**`~`**] *LIST*

Sets the specified segment to the MPOL_INTERLEAVE memory policy using the memory local to the CPUs in *LIST*.

`p`[**`referred`**]**`=`***CPU*

Sets the specified segment to the MPOL_PREFERRED memory policy, preferring to use memory local to the single specified CPU.

`d`[**`efault`**]

Sets the specified segment to the MPOL_DEFAULT memory policy. This is the default.

`v`[**`iew`**] Displays the current memory policy setting for the specified segment.

Additional options that can be used with the mempolicy option include:

`--size=`*SIZE*
`-s` *SIZE* Specifies the size of the segment in bytes.

`--offset` *OFFSET*
`-o` *OFFSET*

Specifies an offset in bytes from the start of an existing segment. This value is rounded up to a pagesize multiple. If the **`-s`** option is also specified, the sum of the values of offset+size must be less than or equal to the total size of the segment.

--user=*USER*
-u *USER* Specifies the login name of the owner of the shared memory segment.

--group=*GROUP*
-g *GROUP*
Specifies the name of the group to which group access to the segment is applicable.

--mode=*MODE*
-m *MODE* Specifies the set of permissions governing access to the shared memory segment. You must use the octal method to specify the permissions; the default mode is 0644.

--strict
-S Outputs an error if any pages in the segment range do not conform to the specified memory policy currently being applied.

--touch
-T Causes a touch (read) to each page in the specified range, enforcing the memory policy early. By default, the policy is applied as applications access these areas and fault in/allocate the pages.

The *key* argument represents a user-chosen identifier for a shared memory segment. This identifier can be either an integer or a standard path name that refers to an existing file. When a pathname is supplied, an `ftok(key,0)` will be used as the key parameter for the **shmget(2)** call.

--tmpfs=*FNAME* / **-t** *FNAME* can be used to specify a tmpfs filesystem filename instead of a key. The **-u, -g** and **-m** options can be used to set or change the file attributes of this segment.

Refer to the man page or the section "The shmconfig Command" in Chapter 3 for other options to **shmconfig**.

If **numactl(8)** is available on your system, it can also be used to set NUMA memory policies.

System Calls

The following system service calls are available. Note that the **numaif.h** header file should be included when making any of these calls. Refer to the man pages for details.

set_mempolicy(2)	Sets a task-wide memory policy for the current process.
get_mempolicy(2)	Gets the memory policy of the current process or memory address.
mbind(2)	Sets a policy for a specific range of address space, including shared memory.

Library Functions

The library, **`/usr/lib64/libnuma.so`**, offers a simple programming interface to the NUMA support. It contains various types of NUMA memory policy and node support routines and alternative interfaces for using the underlying NUMA system service calls. Refer to the **`numa(3)`** man page for details.

Informational Files

When NUMA is enabled in the kernel, each node has a set of information files in the subdirectory **`/sys/devices/system/node/node`**#, where # is the node number (0, 1, 2 etc.). These files are listed below.

`cpumap`	Displays a hexadecimal bitmap of the CPUs in this node; e.g.

```
> cat /sys/devices/system/node/node3/cpumap
08
```

`numastat`	Displays hit/miss statistics for the node. See the next section for explanations of the fields that are displayed.
`meminfo`	Displays various memory statistics for the node, including totals for free, used, high, low and all memory.
`cpu`#	These are the CPU device files associated with the node; e.g.

```
$ ls -l /sys/devices/system/node/node3/cpu3
lrwxrwxrwx 1 root root 0 jan 21 03:01 cpu3
->../../../../devices/system/cpu/cpu3
```

NUMA Hit/Miss Statistics Using numastat

`numastat` is a script that combines the information from all the nodes' **`/sys/devices/system/node/node`#`/numastat`** files:

```
$ numastat
                 node 3    node 2    node 1    node 0
numa_hit          43674     64884     79038     81643
numa_miss             0         0         0         0
numa_foreign          0         0         0         0
interleave_hit     7840      5885      4975      7015
local_node        37923     59861     75202     76404
other_node         5751      5023      3836      5239
```

`numa_hit`	the number of successful memory allocations made from the node
`numa_miss`	the number of memory allocations that could not be made from the node but were instead allocated for another node
`numa_foreign`	the number of allocations that failed to allocate memory from a node but were instead allocated from this node

`interleave_hit`	the number of successful interleaved memory allocations made from this node
`local_node`	the number of memory allocations that were made from the local node
`other_node`	the number of memory allocations that were made to a non-local node

kdb Support

The following **`kdb`** commands have been added or modified to support NUMA. Note that this additional support is only present when the kernel is configured with NUMA support enabled.

`memmap` [*node_id*]	outputs information for all pages in the system, or for only the specified node
`task`	additionally outputs the mempolicy and `il_next` task structure fields
`mempolicy` *addr*	outputs information for the specified mempolicy structure
`pgdat` [*node_id*]	decodes the specified node's zonelists, or if *node_id* is not specified, zone 0
`vmp -v`	additionally outputs mempolicy information for virtual memory areas

Performance Guidelines

Through CPU shielding, CPU biasing and binding an application to specific NUMA nodes, page allocations can be made in the most efficient manner on NUMA systems. Guidelines for working with tasks and shared memory areas are given below.

Task-Wide NUMA Mempolicy

The MPOL_BIND policy is usually the most useful policy for time-critical applications. It is the only policy that lets you deterministically specify the node(s) for page allocations. If the memory allocation cannot be made from the specified node or set of specified nodes, the program will be terminated with a SIGKILL signal.

By combining CPU shielding and CPU biasing with the MPOL_BIND memory policy, a shielded CPU can be created and the application executed on the shielded CPU where the pages for that application will be allocated only from the shielded CPU's NUMA node. Note that pre-existing shared text page sand copy on write data pages may not be local, although copy on write data pages will become local once they are written to.

The **run(1)** command can be used to start up an application on a shielded CPU with the MPOL_BIND memory policy. Alternatively, since pages that are already present in an application's address space are not affected by any subsequent change of NUMA memory policy, the application can set its CPU affinity and NUMA memory policy as soon as possible after it has begun executing with **mpadvise(3)** and **set_mempolicy(2)** or NUMA library function calls.

The following example shows how to use the **run(1)** command bias and mempolicy options to start up an application on a shielded CPU with the MPOL_BIND memory policy with memory allocations coming only from the NUMA node where CPU 2 resides:

```
$ shield -a 2
$ run -b 2 -M b my-app
```

For more information about shielded CPUs and the **shield(1)** command, see Chapter 2 and the **shield(1)** man page.

Shared Memory Segments

It is also generally recommended that the MPOL_BIND memory policy be used for shared memory segments. A shared segment's NUMA memory policy can be specified with the **mbind(2)** system service call or with the **shmconfig(1)** utility.

If a shared memory segment is to be referenced from multiple CPUs, it is possible to specify different MPOL_BIND mempolicy attributes for different portions of a shared memory area in order to maximize memory access performance.

As an example, consider a "low" application that mainly writes to the lower half of a shared memory segment, and a "high" application that mainly writes to the upper half of the same shared memory segment.

1. Create a shared memory segment with a key value of '123'. Change the lower half of the segment to use the MPOL_BIND memory policy with CPU 2's NUMA node for page allocations, and the upper half to use MPOL_BIND with CPU 3's node for page allocations.

   ```
   $ shmconfig -s 0x2000 123
   $ shmconfig -s 0x1000 -M b=2 123
   $ shmconfig -o 0x1000 -M b=3 123
   ```

2. Shield both CPUs 2 and 3.

   ```
   $ shield -a 1,2
   ```

3. Start up the "low" application on CPU 2 with a MPOL_BIND mempolicy using CPU 2's NUMA node for memory allocations, and start up the "high" application on CPU 3 with a MPOL_BIND mempolicy using CPU 3's NUMA node for memory allocations.

   ```
   $ run -b 2 -M b low
   $ run -b 3 -M b high
   ```

Configuration

Only the Opteron processors have NUMA architecture. The K8_NUMA kernel parameter must be enabled for NUMA kernel support. This tunable is accessible under the Processor Type and Features selection in the Kernel Configuration GUI and is enabled by default in all pre-built RedHawk x86_64 kernels.

Note that there is a boot option, numa=off, that can be specified at boot time that will disable NUMA kernel support on a NUMA system. This will create a system with a single node, with all CPUs belonging to that node. It differs from NUMA support not being built into the kernel, in which case there is a flat memory system with no nodes and where the NUMA user interfaces will return errors when called.

When using a K8_NUMA enabled kernel on an Opteron system, the following hardware recommendations are made:

- It is highly recommended that a memory module be installed for each CPU in the system. Otherwise, CPUs without a local memory module must remotely access other memory modules for every memory access, thus degrading system performance.
- Any BIOS-supported memory module interleaving hardware support should be disabled in the BIOS. If not disabled, NUMA support in a K8_NUMA enabled kernel will be disabled, resulting in a single NUMA node containing all the CPUs in the system.

11
Configuring and Building the Kernel

11
Configuring and Building the Kernel

This chapter provides information on how to configure and build a RedHawk Linux 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 (may contain suffixes such as -rc1 or -pre7)
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:

On an iHawk i386 architecture (32-bit):

generic kernel	**`static.config`**
trace kernel	**`trace-static.config`**
debug kernel	**`debug-static.config`**

On an iHawk Opteron architecture (64-bit):

generic kernel	**`static-x86_64.config`**
trace kernel	**`trace-static-x86_64.config`**
debug kernel	**`debug-static-x86_64.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(8)`** tool.

NOTE

> The **`ccur-config`** script must be run as root. If kernel modifications are to be made, the system must be in graphical mode (i.e. run-level 5) or a valid DISPLAY variable must be set.

The following example configures the kernel source tree for building a new kernel based on the RedHawk Linux 2.1 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-2.6.3RedHawk2.1
#./ccur-config trace-static
```

`ccur-config` can also be used for customized kernels by specifying the appropriate custom config file residing in the **`configs`** directory. The **`-k`** *name* option can be used to name a new flavor, and the **-s** option saves the configuration file in the **configs** directory. For example:

```
# ./ccur-config -s -k test debug-static
```

configures a kernel with **-test** as the flavor suffix that is based on the RedHawk i386 **debug-static** kernel and saves the resulting configuration as **configs/test.config**.

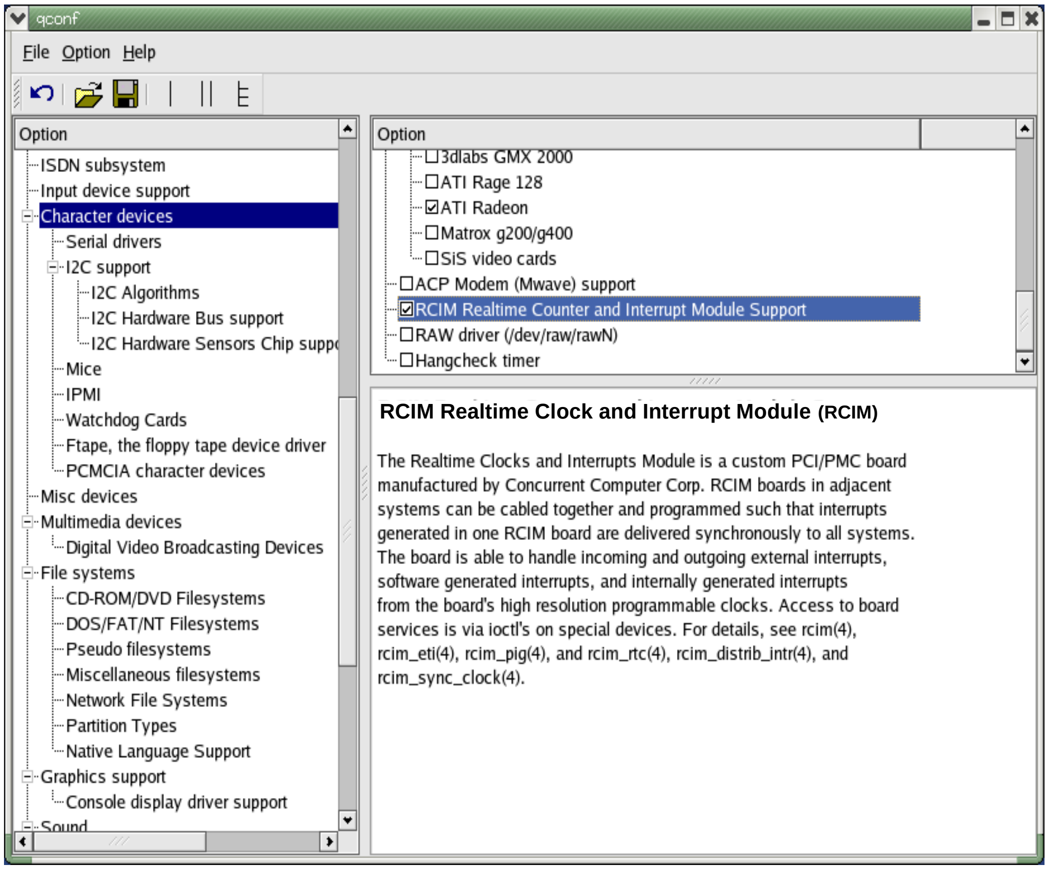
During the execution of **ccur-config** you will be presented with a graphical configuration interface (GUI) in which you can customize many different aspects of the RedHawk Linux kernel. See Screen 11-1 for an example of the Kernel Configuration GUI.

The Save selection from the File menu must be selected to save your changes and exit the program. Note that even if you do not change any configuration parameters, it is still necessary to select Save 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 tunable parameters related to unique RedHawk features and real-time performance are discussed throughout this manual and listed in Appendix B. In addition, when the parameter is selected, information about that parameter is displayed in a separate window of the GUI.

If you do not wish to change kernel parameters, specify the **-n** option to **ccur-config** and the GUI will not appear.

Screen 11-1 Kernel Configuration GUI



☐ 3dlabs GMX 2000
☐ ATI Rage 128
☑ ATI Radeon
☐ Matrox g200/g400
☐ SiS video cards
☐ ACP Modem (Mwave) support
☑ RCIM Realtime Counter and Interrupt Module Support
☐ RAW driver (/dev/raw/rawN)
☐ Hangcheck timer

RCIM Realtime Clock and Interrupt Module (RCIM)

The Realtime Clocks and Interrupts Module is a custom PCI/PMC board manufactured by Concurrent Computer Corp. RCIM boards in adjacent systems can be cabled together and programmed such that interrupts generated in one RCIM board are delivered synchronously to all systems. The board is able to handle incoming and outgoing external interrupts, software generated interrupts, and internally generated interrupts from the board's high resolution programmable clocks. Access to board services is via ioctl's on special devices. For details, see rcim(4), rcim_eti(4), rcim_pig(4), and rcim_rtc(4), rcim_distrib_intr(4), and rcim_sync_clock(4).

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-2.6.3-RedHawk-2.1-custom
```

The final suffix can be changed by specifying the **`-k`** *name* option to **`ccur-config`**. This defines *name* as the REDHAWKFLAVOR variable in the top-level **`Makefile`**, which remains in effect until changed again with **`-k`** or by editing **`Makefile`**. When building multiple kernels from the same kernel source tree, it is important to change the suffix to avoid overwriting existing kernels accidentally.

NOTES

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 a kernel (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 bzImage	Build a standalone kernel.
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/`***kernelversion***`-RedHawk-`***x.x***`-custom`**".
make install	Install the kernel into the **`/boot`** directory along with an associated **`System.map`** file.

NOTE

To completely build and install a new kernel, all of these **`Makefile`** targets must be issued in the order shown above.

For an example of a complete kernel configuration and build session, refer to Figure 11-1.

Figure 11-1 Example of Complete Kernel Configuration and Build Session

```
# cd /usr/src/linux-2.6.3RedHawk2.1
# ./ccur-config -k test debug-static
Configuring version: 2.6.3-RedHawk-2.1-test
Cleaning source tree...
Starting graphical configuration tool...
[ configure kernel parameters as desired ]

Configuration complete.

# make bzImage
# make modules
# make modules_install
# make 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 either one of the pre-existing kernels supplied by Concurrent or a custom kernel.

To build driver modules for a kernel, the following conditions must be met:

- The desired kernel must be the currently running kernel.
- The kernel source directory must be configured properly for the currently running kernel via **`ccur-config`**.

Note that if a custom kernel was built using the procedure outlined in the section “Building a Kernel,” then the kernel source directory is already configured properly and running **`ccur_config`** is not necessary.

The **`-c`** option to **`ccur-config`** can be used to ensure that the kernel source directory is properly configured. This option 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.

The **`-n`** option to **`ccur_config`** can also be specified when it is not necessary to change configuration parameters. With **`-n`**, the configuration GUI does not appear and no configuration customization is performed.

See Figure 11-2 for an example of building a kernel module for a pre-built RedHawk Linux kernel.

Figure 11-2 Example of Building a Kernel Module for a Pre-built RedHawk Kernel

```
# cd /usr/src/linux-2.6.3RedHawk2.1
# ./ccur-config -c

[ Enable the desired driver modules in GUI]

# make REDHAWKFLAVOR=-flavor modules

[ See make output to locate newly built kernel module]
```

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

12
Linux Kernel Crash Dump (LKCD)

12
Linux Kernel Crash Dump (LKCD)

This chapter discusses the Linux Kernel Crash Dump facility, how it is configured and some examples of its use.

WARNING

Crash dumps are disruptive. It is recommended that crash dumps be activated only during testing/debugging. When LKCD is active, any kernel "Oops" causes the system to halt, dump the memory image to swap and reboot. When LKCD is not active, the system may continue running.

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 swap 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)`**.

RedHawk Linux contains an updated version of the LKCD currently available at http://lkcd.sourceforge.net/. For general information about this feature, including documentation, refer to this web site.

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.

WARNING

Dumping to a partition other than a swap partition will destroy the filesystem.

The **/etc/sysconfig/dump** configuration file contains parameters that control the configuration of LKCD. This file may be edited, if needed, to make certain specifications; for example, to modify the method of compressing dumps or to change the directory where dumps are saved. After making modifications to the LKCD configuration file, the user must execute the command **lkcd config** for those changes to take effect.

The DUMP_ACTIVE parameter in the LKCD configuration file enables or disables system dumps. When this variable setting is DUMP_ACTIVE=1, LKCD will cause a dump to be generated when a panic or Oops occurs. To deactivate LKCD, set DUMP_ACTIVE=0.

The type of dump to be performed is set via the DUMP_FLAGS setting. Disruptive disk dump is the default setting (DUMP_FLAGS=0x80000000). This setting saves dumps to the swap partition with an automatic reboot. During the reboot, the image is saved to **/var/log/dump**.

The compression setting for dumps is set via the DUMP_COMPRESS setting. The memory image uses gzip compression by default (DUMP_COMPRESS=2).

To see the configuration of lkcd at any time, execute the command:

lkcd query

```
[root@opteron2 root]# lkcd query
    Configured dump device: 0x803
    Configured dump flags:
    Configured dump level:
KL_DUMP_LEVEL_HEADER|KL_DUMP_LEVEL_KERN|KL_DUMP_LEVEL_USED
Configured dump compression method: KL_DUMP_COMPRESS_GZIP
[root@opteron2 root]#
```

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:	Alt+SysRq+D
Using serial console:	Break followed by D

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 a Break, followed by D

In addition to the SysRq request method to force a crash dump, there is a new “manual dump” feature in this version of LKCD. This looks very similar to a dump forced by a SysRq request. Below is an example of a successful disruptive manual dump. To force a system dump, the **`lkcd`** command is used as shown here:

```
[root@opteron2 root]# lkcd dump
Dumping to block device (8,3) on CPU 1 ...
..|
 42274 dump pages saved of 4096 each in pass 0
.../
 10780 dump pages saved of 4096 each in pass 1

 1355880 dump pages skipped of 4096 each in pass 2

 0 dump pages skipped of 4096 each in pass 3

 Rebooting in 5 seconds ...
```

The above output goes to the console, and will be the last thing you see until the system comes back from reboot in the case of disruptive dumps.

Below shows excerpts of the system rebooting and saving the crash dump before activating the swap drive (disruptive dump):

```
...
Crash dump driver initialized.
block device driver for LKCD registered
...

Your system appears to have shut down uncleanly
Press Y within 1 seconds to force file system integrity check...
Checking root filesystem
/: clean, 251051/2052288 files, 1441830/4096575 blocks
[/sbin/fsck.ext3 (1) -- /] fsck.ext3 -a /dev/sda2
[  OK  ]
Remounting root filesystem in read-write mode:  [  OK  ]
Finding module dependencies:  [  OK  ]
Checking filesystems
/boot: recovering journal
```

```
/boot: clean, 61/128520 files, 61214/514048 blocks
/home: recovering journal
/home: clean, 26/960992 files, 38390/1919759 blocks
/tmp: recovering journal
/tmp: clean, 1571/640000 files, 97607/1279167 blocks
Checking all file systems.
[/sbin/fsck.ext3 (1) -- /boot] fsck.ext3 -a /dev/sda1
[/sbin/fsck.ext3 (1) -- /home] fsck.ext3 -a /dev/sda6
[/sbin/fsck.ext3 (1) -- /tmp] fsck.ext3 -a /dev/sda5
[  OK  ]
Mounting local filesystems:  [  OK  ]

Configuring system for crash dumps [  OK  ]
Saving crash dump (if one exists) [  OK  ]
Activating swap partitions:  [  OK  ]
Enabling local filesystem quotas:  [  OK  ]
Enabling swap space:  [  OK  ]
```

Note that it has been observed that the autoreboot feature for disruptive dumps will hang on occasion. If this occurs, a powercycle is necessary. The system should be able to boot up and save the dump correctly after the powercycle, as long as the hang did not occur in the middle of the dump.

Below is an example of the output when a kernel oops occurs.

```
[root@ercom1 root]# ./crashme
test: no version magic, tainting kernel.
Loading the test driver
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
f8afc025
*pde = 00000000
Oops: 0002 [#1]
CPU:    0
EIP:    0060:[<f8afc025>]    Tainted: PF
EFLAGS: 00010282
EIP is at test_module_init+0x25/0x2e [test]
eax: f8afc05f   ebx: 0804b018   ecx: 00000002   edx: 00000000
esi: f8afc400   edi: 00000000   ebp: f659dfa0   esp: f659df98
ds: 007b   es: 007b   ss: 0068
Process insmod (pid: 1435, threadinfo=f659c000 task=f7608810)
Stack: 00000286 00000000 f659c000 c0144ba9 c0606ac0 00000001 f8afc400
0804b018
       080486dd 00000000 c010b3c1 0804b018 00007532 0804b008 080486dd
00000000
       bfffecc8 00000080 0000007b 0000007b 00000080 ffffe410 00000073
00000246
Call Trace:
 [<c0144ba9>] sys_init_module+0x192/0x37b
 [<c010b3c1>] sysenter_no_trace_1+0x1c/0x33

Code: 88 02 b8 00 00 00 00 c9 c3 55 89 e5 83 ec 08 83 ec 0c 68 77
Dumping to block device (8,3) on CPU 0 ...
.-
 10900 dump pages saved of 4096 each in pass 0
...\
 10372 dump pages saved of 4096 each in pass 1

 240751 dump pages skipped of 4096 each in pass 2

 0 dump pages skipped of 4096 each in pass 3

 Rebooting in 5 seconds ...
```

Using lcrash

lcrash has several functions. It is the utility used to save core dumps and to examine either core files or live memory.

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 running kernel. If you boot a different kernel after 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.

Notice the difference between Example 1 and Example 2 below. In Example 1, **lcrash** is used to read in a core file for analysis. In Example 2, **lcrash** is used to read **/dev/mem**, which provides the capability to examine a still running system.

Example 1

This example illustrates using **lcrash** to analyze a core file. You first need to change into the directory where the dump to be examined is saved: **/var/log/dump/***n*, where *n* is the number of the dump. This example uses a dump saved in **/var/log/dump/1**. Then execute the command **lcrash -n1**.

```
[root@ercom1 1]# pwd
/var/log/dump/1
[root@ercom1 1]# ls
analysis.1  dump.1  index.1  kerntypes.1  lcrash.1  map.1
[root@ercom1 1]# lcrash -n1
lcrash 0.9.2 (xlcrash) build at Jul 14 2004 02:36:01
Lcrash is free software. It is covered by the GNU General Public License.
You are welcome to change it and/or distribute copies of it under certain
conditions. Type “help -C” to see the conditions. Absolutely no warranty
is given for Lcrash. Type “help -W” for warranty details.

      map = map.1
     dump = dump.1
kerntypes = kerntypes.1


Please wait...
        Check dump architecture:
                Dump arch set.
        Init host arch specific data ... Done.
        Init dump arch specific data ... Done.
        Loading system map ... Done.
        Set dump specific data ... Done.
        Loading type info (Kerntypes) ... Done.
        Version of map,dump and types:
                broccoli_Wed_Jul_14_02_19_10_EDT_2004
        Loading ksyms from dump .... Done.
```

```
DUMP INFORMATION:

     architecture: i386
       byte order: little
     pointer size: 32
   bytes per word: 4

   kernel release: 2.6.6
      memory size: 939524096 (0G 896M 0K 0Byte)
   num phys pages: 262103
   number of cpus: 4

>>
```

Example 2

This example illustrates using **lcrash** to analyze a running system. Executing the **lcrash** command with no arguments starts **lcrash** using the files that match the running kernel (symlinks **/boot/System.map** and **/boot/Kerntypes**), and **/dev/mem** instead of a core file (**dump.***n*).

```
[root@ercom1 root]# lcrash
lcrash 0.9.2 (xlcrash) build at Jul 14 2004 02:36:01
Lcrash is free software. It is covered by the GNU General Public License.
You are welcome to change it and/or distribute copies of it under certain
conditions. Type "help -C" to see the conditions. Absolutely no warranty
is given for Lcrash. Type "help -W" for warranty details.

      map = /boot/System.map
     dump = /dev/mem
kerntypes = /boot/Kerntypes


Please wait...
        Check dump architecture:
Warning: Unknown magic number in dump header.
Warning: Unknown dump arch. Setting dump arch to host arch.
        Init host arch specific data ... Done.
        Init dump arch specific data ... Done.
        Loading system map ... Done.
        Set dump specific data ... Done.
        Loading type info (Kerntypes) ... Done.
        Version of map,dump and types:
                broccoli_Wed_Jul_14_02_19_10_EDT_2004
        Loading ksyms from dump .... Done.

DUMP INFORMATION:

     architecture: i386
       byte order: little
     pointer size: 32
   bytes per word: 4

   kernel release: 2.6.6
      memory size: 939524096 (0G 896M 0K 0Byte)
   num phys pages: 262103
   number of cpus: 4

>>
```

Example 3

Available **`lcrash`** commands can be listed using the **`?`** command.

```
>> ?
?                 info              pb                strace
addtypes          ldcmds            pd                symbol
base              load              po                symtab
bt                main              print             t
defcpu            md                ps                task
deftask           mktrace           px                trace
dis               mmap              q                 unload
dt                module            q!                version
dump              mt                quit              vi
findsym           namelist          rd                vtop
fsym              nmlist            report            walk
h                 od                savedump          whatis
help              offset            set
history           p                 sizeof
id                page              stat
>>
```

Typing **`?`** followed by the command name will give help on that command.

```
>> ? mmap
COMMAND: mmap [-f] [-n] [-w outfile] mmap_list

    Display relevant information for each entry in mmap_list.

>> ? deftask
COMMAND: deftask [-w outfile] [task]

    Set the default task if one is indicated.  Otherwise, display the
current value of deftask. When ‘lcrash’ is run on a system core
    dump, deftask gets set automatically to the task that was active when
the system PANIC occurred.When ‘lcrash’ is run on a live
    system, deftask is not set by default.

    The deftask value is used by ‘lcrash’ in a number of ways. The trace
command will display a trace for the default task if one is set.
    Also, the translation of certain virtual addresses (user space) depends
upon deftask being set.

>> ? quit
COMMAND: quit

    Exit lcrash.  Note that q will prompt for confirmation unless a ‘!’ is
appended to the command line.

>>
```

Example 4

This example shows the output of the **`stat`** command. The **`stat`** command displays the kernel's log buffer at the time of the system crash, which may contain clues as to what went wrong. It's also a way to determine if you have a good dump.

```
>> stat
```

(last 24 lines of output shown only)

```
    <4>test: no version magic, tainting kernel.
    <4>Loading the test driver
    <1>Unable to handle kernel NULL pointer dereference at virtual address
00000000
    <4> printing eip:
    <4>f8afc025
    <1>*pde = 00000000
    <4>Oops: 0002 [#1]
    <4>CPU:    0
    <4>EIP:    0060:[<f8afc025>]    Tainted: PF
    <4>EFLAGS: 00010282
    <4>EIP is at test_module_init+0x25/0x2e [test]
    <4>eax: f8afc05f   ebx: 0804b018   ecx: 00000002   edx: 00000000
    <4>esi: f8afc400   edi: 00000000   ebp: f659dfa0   esp: f659df98
    <4>ds: 007b   es: 007b   ss: 0068
    <4>Process insmod (pid: 1435, threadinfo=f659c000 task=f7608810)
    <4>Stack: 00000286 00000000 f659c000 c0144ba9 c0606ac0 00000001 f8afc400
0804b018
    <4>       080486dd 00000000 c010b3c1 0804b018 00007532 0804b008 080486dd
00000000
    <4>       bfffecc8 00000080 0000007b 0000007b 00000080 ffffe410 00000073
00000246
    <4>Call Trace:
    <4> [<c0144ba9>] sys_init_module+0x192/0x37b
    <4> [<c010b3c1>] sysenter_no_trace_1+0x1c/0x33
    <4>
    <4>Code: 88 02 b8 00 00 00 00 c9 c3 55 89 e5 83 ec 08 83 ec 0c 68 77
    <4>Dumping to block device (8,3) on CPU 0 ...
    <4>
>>
```

Example 5

This example shows use of the **`ps`** and **`trace`** commands.

```
>> ps
Address     Uid     Pid    PPid Stat      Flags SIZE:RSS   Command
-----------------------------------------------------------------------------
c05e59a0       0       0       0 0x00 0x00000000     0:0     swapper
f7fa1380       0       1       0 0x01 0x00000100   376:127   init
```

(first two and last three lines shown only)

```
e8b95500       0    2016    1416 0x01 0x00000140  1711:544   sshd
e8b941b0       0    2022    2016 0x01 0x00000100  1066:344   bash
efd70c20       0    2072    2022 0x00 0x00000100  1830:1440  lcrash
-----------------------------------------------------------------------------
77 processes found
```

```
>> trace e8b941b0
================================================================
STACK TRACE FOR TASK: 0xe8b941b0(bash)

 0 schedule+995 [0xc0546299]
 1 sys_wait4+524 [0xc012368f]
 2 sys_waitpid+34 [0xc01237c0]
 3 syscall_call [0xc01042b0]
   ebx: ffffffff ecx: bfffe848 edx: 00000002    esi: 00000002
   edi: 00000000 ebp: bfffe858 eax: 00000007     ds: 007b
    es: 007b     eip: 400daf0e  cs: 0073     eflags: 00000246
================================================================
>>
```

13
Pluggable Authentication Modules (PAM)

13
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 Pluggable Authentication Modules, 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. They are described in greater detail in Appendix C.

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.

NOTE: If capabilities are used, the **`/etc/pam.d/su`** file should be modified as a security precaution to ensure that an invocation such as **`su -l nobody`** *daemon* will impart to *daemon* only the capabilities listed for user **`nobody`**, and will not impart any extra capabilities from the invoking user.

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`**.

NOTE: The path to the PAM files on i386 systems is **`/lib/security`**.
The path on Opteron systems is **`/lib64/security`**.

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 allow roles defined in the **`/etc/security/capability.conf`** file to be assigned to substituted users via **`su(1)`**, and to ensure that those substituted users do not inherit inappropriate capabilities from the invoker of **`su(1)`**, append the following line to **`/etc/pam.d/su`**:

   ```
   session required /lib/security/pam_capability.so
   ```

4. 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 described in the **`cap_from_text(3)`** man page. The capabilities are also described in full detail in Appendix C. 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 INHERIT_CAPS_ACROSS_EXEC configuration option accessible under General Setup on the Kernel Configuration GUI (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
```

14
Device Drivers

14
Device Drivers

This chapter addresses issues relating to user-level and kernel-level device drivers under RedHawk Linux. It includes information about functionality added to RedHawk Linux that facilitates writing device drivers as well as real-time performance issues. Prior knowledge of how to write Linux-based device drivers is assumed.

Information about RedHawk support for a PCI-to-VME bridge device can be found in Chapter 15, “PCI-to-VME Support.”

Understanding Device Driver Types

It is possible to write simple user-level device drivers under RedHawk Linux. A user-level driver can access I/O space to read and write device registers, thus initiating a programmed I/O operation. With the assistance of a skeletal kernel driver, a user-level driver can also initiate actions upon receipt of an interrupt. This is accomplished by supporting functions which allow a signal handler in the user-level driver to be attached to the interrupt routine. Refer to the section “Kernel Skeleton Driver” later in this chapter for the location of a sample kernel driver template for handling an interrupt and sending a signal to a user-level process.

It is not practical to write a user-level driver which does DMA I/O operations under Linux. There are several problems that prohibit DMA operations from user-level; for example, there is currently no supported method for determining the physical address of a user space buffer. Kernel-level device drivers should be used for devices that utilize DMA for I/O operations.

Developing User-level Device Drivers

The sections that follow describe particulars of the RedHawk Linux operating system that affect writing user-level device drivers under RedHawk Linux.

Accessing PCI Resources

During the boot process, devices on the PCI bus are automatically configured, have their interrupts assigned and have their registers mapped into memory regions where the device registers can be accessed via memory-mapped I/O operations. These memory regions are known as base address registers (BARs). A device can have up to six BARs. The content of the BARs vary depending upon the device. Consult the device’s manual for this information.

RedHawk Linux supports a PCI resource file system located in **/proc/bus** that simplifies the code needed to map the registers of a PCI device. This file system provides BAR files representing memory regions that can be mapped into the address space of a program, providing access to the device without having to know the physical address associated with the device. The PCI BAR file system also provides a *config-space* file which can be used to read and write to the device's PCI config space. The first 64 bytes of the *config-space* file are defined by the PCI specification. The remaining 192 bytes are device vendor-specific.

Each PCI hardware device has associated with it a Vendor ID and Device ID. These are fixed values that do not change over time or between systems. Because of the dynamic configuration of PCI devices at boot time, the domain, bus, slot and function numbers remain fixed once the system is booted, but may vary between systems depending on the underlying hardware, even for boards that appear to be plugged into the same PCI bus slot in each system. Paths within the **`/proc/bus/pci`** and BAR file systems are derived from the domain, bus, slot and function numbers assigned by the kernel, and are affected by the physical hardware layout of the host system. Changes, such as physically plugging a board into a different slot, adding a device to the system or modifications to the system BIOS can change the bus and/or slot number assigned to a particular device.

The PCI BAR scan interfaces described below offer a method for finding the bar file associated with a particular device. Without these interfaces, the hardware-dependent nature of these BAR file paths makes the task of programming user-level device drivers somewhat inconvenient, because the driver has to locate the slot address of the appropriate device in order to obtain access to its BAR files.

Using the library interface for the BAR file system and the fixed vendor ID and device ID values, the other values currently associated with the PCI devices can be obtained. These include the BAR file directory path for the device as well as information about each BAR file in that directory. It also returns IDs for vendor, device, class, subclass, IRQ number (if assigned), and domain, bus, slot and function numbers related to each device.

PCI BAR Interfaces

The sections that follow explain the PCI BAR interfaces.

The library scan functions are iterative. If the system has more than one instance of the desired device type, these library functions must be called multiple times. One function is provided that returns the count of all matching devices in the system. Other functions will iteratively return information for devices that match the search criteria. Device information is returned in the `bar_context` type defined in **`/usr/include/pcibar.h`**. This structure is created with a call to **`bar_scan_open`**. Multiple scans can be active concurrently, each having a unique `bar_context`.

The interfaces are briefly described as follows:

`bar_scan_open`	starts a new scan of PCI devices
`bar_scan_next`	obtains the next matching PCI device
`bar_device_count`	returns the number of matching devices remaining in the active scan
`bar_scan_rewind`	restarts a scan
`bar_scan_close`	closes the active scan and frees associated memory
`free_pci_device`	frees all allocated memory associated with a located device
`bar_mmap`	**`mmap`**'s the BAR file with proper page alignment
`bar_munmap`	**`munmap`**'s the **`bar_mmap`**'d BAR file

Note that to use these interfaces, you must link your application with the **`libccur_rt`** library:

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

An example illustrating the use of these functions is provided as **`/usr/share/doc/ccur/examples/pci_barscan.c`**.

bar_scan_open(3)

This function is used to create the initial context for a search of PCI devices. The returned `bar_context` is an opaque pointer type defined in **`/usr/include/pcibar.h`** that designates state data for the iterator interfaces. Its value must be provided to subsequent calls to **`bar_scan_next`**, **`bar_device_count`**, **`bar_scan_rewind`** and **`bar_scan_close`**.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

bar_context bar_scan_open(int vendor_id, int device_id);
```

Arguments are defined as follows:

vendor_id	a vendor identification value defined in **`/usr/include/linux/pci_ids.h`**. or the special value ANY_VENDOR. ANY_VENDOR matches all *vendor_id* values for all devices on the host system.
device_id	a device identification value defined in **`/usr/include/linux/pci_ids.h`**. or the special value ANY_DEVICE. ANY_DEVICE matches all *device_id* values for all devices on the host system.

Refer to the man page for error conditions.

bar_scan_next(3)

This function returns a pointer to a `struct pci_device` object for the next matching PCI device found.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

struct pci_device * bar_scan_next(bar_context ctx);
```

The argument is defined as follows:

ctx	an active `bar_context` returned by **`bar_scan_open`**.

When no further matching devices are available, this function returns NIL_PCI_DEVICE and sets *errno* to zero. Refer to the man page for error conditions.

bar_device_count(3)

This function returns the number of unprocessed devices remaining in an active scan. When called immediately after a call to **`bar_scan_open`** or **`bar_scan_rewind`**, this is the total number of matching devices for the specified *vendor_id* and *device_id*. This value is reduced by 1 upon each call to **`bar_scan_next`**.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

int bar_device_count(bar_context ctx);
```

The argument is defined as follows:

ctx	an active `bar_context` returned by **`bar_scan_open`**.

On success, this function returns a non-negative count of the number of unreported devices that would be returned by subsequent calls to **`bar_scan_next`**. Refer to the man page for error conditions.

bar_scan_rewind(3)

This function resets the specified `bar_context` to the state it was in immediately after the initial call to **`bar_scan_open`**.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

void bar_scan_rewind(bar_context ctx);
```

The argument is defined as follows:

ctx	an active `bar_context` returned by **`bar_scan_open`**. If the value is NIL_BAR_CONTEXT or does not designate a valid `bar_context` object, this call has no effect.

bar_scan_close(3)

This function frees all allocated memory associated with the designated `bar_context`. The value NIL_BAR_CONTEXT is assigned to the `bar_context` object and may no longer be used after this call.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

void bar_scan_close(bar_context ctx);
```

The argument is defined as follows:

ctx	an active `bar_context` returned by **`bar_scan_open`**.

free_pci_device(3)

This function releases all allocated memory associated with the designated `struct pci_device` object.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

void free_pci_device(struct pci_device * dev);
```

The argument is defined as follows:

dev	a valid `struct pci_device` obtained from **`bar_scan_next`**.

bar_mmap(3)

This function can be used to map the specified BAR file into memory. It is a wrapper around **`mmap(2)`** that aligns small BAR files at the start of the mmap'ed BAR data rather than the beginning of the area that is mmap'ed. Use **`bar_munmap(3)`** to unmap files mapped using this function.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

void * bar_mmap(char * barfilepath, void * start, size_t length, int
prot, int flags, int fd, off_t offset);
```

The arguments are defined as follows:

barfilepath	the path of the BAR file to be mmap'ed

Refer to **`mmap(2)`** for a description of the other parameters.

bar_munmap(3)

This function must be used to unmap files that are mapped using **`bar_mmap(3)`**.

Synopsis

```
#include <linux/pci_ids.h>
#include <pcibar.h>

int bar_munmap(void * start, size_t length);
```

Refer to **`munmap(2)`** for a description of the parameters.

Kernel Skeleton Driver

When a device issues interrupts that must be handled by the device driver, it is not possible to build the device driver completely at user level because Linux has no method for attaching a user-level routine to an interrupt. It is however possible to build a simple kernel device driver that handles the device interrupt and issues a signal to the user-level application that is running a user-level driver. Because signals are delivered asynchronously to the execution of a program and because signals can be blocked during critical sections of code – a signal acts much like a user-level interrupt.

The following example of a skeletal kernel-level driver shows how to attach a signal to the occurrence of a device interrupt and the code for the interrupt service routine which will then trigger the signal. The full code for this skeletal driver can be found on a RedHawk installed system in the directory **/usr/share/doc/ccur/examples/driver**. You can use the sample driver, **sample_mod**, as a template for writing a simple kernel-level driver that handles an interrupt and sends a signal to a user-level process.

Understanding the Sample Driver Functionality

The sample driver uses real time clock (rtc) 0 as the hardware device that will generate the interrupts. Rtc 0 is one of the real-time clocks on Concurrent's Real-Time Clock and Interrupt Module (RCIM). The clock counts down to 0 at a predefined resolution and then starts over. Each time the count reaches 0, an interrupt is generated. Some of the setup for real time clock 0 is performed in the module's "init" routine where the device registers are mapped into memory space so that the driver may access those registers. The last section of code shown for the module's "init" routine is the code that attaches the interrupt routine to an interrupt vector.

```
*******************************************************************************
int sample_mod_init_module(void)
{
...
// find rcim board (look for RCIM II, RCIM I, and finally RCIM I old rev)
        dev = pci_find_device(PCI_VENDOR_ID_CONCURRENT, PCI_DEVICE_ID_RCIM_II,dev);
        if (dev == NULL) {                                  //try another id
                 dev = pci_find_device(PCI_VENDOR_ID_CONCURRENT_OLD, PCI_DEVICE_ID_RCIM, dev);
        }
        if (dev == NULL) {                                  //try another id
                 dev = pci_find_device(PCI_VENDOR_ID_CONCURRENT_OLD, PCI_DEVICE_ID_RCIM_OLD, dev);
        }
        if (dev == NULL) {             //no rcim board, just clean up and exit
                unregister_chrdev(major_num,"sample_mod");
                return -ENODEV;
        }

...

      if ((bd_regs = ioremap_nocache(plx_mem_base, plx_mem_size)) == NULL)
           return -ENOMEM;
...

      if ((bd_rcim_regs = ioremap_nocache(rcim_mem_base, rcim_mem_size)) == NULL)
           return -ENOMEM;
...
      sample_mod_irq = dev->irq;
      res = request_irq(sample_mod_irq, rcim_intr, SA_SHIRQ, "sample_mod", &rtc_info);
```

The complete initialization of the rtc 0 device is performed in the module’s “open” method. For this example, the device is automatically set up so that interrupts will be generated by the device. When the device is opened, interrupts associated with rtc 0 are enabled, the device is programmed to count from 10000 to 0 with a resolution of 1 microsecond, and the clock starts counting. It generates an interrupt when the count reaches 0.

```
******************************************************************************
int rcim_rtc_open(struct inode *inode, struct file *filep)
{
      u_int32_t val;

      if (rtc_info.nopens > 0) {
            printk(KERN_ERR "You can only open the device once.\n");
            return -ENXIO;
      }
      rtc_info.nopens++;
      if (!rtc_info.flag)
            return -ENXIO;
      writel(0, &bd_rcim_regs->request);
      writel(ALL_INT_MASK, &bd_rcim_regs->clear);
      writel(RCIM_REG_RTC0, &bd_rcim_regs->arm);
      writel(RCIM_REG_RTC0, &bd_rcim_regs->enable);
      writel(RTC_TESTVAL, &bd_rcim_regs->rtc0_timer);//rtc data reg
      val = RCIM_RTC_1MICRO | RCIM_RTC_START|RCIM_RTC_REPEAT;
      writel(val, &bd_rcim_regs->rtc0_control);
      return 0;
}
*******************************************************************************
```

The user-level driver must specify which signal should be sent when the kernel-level driver receives an interrupt. The user-level driver makes an **ioctl()** call, which is handled by the kernel-level driver’s ioctl method. When the user-level driver calls this **ioctl()** function, it indicates to the kernel-level driver that the user-level process has already set up a signal handler for the specified signal and the user-level driver is now ready to receive a signal.

The calling user-space process specifies the signal number it wishes to receive from the module. The driver remembers the process ID associated with the requested signal number by using the “current” structure. The “signal/process id” pair is stored in the module’s `rtc_info` structure and will later be used by the “notification” mechanism described below.

```
***************************************************************************
int rcim_rtc_ioctl(struct inode *inode, struct file *filep, unsigned int cmd,
unsigned long arg)
{
      if (!rtc_info.flag)
            return (-ENXIO);

      switch (cmd)
      {
            // Attach signal to the specified rtc interrupt
            case RCIM_ATTACH_SIGNAL:
                        rtc_info.signal_num = (int)arg;
                        rtc_info.signal_pid = current->tgid;
                        break;

            default:
                        return (-EINVAL);
      }
      return (0);
}
*****************************************************************************
```

The actual notification is implemented in the module's interrupt handler. When an interrupt is received from rtc 0, this interrupt service routine determines whether to send a signal to a process that has requested it. If there is a registered "process id/signal number" pair in the `rtc_info` structure, the specified signal is sent to the corresponding process using the function `kill_proc()`.

```
*******************************************************************************
int rcim_intr(int irq, void *dev_id, struct pt_regs *regs)
{
      u_int32_t isr;

      isr = readl(&bd_rcim_regs->request);
      writel(0, &bd_rcim_regs->request);
      writel(ALL_INT_MASK, &bd_rcim_regs->clear);

/* Use isr to determine whether the interrupt was generated by rtc 0 only if
   "rcim" module is not built into the kernel. If "rcim" is active, its
   interrupt handler would have cleared "request" register by the time we
   get here. */

//    if (isr & RCIM_REG_RTC0) {
            // Send signal to user process if requested
            if (rtc_info.signal_num && rtc_info.signal_pid &&
                  (kill_proc(rtc_info.signal_pid, rtc_info.signal_num, 1) == -ESRCH))
            {
                  rtc_info.signal_pid = 0;
            }
//    }

      return IRQ_HANDLED;
}
*******************************************************************************
```

When the device is closed, rtc 0 is shut down. The count value is reset to 0 and the clock is stopped. The interrupt/signal attachment is cleared so that no further signal will be sent if further interrupts are received.

```
*******************************************************************************
int rcim_rtc_close(struct inode *inode,struct file *filep)
{
        if (!rtc_info.flag)
                return (-ENXIO);
        rtc_info.nopens--;
        if(rtc_info.nopens == 0) {
                writel(~RCIM_RTC_START, &bd_rcim_regs->rtc0_control);
                writel(0, &bd_rcim_regs->rtc0_timer);
                rtc_info.signal_num = 0;
                rtc_info.signal_pid = 0;
        }
        return 0;
}
*******************************************************************************
```

Testing the Driver

The best way to test the sample kernel module is to build the kernel without the RCIM driver and then load the sample driver. However, this module is designed to work with or without the RCIM driver already built into the kernel.

The RCIM kernel module and the sample kernel module share the same interrupt line. When an interrupt occurs, RCIM's interrupt handler is invoked first and the hardware interrupt register on the RCIM board is cleared. Then the sample module's interrupt handler is invoked.

If both modules are loaded, the second handler will find the interrupt register cleared and if a check for "interrupt source" is performed, the handler will assume that the interrupt came from a device different from rtc 0. To overcome this obstacle, the following line in the sample module's interrupt handler has been commented out when both RCIM and the sample module are loaded:

```
//       if (isr & RCIM_REG_RTC0) { .
```

The code that follows is a simple user-level program which demonstrates how a user-level driver would attach a routine such that this routine is called whenever the RCIM skeletal driver's interrupt fires. The routine "interrupt_handler" is the routine which is called when the RCIM's rtc 0 interrupt fires. This program is terminated by typing Ctrl-C at the terminal where the program is run. Note that this sample code is also available in **`/usr/share/doc/ccur/examples/driver/usersample`**.

In order to load the sample module and successfully run the user sample program, all applications that use the RCIM driver should be aborted.

Below is the **`usersample`** program.

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

#include "sample_mod.h"

static const char *devname = "/dev/sample_mod";
static int nr_interrupts = 0;
static int quit = 0;

void interrupt_handler (int signum)
{
      nr_interrupts++;

      if ((nr_interrupts % 100) == 0) {
            printf (".");
            fflush(stdout);
      }
      if ((nr_interrupts % 1000) == 0)
            printf (" %d interrupts\n", nr_interrupts);
}

void ctrl_c_handler (int signum)
{
      quit++;
}

int main()
{
      int fd;
      struct sigaction intr_sig = { .sa_handler = interrupt_handler };
      struct sigaction ctrl_c_sig = { .sa_handler = ctrl_c_handler };

      sigaction (SIGUSR1, &intr_sig, NULL);
      sigaction (SIGINT, &ctrl_c_sig, NULL);

      if ((fd = open (devname, O_RDWR)) == -1 ) {
            perror ("open");
            exit(1);
      }

      if (ioctl (fd, RCIM_ATTACH_SIGNAL, SIGUSR1) == -1) {
            perror ("ioctl");
            exit(1);
      }

      printf ("waiting for signals...\n");
      while (! quit)
            pause();

      printf ("\nhandled %d interrupts\n", nr_interrupts);
      close(fd);
      exit(0);
}
```

Developing Kernel-level Device Drivers

The sections that follow describe particulars of the RedHawk Linux operating system that affect writing and testing kernel-level device drivers under RedHawk Linux.

Building Driver Modules

Instructions for building driver modules for use with either a pre-existing kernel supplied by Concurrent or a custom kernel are provided in Chapter 10, Configuring and Building the Kernel.

Kernel Virtual Address Space

There are some cases when the amount of kernel virtual address space reserved for dynamic mappings of the kernel support routines **`vmalloc()`** and **`ioremap()`** is not enough to accommodate the requirements of a device. The default value, 128 MB, is enough for all systems except those with I/O boards that have very large onboard memories which are to be ioremap'ed. An example is the VMIC reflective memory board installed on an iHawk system when it is populated with 128 MB of memory.

When 128 MB of reserved kernel virtual address space is not enough, this value can be increased via the VMALLOC_RESERVE tunable, which is located under General Setup on the Kernel Configuration GUI.

Real-Time Performance Issues

A kernel-level 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. The sections that follow provide 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 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. RedHawk 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 (Bottom Halves)" 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 (Bottom Halves)

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, called bottom halves, 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 choices for deferring interrupts: softirqs, tasklets and work queues. Tasklets are built on softirqs and therefore they are similar in how they operate. Work queues operate differently and are built on kernel threads. The decision over which bottom half to use is important. Table 14-1 summarizes the types, which are explained at length in the sections below.

Table 14-1 Types of Bottom Halves

Bottom Half Type	Context	Serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queues	Process	None (scheduled as process context)

Softirqs and Tasklets

Two mechanisms for deferring interrupt processing have different requirements in terms of whether or not the code that is deferred must be reentrant or not. These types of deferrable functions are softirqs and tasklets. 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. 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.

In standard Linux, softirqs and tasklets are usually executed from interrupt context immediately after interrupt handlers transition from interrupt to program level. Occasionally, standard Linux will defer softirq and tasklets to a kernel daemon. Both methods allow softirqs and tasklets to execute with interrupts enabled; however, because they are usually executed from interrupt context, softirqs and tasklets cannot sleep.

RedHawk has been enhanced with an option (that is on by default) to guarantee that softirqs and tasklets are only executed in the context of a kernel daemon. 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 SOFTIRQ_PRI kernel tunable, which is located under General Setup on the Kernel Configuration GUI. 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.

Work Queues

Work queues are another deferred execution mechanism. Unlike softirqs and tasklets, standard Linux always processes work queues in the process context of kernel daemons and therefore the code in a work queue is allowed to sleep.

The kernel daemons that process work queues are called worker threads. Worker threads are always created as a gang of threads, one per CPU, with each thread bound to a single CPU. Work on the work queue is maintained per CPU and is processed by the worker thread on that CPU.

The kernel provides a default work queue that drivers may use. The worker threads that process the default work queue are called **`events/`***cpu*, where *cpu* is the CPU that the thread is bound to.

Optionally, drivers may create private work queues and worker threads. This is advantageous to the driver if the queued work is processor-intensive or performance critical. It also lightens the load on the default worker threads and prevents starving the rest of the work on the default work queue.

Worker threads execute on a CPU when work is placed on the work queue. Therefore, if a hard interrupt has its affinity set to a specific CPU, and the interrupt handler queues work, the corresponding worker thread will also run on that CPU. Worker threads are always created with a nice value of -10 but their priority may be modified with the **`run(1)`** command.

Understanding Priorities

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 from being 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 them 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 and/or interrupts to be disabled. In either case, the worst case process dispatch latency for a process executing on the CPU where these features are disabled is directly impacted by how long they 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 associated with a device driver. If a device driver is multi-threaded, then it is not necessary to lock the BKL before calling **`ioctl`**. RedHawk Linux allows a device driver to specify that the BKL should not be locked before calling **`ioctl`**. 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 the BKL is not locked. Otherwise, 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.

Analyzing Performance

NightTrace, a graphical analysis tool supplied by Concurrent, allows you to graphically display information about important events in your application and in the kernel, and can be used for identifying patterns and anomalies in your application's behavior. The ability to interactively analyze the code under varying conditions is invaluable toward fine-tuning the real-time performance of your device driver.

The process of supplying trace points in user-level code, capturing trace data and displaying the results is fully described in the *NightTrace User's Guide*, publication number 0890398. User and kernel trace events can be logged and displayed for analysis.

Kernel tracing utilizes pre-defined kernel trace events included in the RedHawk Linux trace and debug kernels. User-defined events can be logged using the pre-defined CUSTOM trace event or created dynamically. All are displayed by NightTrace for analysis. Refer to Appendix D for details about kernel trace events.

15
PCI-to-VME Support

15
PCI-to-VME Support

This chapter describes RedHawk Linux support for a PCI-to-VMEbus bridge.

Overview

A PCI-to-VMEbus adapter can be used to connect the iHawk PCI-based system with a VMEbus system. This allows transparent access to all VME memory space and interrupt levels to control and respond to the VME card as though it were plugged directly into the iHawk PCI backplane.

RedHawk Linux includes support for the Model 618-3 PCI-to-VMEbus adapter from SBS Technologies. Using the adapter, memory is shared between the two systems. Two methods are utilized: memory mapping and Direct Memory Access (DMA). Memory mapping supports bi-directional random access bus mastering from either system. This allows programmed I/O access to VMEbus RAM, dual-port memory and VMEbus I/O. On each system, a bus master can access memory in the other system from a window in its own address space. Mapping registers allow PCI devices to access up to 32 MB of VMEbus address space and VMEbus devices to access up to 16 MB of PCI space.

Two DMA techniques are supported: Controller Mode DMA and Slave Mode DMA. Controller mode DMA provides high-speed data transfers from one system's memory directly into the other system's memory. Data transfers can be initiated in both directions by either processor at rates up to 35 MB per second and up to 16 MB per transfer.

VMEbus devices that have their own DMA controllers can use Slave Mode DMA instead of Controller Mode DMA. This allows a VMEbus DMA device to transfer data directly into PCI memory at data rates in excess of 15 MB per second.

The Model 618-3 adapter consists of three parts: the PCI adapter card, the VMEbus adapter card and a fiber optic cable.

The PCI adapter card self-configures at boot time. It responds to and generates A32 memory and I/O accesses and supports D32, D16 and D8 data widths.

The VMEbus adapter card is configured via jumpers. The VMEbus adapter card responds to and generates A32, A24, and A16 accesses and supports D32, D16, and D8 data widths.

Software support for the adapter includes the SBS Linux Model 1003 PCI Adapter Support Software Version 2.2, with modifications for execution and optimization under RedHawk Linux. The software includes a device driver that can access dual-port and/or remote memory space from an application, and example programs to help applications programmers with adapter and system configuration.

Documentation

This chapter provides the information you will need to configure and use this support under RedHawk Linux.

For information beyond the scope of this chapter, refer to the following documents that are included with the RedHawk Linux documentation:

- *SBS Technologies Model 618-3, 618-9U & 620-3 Adapters Hardware Manual* (**sbs_hardware.pdf**)
- *SBS Technologies 946 Solaris, 965 IRIX 6.5, 983 Windows NT/2000, 993 VxWorks & 1003 Linux Support Software Manual* (**sbs_software.pdf**)

Installing the Hardware

The Model 618-3 adapter consists of three parts: the PCI adapter card, the VMEbus adapter card and a fiber optic cable. Instructions for installing these are given below.

Normally, installation and configuration of the hardware is done by Concurrent Computer Corporation. This information is provided for those cases where a PCI-to-VME bridge is added to a system in a post-manufacturing environment.

Unpacking

When unpacking the equipment from the shipping container, refer to the packing list and verify that all items are present. Save the packing material for storing and reshipping the equipment.

NOTE

If the shipping container is damaged upon receipt, request that the carrier's agent be present during unpacking and inspection of the equipment.

Before attempting to install the cards in your system, read the following:

CAUTION

Avoid touching areas of integrated circuitry as static discharge can damage circuits.

Concurrent Computer Corporation strongly recommends that you use an antistatic wrist strap and a conductive foam pad when installing and removing printed circuit boards.

Configuring the Adapter Cards

There are no jumpers to configure on the PCI adapter card.

VME adapter card jumper configuration should take place before the VME adapter card is installed, or when the current settings of the VMEbus attributes that are controlled by the VME adapter card jumpers need to be changed.

Refer to Chapter 10 of the SBS Technologies Hardware Manual for information about configuring the VMEbus adapter card. The following additional information may prove useful:

- The System Jumpers must be set appropriately, based on whether this VME adapter card is used as the system controller in slot 1, or as a non-system controller in some other VME slot.
- To make use of the bt_bind() buffer support or the local memory device support (BT_DEV_LM) that lets devices on the VMEbus access memory on the iHawk system through VME slave windows, the Remote REM-RAM HI and LO jumpers must be set up to indicate the VMEbus base address and range of the VME slave windows out on the VMEbus.

  The base address should be placed on a 16 MB boundary, and the size of this area should typically be set to (but not exceed) 16 MB in order to make use of the total amount of area supported by the SBS hardware; for example, to set up an A32 address range of 0xC0000000 to 0xC1000000, the jumpers should be configured to the settings below:

  To set an A32 address range, the jumpers at the bottom of the REM-RAM should be set to:

  A32 jumper IN
  A24 jumper OUT

  To specify a starting address of 0xC0000000, the row of LO address REM-RAM jumpers should be set to:

  31 and 30 jumpers OUT
  All other LO jumpers IN (29 through 16)

  To specify an ending address of 0xC1000000, the row of HI address REM-RAM jumpers should be set to:

  31, 30 and 24 jumpers OUT
  All other HI jumpers IN (29-25, and 23-16)

Installing the PCI Adapter Card

Use the following procedure to install the PCI adapter in your iHawk system:

1. Ensure that the iHawk system is powered down.
2. Locate a vacant PCI card slot in the chassis that supports a bus master.
3. Remove the metal plate that covers the cable exit at the rear of the chassis.
4. Insert the PCI adapter card into the connector.
5. Fasten the adapter card in place with the mounting screw.
6. Replace the cover.

Installing the VMEbus Adapter Card

NOTE

VMEbus backplanes have jumpers to connect the daisy-chained, bus grant and interrupt acknowledge signals around unused card locations. Make sure these jumpers are removed from the slot in which the adapter card will be installed.

1. Ensure that the VMEbus chassis is powered down.
2. Decide whether the VMEbus adapter card is the system controller. If the VMEbus adapter card is the system controller, it must be installed in slot 1.

   If the adapter card is not the system controller, locate an unoccupied 6U slot in the VMEbus card cage for the adapter.
3. Insert the card into the connector of the selected slot.

Connecting the Adapter Cable

NOTE

Keep the ends of the fiber-optic cable clean. Use alcohol-based fiber-optic wipes to remove minor contaminants such as dust and dirt.

Fiber-optic cables are made of glass: therefore, they may break if crushed or bent in a loop with less than a 2-inch radius.

1. Ensure that the iHawk computer system and the VMEbus chassis are powered off.
2. Remove the rubber boots on the fiber-optic transceivers as well as the ones on the fiber-optic cables. Be sure to replace these boots when cables are not in use.

3. Plug one end of the fiber-optic cable into the PCI adapter card's transceiver.
4. Plug the other end of the fiber-optic cable into the VMEbus adapter card's transceiver.
5. Turn power on to both PCI and VMEbus systems.
6. Ensure that the READY LEDs on both adapter cards are lit. They must be on for the adapter to operate.

Installing the Software

The software is contained on an optional product CD delivered with RedHawk Linux. It is installed using the **install-sbsvme** installation script.

To install the software, perform the following steps:

1. With RedHawk Linux Version 2.1 or later running on the iHawk system, log in as root and take the system down to single-user mode:
   a. Right click on the desktop and select New Terminal.
   b. At the system prompt, type **init 1**.
2. Locate the disc labeled "RedHawk Linux PCI-to-VME Bridge Software Library" and insert it into the CD-ROM drive.
3. To mount the cdrom device, execute the following command:

   ```
   mount /mnt/cdrom
   ```
4. To install, execute the following commands:

   ```
   cd  /mnt/cdrom
   ./install-sbsvme
   ```

   Follow the on-screen instructions until the installation script completes.
5. When the installation completes, execute the following commands:

   ```
   cd  /
   umount /mnt/cdrom
   eject
   ```
6. Remove the disc from the CD-ROM drive and store. Exit single-user mode (Ctrl-D).

Configuration

The sections below discuss configuration of the module under RedHawk Linux and other attributes that can be established at system initialization.

The btp Module

The pre-defined RedHawk kernels have the SBS Technologies Model 618-3 PCI-to-VMEbus bridge configured as a module by default. This can be disabled if desired through the SBSVME option under SBS VMEbus-to-PCI Support on the Kernel Configuration GUI. The module is called "btp."

Device Files and Module Parameter Specifications

The **/dev/btp*** device files are created at initialization via **/etc/init.d/sbsvme**. The attributes for those files are defined in **/etc/sysconfig/sbsvme**. In addition, the following module parameter specifications can be made in this file. The default is no parameters.

`btp_major=`*num*	Specifies the major device number (*num*). By default, it is 0 (zero) which allows the kernel to make the selection. If you supply a nonzero device number, it must not already be in use. The **/proc/devices** file can be examined to determine which devices are currently in use.
`icbr_q_size=`*size*	Specifies the number of ICBR entries (*size*) to be allocated for the interrupt queue. Once set, this value cannot be changed without unloading and reloading the btp driver. The default value is 1 KB of interrupt queue space.
`lm_size=`*size1*, *size2*, ...	Specifies an array of local memory (BT_DEV_LM) sizes in bytes with one for each SBS PCI-to-VME controller (unit) present in the system. If this value is set to 0 (zero), local memory is disabled for that specific unit only. The default value is 64 KB of local memory and the maximum value is 4 MB. Refer to the "Local Memory" section of this chapter for more information.
`trace=`*flag_bits*	Specifies the device driver tracing level. This is used to control which trace messages the btp driver displays. The possible bits to use are the BT_TRC_*xxx* values located in **/usr/include/btp/btngpci.h**. Because tracing has an impact on performance, this feature should be used only for debugging btp driver problems. The default value is 0 (zero) for no trace messages.

The following are examples of btp module parameter specifications:

```
BTP_MOD_PARAMS='bt_major=200 trace=0xff lm_size=0'
BTP_MOD_PARAMS='icbr_q_size=0x1000 lm_size=0x8000,0x4000'
```

VMEbus Mappings

Support for automatically creating and removing PCI-to-VMEbus mappings is included in the **/etc/init.d/sbsvme** initialization script. When mappings are defined in **/etc/sysconfig/sbsvme-mappings**, they are created during "/etc/init.d/sbsvme start" processing and removed during the "stop" processing.

The **/etc/sysconfig/sbsvme-mappings** file contains help information and commented-out templates for creating VMEbus mappings. The template examples can be used to create customized VMEbus mappings, if desired. The mappings are created by writing values to the **/proc/driver/btp/***unit***/vme-mappings** file, which is explained as comments within the **sbsvme-mappings** file and in the section "The /proc File System Interface" later in this chapter.

By making use of the **sbsvme-mappings** file to create PCI-to-VMEbus mappings during system initialization, you may place additional lines in the **/etc/rc.d/ rc.local** script to invoke **shmconfig(1)** to create globally-visible shared memory areas that are bound to VMEbus space. A sample script is provided that illustrates this. Refer to the "Example Applications" section for details.

User Interface

Some modifications to the standard support software have been made for RedHawk Linux. In addition to installation modifications, the following have been added:

- Support for binding multiple buffers of various sizes. In a system with multiple user-level device drivers, this capability allows each driver to allocate its own bind buffer instead of having to share a single bind buffer between multiple devices. This capability also means that by allocating multiple large bind buffers, the total 16 MB area of hardware-supported VMEbus slave window space may be utilized. See the "Bind Buffer Implementation" section for more information. Example programs have been added that demonstrate how to allocate and bind multiple buffers to VMEbus space (see the "Example Applications" section).

- Support for creating and removing VMEbus space mappings that are not associated with a specific process, and obtaining the starting PCI bus address location of that mapping to allow shared memory binding. This can be accomplished in one of two ways:

  - using the bt_hw_map_vme/bt_hw_unmap_vme library functions
  - writing to the **/proc/driver/btp** file system

  See the "Mapping and Binding to VMEbus Space" section for more details. Example programs demonstrate how to create, display and remove VMEbus mappings using both methods (see the "Example Applications" section).

API Functions

Table 15-1 lists the API functions included in the **`libbtp`** library. The functions that have been modified or added to RedHawk Linux are noted and described in the sections that follow. The remaining functions are described in the SBS Technologies Software Manual included with the RedHawk Linux documentation.

Table 15-1 PCI-to-VME Library Functions

Function	Description
bt_str2dev	Convert from string to logical device.
bt_gen_name	Generate the device name.
bp_open	Open a logical device for access.
bt_close	Close the logical device.
bt_chkerr	Check for errors on a unit.
bt_clrerr	Clear errors on a unit.
bt_perror	Print error message to stderr.
bt_strerror	Create a string error message.
bt_init	Initialize a unit.
bt_read	Read data from a logical device.
bt_write	Write data to a logical device.
bt_get_info	Get device configuration settings. **See Note 1 below**.
bt_set_info	Set device configuration settings. **See Note 1 below**.
bt_icbr_install	Install an interrupt call back routine.
bt_icbr_remove	Remove an interrupt call back routine.
bt_lock	Lock a unit.
bt_unlock	Unlock a previously locked unit.
bt_mmap	Create a memory mapped pointer into a logical device.
bt_unmmap	Unmap a memory mapped location.
bt_dev2str	Convert from a logical device type to a string.
bt_ctrl	Call directly into the driver I/O control function.
bt_bind	Bind application supplied buffers. **See Note 1 below.**
bt_unbind	Unbind bound buffers. **See Note 1 below.**
bt_reg2str	Convert register to string.
bt_cas	Compare and swap atomic transactions.

(continued on next page)

Notes:

1 Multiple buffers of various sizes are supported under RedHawk through these functions; see the "Bind Buffer Implementation "section.

2 PCI-to-VME mapping/binding support is unique to RedHawk; see the "Mapping and Binding to VMEbus Space"section.

Table 15-1 PCI-to-VME Library Functions (Continued)

Function	Description
bt_tas	Test and set atomic transaction.
bt_get_io	Read an adapter CSR register.
bt_put_io	Write an adapter CSR register.
bt_or_io	One shot a register.
bt_reset	Remotely reset the VMEbus.
bt_send_irq	Send an interrupt to the remote VMEbus.
bt_status	Return device status.
bt_hw_map_vme	Create a PCI-to-VMEbus mapping. **See Note 2 below.**
bt_hw_unmap_vme	Remove a PCI-to-VMEbus mapping. **See Note 2 below.**

Notes:
1 Multiple buffers of various sizes are supported under RedHawk through these functions; see the “Bind Buffer Implementation “section.
2 PCI-to-VME mapping/binding support is unique to RedHawk; see the “Mapping and Binding to VMEbus Space”section.

Bind Buffer Implementation

The RedHawk sbsvme bind buffer support allows for multiple, different sized kernel bind buffers to be allocated, bt_mmap()ed and bt_bound() to VMEbus space at the same time. This section provides information about this bind buffer support, including how this support differs from the documentation on bind buffers in the SBS Technologies Software Manual.

Note that the only user interface difference between the SBS documentation and the RedHawk bind buffer implementation is in the use of the ‘value’ parameter on the bt_set_info() BT_INFO_KFREE_BUF call, which is discussed below. All other user interfaces are the same as shown in the SBS Technologies Software Manual.

bt_get_info BT_INFO_KMALLOC_BUF

Synopsis

```
bt_error_t bt_get_info(bt_desc_t btd, BT_INFO_KMALLOC_BUF,
     bt_devdata_t *value_p)
```

Multiple bt_get_info() BT_INFO_KMALLOC_BUF command calls can be made to allocate multiple kernel buffers, where each returned buffer address, which is stored at the value_p parameter location, may then be used on subsequent bt_mmap() and bt_bind() calls in order to mmap and bind this buffer to a location on the VMEbus.

BT_INFO_KMALLOC_BUF calls allocate a kernel bind buffer with a size equal to the last value set on the last successful bt_set_info() BT_INFO_KMALLOC_SIZ call. (If no such calls have been made when the BT_INFO_KMALLOC_BUF call is made, then the default size of 64 KB is used.)

Up to BT_KMALLOC_NBUFS (16) kernel buffers can be allocated at the same time with the BT_INFO_KMALLOC_BUF command. If there are already 16 bind buffers allocated, this BT_INFO_KMALLOC_BUF call fails and returns an error value of BT_EINVAL.

Note that if a bt_set_info() BT_INFO_KMALLOC_SIZ call is used to set the bind buffer size to zero, all subsequent BT_INFO_KMALLOC_BUF calls return with an error value of BT_EINVAL until a new bind buffer size is set to a non-zero value via a bt_set_info() BT_INFO_KMALLOC_SIZ call.

If the kernel is unable to allocate enough space for a new kernel bind buffer, this BT_INFO_KMALLOC_BUF call fails and returns an error value of BT_EINVAL.

bt_set_info BT_INFO_KMALLOC_SIZ

Synopsis

```
bt_error_t bt_set_info(bt_desc_t btd, BT_INFO_KMALLOC_SIZ,
     bt_devdata_t value)
```

When the bt_set_info() BT_INFO_KMALLOC_SIZ command is used to set a new bind buffer size, the command only affects future bt_get_info() BT_INFO_KMALLOC_BUF command calls. Any kernel bind buffers that have already been allocated with different bind buffer sizes are NOT affected by the new BT_INFO_KMALLOC_SIZ call.

In this way, different sized kernel bind buffers can be allocated by using a different BT_INFO_KMALLOC_SIZ 'value' parameter after making one or more bt_get_info() BT_INFO_KMALLOC_BUF calls.

It is encouraged, but not required, to use bind buffer sizes for the 'value' parameter that are a power of 2. Since the kernel bind buffer allocation is rounded up to a power of 2, specifying and using a power of 2 'value' parameter value eliminates unused sections of the allocated kernel bind buffers. Note that the initial default value for the kernel bind buffer size is 64 KB.

Typically, the maximum size kernel bind buffer that can be successfully allocated on a subsequent bt_get_info() BT_INFO_KMALLOC_BUF call is 4 MB. However, depending upon the amount of physical memory on the system and the other uses of system memory, it may not always be possible to successfully allocate a 4 MB kernel bind buffer. In this case, multiples of smaller sized bind buffers can be allocated, or alternatively, 4 MB kernel bind buffers can be allocated before other uses of system memory use up the memory resources.

bt_set_info BT_INFO_KFREE_BUF

Synopsis

```
bt_error_t bt_set_info(bt_desc_t btd, BT_INFO_KFREE_BUF,
     bt_devdata_t value)
```

The interface for the bt_set_info() BT_INFO_KFREE_BUF command is slightly different under RedHawk than what is documented in the SBS Technologies Software Manual.

Specifically, the ’value’ parameter is not used in the SBS implementation but the RedHawk implementation uses this parameter in the following ways:

When the ’value’ parameter is zero:

> This call unbinds and frees *all* kernel bind buffers that are not currently bt_mmap()ed from user space. If at least one bind buffer is unbound and freed, a successful status (BT_SUCCESS) is returned.
>
> If no bind buffers are found that can be unbound and freed, this call fails and BT_EINVAL is returned to the caller.

When the ’value’ parameter is not equal to zero:

> This call is for unbinding and freeing up just one specific kernel bind buffer. In this case, the caller’s ’value’ parameter should be equal to the kernel buffer address that was returned at the ’value_p’ parameter location on the previous bt_get_info() BT_INFO_KMALLOC_BUF call.
>
> If the buffer address specified in the ’value’ parameter on this call does not correspond to a valid kernel bind buffer, this call fails and returns an error value of BT_EINVAL.
>
> If the ’value’ parameter on this call corresponds to a valid kernel bind buffer, but that buffer is currently bt_mmap()ed from user space, this call fails and a value of BT_EFAIL is returned. In this case, the buffer must first be bt_unmmap()ed before this call can succeed.

Additional Bind Buffer Information

The following sections describe additional areas where bind buffer support is affected under RedHawk.

The Bigphysarea Patch

The bigphysarea patch discussed in the SBS Technologies Software Manual is not supported or needed in the RedHawk sbsvme btp device driver. By using multiple large bind buffers, it is possible to support the full 16MB of VMEbus slave window space for accessing iHawk memory from the VMEbus.

Unloading the btp Module

The sbsvme ’btp’ kernel module can not be unloaded while there are any kernel bind buffers currently bt_mmap()ed in a process’ address space. Processes must first remove their mappings to kernel bind buffers with bt_unmmap() call(s) before the kernel driver module is unloaded.

When there are no bind buffers currently bt_mmap()ed from user space, the btp kernel module can be unloaded with a “/etc/init.d/sbsvme stop” command, and any kernel bind buffers currently allocated are implicitly unbound (if currently bound) from the hardware VMEbus slave window area and freed up for other future kernel memory allocations.

bt_bind rem_addr_p Parameter

The ’rem_addr_p’ parameter on bt_bind() calls specifies an offset within the remote VMEbus slave window where the caller wishes to bind a kernel bind buffer. Note that this value is an offset, and not an absolute VMEbus physical address. This offset value is from the base VMEbus address defined by the REM-RAM LO jumper setting located on the SBS VME adapter card.

The user can either specify an actual ’rem_addr_p’ offset value, or let the btp driver find an appropriate bind address location by using the BT_BIND_NO_CARE value for the ’rem_addr_p’ parameter. When this value is used, upon successful return from the bt_bind() call the ’rem_addr_p’ memory location contains the offset value where the kernel btp driver bound the bind buffer.

As an example, if the REM-RAM LO jumper settings are set to a value of 0xC0000000 and the offset value is 0x10000, the actual bind address where this buffer can be accessed from the VMEbus would be 0xC0010000.

Local Memory

In addition to the kernel bind buffer support, the btp driver also supports the concept of local memory. This feature is made available through use of the BT_DEV_LM device type, instead of the BT_DEV_A32, BT_DEV_A24, and other VMEbus device types typically used for the bind buffer feature.

The local memory buffer consists of local iHawk memory that is allocated and bound to the VMEbus slave window area when the btp driver is loaded. This memory allocation and binding remains in effect as long as the btp driver is loaded. If the btp driver is unloaded with a “/etc/init.d/sbsvme stop” command, this local memory buffer is unbound from VMEbus space and freed up for other kernel uses.

The local memory buffer is always bound to the bottom area of the VMEbus slave window as defined by the REM-RAM LO jumper settings on the VME adapter card. For example, if the local memory size is 64 KB, and the REM-RAM LO jumper settings are set to a value of 0xC0000000, the local memory buffer is bound to the VMEbus at physical VMEbus addresses 0xC0000000 through 0xC0000FFF.

Note that since the local memory buffer always occupies the bottom area of the VMEbus remote slave window, the kernel bind buffers may not be bound to this area whenever local memory support is enabled. By default, the local memory support is enabled with a local memory buffer size of 64 KB, which leaves 16 MB - 64 KB of VMEbus slave window space for bind buffers (assuming that the REM-RAM LO jumper settings are set to a range that covers 16 MB).

The size of the local memory buffer can be increased by modifying the ’lm_size’ parameter in the **/etc/sysconfig/sbsvme** configuration file (see the “Configuration” section earlier in this chapter. Note that the maximum supported ’lm_size’ value is 4 MB. If a larger value is specified, the btp driver’s buffer allocation does not succeed, and the local memory feature is disabled at btp driver load time.

The local memory support can be disabled by setting the ’lm_size’ btp module parameter to zero. When set to zero, the btp driver does not allocate a local memory buffer, and the entire VMEbus slave window area is free for kernel bind buffer use.

The local memory support is very similar to the bind buffer support:

- Both local memory and bind buffers are accessible from the VMEbus through the slave window area.
- Both the local memory and bind buffer buffer areas can be accessed by specifying the appropriate device type when using the bt_read(), bt_write() and bt_mmap() functions.

The main differences between the local memory and bind buffer support are:

- There may be only one local memory buffer area. This buffer is set up at btp driver load time and remains allocated and bound until the btp driver is unloaded.

  Contrastingly, multiple bind buffers of different sizes can be dynamically allocated and bound, and dynamically unbound and freed.

- The local memory buffer always occupies the bottom of the VMEbus slave window area.

  Contrastingly, for bind buffers the user can either specify the location/offset where each bind buffer is to be bound to VMEbus space, or let the kernel dynamically find the next free location/offset to use.

Mapping and Binding to VMEbus Space

RedHawk provides a method of creating VMEbus space mappings that are not associated with a specific process and remain intact after the process that created the mapping exits. These mappings can be created and removed independently, either through the bt_hw_map_vme and bt_hw_unmap_vme library functions or by writing to a **`/proc`** file system interface.

The unique PCI bus starting address that corresponds to an active VMEbus space area mapping can be obtained and used with **`shmbind(2)`** or **`shmconfig(1)`** to bind this segment to a region of I/O space.

This functionality is described in the sections that follow.

bt_hw_map_vme

This function creates a new PCI-to-VMEbus mapping.

Synopsis

```
bt_error_t bt_hw_map_vme(bt_desc_t btd, void **phys_addr_p,
     bt_devaddr_t vme_addr, size_t map_len, bt_swap_t swapping)
```

Arguments

`btd`	the device descriptor that was returned from a successful bt_open() function call.
`phys_addr_p`	the user space location where the local PCI bus starting/base address for this mapping is returned

`vme_addr`	the starting/base target VMEbus physical address. This address must be aligned on a 4 KB boundary.
`map_len`	the size of hardware mapping to be created. This value is rounded up to a multiple of 4 KB.
`swapping`	the byte swapping method to use for hardware mapping. The BT_SWAP_*xxx* defines included in the **`/usr/include/btp/btngpci.h`** header file can be used.

Return Values

When successful, a value of BT_SUCCESS is returned. The PCI bus address returned at the `phys_addr_p` location can be used with **`shmbind(2)`** or **`shmconfig(1)`** to create a shared memory area that may be used to access this range of remote VMEbus addresses.

When unsuccessful, an appropriate `bt_error_t` value is returned indicating the reason for the failure:

BT_EDESC	An invalid `btd` descriptor was specified. The descriptor must be a descriptor returned from a bt_open() call of a BT_DEV_A32, BT_DEV_A24 or BT_DEV_A16 device type.
BT_EINVAL	An invalid `vme_addr`, `map_len`, `phys_addr_p` or `swapping` parameter was specified.
BT_ENXIO	The sbsvme hardware is not online or not connected properly.
BT_ENOMEM	The required number of sbsvme hardware mapping registers could not be allocated.
BT_ENOMEM	The memory for the kernel data structures that are used for tracking this mapping could not be allocated.

bt_hw_unmap_vme

This function removes a PCI-to-VMEbus mapping previously created with the bt_hw_map_vme function or by writing to the **`/proc/driver/btp/`*unit*`/vme-mappings`** file.

Synopsis

```
bt_error_t bt_hw_unmap_vme(bt_desc_t btd, void *phys_addr)
```

Parameters

`btd`	the device descriptor that was returned from a successful bt_open() function call.
`phys_addr`	the PCI bus starting address for the VMEbus mapping to be removed

Return Values

When successful, a value of BT_SUCCESS is returned.

When unsuccessful, an appropriate `bt_error_t` value is returned indicating the reason for the failure:

BT_EDESC	An invalid `btd` descriptor was specified. The descriptor must be a descriptor that was returned from a bt_open() call of a BT_DEV_A32, BT_DEV_A24 or BT_DEV_A16 device type.
BT_ENOT_FOUND	The mapping specified by the `phys_addr` parameter does not exist.

The /proc File System Interface

When the sbsvme btp kernel module is loaded, the following **`/proc`** file(s) are created:

`/proc/driver/btp/`*unit*`/vme-mappings`

where *unit* is the unit number of the sbsvme PCI bridge card. The first card is unit number 0. On systems with multiple bridges, the second card is unit number 1, etc.

Existing PCI-to-VMEbus mappings can be viewed by reading the file. Mappings can be created and removed by writing to the file. These techniques are described below.

Displaying VMEbus Mappings

Reading the **`vme-mappings`** file using **`cat(1)`** displays all currently established VMEbus mappings. The following output shows two PCI-to-VMEbus mappings:

```
$ cat /proc/driver/btp/0/vme-mappings
pci=0xf8019000 vme=0x00008000 size=0x0001000 space=A16 admod=0x2d swap=5
pci=0xf8011000 vme=0x00fe0000 size=0x0008000 space=A24 admod=0x39 swap=0
```

`pci=`	indicates the local PCI bus address where the mapping begins
`vme=`	indicates the starting VMEbus address
`size=`	indicates the size/length of the mapping
`space=`	indicates the VMEbus address space type for the mapping
`admod=`	indicates the VMEbus address modifier described by the BT_AMOD_*xxx* defines in **`/usr/include/btp/btdef.h`**.
`swap=`	indicates the bit swapping method described by the BT_SWAP_*xxx* defines in **`/usr/include/btp/btngpci.h`**.

Creating VMEbus Mappings

Mappings to VMEbus space can be created by writing to the **`vme-mappings`** file. Note that you must have CAP_SYS_ADMIN privileges to write to this file. To create a mapping, the following three parameters must be specified in the order given here:

`vme=`	specifies the starting, page-aligned VMEbus address to be mapped (e.g., 0xfffff000).

`size=` specifies the size of the mapping, which should be a multiple of a page (e.g., 0x1000). Note that the sbsvme hardware is limited to mapping a total of 32 MB of VMEbus space.

`space=` specifies the VMEbus address space type for the mapping: A32, A24 or A16.

The following optional parameters may also be supplied, in any order, following the required parameters listed above:

`admod=` specifies the VMEbus address modifier described by the BT_AMOD_*xxx* defines in **`/usr/include/btp/btdef.h`**. If not specified, the following default values are used:

BT_AMOD_32	0x0d
BT_AMOD_24	0x3d
BT_AMOD_16	0x2d

`swap=` specifies the bit swapping method described by the BT_SWAP_*xxx* defines in **`/usr/include/btp/btngpci.h`**. If not specified, the default BT_SWAP_DEFAULT value is used.

The following example shows creating two VMEbus mappings by writing to the **`vme-mappings`** file.

```
$ echo "vme=0xe1000000 size=0x10000 space=A32" > /proc/driver/btp/0/vme-mappings
$ echo "vme=0xc0000000 size=0x1000 space=A32 swap=7 admod=0x9" > /proc/driver/btp/0/vme-mappings
```

Note that when the sbsvme btp kernel driver is unloaded with "/etc/init.d/sbsvme stop" (see "VMEbus Mappings"), all current VMEbus mappings are removed before the driver is unloaded. If mappings exist and "modprobe -r btp" is used to unload the driver, the unload will fail until all VMEbus mappings are removed.

Removing VMEbus Mappings

A mapping to VMEbus space can be removed by writing the local PCI bus location of the mapping to the **`vme-mappings`** file. Note that you must have CAP_SYS_ADMIN privileges to write to this file. The PCI bus location is returned by bt_hw_map_vme() and by **`cat`**'ing the **`vme-mappings`** file. For example:

```
$ cat /proc/driver/btp/0/vme-mappings
pci=0xf8019000 vme=0x00008000 size=0x0001000 space=A16 admod=0x2d swap=5
pci=0xf8011000 vme=0x00fe0000 size=0x0008000 space=A24 admod=0x39 swap=0

$ echo "pci=0xf8019000" > /proc/driver/btp/0/vme-mappings

$ cat /proc/driver/btp/0/vme-mappings
pci=0xf8011000 vme=0x00fe0000 size=0x0008000 space=A24 admod=0x39 swap=0
```

Example Applications

Example programs are supplied that demonstrate features of the sbsvme btp device driver and facilitate its use. They can be found in **`/usr/share/doc/ccur/examples/sbsvme`**. The programs are useful tools for:

- debugging
- uploading and downloading binary data
- receiving and counting programmed interrupts
- testing hardware
- creating VMEbus mappings and bindings to shared memory areas

Table 15-2 lists the example programs. An asterisk (*) indicates the program was added to RedHawk Linux and is described in the following sections. Other programs are described in the SBS Technologies Software Manual.

Table 15-2 PCI-to-VME Example Programs

Name	Description	Functions Used

bt_bind	Binds a local buffer to the remote VMEbus, waits for user input, and then prints the first 256 bytes of the bound buffer.	bt_bind()
		bt_unbind()

bt_bind_mult *	Shows how to bind multiple local buffers to the remote VMEbus. Optionally writes values to the local buffers before waiting for user input. After user input occurs, it prints out the first 16 bytes of each page of each of the local buffers.	bt_bind()
		bt_unbind()

bt_bind_multsz *	Shows how to create multiple bind buffers with different sizes.	bt_bind()
		bt_unbind()

bt_cat	Similar to the 'cat' program. Allows reading from the remote VMEbus to stdout, or writing data to the remote VMEbus from stdin.	bt_read()
		bt_write()

bt_datachk	Reads and writes from a device using a specific pattern and then verifies that no data or status errors occurred.	bt_read()
		bt_write()

bt_dumpmem	Reads and prints to stdout 256 bytes of remote VMEbus data.	bt_mmap()
bt_getinfo	A script that gets all the driver parameters and displays their values to stdout.	n/a
bt_hwmap *	Creates a VMEbus mapping.	bt_hw_map_vme()
bt_hwunmap *	Removes a VMEbus mapping.	bt_hw_unmap_vme()

bt_icbr	Registers for and receives interrupts for a given interrupt type.	bt_icbr_install()
		bt_icbr_remove()

bt_info	Gets or sets driver parameters.	bt_get_info()
		bt_set_info()

bt_readmem	Reads and prints to stdout 256 bytes of remote VMEbus data.	bt_read()
bt_reset	Resets the remote VMEbus.	bt_reset()

(continued on next page)

Table 15-2 PCI-to-VME Example Programs (Continued)

Name	Description	Functions Used
bt_revs	Outputs the software driver version and hardware firmware version information to stdout.	bt_open()
bt_sendi	Sends an interrupt to the remote bus.	bt_send_irq()
readdma *	Same as readmem, except this program reads larger amounts of data, which results in the DMA hardware being used in the kernel driver instead of cpu copying the data.	bt_read()

shmat *	Takes a shared memory key parameter to attach and read from a shared memory area. Used by the shmconfig-script program.	shmconfig(1)
		shmat(2)

shmbind *	Creates and attaches to a shared memory area that is mapped to a PCI-to-VMEbus mapping and reads or writes to it.	shmget(2)
		shmbind(2)
		shmat(2)

shmconfig-script *	A script that creates a PCI-to-VMEbus mapping via the **`/proc`** file system and creates a shared memory area that is bound to the VMEbus area.	shmconfig(1)
vme-mappings *	A script that shows how to create, display and remove PCI-to-VMEbus mappings via the **`/proc`** file system.	n/a

writemem *	Writes out 256 bytes of data to the remote VMEbus, reads the 256 bytes of data back from the remote VMEbus and then outputs this data to stdout.	bt_read()
		bt_write()

writedma *	Same as writemem, except this program writes larger amounts of data, which results in the DMA hardware being used in the kernel driver instead of cpu copying the data. This example only writes the data to the remote VMEbus; it does not read the data back from the remote VMEbus.	bt_write()

bt_bind_mult

The bt_bind_mult example application uses the bt_bind() function to bind multiple equally-sized buffers to the remote bus. It waits for user input, then prints the first 4 words of each page of each bound buffer. It also optionally writes data to buffer before waiting.

Usage: bt_bind_mult -[natulws]

OPTION	FUNCTION
-n <nbufs>	Number of buffers to allocate and bind. Default is 2.
-a <vmeaddr>	VME address to bind buffer. Defaults to BT_BIND_NO_CARE.
-t <logdev>	Logical device. (BT_DEV_MEM, BT_DEV_IO, BT_DEV_DEFAULT, etc.) Default is to BT_DEV_DEFAULT.
-u <unit>	Unit number to open. Default is unit 0.
-l <len>	Length of the buffer to bind. Default is one page.
-w <value>	Initially write this value to the first 4 words of each page in the buffer.
-s <swapbits>	Sets the swap bits value for the call to bt_bind(). Note that the symbolic names are not recognized.

bt_bind_multsz

The bt_bind_multsz example application uses the bt_bind() function to bind multiple buffers of various sizes to the remote bus. It waits for user input, then prints the first 4 words of each page of each bound buffer. It also optionally writes data to buffer before waiting.

Usage: bt_bind_multsz -[atuws]

OPTION	FUNCTION
-a <vmeaddr>	VME address to bind buffer. Defaults to BT_BIND_NO_CARE.
-t <logdev>	Logical device. (BT_DEV_MEM, BT_DEV_IO, BT_DEV_DEFAULT, etc.) Default is to BT_DEV_DEFAULT.
-u <unit>	Unit number to open. Default is unit 0.
-w <value>	Initially write this value to the first 4 words of each page in the buffer.
-s <swapbits>	Sets the swap bits value for the call to bt_bind(). Note that the symbolic names are not recognized.

bt_hwmap

The bt_hwmap example application uses the bt_hw_map_vme function to create a hardware mapping to an area of VMEbus space.

Usage: bt_hwmap -a[ltus]

OPTION	FUNCTION
-a <addr>	VMEbus physical address. This argument is required.
-l <len>	Length of VMEbus area to map onto the PCI bus. Default is one page (0x1000).
-t <logdev>	Logical device to access. (BT_DEV_A32, BT_DEV_A24, BT_DEV_A16, BT_DEV_IO, BT_DEV_RR). Default is to BT_DEV_A32.
-u <unit>	Unit number to open. Default is unit 0.
-s <swapbits>	Sets the swap bits value for the call to bt_bind(). Note that the symbolic names are not recognized. Default is BT_SWAP_DEFAULT.

bt_hwunmap

The bt_hwmap example application uses the bt_hw_unmap_vme function to remove a hardware mapping from an area of VMEbus space.

Usage: bt_hwunmap -p[tu]

OPTION	FUNCTION
-p <pciaddr>	Local PCI bus physical address of the mapping to be removed. This argument is required.
-t <logdev>	Logical device. (BT_DEV_A32, BT_DEV_A24, BT_DEV_A16, BT_DEV_IO, BT_DEV_RR). Default is to BT_DEV_A32.
-u <unit>	Unit number to open. Default is unit 0.

readdma

This example program is the same as bt_readmem, except it reads larger amounts of data, which results in the DMA hardware being used in the kernel driver instead of cpu copying the data.

Usage: readdma -[atulo]

OPTION	FUNCTION
-a <addr>	Address at which to start data transfer. Default = 0x00000000.
-t <logdev>	Logical device to access. Default is to BT_DEV_A32.
-u <unit>	Unit number to open. Default is unit 0.
-l <length>	Bytes to read. Round down to pagesize. Default is 0x1000.
-o <outlen>	Number of bytes output at the start of each page boundary. Default is 16 bytes. This value must be <= 409.

shmat

This example program is invoked by the shmconfig-script script. It takes the shared memory 'key' value and attaches to and reads from the shared memory area that is bound to VMEbus space.

Usage: shmat -k shmkey -s size [-o outlen]

OPTION	FUNCTION
-k <shmkey>	Shared memory key value, in decimal, or in hex with a leading '0x' or '0X'.
-s <size>	Size in bytes of the shared memory area.
-o <outlen>	Number of bytes at the start of each shared memory page to output to stdout, in hex. Default is 32 bytes.

shmbind

This example program uses **`shmget(2)`**, **`shmbind(2)`** and **`shmat(2)`** to attach a shared memory area to a PCI-to-VMEbus mapping. You can read or write to the VMEbus space using the shared memory attached area. The PCI-to-VME hardware mapping needs to already be created.

Usage: shmbind -p pci_addr -s size [-r | -w value] [-o len]

OPTION	FUNCTION
-p <pci_addr>	Local PCI bus address where VME mapping is located, in hex.
-s <size>	Size in bytes of the shared memory area to create, in hex.
-r	Read from the shared memory area. (Default.)
-w <value>	Write to the shared memory area, using the specified value, in hex.
-o <len>	Number of bytes at the start of each shared memory page to output to stdout, in hex. Default is 32 bytes.

shmconfig-script

This is an example script of how to use **`shmconfig(1)`** to create a shared memory area that is bound to a specific VMEbus area with a PCI-to-VMEbus mapping. This script invokes the shmat example program after the shared memory area is created.

vme-mappings

This is an example script that shows how to create, examine and remove PCI-to-VMEbus mappings using the **`/proc/driver/btp/`***unit***`/vme-mappings`** file.

writemem

This example program uses the bt_write() Bit 3 Mirror API function to write to any of the Bit 3 logical devices.

Usage: writemem -[atud]

OPTION	**FUNCTION**
-a <addr>	Address at which to start data transfer. Default = 0x00000000.
-t <logdev>	Logical device to access (BT_DEV_RDP, BT_DEV_A32, etc.)
-u <unit>	Unit number to open. Default is unit 0.
-d <value>	Starting data value to write. Default is 0.

All numeric values use C radix notation.

Example: Write the first 256 bytes of data from BT_DEV_RDP starting at address 0x00001000:

```
./writemem -a 0x00001000
```

writedma

This example program is the same as writemem, except it writes larger amounts of data, which results in the DMA hardware being used in the kernel driver instead of cpu copying the data. This example only writes the data to the remote VMEbus; it does not read the data back from the remote VMEbus.

Usage: writedma -[atuld]

OPTION	**FUNCTION**
-a <addr>	Starting VME address. Default = 0x00000000.
-t <logdev>	Logical device to access. Default is to BT_DEV_A32.
-u <unit>	Unit number to open. Default is unit 0.
-l <length>	Number of bytes to write. Round down to pagesize. Default is 0x1000.
-d <value>	Starting data value to write. Default is 0.

A
Example Message Queue Programs

This appendix contains example programs that illustrate the use of the POSIX and System V message queue facilities. Additional example programs are provided online in **/usr/share/doc/ccur/examples**.

POSIX Message Queue Example

The example program given here is written in C. In this program, a parent process opens a POSIX message queue and registers to be notified via a real-time signal when the queue transitions from empty to non-empty. The parent spawns a child and waits on the child until the child sends a message to the empty queue. The child sends the message, closes it's descriptor and exits.

The parent receives the real-time signal and captures the `sigev_value` (`si_value`) as delivered by the `siginfo_t` structure in the signal handler. The parent also tests delivery of the `si_code` (SI_MESGQ) before receiving the child's test message. The parent verifies that delivery of the `si_value` (which is a union) was correct as previously registered by the `sigev_value`. The signal handler also displays the real-time signal value received (SIGRTMAX) using psignal. The psignal function doesn't know how to name SIGRTMAX, so it calls it an unknown signal, prints the value and exits.

To build this program, specify the following:

```
gcc mq_notify_rtsig.c -Wall -g -l ccur_rt -o mq_notify_rtsig
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <unistd.h>
#include <mqueue.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <time.h>
#include <sched.h>
#include <signal.h>
#include <bits/siginfo.h>

#define MSGSIZE 40
#define MAXMSGS 5
#define VAL 1234
```

```
void handlr(int signo, siginfo_t *info, void *ignored);

int val, code;

int main(int argc, char **argv)
{

    struct sigaction act;
    struct sigevent notify;
    struct mq_attr attr;
    sigset_t set;
    char *mqname = "/mq_notify_rtsig";
    char rcv_buf[MSGSIZE];
    mqd_t  mqdes1, mqdes2;
    pid_t pid, cpid;
    int status;

    memset(&attr, 0, sizeof( attr));

    attr.mq_maxmsg = MAXMSGS;
    attr.mq_msgsize = MSGSIZE;

    mq_unlink(mqname);

    mqdes1 = mq_open(mqname, O_CREAT|O_RDWR, 0600, &attr);

    sigemptyset(&set);
    act.sa_flags = SA_SIGINFO;
    act.sa_mask = set;
    act.sa_sigaction = handlr;
    sigaction(SIGRTMAX, &act, 0);

    notify.sigev_notify = SIGEV_SIGNAL;
    notify.sigev_signo = SIGRTMAX;
    notify.sigev_value.sival_int = VAL;

    mq_notify(mqdes1, &notify);

    printf("\nmq_notify_rtsig:\tTesting notification sigev_value\n\n");

    printf("mq_notify_rtsig:\tsigev_value=%d\n",\
     notify.sigev_value.sival_int);


    if( (pid = fork()) < 0) {
     printf("fork: Error\n");
     printf("mq_notify_rtsig: Test FAILED\n");
     exit(-1) ;
    }


    if(pid == 0) { /* child */

     cpid = getpid() ;

     mqdes2 = mq_open(mqname, O_CREAT|O_RDWR, 0600, &attr);

     printf("child:\t\t\tsending message to empty queue\n");

     mq_send(mqdes2, "child-test-message", MSGSIZE, 30);
```

```
        mq_close(mqdes2);

        exit(0);
       }


       else {   /* parent */

        waitpid( cpid, &status, 0); /* keep child status from init */

        printf("parent:\t\t\twaiting for notification\n");

        while(code != SI_MESGQ)
           sleep(1);

        mq_receive(mqdes1, rcv_buf, MSGSIZE, 0);

        printf("parent:\t\t\tqueue transition - received %s\n",rcv_buf);
       }


       printf("mq_notify_rtsig:\tsi_code=%d\n",code);
       printf("mq_notify_rtsig:\tsi_value=%d\n",val);

       if(code != -3 || val != VAL) {
        printf("\nmq_notify_rtsig:\tTest FAILED\n\n");
        return(-1);
       }

       mq_close(mqdes1);
       mq_unlink(mqname);

       printf("\nmq_notify_rtsig:\tTest passed\n\n");

       return(0);

}

void handlr(int signo, siginfo_t *info, void *ignored)
{

       psignal(signo, "handlr:\t\t\t");
       val = info->si_value.sival_int;
       code = info->si_code;

       return;
}
```

System V Message Queue Example

The example program given here 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);
     }
```

B
Kernel Tunables for RedHawk Linux Features

Table B-1 contains a list of unique features in RedHawk Linux and the kernel configuration settings that support them. These include features developed by Concurrent for real-time operation, optional package support and features incorporated from open source patches.

For each function, the Kernel Configuration GUI option and the tunable name are given to help you view and modify the settings as needed. Additionally, the default settings for each feature in each of the RedHawk Linux pre-built kernels are provided. For more information about configuring and building a kernel, see Chapter 11.

Information about individual features is available in various locations. In Table B-1, the following references are provided:

- Page numbers (active hypertext links) where information included in this *RedHawk Linux User's Guide* is provided.
- Names and publication numbers of other appropriate Concurrent documents.

Other sources where information may be obtained include:

- Information provided in a separate help window of the Kernel Configuration GUI that displays when the parameter is selected.
- Text files in the **`Documentation`** directory of the kernel source tree.
- Linux documentation sites on the Internet.

Table B-1 Kernel Tunables for RedHawk Linux Features

Functionality	Kernel Configuration GUI Option	Tunable Name	Default Settings*/ RedHawk Kernels	Concurrent Documentation Reference
Shielded CPUs	General Setup	SHIELD	Y / all	page 2-1
Rescheduling Variables	General Setup	RESCHED_VAR	Y / all	page 5-3
High Resolution Process Accounting	General Setup	HRACCT	Y / debug, trace N / generic	page 7-2
High Resolution POSIX Timers	General Setup	HR_POSIX_TIMERS	Y / all	page 6-2
Frequency Based Scheduler (FBS)				
Enable FBS	Frequency-Based Scheduling	FBSCHED	M / all	FBS User's Guide (0898005)
Max number of schedulers		FBSMNI	10 / all	
Max number of unscheduled processes		FBSUNSCHEDMAX	1 / all	
Max number of simultaneously queried tasks		FBSQUERYMAX	100 / all	
Performance Monitor (PM)	Frequency-Based Scheduling	FBSCHED_PM	Y / debug, trace N / generic	FBS User's Guide (0898005)
RCIM Support	Device Drivers	RCIM	Y / all	RCIM User's Guide (0898007)
Priority Inheritance				
Enable for kernel semaphores as mutexes	General Setup	PRIO_INHERIT	Y / all	page 1-6
Enable for kernel R/W semaphores		RWSEM_PRIO_INHERIT		
POSIX Message Queues	General Setup	POSIX_MQUEUE	Y / all	page 3-2
Post/Wait Support	General Setup	POST_WAIT	Y / all	page 5-37
Inherit Capabilities Across exec	General Setup	INHERIT_CAPS_ACROSS_EXEC	Y / all	page 13-5

*** Y = set, N = not set, M = tunable enabled when kernel module is loaded**

Table B-1 Kernel Tunables for RedHawk Linux Features (Continued)

Functionality	Kernel Configuration GUI Option	Tunable Name	Default Settings*/ RedHawk Kernels	Concurrent Documentation Reference
Memory Mapping				
Process space mmap/usermap support	File Systems	PROCMEM_MMAP	Y / all	page 9-1
File permission access to another process' address space		PROCMEM_ANYONE	Y / all	
Enable writes into another process' address space		PROCMEM_WRITE	Y / all	
Enlarge mmap address range	Processor Type and Features	LARGE_MMAP_SPACE	Y / all (i386 only)	page 9-1
NUMA Support	Processor Type and Features	K8_NUMA SCHED_SMT	Y / all x86_64 only N / all x86_64 only	page 10-1
Interrupt Processing				
Softirq daemon priority	General Setup	SOFTIRQ_PRI	0 / all	page 14-12
Softirq preemption blocking		SOFTIRQ_PREEMPT_BLOCK	Y / all	
Kernel Virtual Address Space for Dynamic Allocation	General Setup	VMALLOC_RESERVE	128 / all	page 14-11
Cross Processor Interrupt Reduction				
TLB flush reduction	General Setup	VMALLOC_TLBFLUSH_ REDUCTION	Y / all	page G-4
Reserve for small vmallocs/ioremaps		VMALLOC_SMALL_RESERVE	46 / all i386 262144 / all x86_64	page G-4
Large threshold for vmallocs/ioremaps		VMALLOC_LARGE_THRESHOLD _SIZE	4 / all	page G-4
Preload vmalloc page tables at boot		VMALLOC_PGTABLE_PRELOAD	Y / generic (i386 only)	page G-4
Graphic Page Preallocation	Device Drivers	PREALLOC_GRAPHICS_PAGES	10240 / all	page G-3

***   Y = set,   N = not set,   M = tunable enabled when kernel module is loaded**

Table B-1 Kernel Tunables for RedHawk Linux Features (Continued)

Functionality	Kernel Configuration GUI Option	Tunable Name	Default Settings*/ RedHawk Kernels	Concurrent Documentation Reference
XFS Filesystem				
Enable XFS	File Systems	XFS_FS	Y / all	page 8-1
Real-time subvolume support		XFS_RT	Y / all	
Kernel Preemption	Processor Type and Features	PREEMPT	Y / all	page 1-6
Ptrace Extensions	General Setup	PTRACE_EXT	Y / all	page 1-6
Kernel Debug				
Include kgdb	Kernel Hacking	KGDB	Y / debug N / trace, generic	page 1-7
Kernel Tracing				
Enable kernel tracing	Kernel Tracing	TRACE	Y / trace, debug N / generic	page D-1
Enable BKL trace events		TRACE_BKL	N / all	
System Dumps				
Enable system dumps	Kernel Hacking	CRASH_DUMP	Y / trace, generic N / debug	page 12-1
Enable RLE compression		CRASH_DUMP_COMPRESS_RLE	Y / trace, generic N / debug	
Enable GZIP compression		CRASH_DUMP_COMPRESS_GZIP	Y / trace, generic N / debug	
nVIDIA Graphics Support	nVIDIA Kernel Support	NVIDIA	M / all	RedHawk Release Notes (0898003)
SBS VMEbus-to-PCI Support	SBS VMEbus-to-PCI Support	SBSVME	M / all	page 15-1
Hyper-threading (i386 kernels only)	Processor Type and Features	X86_HT	Y / all	page 2-22
SNARE Audit Support	General Setup	SNARE_AUDIT	N / all	Guide to SNARE for Linux
*** Y = set, N = not set, M = tunable enabled when kernel module is loaded**				

C
Capabilities in RedHawk Linux

This appendix lists the capabilities included in RedHawk Linux and the permissions that each capability provides.

Overview

Capabilities is a method in Linux where the privileges traditionally associated with superuser are divided into distinct units that can be independently enabled and disabled. An unscrupulous user can use some of the permissions provided by capabilities to defeat the security mechanisms provided by Linux; therefore, this functionality should be used with due caution. Capabilities are defined in **`/usr/include/linux/capability.h`**.

For more information about how capabilities work in Linux, refer to the **`capabilities(7)`** man page. For information about the PAM facility that provides an authentication scheme utilizing capabilities, refer to Chapter 13.

Capabilities

This section describes the permissions provided by each of the capabilities defined under RedHawk Linux. Features from standard Linux as well as features unique to RedHawk Linux are included in this discussion.

CAP_CHOWN This capability overrides the restriction of changing user or group file ownership when the current effective user ID, group ID, or one of the supplemental group IDs do not match the file’s UID/GID attributes.

CAP_DAC_OVERRIDE

Except for the file access restrictions enforced by files marked as immutable or append-only (see **`chattr(1)`**), this capability overrides any file discretionary access control (DAC) restrictions that would normally be enforced with the owner/group/world read/write/execute filesystem permission attributes and Access Control List (ACL) restrictions, if ACL support is configured into the kernel for that filesystem (see **`acl(5)`** for more details).

Read and write access DAC restrictions may always be overridden with this capability. Execute DAC restrictions may be overridden with the capability as long as at least one owner/group/world execute bit is set.

This capability also overrides permission access restrictions when using the **`fbsintrpt(3)`** and **`fbsresume(3)`** commands.

CAP_DAC_READ_SEARCH

This capability overrides any file discretionary access control (DAC) restrictions that would normally be enforced with the owner/group/world read/execute filesystem permission attributes and Access Control List (ACL) restrictions if ACL support is configured into the kernel for that filesystem (see **`acl(5)`** for more details).

This capability always allows read access to files and directories, and search (execute) access to directories.

This capability also overrides permission access restrictions when using the **`fbsintrpt(3)`** and **`fbsresume(3)`** commands.

CAP_FOWNER This capability:

- overrides all Discretionary Access Control (DAC) restrictions regarding file attribute changes where the file owner ID must be equal to the user ID.
- allows the FBS_RMID and FBS_SET **`fbsctl(2)`** commands when the fbs creator user ID and user ID do not match the caller’s effective user ID

This capability does not override Data Access Control (DAC) restrictions.

CAP_FSETID This capability overrides the restriction that the effective group ID (or one of the supplementary group IDs) shall match the file group ID when setting the S_ISGID bit on that file.

CAP_IPC_LOCK This capability allows for the locking of memory through the **`mlock(2)`** and **`mlockall(2)`** system service calls.

It also allows locking and unlocking of shared memory segments through the **`shmctl(2)`** SHM_LOCK and SHM_UNLOCK commands.

CAP_IPC_OWNER

This capability overrides the IPC permission set that is associated with an IPC shared memory segment, message queue or semaphore array. The IPC permissions have the same format and meaning as the read/write owner, group and world permissions associated with files. Note that execute permissions are not used. The **`ipcs(1)`** command may be used to view the owner and permissions of the current IPC resources.

CAP_KILL This capability overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.

This capability also overrides the restriction on KDSIGACCEPT **`ioctl(2)`** calls that requires the calling process to be the owner of the tty or have the CAP_SYS_TTY_CONFIG capability.

CAP_LEASE This capability lets a user take out a lease on a file, with the **`fcntl(2)`** F_SETLEASE command, even when the process’ user ID does not match the file system user ID value.

CAP_LINUX_IMMUTABLE

This capability allows the modification of the S_IMMUTABLE and S_APPEND file attributes. See the **`chattr(1)`** man page for more information on these file attributes.

CAP_MKNOD This capability allows the user to make use of the privileged aspects of **`mknod(1)`**/**`mknod(2)`**. It also allows use of the XFS_IOC_FSSETDM_BY_HANDLE xfs filesystem **`ioctl(2)`** command.

CAP_NET_ADMIN

This capability allows for the following network administration activities:

- setting debug and priority options on sockets
- administration of IP firewall, masquerading and accounting
- interface configuration
- multicasting
- reading and writing of network device hardware registers
- adding/deleting tunnels
- modification of routing tables
- setting TOS (type of service)
- activation of ATM control sockets

CAP_NET_BIND_SERVICE

This capability allows binding to TCP/UDP and Stream Control Transmission Protocol (SCTP) sockets below 1024, and to ATM VCIs below 32.

This capability also causes a reserved port to be used when creating an RPC client transport.

CAP_NET_BROADCAST

This capability is not currently used.

CAP_NET_RAW This capability allows the creation of SOCK_RAW and SOCK_PACKET sockets, and the use of the SO_BINDTODEVICE **`setsockopt(2)`** socket option.

CAP_SETGID This capability overrides the restrictions placed on non-root process' group ID value for the **`setregid(2)`**, **`setgid(2)`**, **`setresgid(2)`**, **`setfsgid(2)`** and **`setgroups(2)`** system services.

This capability also allows a process to send a socket level credential control message that contains a group ID value that does not match the current process' current, effective or saved group ID. (Additionally, the credential control message process ID must match the process' thread group ID or the process must also have the CAP_SYS_ADMIN capability, and the credential control message user ID must match the process' saved, effective or current user ID, or have the CAP_SETUID capability.)

CAP_SETPCAP This capability allows a process to transfer any capability in the process' permitted set to any process ID (PID), and to remove any capability in the process' permitted set from any PID.

CAP_SETUID This capability allows setting the current user ID to any user ID, including the user ID of superuser. Extreme caution should be used in granting this capability.

This capability also allows a process to send a socket level credential control message that contains a user ID value that does not match the current process' current, effective or saved user ID. (Additionally, the credential control message process ID must match the process' thread group ID or the process must also have the CAP_SYS_ADMIN capability, and the credential control message group ID must match the process' saved, effective or current group ID, or have the CAP_SETGID capability.)

This capability also overrides the limitation that processes that are ptraced by this process may not inherit the user or group ID of a "set user or group ID on execution" executable that the ptraced process executes.

CAP_SYS_ADMIN This capability provides the following system administration activities:

- allows use of **`bdflush(2)`**
- overrides the open file limit
- allows examination and configuration of disk quotas
- allows examination and configuration of disk usage on a per user or per group basis under the xfs filesystem (if XFS_QUOTA is enabled)
- allows **`umount()`** and **`mount()`**
- allows copying of a process' namespace during **`fork(2)`**/ **`clone(2)`** calls
- allows **`msgctl(2)`**, **`semctl(2)`** and **`shmctl(2)`** IPC_SET and IPC_RMID commands for message queues, semaphores and shared memory areas that do not have a user ID or creator user ID value that matches the process' effective user ID
- allows **`shmctl(2)`** SHM_PHYSBIND commands for shared memory areas where the user ID or creator user ID of the shared memory area does not match the process' effective user ID
- overrides the limit on the maximum number of processes per process on **`fork(2)`**/**`clone(2)`** calls when the non-root user does not have the CAP_SYS_RESOURCE capability
- allows wakeups on **`pw_post(2)`**, **`pw_postv(2)`**, **`server_wake1(2)`** and **`server_wakevec(2)`** calls when the process(es) to be awakened do not have the same user ID or saved user ID as the calling process' effective user ID or user ID value
- allows use of the RCIM_WRITE_EEPROM and RCIM_TESTIRQ **`ioctl(2)`** RCIM driver commands
- allows use of the system dump **`ioctl(2)`** commands, and the setting of the **`sysctl(2)`** kernel.dump.device variable
- allows configuration of serial ports
- allows **`sethostname(2)`** and **`setdomainname(2)`** calls
- allows the use of **`swapon(8)`** and **`swapoff(8)`** calls

- allows the open of raw volume zero and the CCISS_SETINTINFO and CCISS_SETNODENAME **ioctl(2)** commands in the Disk Array driver for HP SA 5xxx and 6xxx Controllers
- allows **ioctl(2)** commands in the Mylex DAC960 PCI RAID Controller driver
- allows the open of raw volume zero in the Compaq SMART2 Controller Disk Array driver
- allows the use of floppy root-only **ioctl(2)** commands (those commands with bit 0x80 set), and also the FDSETPRM and FDDEFPRM set geometry commands
- allows use of the following block device **ioctl(2**) commands: BLKPG add/delete partition, BLKRRPART re-read partition, BLKRASET set read-ahead for block device, BLKFRASET set filesystem read-ahead, BLKBSZSET set logical block size, BLKFLSBUF flush buffer cache, BLKROSET set device read-only
- allows setting the encryption key on loopback filesystems
- allows network block device **ioctl(2)** commands
- allows modification of the memory type range registers (MTRR)
- allows use of **ioctl(2)** commands for power management when APM is enabled in the kernel
- allows use of some **ioctl(2)** commands for certain BIOS settings
- allows use of the VM86_REQUEST_IRQ **vm86(2)** support
- allows use of the CDROMRESET, CDROM_LOCKDOOR and CDROM_DEBUG **ioctl(2)** CDROM commands
- allows DDIOCSDBG DDI debug **ioctl(2)** on sbpcd CDROM driver
- allows use of the root-only Direct Rendering Manager (DRM) **ioctl(2)** commands and the DRM **mmap(2)** DMA memory command
- allows use of the root-only **ioctl(2)** commands in the Specialix RIO smart serial card driver
- allows reading the first 16 bytes of the VAIO EEProm hardware Sensors chip on the I2C serial bus
- allows writes to the **/proc/ide/ide*n*/config** file, modification of the IDE drive settings, and the following IDE **ioctl(2)** commands: HDIO_DRIVE_TASKFILE (execute raw taskfile), HDIO_SET_NICE (set nice flags), HDIO_DRIVE_RESET (execute a device reset), HDIO_GET_BUSSTATE (get the bus state of the hardware interface), HDIO_SET_BUSSTATE (set the bus state of the hardware interface)
- allows use of the SNDRV_CTL_IOCTL_POWER sound **ioctl(2)** command
- allows the use of various root-only **ioctl(2)** commands for various PCI-based sound cards, such as Live! and Sound Blaster 512
- allows use of the experimental networking SIOCGIFDIVERT and SIOCSIFDIVERT Frame Diverter **ioctl(2)** commands
- allows the sending of the SCM_CREDENTIALS socket level control message, when the user ID of the credentials do not

match the current process' effective, saved or current user ID value

- allows administration of md devices (Multiple Devices - RAID and LVM)
- allows adding and removing a Digital Video Broadcasting interface
- allows the VIDIOC_S_FBUF **ioctl(2)** command for the Philips saa7134-based TV card video4linux device driver, if the CAP_SYS_RAWIO capability is not enabled
- allows the use of the VIDIOCSFBUF and VIDIOC_S_FBUF **ioct(2)** commands in the bttv and Zoran video device drivers, if the CAP_SYS_RAWIO capability is not enabled
- allows the use of the VIDIOCSFBUF **ioctl(2)** command in the planb video device driver if the CAP_SYS_RAWIO capability is not enabled
- allows the use of the VIDIOCSFBUF **ioctl(2)** command in the stradis 4:2:2 mpeg decoder driver
- allows the use of the Intelligent Input/Output (I2O) **ioctl(2)** commands
- allows manufacturer commands in ISDN CAPI support driver
- allows reading up to 256 bytes (non-standardized portions) of PCI bus configuration space, and also allows use of the **pciconfig_read(2)** and **pciconfig_write(2)** system service calls
- allows use of the root-only pcmcia **ioctl(2)** commands
- allows use of the FSACTL_SEND_RAW_SRB **ioctl(2)** command in the aacraid Adaptec RAID driver
- allows read and write to the QLogic ISP2x00 nvram
- allows access to the MegaRAID **ioctl(2)** commands
- allows use of the MTSETDRVBUFFER SCSI tape driver **ioctl(2)** command
- allows write access to the **/proc** SCSI debug file, if SCSI_DEBUG is enabled in the kernel (also requires the CAP_SYS_RAWIO capability)
- allows the sending of arbitrary SCSI commands via the SCSI_IOCTL_SEND_COMMAND **ioctl(2)** command (also requires the CAP_SYS_RAWIO capability)
- allows use of the SCSI scatter-gather SG_SCSI_RESET **ioctl(2)** command, **/proc/sg/allow_dio** and **/proc/sg/def_reserved_size** **write(2)**, (also requires the CAP_SYS_ADMIN capability)
- allows use of the IXJCTL_TESTRAM and IXJCTL_HZ **ioct(2)** commands for the Quicknet Technologies Telephony card driver
- allows some autofs root-only **ioctl**s
- allows getting and setting the extended attributes of filesystem objects (**getfattr(1)**, **setfattr(1)**)
- allows root-only **ioct(2)** commands for NetWare Core Protocol (NCP) filesystems
- allows setting up a new smb filesystem connection

- allows the UDF_RELOCATE_BLOCKS **`ioctl(2)`** command on udf filesystems (used on some CD-ROMs and DVDs)
- allows administration of the random device
- allows binding of a raw character device (**`/dev/raw/raw`***n*) to a block device
- allows configuring the kernel's syslog (printk behavior)
- allows writes to the **`/proc/driver/btp/`***unit#***`/vme-mappings`** file, if SBSVME is enabled in the kernel, to create and remove PCI-to-VMEbus mappings
- allows writes to **`/proc/driver/graphics-memory`** to modify size of the pre-allocated graphics memory pool

CAP_SYS_BOOT This capability allows use of the **`reboot(2)`** system service call.

CAP_SYS_CHROOT

This capability allows use of the **`chroot(2)`** system service call.

CAP_SYS_MODULE

This capability allows the insertion and deletion of kernel modules using **`sys_delete_module(2)`**, **`init_module(2)`**, **`rmmod(8)`** and **`insmod(8)`**.

This capability also lets you modify the kernel capabilities bounding set value, cap_bset, where this value is accessible via the **`sysctl(2)`** kernel.cap-bound parameter.

CAP_SYS_NICE This capability allows:

- raising the scheduling priority on processes with the same user ID
- setting the priority on other processes with a different user ID
- setting the SCHED_FIFO and SCHED_RR scheduling policies for processes that have the same user ID
- changing the scheduling policy of processes with a different user ID
- changing the cpu affinity for processes with a different user ID via the **`sched_setaffinity(2)`** or **`/proc/`***pid***`/affinity`** file
- allows the use of **`fbsconfigure(3)`**

CAP_SYS_PACCT This capability allows configuration of process accounting through the **`acct(2)`** system service call.

CAP_SYS_PTRACE

This capability lets a process **`ptrace(2)`** any other process.

This capability also allows the process to **`ptrace(2)`** setuid executables, regardless of the CAP_SETUID setting.

CAP_SYS_RAWIO This capability allows the following raw I/O activities:

- the **`shmctl(2)`** SHM_PHYSBIND command
- the **`resched_cntl(2)`** RESCHED_SET_VARIABLE command
- **`mmap(2)`** of PCI Bus space and access to the PCI Base Address Registers (BAR)

- **open(2)** of **/dev/port** and **/proc/kcore**
- use of the **ioperm(2)** and **iopl(2)** system service calls
- the filesystem **ioctl(2)** FIBMAP command
- **open(2)** of the **/dev/cpu/microcode** file, if MICROCODE is enabled in the kernel
- the following Disk Array driver for HP SA 5xxx and 6xxx Controllers **ioctl(2)** commands: CCISS_PASSTHRU, CCISS_BIG_PASSTHRU, CCISS_DEREGDISK, CCISS_REGNEWD
- the **open(2)** of Disk Array driver for Compaq SMART2 Controllers, and the IDAPASSTHRU **ioctl(2)** command
- the configuration of IDE controllers, and the following IDE **ioctl(2)** commands: HDIO_DRIVE_TASKFILE, HDIO_DRIVE_CMD, HDIO_DRIVE_TASK, HDIO_SCAN_HWIF, HDIO_UNREGISTER_HWIF
- the Fibre Channel Host Bus Adapter CPQFCTS_SCSI_PASSTHRU **ioctl(2)** command
- write access to the **/proc** SCSI debug file, if SCSI_ DEBUG is enabled in the kernel (CAP_SYS_ADMIN is also required)
- sending of arbitrary SCSI commands via the SCSI_IOCTL_SEND_COMMAND **ioctl(2)** command (CAP_SYS_ADMIN is also required)
- use of the SCSI scatter-gather SG_SCSI_RESET **ioctl(2)** command, **/proc/sg/allow_dio** and **/proc/sg/def_reserved_size** **write(2)** (also requires the CAP_SYS_ADMIN capability)
- the ATMSIGD_CTRL **ioctl(2)** command
- use of the VIDIOCSFBUF and VIDIOC_S_FBUF **ioctl(2)** commands in the bttv and Zoran video device drivers, if the CAP_SYS_ADMIN capability is not enabled
- use of the VIDIOCSFBUF **ioctl(2)** command in the planb video device driver if the CAP_SYS_ADMIN capability is not enabled
- use of the HDLCDRVCTL_SETMODEMPAR and HDLCDRVCTL_CALIBRATE **ioctl(2)** commands in the baycom epp radio and HDLC packet radio network device drivers
- the SIOCSCCCFG, SIOCSCCINI, SIOCSCCSMEM, and SIOCSCCCAL **ioctl(2)** commands in the Z8530 based HDLC cards for AX.25 device driver
- the SIOCYAMSCFG **ioctl(2)** command in the AM radio modem device driver
- the COSAIOSTRT, COSAIODOWNLD, COSAIORMEM and COSAIOBMSET **ioctl(2)** commands for the SRP and COSA synchronous serial card device driver
- the FBIO_ALLOC and FBIO_FREE **ioctl(2)** commands for the SiS frame buffer device driver
- the VIDIOC_S_FBUF **ioctl(2)** command for the Philips saa7134-based TV card video4linux device driver, if the CAP_SYS_ADMIN capability is not enabled

CAP_SYS_RESOURCE

This capability lets the user:

- override disk quota limits
- override the IPC message queue size limit on a **`msgctl(2)`** IPC_SET command
- override the number of processes per process on **`fork(2)`**/ **`clone(2)`** calls, when the non-root user does not have the CAP_SYS_ADMIN capability
- increase this user's resource limits with the **`setrlimit(2)`** system service
- set the real-time clock (rtc) periodic IRQ rate, or enable the periodic IRQ interrupts for a frequency that is greater than 64Hz
- override the limit on the number of console terminal opens/allocations
- override the limit on the number of console keyboard keymaps
- when allocating additional space on ufs, ext2 and ext3 filesystems, override the limit on the amount of reserved space. **Note**: the ext2 filesystem also honors the files system user ID when checking for resource overrides, allowing override using **`setfsuid(2)`** also.
- on ext3 filesystems, modify data journaling mode

CAP_SYS_TIME This capability allows:

- setting or adjusting the time via **`clock_settime(2)`**, **`stime(2)`**, **`settimeofday(2)`** and **`adjtimex(2)`**
- use of the RTC_SET_TIME and RTC_EPOCH_SET **`ioctl(2)`** commands for the **`/dev/rtc`** real-time clock device

CAP_SYS_TTY_CONFIG

This capability allows:

- use of the **`vhangup(2)`** system service
- use of all the console terminal and keyboard **`ioctl(2)`** commands, including cases when the user is not the owner of the console terminal

  Note that the use of the KDKBDREP, KDSETKEYCODE, VT_LOCKSWITCH and VT_UNLOCKSWITCH console terminal and keyboard **`ioctl(2)`** commands require this capability even when the user is the owner of the console terminal.

D
Kernel Trace Events

This appendix lists the pre-defined kernel trace events that are included in the RedHawk Linux trace and debug kernels as well as methods for defining and logging custom events within kernel modules.

Refer to the *NightTrace User's Guide*, publication number 0890398, for a complete description of how to supply trace points in user-level code, capture trace data and display the results.

Pre-defined Kernel Trace Events

Table D-1 provides a list of all the kernel trace events that are pre-defined within the RedHawk Linux trace and debug kernels.

Table D-1 Pre-defined RedHawk Linux Kernel Trace Events

Type of Trace Event	Trace Event Name	Description
System Calls	SYSCALL_ENTRY	A system call was entered. (i386 systems only)
	SYSCALL_EXIT	A system call exited. (i386 systems only)
	SYSCALL32_ENTRY	A 32-bit system call was entered. (Opteron systems only)
	SYSCALL32_EXIT	A 32-bit system call exited. (Opteron systems only)
	SYSCALL64_ENTRY	A 64-bit system call was entered. (Opteron systems only)
	SYSCALL64_EXIT	A 64-bit system call exited. (Opteron systems only)
FBS	FBS_SYSCALL	An FBS system call was made. Possible types include: 0 - fbsop 1 - fbsctl 2 - fbsget 3 - pmctl 4 - pmop 5 - fbswait 6 - fbstrig 7 - fbsavail 8 - fbsdir
	FBS_OVERRUN	A process scheduled on FBS incurred an overrun.

Table D-1 Pre-defined RedHawk Linux Kernel Trace Events (Continued)

Type of Trace Event	Trace Event Name	Description
Traps	TRAP_ENTRY	A trap was entered.
	TRAP_EXIT	A trap exited.
Interrupts	IRQ_ENTRY	An IRQ handler was entered.
	IRQ_EXIT	An IRQ exited.
	SMP_CALL_FUNCTION	A function call was made via cross processor interrupt.
	REQUEST_IRQ	A dynamic IRQ assignment was made.
	SOFT_IRQ_ENTRY	A softirq handler was entered. Possible types include: 1 - conventional bottom-half 2 - real softirq 3 - tasklet action 4 - tasklet hi-action
	SOFT_IRQ_EXIT	A softirq handler exited.
	KERNEL_TIMER	The kernel timer interrupt routine was called.
	GLOBAL_CLI	Linux 2.4 __global_cli() was called. (prior to Redhawk 2.0 only)
	GLOBAL_STI	Linux 2.4 __global_sti() was called. (prior to Redhawk 2.0 only)
Process Management	SCHEDCHANGE	The scheduler made a context switch.
	PROCESS	A process management function was performed. Possible types include: 1 - kernel thread created 2 - fork or clone 3 - exit 4 - wait 5 - signal sent 6 - wakeup
	PROCESS_NAME	This event associates a process ID with a process name prior to a fork, clone, or exec.
File System	FILE_SYSTEM	A file system function was performed. Possible types include: 1 - wait for data buffer started 2 - wait for data buffer finished 3 - exec 4 - open 5 - close 6 - read 7 - write 8 - seek 9 - ioctl 10 - select 11 - poll

Table D-1 Pre-defined RedHawk Linux Kernel Trace Events (Continued)

Type of Trace Event	Trace Event Name	Description
Timers	TIMER	A timer function was performed. Possible types include: 1 - timer expired 2 - set_itimer() system call 3 - schedule_timeout() kernel routine
Work Queues	WORKQUEUE_THREAD	A work queue thread was created.
	WORKQUEUE_WORK	A work queue handler was executed.
Memory Management	MEMORY	A memory management function was performed. Possible types include: 1 - page allocation 2 - page freeing 3 - pages swapped in 4 - pages swapped out 5 - wait for page started 6 - wait for page finished
	GRAPHICS_PGALLOC	An additional graphics bind page was dynamically allocated.
Sockets	SOCKET	A socket function was performed. Possible types include: 1 - generic socket system call 2 - socket created 3 - data sent on socket 4 - data read from socket
IPC	IPC	A System V IPC function was performed. Possible types include: 1 - generic System V IPC call 2 - message queue created 3 - semaphore created 4 - shared memory segment created
Networking	NETWORK	A network function was performed. Possible types include: 1 - packet received 2 - packet transmitted
Big Kernel Lock (BKL)	BKL_LOCK	The Linux big kernel lock was locked.
	BKL_UNLOCK	The Linux big kernel lock was unlocked.
	BKL_CONTEND	The Linux big kernel lock is being contended for.
Custom Event	CUSTOM	This is a user-defined event. **Note:** For information on logging this event and dynamically creating other custom kernel trace events, refer to the section “User-defined Kernel Trace Events” below.

Table D-1 Pre-defined RedHawk Linux Kernel Trace Events (Continued)

Type of Trace Event	Trace Event Name	Description
Kernel Trace Management	BUFFER_START	This event marks the beginning of a trace buffer.
	BUFFER_END	This event marks the end of a trace buffer.
	PAUSE	Tracing was paused.
	RESUME	Tracing was resumed.
	EVENT_MASK	The tracing event mask was changed.
	EVENT_CREATED	A new trace event was dynamically created.
	EVENT_DESTROYED	A dynamically created trace event was destroyed.

User-defined Kernel Trace Events

There is a pre-defined "custom" kernel trace event that can be used for any user-defined purpose. The description for using this CUSTOM kernel trace event is described in the next section. Other user-defined events can be created dynamically using the calls described in the section "Dynamic Kernel Tracing" below.

Pre-defined CUSTOM Trace Event

TRACE_CUSTOM may be used to log the pre-defined CUSTOM trace event. The caller provides an integer identifier (*sub_id*) to differentiate multiple uses of the CUSTOM event. The caller may also provide any arbitrary string of data to be logged with the event.

Synopsis

```
#include <linux/trace.h>
void TRACE_CUSTOM (int sub_id, const void* ptr, int size);
```

Arguments are defined as follows:

sub_id	a user-supplied ID
ptr	a pointer to arbitrary data to be logged with the event
size	the size of the data

Dynamic Kernel Tracing

In addition to the pre-defined CUSTOM kernel trace event described above, user-defined kernel trace events can be dynamically created. All are displayed by NightTrace for analysis.

For dynamic kernel tracing, the following calls are used, which are described below:

- **`trace_create_event`** – allocates an unused trace event ID and associates it with a given name
- **`trace_destroy_event`** – deallocates the event ID
- **`TRACE_EVENT`** – a generic trace point function that may be used to log a dynamic event

trace_create_event

This call allocates an unused trace event ID and associates it with the given name.

Synopsis

```
#include <linux/trace.h>
int trace_create_event (const char* name);
```

The argument is defined as follows:

name	is a unique, user-defined name for the trace event. This name is truncated to 31 characters.

The event ID is returned. An attempt is made to return an ID that was not used (created and destroyed) recently. An EVENT_CREATED trace event is logged with this call.

On failure, one of the following is returned:

-ENOSPC	All dynamic event IDs are in use.
-EINVAL	The given name pointer is NULL or points to a NULL string.
-EEXIST	The given name is non-unique.
-ENOMEM	Memory allocation error.

trace_destroy_event

This call deallocates the trace event ID that was allocated with **`create_trace_event`**.

Synopsis

```
#include <linux/trace.h>
void trace_destroy_event (int id);
```

The argument is defined as follows:

id	the event ID that was allocated with **`create_trace_event`**.

An EVENT_DESTROYED trace event is logged with this call.

TRACE_EVENT

This may be used to log a trace point for the newly-created dynamic trace event.

Synopsis

```
#include <linux/trace.h>
void TRACE_EVENT (int id, const void* ptr, int size);
```

Arguments are defined as follows:

id	the event ID
ptr	a pointer to arbitrary data to be logged with the event
size	the size of the data

E
Migrating 32-bit Code to 64-bit Code

This appendix provides information needed to migrate 32-bit code to 64-bit processing on the AMD Opteron processor.

Introduction

RedHawk Linux Version 2.X can execute on the 64-bit AMD Opteron processor in iHawk 870 Series systems as well as on the 32-bit Intel Pentium Xeon processors in iHawk 860 Series systems. The Opteron version of RedHawk Linux is a full 64-bit operating system that executes both 32-bit and 64-bit applications in native mode on the Opteron processor.

The Opteron processor utilizes the AMD64 Instruction Set Architecture (ISA), which is an extension to the x86 instruction set of the iHawk 860 system. The "long" execution mode of the Opteron processor has two submodes: "64-bit" and "compatibility." Existing 32-bit application binaries can run without recompilation in compatibility mode under RedHawk Linux, or the applications can be recompiled to run in 64-bit mode.

32-bit applications run natively with no "emulation mode" to degrade performance. For this reason, many applications do not need to be ported to 64-bits.

Software optimized for Opteron can make use of the large addressable memory and 64-bit architectural enhancements required by the most demanding applications, such as scientific computing, database access, simulations, CAD tools, etc. If an application would benefit from the larger virtual and physical address space afforded by 64-bit processing, information in this section will help you migrate your code.

Porting existing 32-bit applications to 64-bits involves the following areas, which are discussed in detail in the sections that follow:

- Source code written for 32-bits will likely require modifications to execute in 64-bit mode.
- Binaries that have been compiled for 32-bit operation need to be recompiled for 64-bit before running in 64-bit mode.
- The build process (makefiles, project files, etc.) may need to be updated to build 64-bit executables and add portability checking options for compilation.
- Only 64-bit device drivers can be used with 64-bit operating systems. Applications that install device drivers may not work correctly if there is no 64-bit version of the required driver. All drivers supplied with RedHawk Linux are 64-bit compatible.

In addition, the following are discussed:

- Hints to get the most performance from your applications
- RedHawk Linux functionality differences between 32-bit and 64-bit

The **AMD64 Developer Resource Kit** is a complete resource for programmers porting or developing applications and drivers for the Opteron processor. The AMD64 DRK contains technical information including documentation, white papers, detailed presentations and reference guides. This Kit is available from the **www.amd.com** web site.

Procedures

In order to systematically address modifying your code for porting to 64-bits, follow the guidelines below. All source files should be reviewed and modified accordingly, including header/include files, resource files and makefiles. Specifics regarding these steps are provided in the sections that follow.

- Use `#if defined __x86_64__` for code specific to AMD64 architecture.
- Convert all inline assembly code to use intrinsic functions or native assembly subroutines.
- Modify calling conventions in existing assembly code as needed.
- Review use of any pointer arithmetic and confirm results.
- Review references to pointers, integers and physical addresses and use the variable size data types to accommodate the differences between 32 and 64-bit architectures.
- Examine makefiles to build 64-bit executables and add portability checking options.

Coding Requirements

Data Type Sizes

The main issue with 32-bit and 64-bit portability is that there should be no presumption about the size of an address or its relationship to the size of an `int`, `long`, etc.

Table E-1 shows the sizes of the various ANSI data types under RedHawk Linux on AMD64 systems.

Table E-1 Sizes of Data Types

ANSI Data Type	Size in Bytes
`char`	1
`short`	2
`int`	4
`long`	8
`long long`	8
`intptr_t, uintptr_t`	8
`float`	4
`double`	8
`long double`	16

You can use the `sizeof` operator to get the size of the various data types; for example, if you have a variable `int x` you can get the size of `x` with `sizeof(x)`. This usage works even for structs or arrays. For example, if you have a variable of a struct type with the name `a_struct`, you can use `sizeof(a_struct)` to find out how much memory it is taking up.

Longs

Longs become 64-bit, therefore, you need to examine all direct or implied assignments or comparisons between `long` and `int` values. Examine all casts that allow the compiler to accept assignment and comparison between longs and integers to ensure validity. Use the value of the BITS_PER_LONG macro to determine the size of longs.

If ints and longs must remain different sizes (for example, due to existing public API definitions), implement an assertion that ascertains that the value of the 64-bit item does not exceed the maximum value of the 32-bit item and generate an exception condition to handle the case if it does occur.

Pointers

Pointers become 64-bit, therefore, you also need to examine all direct or implied assignments or comparisons between pointers and `int` values. Remove all casts that allow the compiler to accept assignment and comparison between pointers and integers. Change the type to a type of variable size (equal to pointer size). Table E-2 shows the variable size data types.

Table E-2 Variable Size Data Types

ANSI Data Type	Definition
`intptr_t`	Signed integral type to hold a pointer
`uintptr_t`	Unsigned integral type to hold a pointer
`ptrdiff_t`	Signed type to hold the signed difference of two pointer values
`size_t`	Unsigned value indicating the maximum number of bytes to which a pointer can refer
`ssize_t`	Signed value indicating the maximum number of bytes to which a pointer can refer

Arrays

Under 32-bit code, `int` and `long` could be used to hold the size of arrays. Under 64-bit, arrays can be longer than 4 GB. Instead of `int` or `long`, use the `size_t` data type for portability. It will become 64-bit signed integral type when compiled for 64-bit targets, or 32-bit for 32-bit targets. The return values from both `sizeof()` and `strlen()` are both of type `size_t`.

Declarations

You also need to alter any declarations of variables, parameters or function/method return types that must be changed to 64-bit to use one of the size variant types shown in Table E-2.

Explicit Data Sizes

When it is necessary to explicitly address data size, use the data types in Table E-3. There are no ANSI data types that specifically address data size; these types are specific to Linux.

Table E-3 Fixed Precision Data Types

Data Type	Definition
`int64_t`	64-bit signed integer
`uint64_t`	64-bit unsigned integer
`int32_t`	32-bit signed integer
`uint32_t`	32-bit unsigned integer
`int16_t`	16-bit signed integer
`uint16_t`	16-bit unsigned integer
`int8_t`	8-bit signed integer
`uint8_t`	8-bit unsigned integer

Constants

Constants, especially hex or binary values, are likely to be 32-bit specific. For example, a 32-bit constant 0x80000000 becomes 0x0000000080000000 in 64-bit. Depending upon how it is being used, the results may be undesirable. Make good use of the ~ operator and type suffixes to avoid this problem; for example, the 0x80000000 constant might be better as ~0x7ffffffful instead.

APIs

Code might need to be changed to use 64-bit APIs. Some APIs use data types which the compiler will interpret as 64-bit in conflict with explicit 32-bit data types.

Calling Conventions

Calling conventions specify how processor registers are used by function callers and callees. This applies when porting hand coded assembly code that interoperates with C code and for in-line assembly statements. The Linux calling conventions for the Opteron are given in Table E-4.

Table E-4 Calling Conventions

Register	Status	Use
`%rax`	Volatile	Temporary register; with variable arguments passes information about the number of SSE registers used; first return register
`%rbx`	Non-volatile	Optionally used as base pointer, must be preserved by callee
`%rdi, %rsi, %rdx, %rcx, %r8, %r9`	Volatile	Used to pass integer arguments 1,2,3,4,5,6
`%rsp`	Non-volatile	Stack pointer
`$rbp`	Non-volatile	Optionally used as frame pointer, must be preserved by callee
`%r10`	Volatile	Temporary register, used for passing a function's static chain pointer
`%r11`	Volatile	Temporary register
`%r12-%r15`	Non-volatile	Must be preserved by callee
`%xmm0-%xmm1`	Volatile	Used to pass and return floating point arguments
`%xmm2-%xmm7`	Volatile	Used to pass floating point arguments
`%xmm8-%xmm15`	Volatile	Temporary registers
`%mmx0-%mmx7`	Volatile	Temporary registers
`%st0`	Volatile	Temporary register; used to return long double arguments
`%st1-%st7`	Volatile	Temporary registers
`%fs`	Volatile	Reserved for system use as thread-specific data register

Conditional Compilation

In cases where there is the need to supply conditional code for 32-bit vs. 64-bit execution, the macros in Table E-5 can be used.

Table E-5 Macros for Conditional Compilation

Macro	Definition
`__amd64__`	Compiler will generate code for AMD64
`_i386`	Compiler will generate code for x86

Miscellaneous

A variety of other issues can arise from sign extension, memory allocation sizes, shift counts, and array offsets. Be especially careful about any code that makes assumptions about the semantics of integer overflow.

Compiling

Existing makefiles should build native 64-bit executables on the Opteron processor with little or no modifications.

The following **gcc** switches can be used to catch portability issues. Refer to the **gcc(1)** man page for details.

```
-Werror -Wall -W -Wstrict-prototypes -Wmissing-prototypes
-Wpointer-arith -Wreturn-type -Wcast-qual -Wwrite-strings
-Wswitch -Wshadow -Wcast-align -Wuninitialized -ansi
-pedantic -Wbad-function-cast -Wchar-subscripts -Winline
-Wnested-externs -Wredundant-decl
```

Testing/Debugging

Follow standard RedHawk Linux testing and debugging techniques for 64-bit code.

Performance Issues

The information in this section discusses how to get the best performance from your 64-bit application.

Memory Alignment and Structure Padding

Alignment issues won’t cause exceptions but can cause a performance hit. Misalignment is handled at runtime at the expense of several clock cycles. The performance side-effects of poorly aligned operands can be large.

Data within structures will be aligned on natural boundaries which can lead to inefficient code due to wasted space. Natural alignment means that 2-byte objects are stored on 2-byte boundaries, 4-byte objects on 4-byte boundaries, etc.

For example, the following structure definition will consume 24 bytes when generating 64-bit code:

```
typedef struct _s {
    int x;
    int *p;
    int z;
} s, *ps;
```

The pointer `p` will be aligned on an 8-byte boundary which will cause 4 bytes of padding to be added after the `x` member. In addition, there will be an additional 4 bytes of padding after the `z` member to pad the structure out to an even eight byte boundary.

The most efficient structure packing will be achieved by placing the members from largest to smallest in the structure. The following declaration is more efficient. It will take only 16 bytes and does not require any padding:

```
typedef struct _s }
    int *p;
    int x;
    int z;
} s;
```

Because of potential padding, the safest way to find the constant offset of fields within a structure is to use the `offsetof()` macro, which is defined in **`stddef.h`**.

F
Kernel-level Daemons on Shielded CPUs

The Linux kernel uses many kernel daemons to perform system functions. Some of these daemons are replicated on every CPU in the system. Shielding a CPU from processes will not remove one of these “per-CPU” daemons.

The following daemons can create serious jitter problems on process-shielded CPUs. Fortunately, these daemons can be avoided by configuring and using the system carefully.

kmodule *cpu*

These daemons are created and executed each time a kernel module is unloaded. It is highly recommended that kernel modules are not unloaded while real-time applications are running on the system.

migration/*cpu*

These are the task migration daemons responsible for migrating tasks off a particular CPU. These daemons will run on a process-shielded CPU if a process running on that CPU is forced to migrate off that processor. Forced migration may happen when any of the following interfaces are used:

/proc/*pid***/affinity**
sched_setaffinity(2)
/proc/shield/procs
cpucntl(2)
delete_module(2)

Applications that are running on shielded CPUs should use these interfaces only when background process jitter can be tolerated.

Forced migration is also done by various kernel features, which can be enabled with the CPU_FREQ and NUMA kernel configuration options. These options have been disabled by default in all RedHawk Linux kernel configurations.

kswapd *node*

These are the page swap-out daemons that swap pages out to a swap device to reclaim pages when memory runs low.

When the kernel is built with the NUMA configuration option enabled, there may be several of these daemons, each biased to a single CPU. When a CPU is process-shielded or downed (using **cpu(1)**), the daemon is moved to a non-shielded active CPU. When the CPU is no longer shielded or down, the daemon is moved back.

When NUMA is disabled, there is one system-wide daemon that is not biased to any particular CPUs; therefore, **kswapd** will not run on CPUs shielded from processes and is only a problem on a non-shielded CPU.

NUMA is enabled by default only on prebuilt RedHawk x86_64 kernels.

kapmd This is the Advanced Power Management (APM) daemon that processes power management requests. It is always biased to CPU 0. APM may be disabled with the kernel boot parameter "apm=off" or may be completely eliminated by disabling the APM kernel configuration option. APM has been disabled by default in all RedHawk Linux kernel configurations. Because this daemon is not a per-CPU daemon, it will not run on CPUs shielded from processes and is therefore a problem only on a non-shielded CPU.

The following daemons may execute on process-shielded CPUs. However, because they perform necessary functions on behalf of processes or interrupts that have been biased to that CPU, and because these daemons are only activated as a result of actions initiated by the processes or interrupts that are biased to a shielded CPU, these daemons are considered less problematic in terms of their impact on determinism.

ksoftirqd/*cpu* These are the softirq daemons that execute softirq routines for a particular CPU. One of these daemons will run on a process-shielded CPU if a device driver interrupt handler biased to that CPU uses softirqs either directly or indirectly via tasklets. Softirqs are used directly by the local timer, SCSI, and networking interrupt handlers. Tasklets are used by many device drivers.

The priority of the **ksoftirqd** is determined by the SOFTIRQ_PRI kernel tunable, which is located under General Setup on the Kernel Configuration GUI. 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.

events/*cpu* These are the default work queue threads that perform work on behalf of various kernel services initiated by processes on a particular CPU. They also may perform work that has been deferred by device driver interrupt routines that have been biased to the same CPU. These daemons execute with a nice value of -10.

aio/*cpu* These are work queue threads that complete asynchronous I/O requests initiated with the **io_submit(2)** system call by processes on a particular CPU. These daemons execute with a nice value of -10.

reiserfs/*cpu* These are work queue threads used by the Reiser File System. These daemons execute with a nice value of -10.

xfsdatad/*cpu*
xfslogd/*cpu* These are work queue threads used by the IRIX Journaling File System (XFS). These daemons execute with a nice value of -10.

cio/*cpu*	
kblockd/*cpu*	
wanpipe_wq/*cpu*	These are work queue threads used by various device drivers. These threads perform work on behalf of various kernel services initiated by processes on a particular CPU. They also perform work that has been deferred by device driver interrupt routines that have been biased to the same CPU. These daemons execute with a nice value of -10.

Note also that any third-party driver may create private work queues and work queue threads that are triggered by processes or interrupt handlers biased to a shielded CPU. These daemons are always named *name***/***cpu* and execute with a nice value of -10.

G
Cross Processor Interrupts on Shielded CPUs

This appendix discusses the impact of cross processor interrupts on shielded CPUs and methods to reduce or eliminate these interrupts for best performance.

Overview

On a RedHawk platform configured with one or more shielded CPUs, certain activities on the other CPUs can cause interrupts to be sent to the shielded CPUs. These cross processor interrupts are used as a method for forcing another CPU to handle some per-CPU specific task, such as flushing its own data cache or flushing its own translation look-aside buffer (TLB) cache.

Since cross processor interrupts can potentially cause noticeable jitter for shielded CPUs, it is useful to understand the activities that cause these interrupts to occur, and also how to configure your system so that some of these interrupts can be eliminated.

Memory Type Range Register (MTRR) Interrupts

On Intel P6 family processors (Pentium Pro, Pentium II and later) the Memory Type Range Registers (MTRRs) can be used to control processor access to memory ranges. This is most useful when you have a video (VGA) card on a PCI or AGP bus. Enabling write-combining allows bus write transfers to be combined into a larger transfer before bursting over the PCI/AGP bus. This can increase performance of image write operations by 2.5 times or more.

While the MTRRs provide a useful performance benefit, whenever a new MTRR range is set up or removed, a cross processor interrupt will be sent to all the other CPUs in order to have each CPU modify their per-CPU MTRR registers accordingly. The time that it takes to process this particular interrupt can be quite lengthy, since all the CPUs in the system must first sync-up/handshake before they modify their respective MTRR registers, and they must handshake yet again before they exit their respective interrupt routines. This class of cross processor interrupt can have a severe effect on determinism having been measured at up to three milliseconds per interrupt.

When the X server is first started up after system boot, a MTRR range is set up, and one of these MTRR cross processor interrupts is sent to all other CPUs in the system. Similarly, when the X server exits, this MTRR range is removed, and all other CPUs in the system receive yet another MTRR interrupt.

Three methods can be used to eliminate MTRR related cross processor interrupts during time-critical application execution on shielded CPUs:

1. Reconfigure the kernel so that the MTRR kernel configuration option is disabled. When using the Kernel Configuration GUI, this option is located under the Processor Type and Features section and is referred to as "MTRR (Memory Type Range Register) support". This eliminates MTRR cross processor interrupts since the kernel support for this feature is no longer present. Note that this option has a potentially severe performance penalty for graphic I/O operations.

2. Start up the X server before running the time-critical applications on the shielded CPU(s), and keep the X server running until the time-critical activity has completed. The MTRR interrupts will still occur, but not during time-critical activities.

3. The MTRR range can be preconfigured so that no cross processor interrupts occur. Use the following procedure for preconfiguration of the MTRRs used by the X server:

   a. After the system is booted, but before the X server has started up, examine the current MTRR settings. You need to be in either init state 1 or 3.

```
cat /proc/mtrr
reg00: base=0x00000000 ( 0MB), size=1024MB: write-back, count=1
reg01: base=0xe8000000 (3712MB), size= 128MB: write-combining, count=1
```

   b. After the X server is started up, re-examine the MTRR register settings:

```
cat /proc/mtrr
reg00: base=0x00000000 ( 0MB), size=1024MB: write-back, count=1
reg01: base=0xe8000000 (3712MB), size= 128MB: write-combining, count=2
reg02: base=0xf0000000 (3840MB), size= 128MB: write-combining, count=1
```

   c. In this example, the new X server entry is the last entry, "reg02". If your system has multiple graphics cards, or shows more than one new entry, then these additional entries should also be accommodated with additional **rc.local** script entries.

   d. Now add additional line(s) to your **/etc/rc.d/rc.local** script to account for the X server MTRR entries. In our example we have just one X server entry to account for:

```
 echo "base=0xf0000000 size=0x8000000 type=write-combining" > /proc/mtrr
```

   e. Whenever the hardware configuration is modified on the system, it is a good idea to check that the MTRR entries in **/etc/rc.d/rc.local** are still correct by starting up the X server and using:

```
        cat /proc/mtrr
```

   to examine the MTRR output and check for differences from the previous MTRR settings.

Graphics Interrupts

A number of cross processor interrupts are issued while running graphics applications. These include the following operations:

Accessing Video RAM — Whenever the X server is started on a system containing an AGP, PCI or PCI Express NVIDIA graphics card, two TLB flush cross processor interrupts are issued when the Video RAM area is accessed.

AGP bindings — On an AGP system, a kernel graphics driver such as the NVIDIA driver will allocate a kernel memory buffer and then create an AGP memory binding to that buffer. Whenever these bindings are added or removed during graphics execution, two cross processor interrupts are normally sent to each of the other CPUs in the system in order to first flush their data caches and then flush their kernel TLB translations. This class of cross processor interrupt can have a fairly severe impact that has been measured to be 50 to 250 microseconds per interrupt. These bindings occur when:

- starting up or exiting the X server
- running graphics applications
- switching from a non-graphics tty back to the graphics screen with a Ctrl Alt F# keyboard sequence

DMA buffer allocation — Jitter is also caused by the NVIDIA driver when allocating and releasing physical pages for DMA buffer use.

These types of cross processor interrupts are eliminated or reduced when a pool of cache-inhibited pages is pre-allocated. As graphics buffer allocations and AGP memory bind requests are made, the pages needed to satisfy these requests are taken from the freelist of pages. Since these pages are already cache-inhibited, there is no need to issue additional flush operations when these pages are used. When an allocation or binding is removed, the pages are placed back onto the page freelist, remaining cache-inhibit-clean. Should the pool of pre-allocated pages be empty when a request is made, pages will be dynamically allocated and cross processor interrupts will be issued in the normal fashion. Therefore, it is usually best to pre-allocate enough pages so that the pool of available pages never becomes empty.

To enable this support, the PREALLOC_GRAPHICS_PAGES kernel parameter must have a positive value representing the number of pre-allocated pages in the pool. A value of 10240 is configured by default in all pre-defined RedHawk Linux kernels. If a value of 0 (zero) is supplied for the kernel parameter, this feature is disabled. The PREALLOC_GRAPHICS_PAGES option is located under the Device Drivers -> Character Devices subsection of the Kernel Configuration GUI. This option is valid on systems with supported AGP hardware or with one or more NVIDIA PCI/PCI Express cards. Note that with this feature enabled, the TLB flush cross processor interrupts that normally occur when the Video RAM area is accessed are reduced to a single TLB flush cross processor interrupt at system bootup only.

The **`/proc/driver/graphics-memory`** file can be examined while running graphics applications to observe the maximum amount of graphics memory pages actually in use at any time to determine if the value should be adjusted. For example:

```
$ cat /proc/driver/graphics-memory
Pre-allocated graphics memory:          10240  pages
Total allocated graphics memory:        10240  pages
Graphics memory in use:                    42  pages
Maximum graphics memory used:              42  pages
```

You may write to the file to increase or decrease the number of pages in the pool. This allows you to test your system with various values before changing the kernel configuration parameter. The following example lowers the number of pre-allocated pages in the pool to 5120:

```
$ echo 5120 > /proc/driver/graphics-memory
```

The user must have CAP_SYS_ADMIN capability to write to this file. Note that the page value written to the file must be larger than or equal to the current value of the "Graphics memory in use" field. If the number of currently allocated pages needs to be lowered, exit the X server.

Specifying an unrealistically large value will result in page allocation failures and the allocation will be backed out. After writing to the file, read the file to verify that the page allocation change was successful.

Note that when the NVIDIA driver is loaded or unloaded, a Page Attribute Table (PAT) cross processor interrupt is sent to each CPU. To minimize the jitter involved, avoid loading or unloading the NVIDIA module during time-critical applications. You may pre-load the NVIDIA driver before running time-critical applications, or during system boot with the following command:

```
$ modprobe nvidia
```

Kernel TLB Flush Interrupts

Kernel TLB flush cross processor interrupts occur when various types of kernel virtual space translations are added or removed. For some types of kernel translations, the interrupt is sent when a new translation is created; for other types, the interrupt is sent when the translation is removed. This class of cross processor interrupt has minimal impact that has been measured at less than 10 microseconds per interrupt.

Modifications have been made in RedHawk that reduce the number of times **`vmalloc()`** and **`ioremap()`** kernel TLB flush cross processor interrupts occur. A modified algorithm starts a new search for available kernel virtual space at the end of the last allocated virtual address. TLB flushes occur only when the search wraps to the beginning of the virtual space area. Provisions have been made that prevent an area that is about to be freed from being reallocated before being flushed.

Repeatedly allocating and freeing up kernel virtual space can fragment the virtual area over time. To provide more contiguous allocation space and reduce fragmentation, the virtual area is conceptually divided into two parts: an area where smaller allocations are made, and an area where larger allocations are made. Control over these two sections is provided through two kernel configuration parameters. Allocations to these two separate areas are handled independently. When the end of one of these sections is reached, allocations wrap around to the beginning of that section. When either section wraps, both sections wrap to take advantage of the system-wide TLB flush that occurs.

The following kernel configuration parameters enable and control this functionality. They are accessible under the General Setup option of the Kernel Configuration GUI.

VMALLOC_TLBFLUSH_REDUCTION
: Enables the algorithms for reducing vmalloc TLB flushes.

VMALLOC_SMALL_RESERVE
: Defines the amount of space, in megabytes, to be set aside for small allocations. The value specified as the default is approximately half of the default size of the total vmalloc virtual space area (VMALLOC_RESERVE).

The large allocation area size is equal to the total vmalloc virtual space minus the small reserve size. If the small reserve value needs to be significantly decreased in order to obtain a larger contiguous vmalloc large allocation area, increasing the value of VMALLOC_RESERVE is recommended since decreasing the small reserve size will cause the number of vmalloc-generated kernel TLB flush interrupts to increase. (Note that x86_64 platforms have a fixed 512 GB vmalloc virtual space size and the VMALLOC_RESERVE parameter is not used.) If large contiguous vmalloc requests are not required, specifying a larger small reserve value will slightly decrease the number of vmalloc-generated kernel TLB flush interrupts.

Note that when either the small or large vmalloc allocation areas become full or fragmented and there is not enough free space to satisfy a vmalloc or ioremap request, a message will appear at the console and be logged to the **/var/log/messages** file indicating an allocation failure has occurred and noting which vmalloc virtual area (small or large) was depleted.

Boot parameters can be used to change the size of the total (**vmalloc=***size*) and the small (**vmalloc_sm=***size*) areas during system boot. Increasing the total vmalloc area also increases the large vmalloc allocation area, unless the small allocation area size is also increased. These two boot parameters may be used either together or independently.

VMALLOC_LARGE_THRESHOLD_SIZE
: Defines the size, in megabytes, for determining when an allocation should be made in the large vmalloc virtual area.

VMALLOC_PGTABLE_PRELOAD
: Preloads vmalloc page tables at boot, eliminating kernel page faults that occur when vmalloc space is referenced. Only applicable to i386 generic kernels; on by default.

The **/proc/vmalloc-reserve-info** file can be read to view the current values, including the total and used vmalloc allocations, largest contiguous vmalloc area that is free (chunk) for the small and large areas separately and the large threshold value. The output from **/proc/meminfo** displays the total and used amounts for the entire (small and large) virtual space areas and the larger of the two small/large chunk values.

If this feature is disabled, the following types of activities can cause quite a few of these TLB flush interrupts, and should therefore be avoided or limited while time-critical applications are executing on shielded CPUs:

1. Starting and stopping graphics applications.
2. Starting and stopping the X server.
3. Switching between the graphics terminal screen and other non-graphics terminals with the Ctrl Alt F# keyboard sequence.
4. Changing the screen resolution with the Ctrl Alt - or Ctrl Alt + keyboard sequences.

User Address Space TLB Flush Interrupts

Processes that are biased to execute on a shielded CPU and that share their address space with processes that execute on other CPUs may receive user-space TLB flush cross processor interrupts. Processes that make use of shared memory areas but which are sharing their address space only with processes on the same CPU will *not* observe any cross processor interrupts due to any shared memory activity.

Multithreaded applications that use the pthreads library and Ada applications are examples of shared memory applications – even though the programmer has not explicitly made calls to create shared memory. In these types of programs, the pthreads library and the Ada run time are creating shared memory regions for the user. Therefore, these applications are subject to this type of cross processor interrupt when threads from the same thread group or same Ada program execute on separate CPUs in the system.

A user address TLB flush cross processor interrupt is generated when another process that is sharing the same address space is executing on a different CPU and causes a modification to that address space's attributes. Activities such as memory references that cause page faults, page swapping, **`mprotect()`** calls, creating or destroying shared memory regions, etc., are examples of address space attribute modifications that can cause this type of cross processor interrupt. This class of cross processor interrupt has minimal impact that has been measured at less than 10 microseconds per interrupt. When large amounts of memory are shared, the impact can be more severe.

In order to eliminate these types of cross processor interrupts, users are encouraged to use and write their applications such that time-critical processes executing on shielded CPUs avoid operations which would affect a shared memory region during the time-critical portion of their application. This can be accomplished by locking pages in memory, not changing the memory protection via **`mprotect()`** and not creating new shared memory regions or destroying existing shared memory regions.

H
Serial Console Setup

This appendix provides the steps needed to configure a serial console under RedHawk.

Note that a serial console is needed if you wish to use the **kdb** kernel debugger on a system with a USB keyboard.

1. Modify the GRUB boot command line to include the following kernel option:

   ```
   console=tty#,baud#
   ```

   where `tty#` is the serial port to use for the console and `baud#` is the serial baud rate to use. Generally, this almost always looks like:

   ```
   console=ttyS0,115200
   ```

2. Change the **`/etc/inittab`** file to include the following line:

   ```
   S0:2345:respawn:/sbin/agetty 115200 ttyS0 vt100
   ```

   The `baud#` and `tty#` must match the same values that were given in the boot option in step 1. The final keyword specifies the terminal type, which is almost always vt100 but can be customized if necessary. See the **`agetty(8)`** man page for more information.

   This line can be added anywhere in the file, although it is generally added at the end. The purpose of this line is to get a login on the serial console after the system boots into multi-user mode.

3. If root login is desired on the serial console (generally it is) you must change or remove the **`/etc/securetty`** file. See the **`securetty(5)`** man page for more details.

4. Connect a suitable data terminal device to the serial port and ensure that it is configured to communicate at the chosen baud rate. Depending on the specific device being used, a null-modem may be required.

   Note that an inexpensive Linux PC is an excellent choice for a data terminal device. See the **`minicom(1)`** man page for more information about creating a serial communication session.

   A Windows PC can also be used, but the explanation of that is beyond the scope of this documentation.

Glossary

This glossary defines terms used in RedHawk Linux. Terms in *italics* are also defined here.

affinity

An association between processes or interrupts and the CPUs on which they are allowed to execute. They are prohibited from executing on CPUs not included in their affinity mask. If more than one CPU is included in the affinity mask, the kernel is free to migrate the process or interrupt based on load and other considerations, but only to another CPU in the affinity mask. The default condition is affinity to execute on all CPUs in the system; however, specifications can be made through **`mpadvise(3)`**, **`shield(1)`**, **`sched_setaffinity(2)`** and the **`/proc`** file system. Using affinity with *shielded CPUs* can provide better determinism in application code.

AGP

A bus specification by Intel which gives low-cost 3D graphics cards faster access to main memory on personal computers than the usual PCI bus.

async-safe

When a library routine can be safely called from within signal handlers. A thread that is executing some async-safe code will not *deadlock* if it is interrupted by a signal. This is accomplished by blocking signals before obtaining locks.

atomic

All in a set of operations are performed at the same time and only if they can all be performed simultaneously.

authentication

Verification of the identity of a username, password, process, or computer system for security purposes. *PAM* provides an authentication method on RedHawk Linux.

blocking message operation

Suspending execution if an attempt to send or receive a message is unsuccessful.

blocking semaphore operation

Suspending execution while testing for a semaphore value.

breakpoint

A location in a program at which execution is to be stopped and control of the processor switched to the debugger.

busy-wait

A method of *mutual exclusion* 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. Also known as a *spin lock*.

capabilities

A division of the *privilege*s traditionally associated with superuser into distinct units that can be independently enabled and disabled. The current set of all valid Linux capabilities can be found in **`/usr/include/linux/capability.h`** and detailed in Appendix C. Through *PAM*, a non-root user can be configured to run applications that require privileges only root would normally be allowed.

condition synchronization

Utilizing sleep/wakeup/timer mechanisms to delay a process' progress until an application-defined condition is met. In RedHawk Linux, the **`postwait(2)`** and **`server_block(2)`**/**`server_wake(2)`** system calls are provided for this purpose.

context switch

When a multitasking operating system stops running one process and starts running another.

critical section

A sequence of instructions that must be executed in sequence and without interruption to guarantee correct operation of the software.

deadlock

Any of a number of situations where two or more processes cannot proceed because they are both waiting for the other to release some resource.

deferred interrupt handling

A method by which an interrupt routine defers processing that would otherwise have been done at interrupt level. RedHawk Linux supports *softirqs*, *tasklets* and *work queues*, which execute in the context of a kernel daemon. The priority and scheduling policy of these daemons can be configured so that a high-priority *real-time* task can *preempt* the activity of deferred interrupt functions.

determinism

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 to both the amount of time required to execute a time-critical portion of a user's application and to the amount of time required to execute system code in the kernel.

deterministic system

A system in which it is possible to control the factors that impact *determinism*. Techniques available under RedHawk Linux for maximizing determinism include *shielded CPUs*, *fixed priority scheduling policy*, *deferred interrupt handling*, *load balancing* and unit control of *hyper-threading*.

device driver

Software that communicates directly with a computer hardware component or peripheral, allowing it to be used by the operating system. Also referred to as device module or driver.

direct I/O

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.

discretionary access control

Mechanisms based on usernames, passwords or file access permissions that check the validity of the credentials given them at the discretion of the user. This differs from mandatory controls, which are based on items over which the user has no control, such as the IP address.

execution time

The amount of time it takes to complete a task. Using the high resolution process accounting facility in RedHawk Linux, execution time measurements for each process are broken down into system, user, interrupted system and interrupted user times measured with the high resolution time stamp counter (TSC).

FBS

See *Frequency-Based Scheduler (FBS)*.

fixed priority scheduling policy

A scheduling policy that 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. There are two fixed priority scheduling policies: SCHED_FIFO and SCHED_RR.

flavor

A variation of a single entity. RedHawk Linux has three flavors of pre-built kernels, each containing different characteristics and configurations. A customized kernel would constitute another flavor. The flavor designation is defined in the top level Makefile and appended as a suffix to the kernel name when the kernel is built; e.g, <kernelname>**`-trace`**.

Frequency-Based Scheduler (FBS)

A task synchronization mechanism used to initiate processes at specified frequencies based on a variety of sources, which include high-resolution clocks provided by the Real-Time Clock and Interrupt Module (*RCIM*), an external interrupt source, or the completion of a cycle. The processes are then scheduled using a priority-based scheduler. When used in conjunction with the *Performance Monitor (PM)*, FBS can be used to determine the best way of allocating processors to various tasks for a particular application.

The *NightSim* tool is a graphical interface to the Frequency-Based Scheduler and Performance Monitor.

GRUB

GRand Unified Bootloader. A small software utility that loads and manages multiple operating systems (and their variants). GRUB is the default bootloader for RedHawk Linux.

hyper-threading

A feature of the Intel Pentium Xeon processor that allows for a single physical processor to run multiple threads of software applications simultaneously. Each processor has two sets of architecture state while sharing one set of processor execution resources. Each architecture state can be thought of as a logical CPU resulting in twice as many logical CPUs in a system. A uniprocessor system with hyper-threading enabled has two logical CPUs, making it possible to *shield* one of them from interrupts and background processes. Hyper-threading is enabled by default in all RedHawk Linux i386 pre-built kernels.

info page

Info pages give detailed information about a command or file. Its companion, *man pages*, tend to be brief and provide less explanation than info pages. Info pages are interactive with a navigable menu system. An info page is accessed using the **info(1)** command.

interprocess communication (IPC)

A capability that allows one process to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC methods include pipes, *message queues*, *semaphores*, *shared memory* and sockets.

interprocess synchronization

Mechanisms that allow cooperating processes to coordinate access to the same set of resources. RedHawk Linux supplies a variety of interprocess synchronization tools including *rescheduling variable*s, *busy-wait* and *sleepy-wait mutual exclusion* mechanisms and *condition synchronization* tools.

jitter

The size of the variation in the arrival or departure times of a periodic action. 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. Jitter normally causes no problems as long as the actions all stay within the correct period, but *real-time* tasks generally require that jitter be minimized as much as possible.

journaling file system

A file system whereby disk transactions are written sequentially to an area of disk called a journal or log before being written to their final locations within the filesystem. If a crash occurs before the journal entry is committed, the original data is still on the disk and only new changes are lost. When the system reboots, the journal entries are replayed and the update that was interrupted is completed, greatly simplifying recovery time. Journaling file systems in RedHawk Linux include ext3, xfs and reiserfs.

kernel

The critical piece of an operating system which performs the basic functions on which more advanced functions depend. Linux is based on the kernel developed by Linus Torvalds and a group of core developers. Concurrent has modified the Linux kernel distributed by Red Hat to provide enhancements for *deterministic real-time* processing. RedHawk Linux supplies three pre-built kernels with the following *flavor*s: generic, debug and trace. They reside as files named **vmlinuz-**<kernelversion>**-RedHawk-**<revision.level>**-**<flavor> in the **`/boot`** directory.

Kernel Configuration GUI

The graphical interface from which selections are made for configuring a kernel. In RedHawk Linux, running the **`ccur-config`** script displays the GUI where selections can be made.

load balancing

Moving processes from some CPUs to balance the load across all CPUs.

man page

A brief and concise online document that explains a command or file. A man page is displayed by typing **`man`** followed by a space and then the term you want to read about at the shell prompt. Man pages in RedHawk Linux include those provided with the Red Hat Linux distribution as well as those describing functionality developed by Concurrent.

memory object

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. Memory objects include *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, which eliminates copying the data between the *kernel* and the application.

message queues

An *interprocess communication (IPC)* mechanism that allows one or more processes to write messages which will be read by one or more reading processes. RedHawk Linux includes support for *POSIX* and *System V* message queue facilities.

module

A collection of routines that perform a system-level function. A module may be loaded and unloaded from the running *kernel* as required.

mutex

A *mutual exclusion* device useful for protecting shared data structures from concurrent modifications and implementing *critical section*s. A mutex has two possible states: unlocked (not owned by any thread) and locked (owned by one thread). A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

mutual exclusion

A mechanism that ensures that only one of a set of cooperating processes can be executing in a *critical section* at a time by serializing access to shared resources. Three types of mechanisms are typically used to provide mutual exclusion—those that involve *busy-waiting*, those that involve *sleepy-waiting*, and a combination of the two.

NightProbe

A graphical user interface (GUI) developed by Concurrent that permits *real-time* recording, viewing, and modification of program data within one or more executing programs. It can be used during development and operation of applications, including simulations, data acquisition, and system control.

NightSim

A graphical user interface (GUI) to the *Frequency-Based Scheduler (FBS)* and *Performance Monitor (PM)* facilities.

NightStar Tools

A collection of development tools supplied by Concurrent that provide a graphical interface for scheduling, monitoring, debugging and analyzing run time behavior of *real-time* applications. The toolset includes the *NightSim* periodic scheduler, *NightProbe* data monitor, *NightTrace* event analyzer and *NightView* debugger.

NightTrace

A graphical tool developed by Concurrent used for analyzing the dynamic behavior of multiprocess and/or multiprocessor user applications and operating system activity. The NightTrace toolset consists of an interactive debugging and performance analysis tool, trace data collection daemons, and an Application Programming Interface (API).

NightView

A general-purpose, graphical source-level debugging and monitoring tool designed by Concurrent for *real-time* applications written in C, C++, and Fortran. NightView can monitor, debug, and patch multiple real-time processes running on multiple processors on the local system or on different targets with minimal intrusion.

nonblocking message operation

Not suspending execution if an attempt to send or receive a message is unsuccessful.

nonblocking semaphore operation

Not suspending execution while testing for a *semaphore* value.

PAM

Pluggable Authentication Module. A method that allows a system administrator to set access and *authentication* policies without having to separately recompile individual programs for such features. Under this scheme, a non-root user can be configured to run applications that require *privilege*s only root would normally be allowed.

PCI

(Peripheral Component Interface). A peripheral bus that provides a high-speed data path between the processor and peripheral devices like video cards, sound cards, network interface cards and modems. PCI provides “plug and play” capability, runs at 33MHz and 66 MHz and supports 32-bit and 64-bit data paths.

Performance Monitor (PM)

A facility that makes it possible to monitor use of the CPU by processes that are scheduled on a *frequency-based scheduler.* Values obtained assist in determining how to redistribute processes among processors for improved *load balancing* and processing efficiency. *NightSim* is a graphical interface to the Performance Monitor.

Pluggable Authentication Module (PAM)

See *PAM*.

POSIX

A standard specifying semantics and interfaces for a UNIX-like kernel interface, along with standards for user-space facilities. There is a core POSIX definition which must be supported by all POSIX-conforming operating systems, and several optional standards for specific facilities; e.g., POSIX message queues.

preemption

When a process that was running on a CPU is replaced by a process with a higher priority. Kernel preemption included in RedHawk Linux allows a lower priority process to be preempted, even if operating in kernel space, resulting in improved system response. Process preemption can be controlled through the use of *rescheduling variable*s.

priority inheritance

A mechanism that momentarily passes along the priority of one process to another as needed to avoid *priority inversion*.

priority inversion

When a higher-priority process is forced to wait for the execution of a lower-priority process.

privilege

A mechanism through which users or processes are allowed to perform sensitive operations or override system restrictions. Superuser possesses all (root) privileges. Through *capabilities*, privileges can be enabled or disabled for individual users and processes.

process

An instance of a program that is being executed. Each process has a unique PID, which is that process' entry in the kernel's process table.

process dispatch latency

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.

RCIM

Real-Time Clock and Interrupt Module. A multifunction PCI card designed by Concurrent for fully deterministic event synchronization in multiple CPU applications. The RCIM includes a synchronized clock, multiple programmable real-time clocks, and multiple input and output external interrupt lines. Interrupts can be shared (distributed) across interconnected systems using an RCIM chain.

real-time

Responding to a real-world event and completing the processing required to handle that 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. RedHawk Linux is a true real-time operating system (RTOS) because it can guarantee a certain capability within a specified time constraint.

rescheduling variable

A data structure, allocated on a per-process basis by the application, that controls a single process' vulnerability to rescheduling.

robust mutex

A *mutex* that gives applications a chance to recover if one of the application's threads dies while holding the mutex.

RPM

RPM Package Manager. A management system of tools, databases and libraries used for installing, uninstalling, verifying, querying, and updating computer software packages. See the **`rpm(8)`** man page for complete information.

semaphore

A location in memory whose value can be tested and set by more than one process. A semaphore is 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. RedHawk Linux provides *POSIX* counting semaphores that provide a simple interface to achieve the fastest performance, and *System V* semaphores that provide 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).

shared memory

Memory accessible through more than one process' virtual address map. Using shared memory, processes can exchange data more quickly than by reading and writing using the regular operating system services. RedHawk Linux includes standardized shared memory interfaces derived from *System V* as well as *POSIX*.

shielded CPU

A CPU that is responsible for running high-priority tasks that are protected from the unpredictable processing associated with interrupts and system daemons. Each CPU in a RedHawk Linux system can be individually shielded from background processes, interrupts and/or the local timer interrupt.

shielded CPU model

A model whereby 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.

shielded processor

See *shielded CPU*.

sleepy-wait

A method of *mutual exclusion* such as a *semaphore* that puts a process to sleep if it attempts to obtain a lock that is currently in a locked state

SMP

Symmetric multi-processing. A method of computing which uses two or more processors managed by one operating system, often sharing the same memory and having equal access to input/output devices. Application programs may run on any or all processors in a system.

softirq

A method by which the execution of a function can be delayed until the next available "safe point." Instead of invoking the function, a "trigger" that causes it to be invoked at the next safe point is used instead. A safe point is any time the kernel is not servicing a hardware or software interrupt and is not running with interrupts blocked.

spin lock

A *busy-wait* method of ensuring *mutual exclusion* for a resource. Tasks waiting on a spin lock sit in a busy loop until the spin lock becomes available.

System V

A standard for *interprocess communication (IPC)* objects supported by many UNIX-like systems, including Linux and System V systems. System V IPC objects are of three kinds: System V *message queues*, *semaphore* sets, and *shared memory* segments.

tasklet

A software interrupt routine running when the software interrupt is received at a return to user space or after a hardware interrupt. Tasklets do not run concurrently on multiple CPUs, but are dynamically allocatable.

TLB

Translation Look-aside Buffer. A table used in a virtual memory system, that lists the physical address page number associated with each virtual address page number. A TLB is used in conjunction with a cache whose tags are based on virtual addresses. The virtual address is presented simultaneously to the TLB and to the cache so that cache access and the virtual-to-physical address translation can proceed in parallel

trace event

Logged information for a point of interest (trace point) in an application's source code or in the kernel that can be examined by the *NightTrace* tool for debugging and performance analysis.

work queues

A method of deferred execution in addition to *softirqs* and *tasklets*, but unlike those forms, Linux processes work queues in the process context of kernel daemons and therefore are capable of sleeping.

Index

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

RedHawk™ Linux®

User

User’s Guide

0898004