REACT™ Real-Time for Linux®
Programmer's Guide

009–4746–009

New Features in this Guide

This revision contains the following:

- Information about finding the CPU-to-core numbering scheme in the `/proc/cpuinfo` file; see "Diagnostic Tools" on page 151.
- Minor editorial corrections throughout.

Record of Revision

Version	Description
001	February 2005 Original publication to support REACT real-time for Linux 4.0
002	July 2005 Revision to support REACT real-time for Linux 4.2
003	December 2005 Revision to support REACT real-time for Linux 4.3
004	July 2006 Revision to support REACT real-time for Linux 5.0
005	February 2007 Revision to support REACT real-time for Linux 5.1
006	June 2007 Revision to support REACT real-time for Linux 5.2
007	September 2007 Revision to support REACT real-time for Linux 5.3
008	December 2007 Revision to support REACT real-time for Linux 5.4
009	March 2008 Revision to support REACT real-time for Linux 5.5

Contents

About This Guide . . . . . **xxiii**
Audience xxiii
What This Guide Contains xxiii
Related Publications and Sites xxv
Conventions xxvi
Obtaining Publications xxvi
Reader Comments xxvi

1. Introduction **1**
Real-Time Programs 1
Real-Time Applications 1
Simulators and Stimulators 2
Aircraft Simulators 3
Ground Vehicle Simulators 3
Plant Control Simulators 3
Virtual Reality Simulators 3
Hardware-in-the-Loop Simulators 4
Control Law Processor Stimulator 4
Wave Tank Stimulator 4
Data Collection Systems 4
Process Control Systems 5
REACT Real-Time for Linux 6
REACT Installation Requirements 7

2. Linux and REACT Support for Real–Time Programs **9**

Kernel Facilities 9
Special Scheduling Disciplines 9
Virtual Memory Locking 10
Processes Mapping and CPUs 10
Interrupt Distribution Control 11
Frame Scheduler 11
Clocks and Timers 12
Clocks on Altix ia64 12
Direct RTC Access for Altix ia64 14
ITC Register Access for Altix ia64 14
SGI High-Resolution POSIX Timers 14
Requirements for Altix XE Use of High-Resolution POSIX Timers 15
Using SGI High-resolution POSIX Timers on Altix and Altix XE Servers 16
Interchassis Communication 17
Socket Programming 17
Message-Passing Interface (MPI) 17

3. External Interrupts 19

Abstraction Layer 19
`sysfs` Attribute Files 20
The `/dev/extint#` Device 22
Counting Interrupts 22
Waiting for Interrupts 22
Exclusively Accessing a Device 23
Low-level Driver Interface 26
Driver Registration 26
Implementation Functions 27
When an External Interrupt Occurs 30

Driver Deregistration . . . 31
Interrupt Notification Interface . . . 31
Callout Mechanism . . . 31
Callout Registration . . . 32
Callout Deregistration . . . 33
Making Use of Unsupported Hardware Device Capabilities . . . 33
Low-level Driver Template . . . 34
Example: SGI IOC4 PCI Device . . . 34
Multiple Independent Drivers . . . 34
External Interrupt Output . . . 36
External Interrupt Ingest . . . 38
Physical Interfaces . . . 38

4. CPU Workload . . . 41
Using Priorities and Scheduling Queues . . . 41
Scheduling Concepts . . . 41
Timer Interrupts . . . 42
Real-Time Priority Band . . . 42
`kbar` Kernel Barrier Facility for Altix ia64 Servers . . . 43
Setting Pthread Priority . . . 46
Controlling Kernel and User Threads . . . 46
Minimizing Overhead Work . . . 47
Avoid the Clock Processor (CPU 0) . . . 47
Reduce the System Flush Duration . . . 48
Redirect Interrupts . . . 48
Select the Console Node for SAL Console Driver Interrupt for Altix ia64 Servers . . . 49
Restrict, Isolate, and Shield CPUs . . . 49

Restricting a CPU from Scheduled Work 50
Isolating a CPU from Scheduler Load Balancing 50
Shielding a CPU from Timer Interrupts 50
Avoid Kernel Module Insertion and Removal 52
Avoid Filesystem Mounts 52
Understanding Interrupt Response Time 52
Maximum Response Time Guarantee 52
Components of Interrupt Response Time 53
Hardware Latency 54
Software Latency 54
Kernel Critical Sections 54
Interrupt Threads Dispatch 55
Device Service Time 55
Interrupt Service Routines 55
User Threads Dispatch 55
Mode Switch 55
Minimizing Interrupt Response Time 56

5. Using the Frame Scheduler 57

Frame Scheduler Concepts 57
Frame Scheduler Basics 58
Thread Programming Model 59
Frame Scheduling 59
Controller Thread 61
Frame Scheduler API 62
Interrupt Information Templates 63
Library Interface for C Programs 64
Thread Execution 66

Scheduling Within a Minor Frame . . . 67
Scheduler Flags `frs_run` and `frs_yield` . . . 67
Detecting Overrun and Underrun . . . 68
Estimating Available Time . . . 68
Synchronizing Multiple Schedulers . . . 69
Starting a Single Scheduler . . . 69
Starting Multiple Schedulers . . . 70
Pausing Frame Schedulers . . . 71
Managing Activity Threads . . . 71
Selecting a Time Base . . . 72
High-Resolution Timer . . . 73
External Interrupts as a Time Base . . . 73
Using the Scheduling Disciplines . . . 74
Real-Time Discipline . . . 74
Background Discipline . . . 75
Underrunable Discipline . . . 76
Overrunnable Discipline . . . 76
Continuable Discipline . . . 76
Using Multiple Consecutive Minor Frames . . . 77
Designing an Application for the Frame Scheduler . . . 78
Preparing the System . . . 79
Implementing a Single Frame Scheduler . . . 80
Implementing Synchronized Schedulers . . . 81
Synchronized Scheduler Concepts . . . 82
Master Controller Thread . . . 82
Slave Controller Thread . . . 83
Handling Frame Scheduler Exceptions . . . 84

Exception Types 84
Exception Handling Policies 85
Injecting a Repeat Frame 85
Extending the Current Frame 85
Dealing With Multiple Exceptions 86
Setting Exception Policies 86
Querying Counts of Exceptions 87
Using Signals Under the Frame Scheduler 89
Handling Signals in the Frame Scheduler Controller 89
Handling Signals in an Activity Thread 90
Setting Frame Scheduler Signals 90
Handling a Sequence Error 91
Using Timers with the Frame Scheduler 92

6. Disk I/O Optimization 93
Memory-Mapped I/O 93
Asynchronous I/O 93
Conventional Synchronous I/O 94
Asynchronous I/O Basics 94

7. PCI Devices 95

8. User-Level Interrupts 99
Overview of ULI 100
Restrictions on the ULI Handler 101
Planning for Concurrency: Declaring Global Variables 103
Using Multiple Devices 103
Setting Up ULI 103

Opening the Device Special File 104
Locking the Program Address Space 104
Registering the Interrupt Handler 105
Registering a Per-IRQ Handler 106
Interacting With the Handler 106
Achieving Mutual Exclusion 107

9. REACT System Configuration 109

Configuration Overview 109
`reactconf.pl` Command-Line Summary 111
Generating a REACT System Configuration 113
Interactive Mode 113
Noninteractive Mode 117
Example `reactconf.pl` Interactive-Mode Output 119
Running with REACT Enabled 123
Running a Process on a Restricted CPU 123
Dynamically Changing the Configuration 124
Granting Users Access to Select REACT Features 125
Linux Capabilities and the SGI ProPack Captools Package 126
Capabilities Provided with Standard Linux 126
Enhanced Capabilities Provided with the SGI ProPack Captools Package 127
Captools Policy Specification and Usage 132
Captools Usage Example for REACT 134
Viewing the Current REACT Configuration 135
Disabling and Enabling REACT 135
Disabling REACT 135
Enabling REACT 136

10. SGI Linux Trace 137
Overview of SGI Linux Trace 137
Installing SGI Linux Trace 138
Gathering Trace Data 139
Invoking the `tracevisualizer` GUI 139
Recording Events 140
Trace Files and Event Types 141
Exiting from the `tracedaemon` Command Line 146
Monitoring Trace Events 146
Opening a Trace File 146
Zooming In On An Event 147
Changing the Time Frame and Display 147
Seeing Process Details 148
Seeing All Event Trace Details 148
Filtering Events Based on CPU 148
Exiting from the `tracevisualizer` GUI 148
Removing SGI Linux Trace 148

11. Troubleshooting 151
Diagnostic Tools 151
Problem Removing `/rtcpus` 154
Problem Removing `/boot` Cpuset 155

Appendix A. Example Application 157
Setting Up External Interrupts 159
Building and Loading the Kernel Module 160
Building the User-Space Application 161

Running the Sample Application 161
Matrix Multiply Mode Examples 162
Netlink Socket Benchmark Mode Examples 163
`set_affinity` code 164

Appendix B. High-Resolution Timer Example for Altix ia64 Servers . . . 167

Appendix C. Sample User-Level Interrupt Programs 173
`uli_sample` Sample Program 173
`uli_ei` Sample Program 174

Appendix D. Reading MAC Addresses Sample Program 175

Appendix E. IRIX Differences 179

Glossary 181

Index 191

Figures

Figure 3-1 Output and Input Connectors for the Internal Driver Circuit 39
Figure 4-1 `kbar`(3) Kernel Barrier Facility 45
Figure 4-2 Components of Interrupt Response Time 53
Figure 5-1 Major and Minor Frames 60
Figure 8-1 ULI Functional Overview 99
Figure 8-2 ULI Handler Functions 102
Figure A-1 Example Work Flow 159

Tables

Table 3-1 Register Format 37
Table 5-1 Frame Scheduler Operations 64
Table 5-2 Signal Numbers Passed in `frs_signal_info_t` 90
Table 8-1 Common Arguments for Registration Functions 100
Table 10-1 Trace Events that are Recorded 143

Examples

Example 3-1 Searching for an Unused External Interrupt Device 24

Example 5-1 Skeleton of an Activity Thread 66

Example 5-2 Alternate Skeleton of an Activity Thread 67

Example 5-3 Function to Set `INJECTFRAME` Exception Policy 86

Example 5-4 Function to Set `STRETCH` Exception Policy 87

Example 5-5 Function to Return a Sum of Exception Counts (pthread Model) 88

Example 5-6 Function to Set Frame Scheduler Signals 91

Example 5-7 Minimal Activity Process as a Timer 92

Example 9-1 `reactconf.pl` Output 119

Example B-1 High-Resolution Timer 167

Example D-1 Reading MAC Addresses on an Altix Server 175

About This Guide

A *real-time program* is one that must maintain a fixed timing relationship to external hardware. In order to respond to the hardware quickly and reliably, a real-time program must have special support from the system software and hardware. This guide describes the facilities of REACT real-time for Linux.

Audience

This guide is written for real-time programmers. You are assumed to be:

- An expert in the C programming language
- Knowledgeable about the hardware interfaces used by your real-time program
- Familiar with system-programming concepts such as interrupts, device drivers, multiprogramming, and semaphores

You are not assumed to be an expert in Linux system programming, although you do need to be familiar with Linux as an environment for developing software.

What This Guide Contains

This guide contains the following:

- Chapter 1, "Introduction" on page 1 describes the important classes of real-time programs and applications, summarizes the features that REACT provides, and lists installation requirements
- Chapter 2, "Linux and REACT Support for Real–Time Programs" on page 9 provides an overview of how Linux and REACT support real-time programs
- Chapter 3, "External Interrupts" on page 19 discusses the external interrupts feature and, as an example, the SGI IOC4 PCI device.
- Chapter 4, "CPU Workload" on page 41 describes how you can isolate a CPU and dedicate almost all of its cycles to your program's use

- Chapter 5, "Using the Frame Scheduler" on page 57 describes how to structure a real-time program as a family of independent, cooperating activities, running on multiple CPUs, scheduled in sequence at the frame rate of the application
- Chapter 6, "Disk I/O Optimization" on page 93 describes how to set up disk I/O to meet real-time constraints, including the use of memory-mapped and asynchronous I/O
- Chapter 7, "PCI Devices" on page 95 discusses the Linux PCI interface
- Chapter 8, "User-Level Interrupts" on page 99 discusses the facility that is intended to simplify the creation of device drivers for unsupported devices
- Chapter 9, "REACT System Configuration" on page 109 explains how to configure restricted and isolated CPUs on a system running the REACT real-time for Linux product by using the `/usr/sbin/reactconf.pl` script
- Chapter 10, "SGI Linux Trace" on page 137 discusses the feature that generates traces for kernel events such as interrupt handling, scheduling, and system calls.
- Chapter 11, "Troubleshooting" on page 151 discusses diagnostic tools that apply to real-time applications and common problems
- Appendix A, "Example Application" on page 157 provides excerpts of application modules to be used with REACT
- Appendix B, "High-Resolution Timer Example for Altix ia64 Servers" on page 167 demonstrates the use of SGI high-resolution timers
- Appendix C, "Sample User-Level Interrupt Programs" on page 173 contains a sample program that shows how user-level interrupts are used
- Appendix D, "Reading MAC Addresses Sample Program" on page 175 provides a sample program for reading the MAC address from an Ethernet card
- Appendix E, "IRIX Differences" on page 179 summarizes some differences between REACT for IRIX and REACT for Linux

Related Publications and Sites

The following may be useful:

- Available from the online SGI Technical Publications Library:
  - The user guide for your SGI system
  - *Linux Configuration and Operations Guide*
  - *Linux Device Driver Programmer's Guide-Porting to SGI Altix Systems*
  - *SGI L1 and L2 Controller Software User's Guide*
  - *SGI ProPack 5 for Linux Service Pack 5 Start Here*
  - *SUSE LINUX Enterprise Server for SGI Altix Systems*
  - *Porting IRIX Applications to SGI Altix Platforms: SGI ProPack for Linux*
  - *The Linux Programmer's Guide* (Sven Goldt, Sven van der Meer, Scott Burkett, Matt Welsh)
  - *The Linux Kernel* (David A Rusling)
  - *Linux Kernel Module Programming Guide* (Ori Pomerantz)
- *Linux Device Drivers*, third edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, February 2005 (ISBN: 0-596-00590-3):

  http://www.oreilly.com/catalog/linuxdrive3/

For more information about the SGI Altix XE x86_64 and SGI Altix ia64 series, see the following sites:

- http://www.sgi.com/products/servers/altix/xe
- http://www.sgi.com/products/servers/altix
- http://www.sgi.com/products/servers/altix/350

Conventions

The following conventions are used throughout this document:

Convention	Meaning
[ ]	Brackets enclose optional portions of a command or directive line.
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
...	Ellipses indicate that a preceding element can be repeated.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
variable	Italic typeface denotes variable entries and words or concepts being defined.

Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- You can view man pages by typing `man` *title* at a command line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the

front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  SGI
  Technical Publications
  1140 East Arques Avenue
  Sunnyvale, CA 94085–4602

SGI values your comments and will respond to them promptly.

Introduction

This chapter discusses the following:

- "Real-Time Programs" on page 1
- "Real-Time Applications" on page 1
- "REACT Real-Time for Linux" on page 6
- "REACT Installation Requirements" on page 7

Real-Time Programs

A *real-time program* is any program that must maintain a fixed, absolute timing relationship with an external hardware device. A *hard real-time program* is a program that experiences a catastrophic error if it misses a deadline. A *firm real-time program* is a program that experiences a significant error if it misses a deadline but is able to recover from the error and can continue to execute. A *soft real-time program* is a program that can occasionally miss a deadline with only minor adverse effects.

A *normal-time program* is a correct program when it produces the correct output, no matter how long that takes. Normal-time programs do not require a fixed timing relationship to external devices. You can specify performance goals for a normal-time program (such as “respond in at most 2 seconds to 90% of all transactions”), but if the program does not meet the goals, it is merely slow, not incorrect.

Real-Time Applications

The following are examples of real-time applications:

- "Simulators and Stimulators" on page 2
- "Data Collection Systems" on page 4
- "Process Control Systems" on page 5

Simulators and Stimulators

A *simulator* or a *stimulator* maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model. It must process inputs in real time in order to be accurate. The difference between them is that a simulator provides visual output while a stimulator provides nonvisual output. SGI systems are well-suited to programming many kinds of simulators and stimulators.

Simulators and stimulators have the following components:

- An internal model of the world, or part of it; for example, a model of a vehicle traveling through a specific geography, or a model of the physical state of a nuclear power plant.
- External devices to supply control inputs; for example, a steering wheel, a joystick, or simulated knobs and dials. (This does not apply to all stimulators.)
- An operator (or hardware under test) that closes the feedback loop by moving the controls in response to what is shown on the display. A *feedback loop* provides input to the system in response to output from the system. (This does not apply to all stimulators.)

Simulators also have the external devices to display the state of the model; for example, video displays, audio speakers, or simulated instrument panels.

The real-time requirements vary depending on the nature of these components. The following are key performance requirements:

- *Frame rate* is the rate at which the simulator updates the display, whether or not the simulator displays its model on a video screen. Frame rate is given in cycles per second (*hertz*, abbreviated *Hz*). Typical frame rates run from 15 Hz to 60 Hz, although rates higher and lower than these are used in special situations.

  The inverse of frame rate is *frame interval*. For example, a frame rate of 60 Hz implies a frame interval of 1/60 second, or 16.67 milliseconds (ms). To maintain a frame rate of 60 Hz, a simulator must update its model and prepare a new display in less than 16.67 ms.
- *Transport delay* is the number of frames that elapses before a control motion is reflected in the display. When the transport delay is too long, the operator perceives the simulation as sluggish or unrealistic. If a visual display in a simulator lags behind control inputs, a human operator can become physically ill. In the case where the operator is physical hardware, excessive transport delay can cause the control loop to become unstable.

Aircraft Simulators

Simulators for real or hypothetical aircraft or spacecraft typically require frame rates of 30 Hz to 120 Hz and transport delays of 1 or 2 frames. There can be several analogue control inputs and possibly many digital control inputs (simulated switches and circuit breakers, for example). There are often multiple video display outputs (one each for the left, forward, and right "windows") and possibly special hardware to shake or tilt the "cockpit." The display in the "windows" must have a convincing level of detail.

Ground Vehicle Simulators

Simulators for automobiles, tanks, and heavy equipment have been built with SGI systems. Frame rates and transport delays are similar to those for aircraft simulators. However, there is a smaller world of simulated "geography" to maintain in the model. Also, the viewpoint of the display changes more slowly, and through smaller angles, than the viewpoint from an aircraft simulator. These factors can make it somewhat simpler for a ground vehicle simulator to update its display.

Plant Control Simulators

A simulator can be used to train the operators of an industrial plant such as a nuclear or conventional power-generation plant. Power-plant simulators have been built using SGI systems.

The frame rate of a plant control simulator can be as low as 1 or 2 Hz. However, the number of control inputs (knobs, dials, valves, and so on) can be very large. Special hardware may be required to attach the control inputs and multiplex them onto the PCI bus. Also, the number of display outputs (simulated gauges, charts, warning lights, and so on) can be very large and may also require custom hardware to interface them to the computer.

Virtual Reality Simulators

A virtual reality simulator aims to give its operator a sense of presence in a computer-generated world. A difference between a vehicle simulator and a virtual reality simulator is that the vehicle simulator strives for an exact model of the laws of physics, while a virtual reality simulator typically does not.

Usually the operator can see only the simulated display and has no other visual referents. Because of this, the frame rate must be high enough to give smooth, nonflickering animation; any perceptible transport delay can cause nausea and

disorientation. However, the virtual world is not required (or expected) to look like the real world, so the simulator may be able to do less work to prepare the display than does a vehicle simulator

SGI systems, with their excellent graphic and audio capabilities, are well suited to building virtual reality applications.

Hardware-in-the-Loop Simulators

The operator of a simulator need not be a person. In a *hardware-in-the-loop* (HWIL) simulator, the human operator is replaced by physical hardware such as an aircraft autopilot or a missile guidance computer. The inputs to the system under test are the simulator's output. The output signals of the system under test are the simulator's control inputs.

Depending on the hardware being exercised, the simulator may have to maintain a very high frame rate, up to several thousand Hz. SGI systems are excellent choices for HWIL simulators.

Control Law Processor Stimulator

An example of a *control law processor* is one that simulates the effects of Newton's law on an aircraft flying through the air. When the rudder is turned to the left, the information that the rudder had turned, the velocity, and the direction is fed into the control law processor. The processor calculates and returns a response that represents the physics of motion. The pilot in the simulator cockpit will feel the response and the instruments will show the response. However, a human did not actually interact directly with the processor; it was a machine-to-machine interaction.

Wave Tank Stimulator

A wave tank simulates waves hitting a ship model under test. The stimulator must "push" the water at a certain rhythm to keep the waves going. An operator may adjust the frequency and amplitude of the waves, or it could run on a preprogrammed cycle.

Data Collection Systems

A *data collection system* receives input from reporting devices (such as telemetry receivers) and stores the data. It may be required to process, reduce, analyze, or

compress the data before storing it. It must respond in real time to avoid losing data. SGI systems are suited to many data collection tasks.

A data collection system has the following major parts:

- Sources of data such as telemetry (the PCI bus, serial ports, SCSI devices, and other device types can be used).
- A repository for the data. This can be a raw device (such as a tape), a disk file, or a database system.
- Rules for processing. The data collection system might be asked only to buffer the data and copy it to disk. Or it might be expected to compress the data, smooth it, sample it, or filter it for noise.
- Optionally, a display. The data collection system may be required to display the status of the system or to display a summary or sample of the data. The display is typically not required to maintain a particular frame rate, however.

The first requirement on a data collection system is imposed by the *peak data rate* of the combined data sources. The system must be able to receive data at this peak rate without an *overrun*; that is, without losing data because it could not read the data as fast as it arrived.

The second requirement is that the system must be able to process and write the data to the repository at the *average data rate* of the combined sources. Writing can proceed at the average rate as long as there is enough memory to buffer short bursts at the peak rate.

You might specify a desired frame rate for updating the display of the data. However, there is usually no real-time requirement on display rate for a data collection system. That is, the system is correct as long as it receives and stores all data, even if the display is updated slowly.

Process Control Systems

A *process control system* monitors the state of an industrial process and constantly adjusts it for efficient, safe operation. It must respond in real time to avoid waste, damage, or hazardous operating conditions.

An example of a process control system would be a power plant monitoring and control system required to do the following:

- Monitor a stream of data from sensors
- Recognize a dangerous situation has occurred
- Visualize the key data, such as by highlighting representations of down physical equipment in red and sending audible alarms

The danger must be recognized, flagged, and responded to quickly in order for corrective action to be taken appropriately. This entails a real-time system. SGI systems are suited for many process control applications.

REACT Real-Time for Linux

REACT provides the following:

- SGI Linux Trace debug kernel to provide trace information for analyzing the impact of kernel operations on application performance.
- The `reactconf.pl` and `react-permissions.pl` scripts to help you easily generate and configure a real-time system and facilitate capability access and manipulation, going beyond that of standard Linux. See Chapter 9, "REACT System Configuration" on page 109.
- User-level interrupts to allow you to handle hardware interrupts from a user process.
- A frame scheduler that makes it easier to structure a real-time program as a family of independent, cooperating activities that are running on multiple CPUs and are scheduled in sequence at the frame rate of the application.

  Note: *CPU* refers to cores (not sockets).

- The `kbar`(3) kernel barrier facility, which provides for the fast wake-up of many blocked user threads.

Note: Real–time programs using REACT should be written in the C language, which is the most common language for system programming on Linux.

REACT Installation Requirements

The following RPMs are used for REACT:

- **Required** RPMs:
  - Cpuset and bitmask:

    ```
    cpuset-utils
    libbitmask
    libcpuset
    ```

  - External interrupts (see Chapter 3, "External Interrupts" on page 19):

    ```
    extint
    sgi-extint-kmp-*
    ```

  - REACT configuration (see Chapter 9, "REACT System Configuration" on page 109):

    ```
    react-configuration
    ```

- **Optional** RPMs:
  - Kernel barrier facility (see "`kbar` Kernel Barrier Facility for Altix ia64 Servers" on page 43):

    ```
    kbar
    sgi-kbar-kmp-*
    ```

  - IRIX compatibility and `sgi-shield` (see "Shielding a CPU from Timer Interrupts" on page 50):

    ```
    libsgirt
    sgi-shield-kmp-*
    ```

  - Frame scheduler (see Chapter 5, "Using the Frame Scheduler" on page 57):

    ```
    frs
    sgi-frs-kmp-*
    ```

  - User-level interrupts (see Chapter 8, "User-Level Interrupts" on page 99):

    ```
    uli
    sgi-uli-kmp-*
    ```

– SGI Linux Trace (see Chapter 10, "SGI Linux Trace" on page 137):

```
TraceToolkit
kernel-sltdebug
kernel-sltdebug-source
```

For installation instructions, see the *SGI ProPack 5 for Linux Service Pack 5 Start Here*.

Linux and REACT Support for Real–Time Programs

This chapter provides an overview of how Linux and REACT support real-time programs:

- "Kernel Facilities" on page 9
- "Frame Scheduler" on page 11
- "Clocks and Timers" on page 12
- "Interchassis Communication" on page 17

Kernel Facilities

The Linux kernel has a number of features that are valuable when you are designing a real-time program. These are described in the following sections:

- "Special Scheduling Disciplines" on page 9
- "Virtual Memory Locking" on page 10
- "Processes Mapping and CPUs" on page 10
- "Interrupt Distribution Control" on page 11

Special Scheduling Disciplines

The default Linux scheduling algorithm is designed to ensure fairness among time-shared users. The priorities of time-shared threads are largely determined by the following:

- Their `nice` value
- The degree to which they are CPU-bound versus I/O-bound

While a time-share scheduler is effective at scheduling most standard applications, it is not suitable for real time. For deterministic scheduling, Linux provides the following POSIX real-time policies:

- First-in-first-out
- Round-robin

These policies share a real-time priority band consisting of 99 priorities. For more information about scheduling, see "Real-Time Priority Band" on page 42 and the `sched_setscheduler`(2) man page.

Virtual Memory Locking

Linux allows a task to lock all or part of its virtual memory into physical memory so that it cannot be paged out and so that a page fault cannot occur while it is running.

Memory locking prevents unpredictable delays caused by paging, but the locked memory is not available for the address spaces of other tasks. The system must have enough physical memory to hold the locked address space and space for a minimum of other activities.

Examples of system calls used to lock memory are `mlock`(2) and `mlockall`(2).

Processes Mapping and CPUs

Normally, Linux tries to keep all CPUs busy, dispatching the next ready process to the next available CPU. Because the number of ready processes changes continuously, dispatching is a random process. A normal process cannot predict how often or when it will next be able to run. For normal programs, this does not matter as long as each process continues to run at a satisfactory average rate. However, real-time processes cannot tolerate this unpredictability. To reduce it, you can dedicate one or more CPUs to real-time work by using the following steps:

1. Restrict one or more CPUs from normal scheduling so that they can run only the processes that are specifically assigned to them.
2. Isolate one or more CPUs from the effects of scheduler load-balancing.
3. Assign one or more processes to run on the restricted CPUs.

A process on a dedicated CPU runs when it needs to run, delayed only by interrupt service and by kernel scheduling cycles.

Interrupt Distribution Control

In normal operations, a CPU receives frequent interrupts:

- I/O interrupts from devices attached to, or near, the CPU
- Timer interrupts that occur on every CPU
- Console interrupts that occur on the CPU servicing the system console

These interrupts can make the execution time of a process unpredictable. I/O interrupt control is done by `/proc` filesystem manipulation. For more information on controlling I/O interrupts, see "Redirect Interrupts" on page 48.

You can minimize console interrupt effects with proper real-time thread placement. You should not run time-critical threads on the CPU that is servicing the system console.

You can see where console interrupts are being serviced by examining the `/proc/interrupts` file. For example:

```
[root@linux root]# head -1 /proc/interrupts && grep 'SAL console' /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
233:          0      12498          0          0          SN hub  SAL console driver
```

The above shows that 12,498 console driver interrupts have been serviced by CPU 1. In this case, CPUs 2 and 3 would be much better choices for running time-critical threads because they are not servicing console interrupts.

Timer processing is always performed on the CPU from which the timer was started, such as by executing a POSIX `timer_settime()` call. You can avoid the effects of timer processing by not allowing execution of any threads other than time-critical threads on CPUs that have been designated as such. If your time-critical threads start any timers, the timer processing will result in additional latency when the timeout occurs.

Frame Scheduler

Many real-time programs must sustain a fixed frame rate. In such programs, the central design problem is that the program must complete certain activities during every frame interval.

The *frame scheduler* is a process execution manager that schedules activities on one or more CPUs in a predefined, cyclic order. The scheduling interval is determined by a repetitive time base, usually a hardware interrupt.

The frame scheduler makes it easy to organize a real-time program as a set of independent, cooperating threads. You concentrate on designing the activities and implementing them as threads in a clean, structured way. It is relatively easy to change the number of activities, their sequence, or the number of CPUs, even late in the project. For more information, see Chapter 5, "Using the Frame Scheduler" on page 57.

Clocks and Timers

This section discusses the following:

- "Clocks on Altix ia64" on page 12
- "Direct RTC Access for Altix ia64" on page 14
- "ITC Register Access for Altix ia64" on page 14
- "SGI High-Resolution POSIX Timers" on page 14

Clocks on Altix ia64

SGI Altix ia64 systems provide a systemwide clock called a *real-time clock* (RTC) that is accessible locally on every node. The RTC provides a raw time source that is incremented in 50-ns intervals. The RTC is 55 bits wide, which ensures that it will not wrap around zero unless the system has been running for more than half a century. RTC values are mapped into the local memory of each node. Multiple nodes accessing the RTC value will not reduce the performance of the clock functions. RTCs are synchronized among all of the nodes in an SGI system using a special pin on the NUMAlink cable.

The RTC is the basis for system time, which may be obtained via the `clock_gettime` function call that is implemented in conformance with the POSIX standard. `clock_gettime` takes an argument that describes which clock is wanted.

The following clock values are typically used:

- `CLOCK_REALTIME` is the actual current time that you would obtain from any ordinary clock. However, `CLOCK_REALTIME` is set during startup and may be corrected during the operation of the system. This implies that time differences observed by an application using `CLOCK_REALTIME` may be affected by the initial setting or the later correction of time (via `clock_settime`) and therefore may not accurately reflect time that has passed for the system.
- `CLOCK_MONOTONIC` starts at zero during bootup and is continually increasing. `CLOCK_MONOTONIC` will not be affected by time corrections and the initial time setup during boot. If you require a continually increasing time source that always reflects the real time that has passed for the system, use `CLOCK_MONOTONIC`.

The `clock_gettime` function is a *fastcall* version that was optimized in assembler and bypasses the context switch typically necessary for a full system call. SGI recommends that you use `clock_gettime` for all time needs.

Both `CLOCK_REALTIME` and `CLOCK_MONOTONIC` report a resolution via the `clock_getres()` function call that is in the range 1 ms through 10 ms, corresponding to the timer tick frequency in the range 1000 Hz through 100 Hz. The `clock_getres()` function call is POSIX compliant.

You can use either `CLOCK_REALTIME` or `CLOCK_MONOTONIC` to generate signals via the `timer_create` function.

Note: Linux can only deliver signals based on `CLOCK_REALTIME` and `CLOCK_MONOTONIC` in intervals of the timer tick. In order to fulfill the mandates of the POSIX standard, no higher resolution can be reported via `clock_getres()`. However, the actual resolution of both clocks is the full resolution of the RTC. Time can be measured in 50-ns intervals with both clocks, but it is not possible to generate signals with this accuracy using these clocks. For information on generating signals with greater accuracy than the timer tick allows, see "SGI High-Resolution POSIX Timers" on page 14.

To determine the tick frequency, use the `sysconf(_SC_CLK_TCK)` function. The `sysconf(_SC_CLK_TCK)` will always return the right value on SGI Altix systems.

Direct RTC Access for Altix ia64

In some situations, the overhead of the `clock_gettime` fastcall may be too high. In that case, direct memory-mapped access to the RTC counter is useful. (See the comments in `mmtimer.h`.)

Note: Measurements have shown that the code generated by a function written to obtain the RTC value and then calculate the nanoseconds that have passed is slower than the fastcall for `clock_gettime`. Direct use of the RTC is only advisable for timestamps.

Like `CLOCK_MONOTONIC`, the RTC counter is monotonically increasing from bootup and is not affected by setting the time.

ITC Register Access for Altix ia64

The Itanium processor provides a 64-bit counter incremented by the processor clock called the *interval time counter* (ITC). ITC register accesses are very fast compared to the RTC (which must retrieve a value from memory) and the ITC typically allows measurements of much smaller time intervals than the RTC.

However, the ITC is a local processor-based counter. The clock frequencies of processors in an SGI system are not synchronized and may be running at different frequencies. Therefore, if you want to measure an interval by using the ITC, you must ensure that the process is not migrating to a different processor. Furthermore, in order to obtain meaningful time information from differences in ITC values, you must know the local ITC frequency.

In contrast to the ITC register, the RTC counter is a global time source and is not subject to these limitations.

SGI High-Resolution POSIX Timers

You can use POSIX timers to generate signals at higher resolution than the system timer tick on SGI systems by using the SGI `CLOCK_SGI_CYCLE` clock ID. POSIX timers with the `CLOCK_SGI_CYCLE` clock ID on Altix ia64 systems use the RTC as a time source; on Altix XE x86_64 systems, they use the internal clock on the PCI-RT-Z external interrupt card. The REACT frame scheduler uses this hardware timer for its high-resolution timer solution. Frame scheduler applications cannot use POSIX high-resolution timers. For more information, see "Clocks on Altix ia64" on page 12..

Requirements for Altix XE Use of High-Resolution POSIX Timers

You must have at least one SGI PCI-RT-Z card installed in order to use the high-resolution POSIX timers (using `CLOCK_SGI_CYCLE`) on Altix XE x86_64 servers, because they are based off of the clock on the PCI-RT-Z card rather than hardware internal to the Altix XE server. You must do the following before using the high-resolution POSIX timers on an Altix XE:

1. Select which PCI-RT-Z card to use for high resolution timers by using the `dmesg`(8) command:

   ```
   [root@linux root]# dmesg | grep "IOC4.*PCI clock"
    IOC4 0000:08:03.0: PCI clock is 30 ns.
    IOC4 0000:0d:01.0: PCI clock is 15 ns.
   ```

   The output above shows two PCI-RT-Z cards and the PCI bus period for each card. In the above case, you can achieve higher resolution timers with the 15–ns card (`0000:0d:01.0`) than with the 30–ns card, but this does not mean that the timer resolution will be 15 nsec (see step 4).

   Alternatively, you can list the available PCI-RT-Z cards using the `lspci`(8) command:

```
[root@linux root]# lspci -D  | grep IOC4
0000:08:03.0 Co-processor: Silicon Graphics, Inc. IOC4 I/O controller (rev 3e)
0000:0d:01.0 Co-processor: Silicon Graphics, Inc. IOC4 I/O controller (rev 3e)
```

   Note: If neither of the above commands generates any output, ensure that the `ioc4` module is loaded by using the following command:

   ```
   [root@linux root]# lsmod | grep ioc4
   ```

2. Load the external interrupt modules by using the `modprobe`(8) command, specifying the card that you want to use for high-resolution timers:

   ```
   [root@linux root]# modprobe ioc4_extint timers=0000:0d:01.0
   ```

   Card `0000:0d:01.0` will now be unavailable for use by other applications such as external interrupts or the frame scheduler, and will not have a `/dev/extint*` device associated with it.

3. Load the Altix XE high-resolution timers module:

   ```
   [root@linux root]# modprobe xe-timers
   ```

4. View the timer resolution in the log buffer:

   ```
   [root@linux root]# dmesg | grep "SGI Altix XE Timer:"
   SGI Altix XE Timer: v1.0
   SGI Altix XE Timer: resolution 7834 nsec
   ```

   The above output shows the timer resolution you can expect from the chosen card is 7.834 usec.

For lowest latencies, redirect the `ioc4_extint` IRQ to a real-time CPU. For example, the following command lines will move IRQ 82 to CPU 2 (which has a CPU bitmask of 4):

```
[root@linux root]# cat /proc/interrupts | grep ioc4_extint
74:          0          0          0          0   IO-APIC-level  ioc4_extint
82:        242          0          0          0   IO-APIC-level  ioc4_extint
[root@linux root]# echo 4 > /proc/irq/82/smp_affinity
```

Using SGI High-resolution POSIX Timers on Altix and Altix XE Servers

To use SGI high-resolution POSIX timers, do the following:

1. Set up the timer by using `timer_create()` with the kernel value for `CLOCK_SGI_CYCLE` as the clock ID. For more information on `timer_create()`, see the `timer_create`(3p) man page.
2. Use the returned `timer_t` value for the other POSIX timer functions, such as `timer_settime()`.

The `CLOCK_SGI_CYCLE` clock ID is only valid with the `timer_create()` call. It is an invalid argument with other functions that accept clock IDs, such as `clock_gettime()`.

There is a limit of three outstanding high-resolution timers allowed at any given time per hardware node, which means per pair of CPUs on most systems. Attempting to set more than three timers per node will result in `timer_settime()` returning `EBUSY`.

For more information, see Appendix B, "High-Resolution Timer Example for Altix ia64 Servers" on page 167.

Interchassis Communication

This section discusses socket programming and the Message-Passing Interface (MPI). The performance of both sockets and MPI depends on the speed of the underlying network. The network that connects nodes (systems) in an array product has a very high bandwidth.

Socket Programming

One standard, portable way to connect processes in different computers is to use the BSD-compatible socket I/O interface. You can use sockets to communicate within the same machine, between machines on a local area network, or between machines on different continents.

Message-Passing Interface (MPI)

The Message-Passing Interface (MPI) is a standard architecture and programming interface for designing distributed applications. For the MPI standard, see:

http://www.mcs.anl.gov/mpi

SGI supports MPI in SGI ProPack.

External Interrupts

Real-time processes often require the ability to react to an external event. *External interrupts* are a way for a real-time process to receive a real-world external signal.

An external interrupt is generated via a signal applied to the external interrupt socket on systems supporting such a hardware feature, such as the IO9 card on an SGI Altix system, which has a 1/8-inch stereo-style jack into which a 0-5V signal can be fed. An exterior piece of hardware can assert this line, causing the card's IOC4 chip to generate an interrupt.

This chapter discusses the external interrupts feature and, as an example, the SGI IOC4 PCI device.

This section discusses the following:

- "Abstraction Layer" on page 19
- "Making Use of Unsupported Hardware Device Capabilities " on page 33
- "Low-level Driver Template" on page 34
- "Example: SGI IOC4 PCI Device" on page 34

Abstraction Layer

Various external interrupt hardware might implement the external interrupt feature in very different ways. The *external interrupt abstraction layer* provides the ability to determine when an interrupt occurs, count the number of interrupts, and select the source of those interrupts without depending upon specifics of the device being used.

This section discusses the following:

- "`sysfs` Attribute Files" on page 20
- "The `/dev/extint#` Device" on page 22
- "Low-level Driver Interface" on page 26
- "Interrupt Notification Interface" on page 31

sysfs Attribute Files

The external interrupt abstraction layer provides a character device and the following sysfs attribute files to control operation:

```
dev
mode
modelist
period
provider
quantum
source
sourcelist
```

Assuming the usual /sys mount-point for sysfs, the attribute files are located in the following directory:

```
/sys/class/extint/extint#/
```

The extint# component of the path is determined by the extint driver itself. The # character is replaced by a number (possibly multidigit), one per external interrupt device, beginning at 0. For example, if there were three devices, there would be three directories:

```
/sys/class/extint/extint0/
/sys/class/extint/extint1/
/sys/class/extint/extint2/
```

The attribute files are as follows:

File	Description
dev	Contains the major and minor number of the abstracted external interrupt device. If sysfs, hotplug, and udev are configured appropriately, udev will automatically create a /dev/extint# character special device file with this major and minor number. If you prefer, you may manually invoke mknod(1) to create the character special device file. Once created, this device file provides a counter that can be used by applications in a variety of ways. See "The /dev/extint# Device" on page 22.
mode	Contains the shape of the output signal for interrupt generation. For example, SGI's IOC4 chip can set the output to one of the following:

`high`, `low`, `pulse`, `strobe`, or `toggle`. For more information, see "External Interrupt Output" on page 36.

`modelist` Contains the list of available **valid** output modes, one per line. These strings are the legal valid values that can be written to the `mode` attribute. For more information, see "External Interrupt Output" on page 36.

Note: For the SGI IOC4 chip, there are other values that may be read from the `mode` attribute file that do not appear in `modelist`; these represent **invalid** hardware states. Only the modes present from the `modelist` are valid settings to be written to the mode attribute.

`period` Contains the repetition interval for periodic output signals (such as repeated strobes, automatic toggling). This period is specified in nanoseconds, and is written as a string. For more information, see "External Interrupt Output" on page 36.

`provider` Contains an indication of which low-level hardware driver and device instance are attached to the external interrupt interface. This string is free-form and is determined by the low-level driver. For example, the SGI IOC4 low-level driver will return a string of the form `ioc4_intout#`.

Note: The # value in `ioc4_intout#` is not necessarily the same number used for `extint#`, particularly if multiple different low-level drivers are in use (for example, IOC3 and IOC4).

`quantum` Contains the interval to which any writes of the `period` attribute will be rounded. Because external interrupt output hardware may not support nanosecond granularity for output periods, this attribute allows you to determine the supported granularity. The behavior of the interrupt output (when a value that is not a multiple of the quantum is written to the `period` attribute) is determined by the specific low-level external interrupt drive. However, generally the low-level driver should round to the nearest available quantum multiple. For example, suppose the `quantum` value is `7800`. If a value of `75000` was written into the `period` attribute, this would represent 9.6 quantums. The actual period will be rounded to 10 quantums, or 78000 nanoseconds. The actual period will be returned upon subsequent reads from the `period`

attribute. For more information, see "External Interrupt Output" on page 36.

`source` Contains the hardware source of interrupts. For example, SGI's IOC4 chip can trigger either from the external pin or from an internal loopback from its interrupt output section.

`sourcelist` Contains the list of available interrupt sources, one per line. These strings are the legal values that can be written to the `source` attribute file.

The `/dev/extint#` Device

This section discusses the operations that an application can perform with the read-only external interrupt device file `/dev/extint#`:

- "Counting Interrupts" on page 22
- "Waiting for Interrupts" on page 22
- "Exclusively Accessing a Device" on page 23

Counting Interrupts

A process may use `mmap`(2) to memory-map a single memory page from the external interrupt device file into the process' address space. At the beginning of this page, a counter of an `unsigned long` type is maintained. This counter is incremented with each external interrupt received by the device.

Alternatively, the `read`(2) system call returns a string representation of the counter's current value.

Waiting for Interrupts

The `poll`(2) and `select`(2) system calls allow a process to wait for an interrupt to trigger:

- `poll()` indicates whether an interrupt has occurred since the last `open`(2) or `read()` of the file
- `select()` blocks until the next interrupt is received

Exclusively Accessing a Device

The `flock`(2) system call with the options `LOCK_EX|LOCK_MAND` ensures exclusive write access to the device attribute files (for example, `/sys/class/extint/extint#/mode`).

Note: You must define the `_GNU_SOURCE` macro before including the header files in order to use the `LOCK_MAND` flag on the call to `flock`(2).

When this lock is obtained, only a process that has access to the corresponding file descriptor will be able to write to the attribute files for that device. Any other process that attempts a `write`(2) system call on one of these attribute files will fail with `errno` set to `EAGAIN`.

The `flock()` system call will block until there are no other processes that have the device file open and until no other `flock()` is active on the device. However, if `LOCK_NB` is passed to `flock()`, the call will fail and `errno` will be set to `EWOULDBLOCK`.

While a lock is in place, any attempt to call `open`(2) on the device will block. However, if `O_NONBLOCK` is passed to `open()`, the call will fail and `errno` will be set to `EWOULDBLOCK`.

To release the lock, call `flock()` with the `LOCK_UN` argument. The lock will also be automatically dropped when the last user of the corresponding file descriptor closes the file, including via a process exit. The lock will persist if the file descriptor is inherited across `fork`(2) or `exec`(2) system calls.

Note: You **must not** pass the `LOCK_MAND` flag along with the `LOCK_UN` flag. The `flock()` system call behavior is unspecified in this case.

Example 3-1 illustrates a method of searching for an unused external interrupt device that can be used exclusively by that program.

Example 3-1 Searching for an Unused External Interrupt Device

```
#define _GNU_SOURCE

#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <string.h>
int main(void) {
        char devfile[PATH_MAX];
        int i = 0;
        int fd;
        int found = 0;
        int status;

try_again:
        /* Search for free /dev/extint# device */
        while (i <= 255) {
                sprintf(devfile, "/dev/extint%d", i);
                i++;

                fd = open(devfile, O_RDONLY|O_NONBLOCK);
                if (fd >= 0) {
                        /* Found a unlocked device. */
                        found = 1;
                        break;
                }

                /* An error occurred.  Check why. */
                if (EWOULDBLOCK == errno) {
                        /* Found a locked device. */
                        printf("Tried %s, but it is locked.\n", devfile);
                }
                /* Some other type of error, just try next device.
                 * But don't complain about non-existent devices.
                 */
                if (ENOENT != errno)
```

```
                printf("Unexpected error opening %s: %s\n",
                        devfile, strerror(errno));
}

if (!found) {
        printf("Could not find unlocked extint device to use.\n");
        return 1;
}

/* Try locking this device to gain exclusive access. */
status = flock(fd, LOCK_EX|LOCK_MAND|LOCK_NB);
if (status != 0) {
        if (EWOULDBLOCK == errno) {
                /* The device was available, but another process
                 * has locked it between the time we opened it
                 * and made the flock() call.
                 */
                printf("Opened %s, but someone else locked it.\n",
                        devfile);
        } else {
                /* Some other error occurred. */
                printf("Unexpected error locking %s: %s\n",
                        devfile, strerror(errno));
        }
        /* Try the next device. */
        found = 0;
        close(fd);
        goto try_again;
}

/* Successfully gained exclusive use of device */
printf("Exclusive use of %s established.\n", devfile);

/* Application code begins... */

/* ... application code ends. */

/* Unlock and close external interrupt device */
flock(fd, LOCK_UN);
close(fd);
```

```
        /* Successful run */
        return 0;
}
```

Low-level Driver Interface

The `extint_properties` and `extint_device` structures provide the low-level driver interface to the abstraction layer driver. The `/usr/local/include/extint.h` file defines the structures and function prototypes.

This section discusses the following:

- "Driver Registration" on page 26
- "Implementation Functions" on page 27
- "When an External Interrupt Occurs" on page 30
- "Driver Deregistration" on page 31

Driver Registration

To register the low-level driver with the abstraction layer, use the following call:

```
struct extint_device*
extint_device_register(struct extint_properties *ep,
                       void *devdata);
```

The `ep` argument is a pointer to an `extint_properties` structure that specifies the particular low-level driver functions that the abstraction layer should call when reading/writing the attributes described in "`sysfs` Attribute Files" on page 20.

The `devdata` argument is an opaque pointer that is stored by the `extint` code. To retrieve or modify this value, use the following calls:

```
void* extint_get_devdata(const struct extint_device *ed);
void extint_set_devdata(struct extint_device *ed, void* devdata);
```

The low-level driver uses this value to determine which of multiple devices it is operating upon.

The return value is one of the following:

- A pointer to a `struct extint_device` (which should be saved for later interrupt notification and driver deregistration).
- A negative error value (in case of registration failure). The driver should be prepared to deal with such failures.

Implementation Functions

The `struct extint_properties` call table is as follows:

```
struct extint_properties {
        /* Owner module */
        struct module *owner;

        /* Get/set generation mode */
        ssize_t (*get_mode)(struct extint_device * ed, char *buf);
        ssize_t (*set_mode)(struct extint_device * ed, const char *buf,
                            size_t count);

        /* Get generation mode list */
        ssize_t (*get_modelist)(struct extint_device * ed, char *buf);

        /* Get/set generation period */
        unsigned long (*get_period)(struct extint_device * ed);
        ssize_t (*set_period)(struct extint_device * ed, unsigned long period);

        /* Get low-level provider name */
        ssize_t (*get_provider)(struct extint_device *ed, char *buf);

        /* Generation period quantum */
        unsigned long (*get_quantum)(struct extint_device * ed);

        /* Get/set ingest source */
        ssize_t (*get_source)(struct extint_device * ed, char *buf);
        ssize_t (*set_source)(struct extint_device * ed, const char *buf,
                              size_t count);

        /* Get ingest source list */
        ssize_t (*get_sourcelist)(struct extint_device * ed, char *buf);

        /* Arm/disarm timer */
        int64_t (*arm_timer)(struct extint_device * ed, int64_t ns, int when);
```

```
        void (*disarm_timer)(struct extint_device * ed);
};
```

Note: Additional fields not of interest to the low-level external interrupt driver may be present. You should include `/usr/local/include/extint.h` to acquire these structure definitions.

The `owner` value should be set to the module that contains the functions pointed to by the remaining structure members. The remaining functions implement low-level aspects of the abstraction layer attributes. They all take a pointer to the `struct extint_device` as was returned from the registration function. In all of these functions, you can retrieve the value passed as the `devdata` argument to the registration function by using the following call:

```
extint_get_devdata(ed);
```

You can update the value by using the following call:

```
extint_set_devdata(ed, newvalue);
```

Typically, this value is a pointer to driver-specific data for the individual device being operated upon. It may, for example, contain pointers to mapped PCI regions where control registers reside.

Field	Description
`owner`	Specifies the module that contains the functions pointed to by the remaining structure members.
`get_mode`	Writes the current `mode` attribute of the abstraction layer into the single-page-sized buffer passed as the second argument and returns the length of the written string.
`set_mode`	Reads the `mode` attribute of the abstraction layer as specified in the buffer (passed as the second argument and as sized by the third) and returns the number of characters consumed (or a negative error number in event of failure). It also causes the output mode to be set as requested.
`get_modelist`	Writes strings representing the available interrupt output generation modes into the single-page-sized buffer passed as the second argument, one mode per line. It returns the number of bytes written into this

	buffer. This implements the `modelist` attribute of the abstraction layer.
`get_period`	Returns an `unsigned long` that represents the current repetition period, in nanoseconds. This implements the `period` attribute of the abstraction layer.
`set_period`	Accepts an `unsigned long` as the new value for the repetition period, specified in nanoseconds, and returning either `0` or a negative error number indicating a failure. If the requested repetition period is not a value that can be exactly set into the underlying hardware, the driver is free to adjust the value as it sees fit, although typically it should round the value to the nearest available value. This implements the `period` attribute of the abstraction layer.
`get_provider`	Writes a human-readable string that identifies the low-level driver and a particular instance of a driven hardware device. For example, if the low-level driver provides its own additional device files for extra functionality not present in the abstraction layer, this routine might emit the name of the driver module and the names (or device numbers) of the low-level driver's own character special device files. This implements the `provider` attribute of the abstraction layer.
`get_quantum`	Returns an `unsigned long` that represents the granularity to which the interrupt output repetition period can be set, in nanoseconds. This implements the `quantum` attribute of the abstraction layer.
`get_source`	Writes the current interrupt input source into the single-page-sized buffer passed as the second argument and returns the length of the written string. This implements the `source` attribute of the abstraction layer.
`set_source`	Reads the source specified in the buffer (passed as the second argument and as sized by the third) and returns the number of characters consumed or a negative error number in event of failure. It also causes the input source to be selected as requested. This implements the `source` attribute of the abstraction layer.

`get_sourcelist`	Writes strings representing the available interrupt input sources into the single-page-sized buffer passed as the second argument, one source per line. It returns the number of bytes written into this buffer. This implements the `sourcelist` attribute of the abstraction layer.
`arm_timer`	Sets up the external interrupt device to generate an interrupt at a specified time. The time is specified in nanoseconds via the second argument. Whether the time is relative to the moment the function is called or absolute system time (as returned by `getnstimeofday()`) is controlled by the third parameter, which may be set to the values `EXTINT_TIMER_RELATIVE` or `EXTINT_TIMER_ABSOLUTE`. Interrupt notifications occurs through the standard external interrupt callout mechanism described in "Interrupt Notification Interface" on page 31. This field may be set to `NULL` if the low-level driver does not support timer functionality.
`disarm_timer`	Cancels a pending interrupt, if any, scheduled to be delivered due to a prior call to the `arm_timer()` function. If the previously scheduled interrupt has already occurred, it is not necessary to call `disarm_timer()`, and calling `disarm_timer()` when no interrupt is pending should be harmless. This field may be set to `NULL` if the low-level driver does not support timer functionality.

When an External Interrupt Occurs

When an external interrupt signal triggers an interrupt that is handled by the low-level driver, the driver should make the following call:

```
void
extint_interrupt(struct extint_device *ed);
```

This allows the abstraction layer to perform any appropriate abstracted actions, such as update the interrupt count or trigger `poll/select` actions. The sole argument is the `struct extint_device` that was returned from the registration call.

Driver Deregistration

When the driver desires to deregister a particular device previously registered with the abstraction layer, it should make the following call:

```
void
extint_device_unregister(struct extint_device *ed);
```

The sole argument is the `struct extint_device` that was returned from the registration call. There is no error return from this call, but if invalid data is passed to it, the likelihood of a kernel panic is very high.

Interrupt Notification Interface

In addition to the user-visible aspects of the external interrupt abstraction layer, there is a kernel-only interface available for interrupt notification. This interface provides the ability for other kernel modules to register a callout to be invoked whenever an external interrupt is ingested for a particular device.

This section discusses the following:

- "Callout Mechanism" on page 31
- "Callout Registration" on page 32
- "Callout Deregistration" on page 33

Callout Mechanism

For systems (not just applications) that are critically interested in responding as quickly as possible to an externally triggered event, waiting for a poll/select operation, or even busy-waiting on the value of the interrupt counter to change, may have unexpected harmful effects (such as tying up a CPU spinning on a value) or may not provide appropriate response times.

A callout mechanism lets you write your own kernel module in order to gain minimal-latency notification of events and react accordingly. It also provides an extension capability that might be of interest in certain situations. For example, there could be an application that requires an interrupt counter page similar to the one maintained by the abstraction layer, but that starts counting at 0 when the device special file is opened. Or, there could be an application that requires a signal to be generated and delivered to the process when an interrupt is ingested. These examples

are more esoteric than the simple counter page, and are best provided by a separate module rather than cluttering the main external interrupt abstraction code.

Callout Registration

To register a callout to be invoked upon interrupt ingest, allocate a `struct extint_callout`, fill it in, and pass it to the following call:

```
int
extint_callout_register(struct extint_device *ed,
                        struct extint_callout *ec);
```

The first argument is the `struct extint_device` corresponding to the particular abstracted external interrupt hardware device of interest. How this structure is found is up to the caller; however, the `file_to_extint_device` function will convert a `struct` file pointer to a `struct extint_device` pointer. This function will return `-EINVAL` if an inappropriate file descriptor is passed to it.

The second argument is one of the following structures:

```
struct extint_callout {
        struct module* owner;
        void (*function)(void *);
        void *data;
};
```

Note: Additional fields not of interest to the external interrupt user may be present. You should include `/usr/local/include/extint.h` to acquire these structure definitions.

The `owner` field should be set to the module containing the function and data pointed to by the remaining fields.

The `function` pointer is a callout function that is to be invoked whenever an interrupt is ingested by the abstraction layer for the device of interest. The `data` field is the only argument passed to it; it is used opaquely and is provided solely for use by the caller. That is, the abstraction layer will invoke the following upon each interrupt of the specified device:

```
ec->function(data);
```

You can register multiple callouts for the same abstracted external interrupt device. They will be invoked in no guaranteed order, but will be invoked one at a time.

The interrupt counter will be incremented before the callouts are invoked, but before any signal/poll notifications occur.

The module specified by the `owner` field in the callout structure, as well as the module corresponding to the low-level external interrupt device driver, will have their reference counts increased by one until the callout is deregistered.

Callout Deregistration

To remove a callout, call the following with the same arguments as provided during callout registration:

```
extern void
extint_callout_unregister(struct extint_device *ed,
                          struct extint_callout *ec);
```

You can remove both active and orphaned callouts in this manner with no distinction between the two.

The callout function must continue to be able to be invoked until the call to `extint_callout_unregister` completes.

Making Use of Unsupported Hardware Device Capabilities

If your hardware device supports capabilities that are not provided for in the abstraction layer, you can do one of the following:

- Add a new attribute to the abstraction layer by modifying `struct extint_properties` to add appropriate interface routines and update any existing drivers as necessary.
- Have the low-level driver create its own device class and corresponding attributes and/or character special devices. This method is preferred and is required if the capability is dependent on the hardware in a method that cannot be abstracted.

For example, the SGI IOC4 has the ability to map the interrupt output control register directly into a user application to avoid the kernel overhead of reading/writing the abstracted attribute files. Using this capability means that the application must have intimate knowledge of the format of the control register, something that cannot be abstracted away by the kernel and is very specific to this particular I/O controller chip. This capability is provided by the `ioc4_extint` driver, which supplys its own character special device along with an `ioc4_intout` device class.

Low-level Driver Template

You can use the `ioc4_extint.c` file as a template for a low-level driver. The file is shipped as part of the `extint` source RPM.

Note: In addition to providing the abstraction interface, this low-level driver creates an IOC4-specific character special device and an IOC4-specific device class.

Example: SGI IOC4 PCI Device

This section describes the following for the SGI IOC4 PCI device:

- "Multiple Independent Drivers" on page 34
- "External Interrupt Output" on page 36
- "External Interrupt Ingest" on page 38
- "Physical Interfaces" on page 38

For more information, see the `Documentation/sgi-ioc4.txt` file, which is installed with the Linux source code corresponding to the real-time kernel.

Multiple Independent Drivers

The IOC4 external interrupt driver is not a typical PCI device driver. Due to certain design features of the IOC4 controller, typical PCI probing and removal functions are not appropriate. Instead, the IOC4 external interrupt driver interfaces with a core IOC4 driver that takes care of the usual PCI-level driver functionality. (An overview is provided below; for more details, see the `Documentation/sgi-ioc4.txt` file in the kernel source code.) However, the IOC4 external interrupt driver does interface very cleanly with the external interrupt abstraction layer, which is within the scope of the following discussion.

The IOC4 driver actually consists of the following independent drivers:

`ioc4` — The core driver for IOC4. It is responsible for initializing the basic functionality of the chip and allocating the PCI resources that are shared between the IOC4 functions.

This driver also provides registration functions that the other IOC4 drivers can call to make their presence known. Each driver must provide a probe and a remove function, which are invoked by the core driver at appropriate times. The interface for the probe and remove operations is not precisely the same as the PCI device probe and remove operations, but is logically the same operation.

`sgiioc4` — The IDE driver for IOC4. It hooks up to the `ioc4` driver via the appropriate registration, probe, and remove functions.

`ioc4_serial` — The serial driver for IOC4. It hooks up to the `ioc4` driver via the appropriate registration, probe, and remove functions.

`ioc4_extint` — The external interrupts driver for IOC4.

IOC4-based I/O controller cards provide an electrical interface to the outside world that can be used to ingest and generate a simple signal for the following purposes:

- On the output side, one of the jacks can provide a small selection of output modes (low, high, a single strobe, toggling, and pulses at a specified interval) that create a 0-5V electrical output.
- On the input side, one of the jacks will cause the IOC4 to generate a PCI interrupt on the transition edge of an electrical signal.

This driver registers with the `extint` abstracted external interrupt driver and lets it take care of the user-facing details.

External Interrupt Output

The output section provides several modes of output:

`high` Sets the output to logic `high`. The `high` state of the card's electrical output is actually a low voltage (0V).

`low` Sets the output to logic `low`. The `low` state of the card's electrical output is actually a high voltage (+5V).

`pulse` Sets the output to logic `high` for 3 ticks then returns to logic `low` for an interval configured by the `period` setting, then repeats. The mode is configurable by the abstraction layer device's `mode` attribute. The abstraction layer device's `modelist` attribute contains available modes.

`strobe` Sets the output to logic `high` for 3 ticks, then returns to logic `low`. A *tick* is the PCI clock signal divided by 520.

`toggle` Alternates the output between logic `low` and logic `high` as configured by the `period` setting.

The `period` can be set to a range of values determined by the PCI clock speed of the IOC4 device. For the `toggle` and `pulse` output modes, this period determines how often the toggle or pulse occurs. The output period can be set only to a multiple of this length (rounding will occur automatically in the driver). The `pulse` and `strobe` output modes have a logic `high` pulse width equal to three ticks. The period should be configurable by the abstraction layer device's `period` attribute, and the tick length can be found from the abstraction layer device's `quantum` attribute.

Note: For reference, on a 66-MHz PCI bus, the tick length is 7.8 microseconds. On a 33-MHz PCI bus, the tick length is 15.6 microseconds. However, the IOC4 driver calibrates itself to a more precise value than these somewhat coarse numbers, depending on actual bus speed, which may vary slightly from bus to bus or even reboot to reboot. However, IOC4 is only officially supported when running at 66-MHz.

One device file is provided, which can be memory mapped. The first 32-bit quantity in the mapped area is aliased to the hardware register that controls output. Direct manipulation of the register, both for reading and writing, may be performed in order to avoid the kernel overhead that would be necessary if using the abstracted interfaces. Assuming the typical `sysfs` mount point, the device number files for these devices can be found at:

```
/sys/class/ioc4_intout/intout#/dev
```

This capability is not abstracted into the external interrupt abstraction layer because it is critical for an application to know that this is an IOC4 device in order to determine the format of the mapped register. Table 3-1 shows the register format.

Table 3-1 Register Format

Bits	Field	Read/Write Options	Description
15:0	`COUNT`	`RW`	Reloaded into the counter each time it reaches 0x0. The count period is actually `(COUNT+1)`.
18:16	`MODE`	`RW`	Sets the mode for `INT_OUT` control: • `000` loads a `0` to `INT_OUT` • `100` loads a `1` to `INT_OUT` • `101` pulses `INT_OUT` high for 3 ticks • `110` pulses `INT_OUT` for 3 ticks every `COUNT` • `111` toggles `INT_OUT` for 3 ticks every `COUNT` • `001`, `010`, and `011` are undefined
29:19	(reserved)	`RO`	Read as `0`, writes are ignored.
30	`DIAG`	`RW`	Bypass clock base divider. Operation when `DIAG` is set to a value of 1 is strictly unsupported.
31	`INT_OUT`	`RO`	Current state of `INT_OUT` signal.

Note: There are the following considerations:

- The register should always be read and written as a 32-bit word in order to avoid concerns about big-endian and little-endian differences between the CPU and the IOC4 device.
- The `/dev/intout#` file may be memory-mapped only on kernels with a system page size of 16 KB or smaller. Due to technical constraints, it is not made available on kernels with a system page size larger than 16 KB.

External Interrupt Ingest

The ingest section provides one control, the source of interrupt signals. The `external` source is a circuit connected to the external jack provided on IOC4-based I/O controller cards. The `loopback` source is the output of the IOC4's interrupt output section. The source is configurable by the abstraction layer device's `source` attribute. You can find available sources in the abstraction layer device's `sourcelist` attribute.

For example, to set up loopback mode:

```
[root@linux root]# echo loopback >/sys/class/extint/extint0/source
[root@linux root]# echo 100000000 >/sys/class/extint/extint0/period
[root@linux root]# echo toggle >/sys/class/extint/extint0/mode
```

Note: The IO10 card does not provide the 1/8–inch stereo connector interface for external interrupts, and thus can only use `loopback` as its source.

Physical Interfaces

Use a two-conductor shielded cable to connect external interrupt output and input, with the two cable conductors wired to the +5V and interrupt conductors and the sleeves connected to the cable shield at both ends to maintain EMI integrity.

All IOC4-based external interrupt implementations use female 1/8-inch audio jacks. The wiring for the input jack is as follows:

- Tip: +5V input
- Ring: interrupt input (active low, optoisolated)
- Sleeve: chassis ground/cable shield

The input signal passes through an optoisolator that has a damping effect. The input signal must be of sufficient duration to drive the output of the optoisolator low in order for the interrupt to be recognized by the receiving machine. Current experimentation shows that the threshold is about 2.5 microseconds. To be safe, the driver sets its default outgoing pulse width to 10 microseconds. Any hardware not from SGI that is driving this line should do the same.

Figure 3-1 shows the internal driver circuit for the output connector and the internal receiver circuit for the input connector.

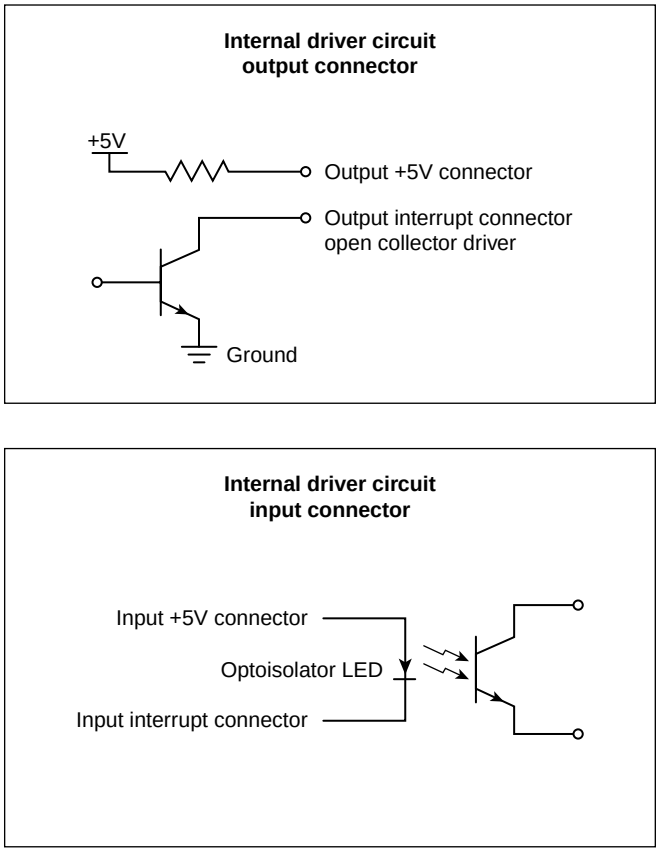

Figure 3-1 Output and Input Connectors for the Internal Driver Circuit

You can wire an output connector directly to an input connector, taking care to connect the +5V output to the +5V input and the interrupt output to the interrupt input. If some other device is used to drive the input, it must be a +5V-source current limited with a 420–ohm resistor in series in order to avoid damaging the optoisolator.

CPU Workload

This chapter describes how to use Linux kernel features to make the execution of a real-time program predictable. Each of these features works in some way to dedicate hardware to your program's use, or to reduce the influence of unplanned interrupts on it:

- "Using Priorities and Scheduling Queues" on page 41
- "Minimizing Overhead Work" on page 47
- "Understanding Interrupt Response Time" on page 52
- "Minimizing Interrupt Response Time" on page 56

Using Priorities and Scheduling Queues

The default Linux scheduling algorithm is designed for a conventional time-sharing system. It also offers additional real-time scheduling disciplines that are better-suited to certain real-time applications.

This section discusses the following:

- "Scheduling Concepts" on page 41
- "`kbar` Kernel Barrier Facility for Altix ia64 Servers" on page 43
- "Setting Pthread Priority" on page 46
- "Controlling Kernel and User Threads" on page 46

Scheduling Concepts

In order to understand the differences between scheduling methods, you must understand the following basic concepts:

- "Timer Interrupts" on page 42
- "Real-Time Priority Band" on page 42

For information about time slices and changing the time-slice duration, see the information about the CPU scheduler in the *Linux Configuration and Operations Guide*.

Timer Interrupts

In normal operation, the kernel pauses to make scheduling decisions every several millisecond (ms) in every CPU. You can determine the frequency of this interval with the `sysconf(_SC_CLK_TCK)` function (see "Clocks on Altix ia64" on page 12). Every CPU is normally interrupted by a timer every timer interval. (However, the CPUs in a multiprocessor are not necessarily synchronized. Different CPUs may take timer interrupts at different times.)

During the timer interrupt, the kernel updates accounting values, does other housekeeping work, and chooses which process to run next—usually the interrupted process, unless a process of superior priority has become ready to run. The timer interrupt is the mechanism that makes Linux scheduling preemptive; that is, it is the mechanism that allows a high-priority process to take a CPU away from a lower-priority process.

Before the kernel returns to the chosen process, it checks for pending signals and may divert the process into a signal handler.

Real-Time Priority Band

A real-time thread can select one of a range of 99 priorities (1-99) in the real-time priority band, using POSIX interfaces `sched_setparam()` or `sched_setscheduler()`. The higher the numeric value of the priority, the more important the thread. For more information, see the `sched_setscheduler`(2) man page.

Many soft real-time applications must execute ahead of time-share applications, so a lower priority range is best suited. Because time-share applications are scheduled at lower priority than real-time applications, a thread running at the lowest real-time priority (1) still executes ahead of all time-share applications.

Note: Applications cannot depend on system services if they are running ahead of system threads without observing system responsiveness timing guidelines.

Within a program it is usually best to follow the principles of *rate-monotonic scheduling*. However, you can use the following list as a guideline for selecting scheduling priorities in order to coordinate among different programs:

Priority	Description
99	Reserved for critical kernel threads and should not be used by applications (99 is the highest real-time priority)
90 - 98	Hard real-time user threads
60 - 89	High-priority operating system services
40 - 59	Firm real-time user threads
31 - 39	Low-priority operating system services
1 - 30	Soft real-time user threads

Real-time users can use tools such as `strace`(1) and `ps`(1) to observe the actual priorities and dynamic behaviors.

`kbar` Kernel Barrier Facility for Altix ia64 Servers

Note: This section does not apply to Altix XE x86_64 servers.

The `kbar`(3) kernel barrier facility provides for the fast wake-up of many blocked user threads. When the barrier is signaled, the operating system will use a configurable number of CPUs to quickly wake all blocked threads. A maximum of 64 barriers are supported system-wide.

Note: The barriers do not behave precisely as traditional barriers. A specific number of threads do not have to be blocked on the barrier for them to be woken.

To use `kbar`, you must load the `kbar` Linux kernel module into the Linux kernel. To do this, enter the following as `root`:

```
[root@linux root]# modprobe kbar
```

Synopsis:

```
#include <bitmask.h>
#include <sn/kbar.h>

link with -lkbar

int kbar_open(struct bitmask * mask, int discipline);
int kbar_close(int fd);
int kbar_wait(int fd);
int kbar_signal(int fd);
```

where:

- `kbar_open` creates a kernel-supported barrier. It takes as its arguments a per-CPU bitmask of the CPUs that it should use to wake blocked threads and an assignment discipline. You can generate these bitmasks by using `bitmask_routines` available in `<bitmask.h>`, which ships as part of SGI ProPack for Linux.

  The assignment discipline can be one of the following:

  – `KBAR_LOCAL`, which tries to wake threads with worker CPUs near where they last ran

  – `KBAR_BALANCED`, which tries to balance the waking of threads across worker CPUs

  `kbar_open` returns a file descriptor with which all future interactions with that barrier will be made. If any other process is given access to that file descriptor (by such means as fork or interprocess communication), it can also take part in the barrier. On error, `kbar_open` returns `-1` and sets `errno` to one of the following:

`EFAULT`	There was a memory error in accessing the bitmask argument
`EINVAL`	There was an error with the configuration arguments
`ENOENT`	The `kbar` Linux kernel module is not loaded
`ENOMEM`	There was not enough memory to create the barrier
`ENOSPC`	The maximum system-wide number of barriers have already been created

- `kbar_close` removes access to a barrier previously allocated by `kbar_open`. If the file descriptor has not been duplicated or is the last copy, then the barrier is deallocated. If user threads are still blocked on the barrier when `kbar_close` is called, the barrier will not be destroyed until they are all interrupted. Further

access to the barrier through that file descriptor will no longer be possible. On error, `kbar_close` returns `-1` and sets `errno` to the following:

`EBADF` `fd` is not a valid open file descriptor

- `kbar_wait` causes the calling thread to block on a barrier that was previously allocated by `kbar_open`. The thread will remain blocked until some other thread calls `kbar_signal` on the barrier or until it receives an unblocked signal.

  On error, `kbar_wait` returns `-1` and sets `errno` to the following:

  `EINTR` The thread was interrupted by a signal

- `kbar_signal` causes the operating system to use the previously designated helper CPUs to wake all the threads blocked on the barrier at the current time.

  On error, `kbar_signal` returns `-1` and sets `errno` to the following:

  `EBUSY` Another thread is in the process of waking the barrier's threads

Figure 4-1 shows the flow of the `kbar_wait` and `kbar_signal` functions. Threads block on the barrier with `kbar_wait` and are later scheduled again with `kbar_signal`.

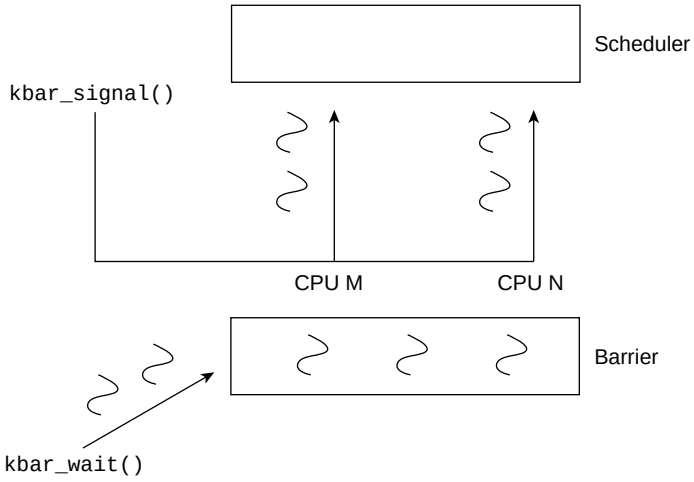

Figure 4-1 `kbar`(3) Kernel Barrier Facility

For examples, see the `/usr/share/react/kbar/examples/` directory.

Setting Pthread Priority

The Linux pthreads library shipped with 2.6 Linux is known as the *new pthreads library (NPTL)*. By default, a newly created pthread receives its priority from the same scheduling policy and scheduling priority as the pthread that created it; new pthreads will ignore the values in the attributes structure.

You can set the priority and scheduling policy of pthreads as follows:

- To change a running pthread, the pthread must call `pthread_setschedparam()`.
- To set the scheduling attributes that a pthread will start with when it is created, use the `pthread_attr_setschedpolicy()` and `pthread_attr_setschedparam()` library calls to configure the attributes structure that will later be passed to `pthread_create()`.

  The `pthread_attr_setinheritsched()` library call acts on the `pthread_attr_t` structure that will later be passed to `pthread_create()`. You can configure it with one of the following settings:

  – `PTHREAD_EXPLICIT_SCHED` causes pthreads to use the scheduling values set in the structure

  – `PTHREAD_INHERIT_SCHED` causes pthreads to inherit the scheduling values from their parent pthread

Controlling Kernel and User Threads

In some situations, kernel threads and user threads must run on specific processors or with other special behavior. Most user threads and a number of kernel threads do not require any specific CPU or node affinity, and therefore can run on a select set of nodes. The SGI ProPack bootcpuset feature controls the placement of both kernel and user threads that do not require any specific CPU or node affinity. By placing these threads out of the way of your time-critical application threads, you can minimize interference from various external events.

As an example, an application might have two time-critical interrupt servicing threads, one per CPU, running on a four-processor machine. You could set up CPUs 0 and 1 as a bootcpuset and then run the time-critical threads on CPUs 2 and 3.

Note: You must have the SGI `cpuset-*.rpm` RPM installed to use bootcpusets. For configuration information, see the `bootcpuset`(8) man page.

You must use the `reactconf.pl` configuration script for this procedure; see Chapter 9, "REACT System Configuration" on page 109.

Minimizing Overhead Work

A certain amount of CPU time must be spent on general housekeeping. Because this work is done by the kernel and triggered by interrupts, it can interfere with the operation of a real-time process. However, you can remove almost all such work from designated CPUs, leaving them free for real-time work.

First decide how many CPUs are required to run your real-time application. Then apply the following steps to isolate and restrict those CPUs:

- "Avoid the Clock Processor (CPU 0)" on page 47
- "Reduce the System Flush Duration" on page 48
- "Redirect Interrupts" on page 48
- "Select the Console Node for SAL Console Driver Interrupt for Altix ia64 Servers" on page 49
- "Restrict, Isolate, and Shield CPUs" on page 49
- "Avoid Kernel Module Insertion and Removal" on page 52
- "Avoid Filesystem Mounts" on page 52

Note: The steps are independent of each other, but each must be done to completely free a CPU.

Avoid the Clock Processor (CPU 0)

Every CPU takes a timer interrupt that is the basis of process scheduling. However, CPU 0 does additional housekeeping for the whole system on each of its timer interrupts. Real-time users are therefore advised not to use CPU 0 for running real-time processes.

Reduce the System Flush Duration

In SGI systems running Linux, the scalable hub (SHub) ASIC is responsible for memory transactions with the processor front-side bus. Periodically, the SHub initiates a system flush, which can impact real-time performance. The system flush duration by default is set to a value appropriate for more general purpose computing, and this default value can interfere with extremely time-sensitive threads that require interrupt response times measured in microseconds. You can set the system flush duration to a value appropriate for real-time applications by following step 8 in Chapter 9, "REACT System Configuration" on page 109.

Redirect Interrupts

To minimize latency of real-time interrupts, it is often necessary to direct them to specific real-time processors. It is also necessary to direct other interrupts away from specific real-time processors. This process is called *interrupt redirection*.

The `reactconf.pl` script will assist you in redirecting interrupts. The process involves writing a bitmask to the `/proc/irq/`*interruptnumber*`/smp_affinity` file, which shows a bitmask of the CPUs that are allowed to receive this interrupt. A `1` in the least-significant bit in this mask denotes that CPU 0 is allowed to receive the interrupt. The most-significant bit denotes the highest-possible CPU that the booted kernel could support.

For example, to redirect interrupt 62 to CPU 1, enter the following:

```
[root@linux root]# echo 1 > /proc/irq/62/smp_affinity
```

You can examine the `/proc/interrupts` file to discover where interrupts are being received on your system.

An example of how to redirect interrupts is demonstrated by the `reactconf.pl` configuration script. SGI recommends that someone with knowledge of the system configuration use this script to redirect only the interrupts that must be moved. For more information, see Chapter 9, "REACT System Configuration" on page 109.

Select the Console Node for SAL Console Driver Interrupt for Altix ia64 Servers

Note: This section does not apply to Altix XE servers.

The console node you select for the system abstraction layer (SAL) console driver interrupts depends upon whether your system has an L2 system controller or not:

- If your system has an L2 controller, the SAL console driver interrupt will always appear on a CPU on the first node.
- If your system does not have an L2 controller, the SAL console driver generates interrupts that will be directed toward a single CPU on the node where the console is attached. SGI recommends that you attach the console to a node that will not be used for time-critical threads. Because the clock processor always runs on CPU 0, SGI recommends that you use node 0 as the console node.

  Note: You cannot select which CPU on the console node will receive interrupts.

For more information, see the *SGI L1 and L2 Controller Software User's Guide*.

Restrict, Isolate, and Shield CPUs

In general, the Linux scheduling algorithms run a process that is ready to run on any CPU. For best performance of a real-time process or for minimum interrupt response time, you must use one or more CPUs without competition from other scheduled processes. You can exert the following levels of increasing control:

- *Restricted*, which prevents the CPU from running scheduled processes.
- *Isolated*, which removes the CPU from load balancing considerations, a time-consuming scheduler operation.
- *Shielded*, which switches off the timer (scheduler) interrupts that would normally be scheduled on the CPU. These are a source of jitter, but only a minor source of interrupt response latency. Shielding should only be done for short periods where basically jitter-free program execution is required.

Generally, you would want to prepare a CPU for time-critical real-time operation by both restricting and isolating it. You should use the `reactconf.pl` configuration script to perform the steps required to restrict and isolate a CPU. For more information, see Chapter 9, "REACT System Configuration" on page 109.

Restricting a CPU from Scheduled Work

You can restrict one or more CPUs from running scheduled processes. The only processes that can use a restricted CPU are those processes that you assign to it, along with certain per-CPU kernel threads.

To restrict one or more CPUs, use the `reactconf.pl` configuration script documented in Chapter 9, "REACT System Configuration" on page 109.

After restricting a CPU, you can assign processes to it using the SGI `cpuset` command. See "Running a Process on a Restricted CPU" on page 123.

To remove the CPU restriction, allowing the CPU to execute any scheduled process, rerun the `reactconf.pl` command with a different set of CPUs or use the procedure in "Disabling REACT" on page 135.

Isolating a CPU from Scheduler Load Balancing

You can isolate a CPU so that it is not subject to the effects of scheduler load balancing. Isolating a CPU removes one source of unpredictable delays from a real-time program and helps further minimize the latency of interrupt handling.

To isolate one or more CPUs, use the `reactconf.pl` configuration script documented in Chapter 9, "REACT System Configuration" on page 109. Each `rtcpu` is set to be `cpu_exclusive`.

Normally, you would also want to restrict the isolated CPUs. See "Restricting a CPU from Scheduled Work" on page 50.

Shielding a CPU from Timer Interrupts

You can shield a CPU from the normally scheduled Linux timer (scheduler) interrupts. For more information on timer interrupts, see "Timer Interrupts" on page 42.

Timer interrupts are a source of interrupt response latency (usually several usec). Shielding is done dynamically from program control, and should only be done for short periods where essentially jitter-free program execution is required.

When a CPU's timer interrupts are switched off, scheduling on that CPU ceases. A thread must not yield the CPU (sleep) unless it expects to be awoken by an external event such as an I/O interrupt or if timer interrupts will be switched back on before it must be scheduled again.

Note: Be aware of the following:

- Prolonged periods of shielding might eventually result in system resource depletion.

  System resource depletion usually takes the form of out-of-memory conditions, eventually causing forced shutdown of the application. The kernel ring buffer will indicate this situation by showing a stack trace for the application and a `No available memory in cpuset:` message. To view the kernel ring buffer, run the `dmesg` command.
- You should ensure that all threads are placed in their appropriate cpusets prior to calling `cpu_shield()` anywhere on the system. Movement between cpusets will be held off during periods where any processor's timer interrupts are switched off. After timer interrupts for all processors are switched back on, any pending cpuset thread movement will occur.

To shield a CPU from timer interrupts, do the following:

1. Load the `sgi-shield` kernel module.

   For example:

   ```
   [root@linux root]# modprobe sgi-shield
   ```

2. From your application, call the `cpu_shield()` function with the `SHIELD_STOP_INTR` flag and the desired CPU number. Your program must link in the `sgirt` library to access the `cpu_shield()` function. For more information, see the `libsgirt`(3) man page.

   For example, to switch off timer interrupts on CPU 3, perform the following function call from the application:

   ```
   cpu_shield(SHIELD_STOP_INTR, 3)
   ```

To unshield the CPU, call the `cpu_shield()` function with the `SHIELD_START_INTR` flag and the desired CPU number.

For example, when shielding CPU 3 is no longer necessary, perform the following call from the application:

```
cpu_shield(SHIELD_START_INTR, 3)
```

Avoid Kernel Module Insertion and Removal

The insertion and removal of Linux kernel modules (such as by using `modprobe` or `insmod/rmmod`) requires that a kernel thread be started on all active CPUs (including isolated CPUs) in order to synchronously stop them. This process allows safe lockless-module list manipulation. However, these kernel threads can interfere with thread wakeup and, for brief periods, the ability to receive interrupts.

While a time-critical application is running, you must avoid Linux kernel module insertion and removal. All necessary system services should be running prior to starting time-critical applications.

Avoid Filesystem Mounts

The process of mounting of a filesystem can interfere with response times for a number of CPUs. These delays do not happen after the mount has completed. There is no delay for disk accesses.

Prior to running a time-critical application, you should complete all filesystem mounts that may be necessary during application execution.

Understanding Interrupt Response Time

Interrupt response time is the time that passes between the instant when a hardware device raises an interrupt signal and the instant when (interrupt service completed) the system returns control to a user process. SGI guarantees a maximum interrupt response time on certain systems, but you must configure the system properly in order to realize the guaranteed time.

Maximum Response Time Guarantee

In properly configured systems, interrupt response time is guaranteed not to exceed 30 microseconds (usecs) for SGI systems running Linux.

This guarantee is important to a real-time program because it puts an upper bound on the overhead of servicing interrupts from real-time devices. You should have some idea of the number of interrupts that will arrive per second. Multiplying this by 30 usecs yields a conservative estimate of the amount of time in any one second devoted to interrupt handling in the CPU that receives the interrupts. The remaining time is available to your real-time application in that CPU.

Components of Interrupt Response Time

The total interrupt response time includes the following sequential parts:

Hardware latency	The time required to make a CPU respond to an interrupt signal.
Software latency	The time required to dispatch an interrupt thread.
Device service time	The time the device driver spends processing the interrupt and dispatching a user thread.
Mode switch	The time it takes for a thread to switch from kernel mode to user mode.

Figure 4-2 diagrams the parts discussed in the following sections.

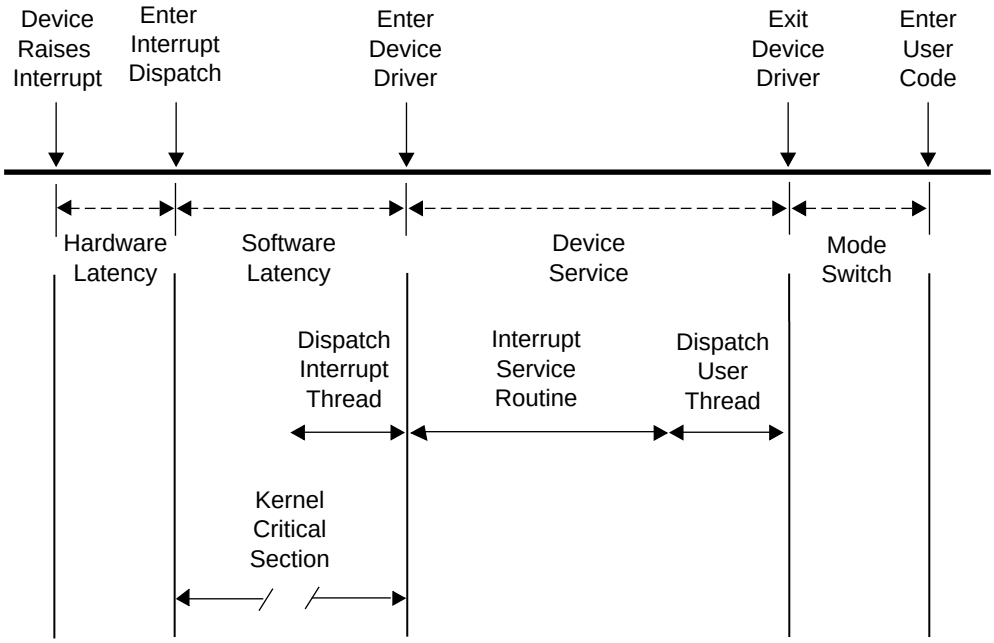

Figure 4-2 Components of Interrupt Response Time

Hardware Latency

When an I/O device requests an interrupt, it activates a line in the PCI bus interface. The bus adapter chip places an interrupt request on the system internal bus and a CPU accepts the interrupt request.

The time taken for these events is the hardware latency, or *interrupt propagation delay*.

For more information, see Chapter 7, "PCI Devices" on page 95.

Software Latency

Software latency is affected by the following:

- "Kernel Critical Sections" on page 54
- "Interrupt Threads Dispatch" on page 55

Kernel Critical Sections

Certain sections of kernel code depend on exclusive access to shared resources. Spin locks are used to control access to these critical sections. Once in a critical section, interrupts are disabled. New interrupts are not serviced until the critical section is complete.

There is no guarantee on the length of kernel critical sections. In order to achieve 30-usec response time, your real-time program must avoid executing system calls on the CPU where interrupts are handled. The way to ensure this is to restrict that CPU from running normal processes. For more information, see "Restricting a CPU from Scheduled Work" on page 50 and "Isolating a CPU from Scheduler Load Balancing" on page 50.

You may need to dedicate a CPU to handling interrupts. However, if the interrupt-handling CPU has power well above that required to service interrupts (and if your real-time process can tolerate interruptions for interrupt service), you can use the restricted CPU to execute real-time processes. If you do this, the processes that use the CPU must avoid system calls that do I/O or allocate resources, such as `fork()`, `brk()`, or `mmap()`. The processes must also avoid generating external interrupts with long pulse widths.

In general, processes in a CPU that services time-critical interrupts should avoid all system calls except those for interprocess communication and for memory allocation within an arena of fixed size.

Interrupt Threads Dispatch

The primary function of interrupt dispatch is to determine which device triggered the interrupt and dispatch the corresponding interrupt thread. Interrupt threads are responsible for calling the device driver and executing its interrupt service routine.

While the interrupt dispatch is executing, all interrupts at or below the current interrupt's level are masked until it completes. Any pending interrupts are dispatched before interrupt threads execute. Thus, the handling of an interrupt could be delayed by one or more devices.

In order to achieve 30-usec response time on a CPU, you must ensure that the time-critical devices supply the only device interrupts directed to that CPU. For more information, see "Redirect Interrupts" on page 48.

Device Service Time

Device service time is affected by the following:

- "Interrupt Service Routines"
- "User Threads Dispatch"

Interrupt Service Routines

The time spent servicing an interrupt should be negligible. The interrupt handler should do very little processing; it should only wake up a sleeping user process and possibly start another device operation. Time-consuming operations such as allocating buffers or locking down buffer pages should be done in the request entry points for `read()`, `write()`, or `ioctl()`. When this is the case, device service time is minimal.

User Threads Dispatch

Typically, the result of the interrupt is to make a sleeping thread runnable. The runnable thread is entered in one of the scheduler queues. This work may be done while still within the interrupt handler.

Mode Switch

A number of instructions are required to exit kernel mode and resume execution of the user thread. Among other things, this is the time when the kernel looks for software signals addressed to this process and redirects control to the signal handler.

If a signal handler is to be entered, the kernel might have to extend the size of the stack segment. (This cannot happen if the stack was extended before it was locked.)

Minimizing Interrupt Response Time

You can ensure interrupt response time of 30 usecs or less for one specified device interrupt on a given CPU provided that you configure the system as follows:

- The CPU does not receive any other `SN hub` device interrupts
- The interrupt is handled by a device driver from a source that promises negligible processing time
- The CPU is isolated from the effects of load balancing
- The CPU is restricted from executing general Linux processes
- Any process you assign to the CPU avoids system calls other than interprocess communication and allocation within an arena
- Kernel module insertion and removal is avoided

When these things are done, interrupts are serviced in minimal time.

Using the Frame Scheduler

The frame scheduler makes it easy to structure a real-time program as a family of independent, cooperating activities that are running on multiple CPUs and are scheduled in sequence at the frame rate of the application.

Note: With Altix XE x86_64 architecture, the CC clock source is supplied by the PCI-RT-Z card. SHUB hardware timers are not available on Altix XE platforms. On Altix XE platforms, you must have one PCI-RT-Z card per asynchronous frame scheduler. Multiple frame schedulers running synchronously can use a single PCI-RT-Z card, however.

This chapter discusses the following:

- "Frame Scheduler Concepts" on page 57
- "Selecting a Time Base" on page 72
- "Using the Scheduling Disciplines" on page 74
- "Using Multiple Consecutive Minor Frames" on page 77
- "Designing an Application for the Frame Scheduler" on page 78
- "Preparing the System" on page 79
- "Implementing a Single Frame Scheduler" on page 80
- "Implementing Synchronized Schedulers" on page 81
- "Handling Frame Scheduler Exceptions" on page 84
- "Using Signals Under the Frame Scheduler" on page 89
- "Using Timers with the Frame Scheduler" on page 92

Frame Scheduler Concepts

One frame scheduler dispatches selected threads at a real-time rate on one CPU. You can also create multiple, synchronized frame schedulers that dispatch concurrent threads on multiple CPUs.

This section discusses the following:

- "Frame Scheduler Basics" on page 58
- "Thread Programming Model" on page 59
- "Frame Scheduling" on page 59
- "Controller Thread" on page 61
- "Frame Scheduler API" on page 62
- "Interrupt Information Templates" on page 63
- "Library Interface for C Programs" on page 64
- "Thread Execution" on page 66
- "Scheduling Within a Minor Frame" on page 67
- "Synchronizing Multiple Schedulers" on page 69
- "Starting a Single Scheduler" on page 69
- "Starting Multiple Schedulers" on page 70
- "Pausing Frame Schedulers" on page 71
- "Managing Activity Threads" on page 71

Frame Scheduler Basics

When a frame scheduler dispatches threads on one CPU, it does not completely supersede the operation of the normal Linux scheduler. The CPUs chosen for frame scheduling must be restricted and isolated (see "Restrict, Isolate, and Shield CPUs" on page 49). You do not have to set up cpusets for the frame-scheduled CPUs because the frame scheduler will set up cpusets named `rtcpu`*N* (where *N* is the CPU number) if this has not already been done. For more control over cpuset parameters, you can create your own cpusets for the frame scheduler to use (one per CPU, and one CPU per cpuset), by naming them exactly as mentioned above.

If you already have cpusets named `rtcpu`*N* but they include other than only the CPU number in question, the frame scheduler will return an `EEXIST` error.

Note: REACT for Linux does not support Vsync, device-driver, or system-call time bases.

For more information, see "Isolating a CPU from Scheduler Load Balancing" on page 50 and "Preparing the System" on page 79.

Thread Programming Model

The frame scheduler supports pthreads.

In this guide, a *thread* is defined as an independent flow of execution that consists of a set of registers (including a program counter and a stack). A *pthread* is defined by the POSIX standard. Pthreads within a process use the same global address space.

A traditional Linux process has a single active thread that starts after the program is executed and runs until the program terminates. A multithreaded process may have several threads active at one time. Hence, a process can be viewed as a receptacle that contains the threads of execution and the resources they share (that is, data segments, text segments, file descriptors, synchronizers, and so forth).

Frame Scheduling

Instead of scheduling threads according to priorities, the frame scheduler dispatches them according to a strict, cyclic rotation governed by a repetitive time base. The time base determines the fundamental frame rate. (See "Selecting a Time Base" on page 72.) Some examples of the time base are as follows:

- A specific clocked interval in microseconds
- An external interrupt (see "External Interrupts as a Time Base" on page 73)
- The Vsync (vertical retrace) interrupt from the graphics subsystem
- A device interrupt from a specially modified device driver
- A system call (normally used for debugging)

Note: REACT for Linux does not support Vsync, device-driver, or system-call time bases.

The interrupts from the time base define *minor frames*. Together, a fixed number of minor frames make up a *major frame*. The length of a major frame defines the application's true frame rate. The minor frames allow you to divide a major frame into subframes. Figure 5-1 shows major and minor frames.

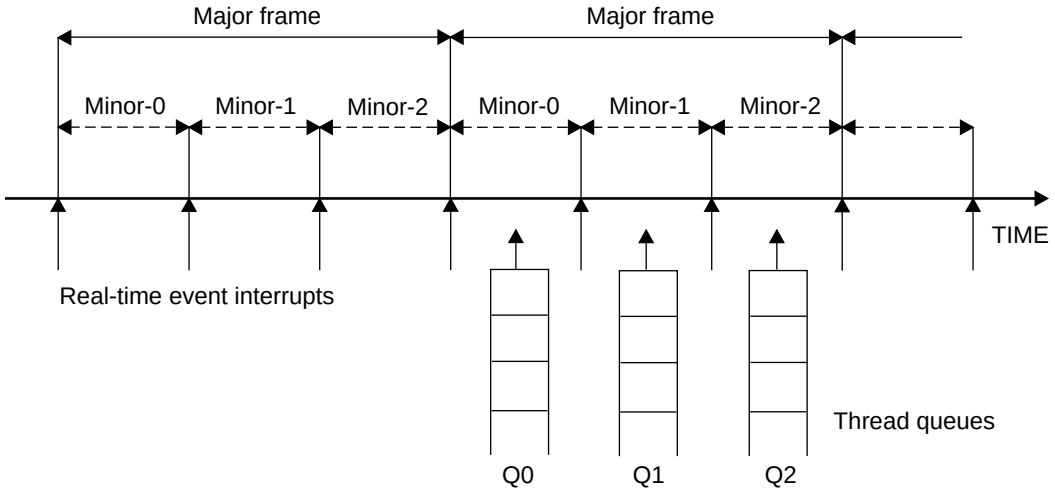

Figure 5-1 Major and Minor Frames

In the simplest case, there is a single frame rate, such as 60 Hz, and every activity the program performs must be done once per frame. In this case, the major and minor frame rates are the same.

In other cases, there are some activities that must be done in every minor frame, but there are also activities that are done less often, such as in every other minor frame or in every third one. In these cases, you define the major frame so that its rate is the rate of the least-frequent activity. The major frame contains as many minor frames as necessary to schedule activities at their relative rates.

As pictured in Figure 5-1, the frame scheduler maintains a queue of threads for each minor frame. You must queue each activity thread of the program to a specific minor frame. You determine the order of cyclic execution within a minor frame by the order in which you queue threads. You can do the following:

- Queue multiple threads in one minor frame. They are run in the queued sequence within the frame. All must complete their work within the minor frame interval.

- Queue the same thread to run in more than one minor frame. For example, suppose that thread `double` is to run twice as often as thread `solo`. You would queue `double` to Q0 and Q2 in Figure 5-1, and queue `solo` to Q1.
- Queue a thread that takes more than a minor frame to complete its work. If thread `sloth` needs more than one minor interval, you would queue it to Q0, Q1, and Q2, such that it can continue working in all three minor frames until it completes.
- Queue a background thread that is allowed to run only when all others have completed, to use up any remaining time within a minor frame.

All of these options are controlled by scheduling disciplines you specify for each thread as you queue it. For more information, see "Using the Scheduling Disciplines" on page 74.

Typically, a frame scheduler is driven by a single interrupt source and contains minor frames having the same duration, but a variable frame scheduler may be used to implement a frame scheduler having multiple interrupt sources and/or minor frames of variable duration. For more information, see the `frs_create_vmaster()` function.

The relationship between threads and a frame scheduler depends upon the thread model in use:

- The pthread programming model requires that all threads scheduled by the frame scheduler reside in the same process.
- The `fork()` programming model does not require that the participating threads reside in the same process.

See "Implementing a Single Frame Scheduler" on page 80 for details.

Controller Thread

The thread that creates a frame scheduler is called the *frame scheduler controller thread*. It is privileged in these respects:

- Its identifier is used to identify its frame scheduler in various functions. The frame scheduler controller thread uses a pthread ID.
- It can receive signals when errors are detected by the frame scheduler (see "Using Signals Under the Frame Scheduler" on page 89).

- It cannot itself be queued to the frame scheduler. It continues to be dispatched by Linux and executes on a CPU other than the one that the frame scheduler uses.

Frame Scheduler API

For an overview of the frame scheduler API, see the `frs`(3) man page, which provides a complete listing of all the frame scheduler functions. Separate man pages for each of the frame scheduler functions provide the API details. The API elements are declared in `/usr/include/frs.h`. The following are some important types that are declared in `/usr/include/frs.h`:

`typedef frs_fsched_info_t`	A structure containing information about one scheduler (including its CPU number, interrupt source, and time base) and number of minor frames. Used when creating a frame scheduler.
`typedef frs_t`	A structure that identifies a frame scheduler.
`typedef frs_queue_info_t`	A structure containing information about one activity thread: the frame scheduler and minor frame it uses and its scheduling discipline. Used when enqueuing a thread.
`typedef frs_recv_info_t`	A structure containing error recovery options.
`typedef frs_intr_info_t`	A structure that `frs_create_vmaster()` uses for defining interrupt information templates (see Table 5-1 on page 64).

Additionally, the pthreads interface adds the following types, as declared in `/usr/include/sys/pthread.h`:

`typedef pthread_t`	An integer identifying the pthread ID.
`typedef pthread_attr_t`	A structure containing information about the attributes of the frame scheduler controller thread.

Interrupt Information Templates

Variable frame schedulers may drive each minor frame with a different interrupt source, as well as define a different duration for each minor frame. These two characteristics may be used together or separately, and are defined using an interrupt information template.

An *interrupt information template* consists of an array of `frs_intr_info_t` data structures, where each element in the array represents a minor frame. For example, the first element in the array represents the interrupt information for the first minor frame, and so on for *n* minor frames.

The `frs_intr_info_t` data structure contains two fields for defining the interrupt source and its qualifier: `intr_source` and `intr_qualifier`.

The following example demonstrates how to define an interrupt information template for a frame scheduler having minor frames of different duration. Assume the application requires four minor frames, where each minor frame is triggered by the synchronized clock timer, and the duration of each minor frame is as follows: 100 ms, 150 ms, 200 ms, and 250 ms. The interrupt information template may be defined as follows:

```
frs_intr_info_t intr_info[4];
intr_info[0].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[0].intr_qualifier = 100000;
intr_info[1].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[1].intr_qualifier = 150000;
intr_info[2].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[2].intr_qualifier = 200000;
intr_info[3].intr_source    = FRS_INTRSOURCE_CCTIMER;
intr_info[3].intr_qualifier = 250000;
```

For detailed programming examples, demonstrating the use of variable frame schedulers, see the `/usr/share/react/frs/examples` directory and the `frs_create_vmaster`(3) man page.

Library Interface for C Programs

Table 5-1 summarizes the API library functions in the `/usr/lib/libfrs.a` file.

Table 5-1 Frame Scheduler Operations

Operation	Use	Frame Scheduler API
Create a frame scheduler	Process setup	`frs_t* frs_create(`*cpu*`, (int int` *intr_source* `int` *intr_qualifier*`, int,` *n_minors*`, pid_t` *sync_master_pid*`, int`*num_slaves*`);`
	Process or pthread setup	`frs_t* frs_create_master(int` *cpu*`, int` *intr_source*`, int` *intr_qualifier*`, int` *n_minors*`, int` *num_slaves*`);`
	Process or pthread setup	`frs_t* frs_create_slave(int` *cpu*`, frs_t*` *sync_master_frs*`);`
	Process or pthread setup	`frs_t* frs_create_vmaster(int` *cpu*`, int` *n_minors*`, int` *n_slaves*`, frs_intr_info_t *`*intr_info*`);`
Queue to a frame scheduler minor frame	Process setup	`int frs_enqueue(frs_t*` *frs*`, pid_t` *pid*`, int` *minor_frame*`, unsigned int` *discipline*`);`
	Pthread setup	`int frs_pthread_enqueue(frs_t*` *frs*`, pthread_t` *pthread*`, int` *minor_frame*`, unsigned int` *discipline*`);`
Insert into a queue, possibly changing discipline	Process setup	`int frs_pinsert(frs_t*` *frs*`, int` *minor_frame*`, pid_t` *target_pid*`, int` *discipline*`, pid_t` *base_pid*`);`
	Pthread setup	`int frs_pthread_insert(frs_t*` *frs*`, int` *minor_index*`, pthread_t` *target_pthread*`, int` *discipline*`, pthread_t` *base_pthread*`);`
Set error recovery options	Process setup	`int frs_setattr(frs_t*` *frs*`, int` *minor_frame*`, pid_t` *pid*`, frs_attr_t` *attribute*`, void*` *param*`);`
	Pthread setup	`int frs_pthread_setattr(frs_t*` *frs*`, int` *minor_frame*`, pthread_t` *pthread*`, frs_attr_t` *attribute*`, void*` *param*`);`
Join a frame scheduler (activity is ready to start)	Process or pthread execution	`int frs_join(frs_t*` *frs*`);`

Operation	Use	Frame Scheduler API
Start scheduling (all activities queued)	Process or pthread execution	`int frs_start(frs_t* frs);`
Yield control after completing activity	Process or pthread execution	`int frs_yield(void);`
Pause scheduling at end of minor frame	Process or pthread execution	`int frs_stop(frs_t* frs);`
Resume scheduling at next time-base interrupt	Process or pthread execution	`int frs_resume(frs_t* frs);`
Trigger a user-level frame scheduler interrupt	Process or pthread execution	`int frs_userintr(frs_t* frs);`
Interrogate a minor frame queue	Process or pthread query	`int frs_getqueuelen(frs_t* frs, int minor_index);`
	Process query	`int frs_readqueue(frs_t* frs, int minor_frame, pid_t *pidlist);`
	Pthread query	`int frs_pthread_readqueue(frs_t* frs, int minor_frame, pthread_t *pthreadlist);`
Retrieve error recovery options	Process query	`int frs_getattr(frs_t* frs, int minor_frame, pid_t pid, frs_attr_t attribute, void* param);`
	Pthread query	`int frs_pthread_getattr(frs_t* frs, int minor_frame, pthread_t pthread, frs_attr_t attribute, void* param);`
Destroy frame scheduler and send `SIGKILL` to its frame scheduler controller	Process or pthread teardown	`int frs_destroy(frs_t* frs);`
Remove a process or thread from a queue	Process teardown	`int frs_premove(frs_t* frs, int minor_frame, pid_t remove_pid);`
	Pthread teardown	`int frs_pthread_remove(frs_t* frs, int minor_frame, pthread_t remove_pthread);`
Register a thread	Pthread setup	`int frs_pthread_register(void)`

Thread Execution

Example 5-1 shows the basic structure of an activity thread that is queued to a frame scheduler.

Example 5-1 Skeleton of an Activity Thread

```
/* Initialize data structures etc. */
frs_join(scheduler-handle)
do
{
   /* Perform the activity. */
   frs_yield();
} while(1);
_exit();
```

When the thread is ready to start real-time execution, it calls `frs_join()`. This call blocks until all queued threads are ready and scheduling begins. When `frs_join()` returns, the thread is running in its first minor frame. For more information, see "Starting Multiple Schedulers" on page 70 and the `frs_join`(3) man page.

Note: Each thread of a pthreaded application (including the controller thread) must first call `frs_pthread_register()` before making any other calls to the frame scheduler. In addition, each activity thread must complete its call to `frs_pthread_register` before the controller thread calls `frs_pthread_enqueue`.

The thread then performs whatever activity is needed to complete the minor frame and calls `frs_yield()`. This gives up control of the CPU until the next minor frame where the thread is queued and executes. For more information, see the `frs_yield`(3) man page.

An activity thread is never preempted by the frame scheduler within a minor frame. As long as it yields before the end of the frame, it can do its assigned work without interruption from other activity threads. (However, it can be interrupted by hardware interrupts, if they are allowed in that CPU.) The frame scheduler preempts the thread at the end of the minor frame.

When a very short minor frame interval is used, it is possible for a thread to have an overrun error in its first frame due to cache misses. A simple variation on the basic structure shown in Example 5-1 is to spend the first minor frame touching a set of important data structures in order to “warm up” the cache. This is sketched in Example 5-2.

Example 5-2 Alternate Skeleton of an Activity Thread

```
/* Initialize data structures etc. */
frs_join(scheduler-handle); /* Much time could pass here. */
/* First frame: merely touch important data structures. */
do
{
   frs_yield();
   /* Second and later frames: perform the activity. */
} while(1);
_exit();
```

When an activity thread is scheduled on more than one minor frame in a major frame, it can be designed to do nothing except warm up the cache in the entire first major frame. To do this, the activity thread function must know how many minor frames it is scheduled on and call `frs_yield()` a corresponding number of times in order to pass the first major frame.

Scheduling Within a Minor Frame

Threads in a minor frame queue are dispatched in the order that they appear on the queue (priority is irrelevant). Queue ordering can be modified as follows:

- Appending a thread at the end of the queue with `frs_pthread_enqueue()` or `frs_enqueue()`
- Inserting a thread after a specific target thread via `frs_pthread_insert()` or `frs_pinsert()`
- Deleting a thread in the queue with `frs_pthread_remove()` or `frs_premove()`

See "Managing Activity Threads" on page 71 and the `frs_enqueue`(3), `frs_pinsert`(3), and `frs_premove`(3) man pages.

Scheduler Flags `frs_run` and `frs_yield`

The frame scheduler keeps two status flags per queued thread: `frs_run` and `frs_yield`:

- If a thread is ready to run when its turn comes, it is dispatched and its `frs_run` flag is set to indicate that this thread has run at least once within this minor frame.

- When a thread yields, its `frs_yield` flag is set to indicate that the thread has released the processor. It is not activated again within this minor frame.

If a thread is not ready (usually because it is blocked waiting for I/O, a semaphore, or a lock), it is skipped. Upon reaching the end of the queue, the scheduler goes back to the beginning, in a round-robin fashion, searching for threads that have not yielded and may have become ready to run. If no ready threads are found, the frame scheduler goes into idle mode until a thread becomes available or until an interrupt marks the end of the frame.

Detecting Overrun and Underrun

When a time base interrupt occurs to indicate the end of the minor frame, the frame scheduler checks the flags for each thread. If the `frs_run` flag has not been set, that thread never ran and therefore is a candidate for an *underrun exception*. If the `frs_run` flag is set but the `frs_yield` flag is not, the thread is a candidate for an *overrun exception*.

Whether these exceptions are declared depends on the scheduling discipline assigned to the thread. For more information, see "Using the Scheduling Disciplines" on page 74.

At the end of a minor frame, the frame scheduler resets all `frs_run` flags, except for those of threads that use the continuable discipline in that minor frame. For those threads, the residual `frs_yield` flags keeps the threads that have yielded from being dispatched in the next minor frame.

Underrun and overrun exceptions are typically communicated via Linux signals. For more information, see "Using Signals Under the Frame Scheduler" on page 89.

Estimating Available Time

It is up to the application to make sure that all the threads queued to any minor frame can actually complete their work in one minor-frame interval. If there is too much work for the available CPU cycles, overrun errors will occur.

Estimation is somewhat simplified by the fact that a restricted CPU will only execute threads specifically pinned to it, along with a few CPU-specific kernel threads. You must estimate the maximum time each thread can consume between one call to `frs_yield()` and the next.

Frame scheduler threads do compete for CPU cycles with I/O interrupts on the same CPU. If you direct I/O interrupts away from the CPU, the only competition for CPU

cycles (other than a very few essential interrupts and CPU-specific kernel threads) is the overhead of the frame scheduler itself, and it has been carefully optimized to reduce overhead.

Alternatively, you may assign specific I/O interrupts to a CPU used by the frame scheduler. In that case, you must estimate the time that interrupt service will consume and allow for it.

Synchronizing Multiple Schedulers

When the activities of one frame cannot be completed by one CPU, you must recruit additional CPUs and execute some activities concurrently. However, it is important that each of the CPUs have the same time base, so that each starts and ends frames at the same time.

You can create one master frame scheduler that owns the time base and one CPU, and as many synchronized (slave) frame schedulers as you need, each managing an additional CPU. The slave schedulers take their time base from the master, so that all start minor frames at the same instant.

Each frame scheduler requires its own controller thread. Therefore, to create multiple, synchronized frame schedulers, you must create a controller thread for the master and each slave frame scheduler.

Each frame scheduler has its own queues of threads. A given thread can be queued to only one CPU. (However, you can create multiple threads based on the same code, and queue each to a different CPU.) All synchronized frame schedulers use the same number of minor frames per major frame, which is taken from the definition of the master frame scheduler.

Starting a Single Scheduler

A single frame scheduler is created when the frame scheduler controller thread calls `frs_create_master()` or `frs_create()`. The frame scheduler controller calls `frs_pthread_enqueue()` or `frs_enqueue()` one or more times to notify the new frame scheduler of the threads to schedule in each of the minor frames. The frame scheduler controller calls `frs_start()` when it has queued all the threads. Each scheduled thread must call `frs_join()` after it has initialized and is ready to be scheduled.

Each activity thread must be queued to at least one minor frame before it can join the frame scheduler via `frs_join()`. After all threads have called `frs_join()` and the controller has called `frs_start()`, scheduling of worker threads in the first minor frame occurs after the second interrupt arrives.

Note: The first interrupt is used to drive the frame scheduler's internal processing during which time no scheduling occurs.

For more information about these functions, see the `frs_enqueue`(3), `frs_join`(3), and `frs_start`(3) man pages.

Starting Multiple Schedulers

A frame scheduler cannot start dispatching activities until the following have occurred:

- The frame scheduler controller has queued all the activity threads to their minor frames
- All the queued threads have done their own initial setup and have joined

When multiple frame schedulers are used, none can start until all are ready.

Each frame scheduler controller notifies its frame scheduler that it has queued all activities by calling `frs_start()`. Each activity thread signals its frame scheduler that it is ready to begin real-time processing by calling `frs_join()`.

A frame scheduler is ready when it has received one or more `frs_pthread_enqueue()` or `frs_enqueue()` calls, a matching number of `frs_join()` calls, and an `frs_start()` call for each frame scheduler. Each slave frame scheduler notifies the master frame scheduler when it is ready. When all the schedulers are ready, the master frame scheduler gives the downbeat and the first minor frame begins.

Note: After all threads have called `frs_join()` and the controller has called `frs_start()`, scheduling of worker threads in the first minor frame does not occur until the second interrupt arrives. The first interrupt is used to drive the frame scheduler's internal processing during which time no scheduling occurs.

Pausing Frame Schedulers

Any frame scheduler can be made to pause and restart. Any thread (typically but not necessarily the frame scheduler controller) can call `frs_stop()`, specifying a particular frame scheduler. That scheduler continues dispatching threads from the current minor frame until all have yielded. Then it goes into an idle loop until a call to `frs_resume()` tells it to start. It resumes on the next time-base interrupt, with the next minor frame in succession. For more information, see the `frs_stop`(3) and `frs_resume`(3) man pages.

Note: If there is a thread running background discipline in the current minor frame, it continues to execute until it yields or is blocked on a system service. See "Background Discipline" on page 75.

Because a frame scheduler does not stop until the end of a minor frame, you can stop and restart a group of synchronized frame schedulers by calling `frs_stop()` for each one before the end of a minor frame. There is no way to restart all of a group of schedulers with the certainty that they start up on the same time-base interrupt.

Managing Activity Threads

The frame scheduler control thread identifies the initial set of activity threads by calling `frs_pthread_enqueue()` or `frs_enqueue()` prior to starting the frame scheduler. All the queued threads must call `frs_join()` before scheduling can begin. However, the frame scheduler controller can change the set of activity threads dynamically while the frame scheduler is working, using the following functions:

`frs_getqueuelen()`	Gets the number of threads currently in the queue for a specified minor frame
`frs_pthread_readqueue()` or `frs_readqueue()`	Returns the ID values of all queued threads for a specified minor frame as a vector of integers
`frs_pthread_remove()` or `frs_premove()`	Removes a thread (specified by its ID) from a minor frame queue
`frs_pthread_insert()` or `frs_pinsert()`	Inserts a thread (specified by its ID and discipline) into a given position in a minor frame queue

Using these functions, the frame scheduler controller can change the queueing discipline (overrun, underrun, continuable) of a thread by removing it and inserting it with a new discipline. The frame scheduler controller can suspend a thread by removing it from its queue or can restart a thread by putting it back in its queue.

Note: When an activity thread is removed from the last or only queue it was in, it no longer is dispatched by the frame scheduler. When an activity thread is removed from a queue, a signal may be sent to the removed thread (see "Handling Signals in an Activity Thread" on page 90). If a signal is sent to it, it begins executing in its specified or default signal handler; otherwise, it begins executing following `frs_yield()`. After being returned to the Linux scheduler, a call to a frame scheduler function such as `frs_yield()` returns an error (this also can be used to indicate the resumption of normal scheduling).

The frame scheduler controller can also queue new threads that have not been scheduled before. The frame scheduler does not reject an `frs_pthread_insert()` or `frs_pinsert()` call for a thread that has not yet joined the scheduler. However, a thread must call `frs_join()` before it can be scheduled. For more information, see the `frs_pinsert`(3) man page.

If a queued thread is terminated for any reason, the frame scheduler removes the thread from all queues in which it appears.

Selecting a Time Base

The program specifies an interrupt source for the time base when it creates the master (or only) frame scheduler. The master frame scheduler initializes the necessary hardware resources and redirects the interrupt to the appropriate CPU and handler.

The frame scheduler time base is fundamental because it determines the duration of a minor frame, and hence the frame rate of the program. This section explains the different time bases that are available.

When you use multiple, synchronized frame schedulers, the master frame scheduler distributes the time-base interrupt to each synchronized CPU. This ensures that minor-frame boundaries are synchronized across all the frame schedulers.

This section discusses the following:

- "High-Resolution Timer" on page 73
- "External Interrupts as a Time Base" on page 73

High-Resolution Timer

The real-time clock (RTC) is synchronous across all processors and is ideal to drive synchronous schedulers. REACT uses the RTC for its frame scheduler high-resolution timer solution.

Note: Frame scheduler applications cannot use POSIX high-resolution timers.

To use the RTC, specify `FRS_INTRSOURCE_CCTIMER` and the minor frame interval in microseconds to `frs_create_master()` or `frs_create()`. The maximum frame rate supported by a timer is 2000 Hz.

The high-resolution timers in all CPUs are synchronized automatically.

Note: Altix XE servers do not have a built-in high-resolution timer. A `pci-rt` external interrupt card is supplied by SGI and is required for generation of the high-resolution timer interrupts. Each `pci-rt` can generate interrupts at one set frequency, so a `pci-rt` card is required for each asynchronous frame scheduler running on a system.

External Interrupts as a Time Base

To use external interrupts as a time base, do the following:

1. Load `ioc4_extint` to load the external interrupts modules.
2. Open the appropriate external interrupts device file. For example:

```
if ((fd = open("/dev/extint0", O_RDONLY)) < 0) {
        perror("Open EI control file");
        return 1;
}
```

3. Specify `FRS_INTRSOURCE_EXTINTR` as the `intr_source` and pass the returned file descriptor as the `intr_qualifier` to `frs_create_master` or `frs_create`.

The CPU receiving the interrupt allocates it simultaneously to the synchronized schedulers. If other IOC4 devices are also in use, you should redirect IOC4 interrupts to a non-frame-scheduled CPU in order to avoid jitter and delay.

Note: After all threads have called `frs_join()` and the controller has called `frs_start()`, scheduling of worker threads in the first minor frame does not occur until the second interrupt arrives. The first interrupt is used to drive the frame scheduler's internal processing during which time no scheduling occurs.

For more information, see Chapter 3, "External Interrupts" on page 19.

Using the Scheduling Disciplines

When a frame scheduler controller thread queues an activity thread to a minor frame using `frs_pthread_enqueue()` or `frs_enqueue()`, it must specify a *scheduling discipline* that tells the frame scheduler how the thread is expected to use its time within that minor frame.

The disciplines are as follows:

- "Real-Time Discipline" on page 74
- "Background Discipline" on page 75
- "Underrunable Discipline" on page 76
- "Overrunnable Discipline" on page 76
- "Continuable Discipline" on page 76

Real-Time Discipline

In the real-time discipline, an activity thread starts during the minor frame in which it is queued, completes its work, and yields within the same minor frame. If the thread is not ready to run (for example, if it is blocked on I/O) during the entire minor frame, an *underrun exception* is said to occur. If the thread fails to complete its work and yield within the minor frame interval, an *overrun exception* is said to occur.

Note: If an activity thread becomes blocked by other than an `frs_yield()` call (and therefore is not ready to run) and later becomes unblocked outside of its minor frame slot, it will run assuming that no other threads are available to run (similar to "Background Discipline" on page 75) until it yields or a new minor frame begins.

This model could describe a simple kind of simulator in which certain activities (such as poll the inputs, calculate the new status, and update the display) must be repeated in the same order during every frame. In this scenario, each activity must start and must finish in every frame. If one fails to start, or fails to finish, the real-time program is broken and must take action.

However, realistic designs need the flexibility to have threads with the following characteristics:

- Need not start every frame; for instance, threads that sleep on a semaphore until there is work for them to do
- May run longer than one minor frame
- Should run only when time is available, and whose rate of progress is not critical

The other disciplines are used, in combination with real-time and with each other, to allow these variations.

Background Discipline

The background discipline is mutually exclusive with the other disciplines. The frame scheduler dispatches a background thread only when all other threads queued to that minor frame have run and have yielded. Because the background thread cannot be sure it will run and cannot predict how much time it will have, the concepts of underrun and overrun do not apply to it.

Note: A thread with the background discipline must be queued to its frame following all non-background threads. Do not queue a real-time thread after a background thread.

Underrunable Discipline

You specify the underrunable discipline in the following cases:

- When a thread needs to run only when an event has occurred, such as a lock being released or a semaphore being posted
- When a thread may need more than one minor frame (see "Using Multiple Consecutive Minor Frames" on page 77)

You specify the underrunable discipline with the real-time discipline to prevent detection of underrun exceptions. When you specify real time plus underrunable, the thread is not required to start in that minor frame. However, if it starts, it is required to yield before the end of the frame or an overrun exception is raised.

Overrunnable Discipline

You specify the overrunnable discipline in the following cases:

- When it truly does not matter if the thread fails to complete its work within the minor frame—for example, a calculation of a game strategy that, if it fails to finish, merely makes the computer a less dangerous opponent
- When a thread may need more than one minor frame (see "Using Multiple Consecutive Minor Frames" on page 77)

You specify an overrunnable discipline with a real-time discipline to prevent detection of overrun exceptions. When you specify overrunnable plus real-time, the thread is not required to call `frs_yield()` before the end of the frame. Even so, the thread is preempted at the end of the frame. It does not have a chance to run again until the next minor frame in which it is queued. At that time it resumes where it was preempted, with no indication that it was preempted.

Continuable Discipline

You specify continuable discipline with real-time discipline to prevent the frame scheduler from clearing the flags at the end of this minor frame (see "Scheduling Within a Minor Frame" on page 67).

The result is that, if the thread yields in this frame, it need not run or yield in the following frame. The residual `frs_yield` flag value, carried forward to the next

frame, applies. You specify continuable discipline with other disciplines in order to let a thread execute just once in a block of consecutive minor frames.

Using Multiple Consecutive Minor Frames

There are cases when a thread sometimes or always requires more than one minor frame to complete its work. Possibly the work is lengthy, or possibly the thread could be delayed by a system call or a lock or semaphore wait.

You must decide the absolute maximum time the thread could consume between starting up and calling `frs_yield()`. If this is unpredictable, or if it is predictably longer than the major frame, the thread cannot be scheduled by the frame scheduler. Hence, it should probably run on another CPU under the Linux real-time scheduler.

However, when the worst-case time is bounded and is less than the major frame, you can queue the thread to enough consecutive minor frames to allow it to finish. A combination of disciplines is used in these frames to ensure that the thread starts when it should, finishes when it must, and does not cause an error if it finishes early.

The discipline settings should be as follows:

Frame	Description
First	Real-time + overrunnable + continuable The thread must start in this frame (not underrunable) but is not required to yield (overrunnable). If it yields, it is not restarted in the following minor frame (continuable).
Intermediate	Realtime + underrunable + overrunnable + continuable The thread need not start (it might already have yielded, or might be blocked) but is not required to yield. If it does yield or if it had yielded in a preceding minor frame, it is not restarted in the following minor frame (continuable).
Final	Realtime + underrunable The thread need not start (it might already have yielded) but if it starts, it must yield in this frame (not overrunnable). The thread can start a new run in the next minor frame to which it is queued (not continuable).

A thread can be queued for one or more of these multiframe sequences in one major frame. For example, suppose that the minor frame rate is 60 Hz and a major frame

contains 60 minor frames (1 Hz). You have a thread that should run at a rate of 5 Hz and can use up to 3/60 second at each dispatch. You can queue the thread to 5 sequences of 3 consecutive frames each. It could start in frames 0, 12, 24, 36, and 48. Frames 1, 13, 25, 37, and 49 could be intermediate frames, and 2, 14, 26, 38, and 50 could be final frames.

Designing an Application for the Frame Scheduler

When using the frame scheduler, consider the following guidelines when designing a real-time application:

1. Determine the programming model for implementing the activities in the program, choosing among POSIX threads or SVR4 `fork()` calls. (You cannot mix pthreads and other disciplines within the program.)
2. Partition the program into activities, where each activity is an independent piece of work that can be done without interruption.

   For example, in a simple vehicle simulator, activities might include the following:

   - Poll the joystick
   - Update the positions of moving objects
   - Cull the set of visible objects
3. Decide the relationships among the activities, as follows:
   - Some must be done once per minor frame, others less frequently
   - Some must be done before or after others
   - Some may be conditional (for example, an activity could poll a semaphore and do nothing unless an event had completed)
4. Estimate the worst-case time required to execute each activity. Some activities may need more than one minor frame interval (the frame scheduler allows for this).
5. Schedule the activities. If all are executed sequentially, will they complete in one major frame? If not, choose activities that can execute concurrently on two or more CPUs, and estimate again. You may have to change the design in order to get greater concurrency.

When the design is complete, implement each activity as an independent thread that communicates with the others using shared memory, semaphores, and locks.

A controller thread creates, stops, and resumes the frame scheduler. The controller thread can also interrogate and receive signals from the frame scheduler.

A frame scheduler seizes its assigned CPU, isolates it, and controls the scheduling on it. It waits for all queued threads to initialize themselves and join the scheduler. The frame scheduler begins dispatching the threads in the specified sequence during each frame interval. Errors are monitored (such as a thread that fails to complete its work within its frame) and a specified action is taken when an error occurs. Typically, the error action is to send a signal to the controller thread.

Preparing the System

Before a real-time program executes, you must do the following:

1. Choose the CPUs that the real-time program will use. CPU 0 (at least) must be reserved for Linux system functions.
2. Decide which CPUs will handle I/O interrupts. By default, Linux distributes I/O interrupts across all available processors as a means of balancing the load (referred to as *spraying interrupts*). You should redirect I/O interrupts away from CPUs that are used for real-time programs. For more information, see "Redirect Interrupts" on page 48.
3. If you are using an external interrupt as a time base, make sure it is redirected to the CPU of the master frame scheduler. For more information, see "External Interrupts as a Time Base" on page 73.
4. Make sure that none of the real-time CPUs is managing the clock. Normally, the responsibility of handling 10–ms scheduler interrupts is given to CPU 0. For more information, see "Avoid the Clock Processor (CPU 0)" on page 47.
5. Restrict and isolate the real-time CPUs, as described in "Restrict, Isolate, and Shield CPUs" on page 49.
6. Load the `frs` kernel module:

   ```
   [root@linux root]# modprobe frs
   ```

   Note: You must perform this step after each system boot.

7. If you are using external interrupts as a time base or if you are running the frame scheduler on an Altix XE server, you must load the `ioc4_extint` kernel module:

```
[root@linux root]# modprobe ioc4_extint
```

Implementing a Single Frame Scheduler

When the activities of a real-time program can be handled within a major frame interval by a single CPU, the program requires only one frame scheduler. The programs found in `/usr/share/react/frs/examples` provide examples of implementing a single frame scheduler.

Typically, a program has a top-level controller thread to handle startup and termination, and one or more activity threads that are dispatched by the frame scheduler. The activity threads are typically lightweight pthreads, but that is not a requirement; they can also be created with `fork()`. (They need not be children of the controller thread.) For examples, see `/usr/share/react/frs/examples`.

In general, these are the steps for setting up a single frame scheduler:

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, semaphores, locks, and other resources.
2. Lock the shared address space segments. (When `fork()` is used, each child process must lock its own address space.)
3. If using pthreads, create a controller thread; otherwise, the initial thread of execution may be used as the controller thread.
   - Create a controller thread using `pthread_create()` and the attribute structure you just set up. See the `pthread_create`(3P) man page for details.
   - Exit the initial thread, because it cannot execute any frame scheduler operations.
4. Create the frame scheduler using `frs_create_master()`, `frs_create_vmaster()`, or `frs_create()`. See the `frs_create`(3) man page.
5. Create the activity threads using one of the following interfaces, depending on the thread model being used:
   - `pthread_create()`
   - `fork()`

6. Queue the activity threads on the target minor frame queues, using `frs_pthread_enqueue()` or `frs_enqueue()`.
7. Optionally, initialize the frame scheduler signal handler to catch frame overrun, underrun, and activity dequeue events (see "Setting Frame Scheduler Signals" on page 90 and "Setting Exception Policies" on page 86). The handlers are set at this time, after creation of the activity threads, so that the activity threads do not inherit them.
8. Use `frs_start()` to enable scheduling. For more information, see Table 5-1.
9. Have the activity threads call `frs_join()`. The frame scheduler begins scheduling processes as soon as all the activity threads have called `frs_join()`.
10. Wait for error signals from the frame scheduler and for the termination of child processes.
11. Use `frs_destroy()` to terminate the frame scheduler.
12. Perform program cleanup as desired.

See `/usr/share/react/frs/examples`.

Implementing Synchronized Schedulers

When the real-time application requires the power of multiple CPUs, you must add one more level to the program design for a single CPU. The program creates multiple frame schedulers, one master and one or more synchronized slaves.

This section discusses the following:

- "Synchronized Scheduler Concepts" on page 82
- "Master Controller Thread" on page 82
- "Slave Controller Thread" on page 83

Synchronized Scheduler Concepts

The first frame scheduler provides the time base for the others. It is called the *master scheduler*. The other schedulers take their time base interrupts from the master, and so are called *slaves*. The combination is called a *sync group*.

No single thread may create more than one frame scheduler. This is because every frame scheduler must have a unique frame scheduler controller thread to which it can send signals. As a result, the program has the following types of threads:

- A master controller thread that sets up global data and creates the master frame scheduler
- One slave controller thread for each slave frame scheduler
- Activity threads

The master frame scheduler must be created before any slave frame schedulers can be created. Slave frame schedulers must be specified to have the same time base and the same number of minor frames as the master.

Slave frame schedulers can be stopped and restarted independently. However, when any scheduler (master or slave) is destroyed, all are immediately destroyed.

Master Controller Thread

The master controller thread performs these steps:

1. Initializes a global resource. One global resource is the thread ID of the master controller thread.
2. Creates the master frame scheduler using either the `frs_create_master()` or `frs_create_vmaster()` call and stores its handle in a global location.
3. Creates one slave controller thread for each synchronized CPU to be used.
4. Creates the activity threads that will be scheduled by the master frame scheduler and queues them to their assigned minor frames.
5. Sets up signal handlers for signals from the frame scheduler. See "Using Signals Under the Frame Scheduler" on page 89.
6. Uses `frs_start()` to tell the master frame scheduler that its activity threads are all queued and ready to commence scheduling. See Table 5-1.

The master frame scheduler starts scheduling threads as soon as all threads have called `frs_join()` for their respective schedulers.

7. Waits for error signals.
8. Uses `frs_destroy()` to terminate the master frame scheduler.
9. Performs any desired program cleanup.

Slave Controller Thread

Each slave controller thread performs these steps:

1. Creates a synchronized frame scheduler using `frs_create_slave()`, specifying information about the master frame scheduler stored by the master controller thread. The master frame scheduler must exist. A slave frame scheduler must specify the same time base and number of minor frames as the master frame scheduler.
2. Changes the frame scheduler signals or exception policy, if desired. See "Setting Frame Scheduler Signals" on page 90 and "Setting Exception Policies" on page 86.
3. Creates the activity threads that are scheduled by this slave frame scheduler and queues them to their assigned minor frames.
4. Sets up signal handlers for signals from the slave frame scheduler.
5. Use `frs_start()` to tell the slave frame scheduler that all activity threads have been queued.

   The slave frame scheduler notifies the master when all threads have called `frs_join()`. When the master frame scheduler starts broadcasting interrupts, scheduling begins.
6. Waits for error signals.
7. Uses `frs_destroy()` to terminate the slave frame scheduler.

For an example of this kind of program structure, refer to `/usr/share/react/frs/examples`.

Tip: In this design sketch, the knowledge of which activity threads to create, and on which frames to queue them, is distributed throughout the code, where it might be hard to maintain. However, it is possible to centralize the plan of schedulers, activities, and frames in one or more arrays that are statically initialized. This improves the maintainability of a complex program.

Handling Frame Scheduler Exceptions

The frame scheduler control thread for a scheduler controls the handling of the overrun and underrun exceptions. It can specify how these exceptions should be handled and what signals the frame scheduler should send. These policies must be set before the scheduler is started. While the scheduler is running, the frame scheduler controller can query the number of exceptions that have occurred.

This section discusses the following:

- "Exception Types" on page 84
- "Exception Handling Policies" on page 85
- "Setting Exception Policies" on page 86
- "Querying Counts of Exceptions" on page 87

Exception Types

The overrun exception indicates that a thread failed to yield in a minor frame where it was expected to yield and was preempted at the end of the frame. An overrun exception indicates that an unknown amount of work that should have been done was not done, and will not be done until the next frame in which the overrunning thread is queued.

The underrun exception indicates that a thread that should have started in a minor frame did not start. The thread may have terminated or (more likely) it was blocked in a wait because of an unexpected delay in I/O or because of a deadlock on a lock or semaphore.

Exception Handling Policies

The frame scheduler control thread can establish one of four policies for handling overrun and underrun exceptions. When it detects an exception, the frame scheduler can do the following:

- Send a signal to the controller
- Inject an additional minor frame
- Extend the frame by a specified number of microseconds
- Steal a specified number of microseconds from the following frame

By default, it sends a signal. The scheduler continues to run. The frame scheduler control thread can then take action, such as terminating the frame scheduler. For more information, see "Setting Frame Scheduler Signals" on page 90.

Injecting a Repeat Frame

The policy of injecting an additional minor frame can be used with any time base. The frame scheduler inserts another complete minor frame, essentially repeating the minor frame in which the exception occurred. In the case of an overrun, the activity threads that did not finish have another frame's worth of time to complete. In the case of an underrun, there is that much more time for the waiting thread to wake up. Because exactly one frame is inserted, all other threads remain synchronized to the time base.

Extending the Current Frame

The policies of extending the frame, either with more time or by stealing time from the next frame, are allowed only when the time base is a high-resolution timer. For more information, see "Selecting a Time Base" on page 72.

When adding time, the current frame is made longer by a fixed amount of time. Because the minor frame becomes a variable length, it is possible for the frame scheduler to drop out of synchronization with an external device.

When stealing time from the following frame, the frame scheduler returns to the original time base at the end of the following minor frame provided that the threads queued to that following frame can finish their work in a reduced amount of time. If they do not, the frame scheduler steals time from the next frame.

Dealing With Multiple Exceptions

You decide how many consecutive exceptions are allowed within a single minor frame. After injecting, stretching, or stealing time that many times, the frame scheduler stops trying to recover and sends a signal instead.

The count of exceptions is reset when a minor frame completes with no remaining exceptions.

Setting Exception Policies

The `frs_pthread_setattr()` or `frs_setattr()` function is used to change exception policies. This function must be called before the frame scheduler is started. After scheduling has begun, an attempt to change the policies or signals is rejected.

In order to allow for future enhancements, `frs_pthread_setattr()` or `frs_setattr()` accepts arguments for minor frame number and thread ID; however it currently allows setting exception policies only for all policies and all minor frames. The most significant argument to it is the `frs_recv_info` structure, declared with the following fields:

```
typedef struct frs_recv_info {
    mfbe_rmode_t  rmode;      /* Basic recovery mode */
    mfbe_tmode_t  tmode;      /* Time expansion mode */
    uint          maxcerr;    /* Max consecutive errors */
    uint          xtime;      /* Recovery extension time */
} frs_recv_info_t;
```

The recovery modes and other constants are declared in `/usr/include/frs.h`. The function in Example 5-3 sets the policy of injecting a repeat frame. The caller specifies only the frame scheduler and the number of consecutive exceptions allowed.

Example 5-3 Function to Set `INJECTFRAME` Exception Policy

```
int
setInjectFrameMode(frs_t *frs, int consecErrs)
{
  frs_recv_info_t work;
  bzero((void*)&work,sizeof(work));
  work.rmode = MFBERM_INJECTFRAME;
  work.maxcerr = consecErrs;
  return frs_setattr(frs,0,0,FRS_ATTR_RECOVERY,(void*)&work);
}
```

The function in Example 5-4 sets the policy of stretching the current frame (a function to set the policy of stealing time from the next frame is nearly identical). The caller specifies the frame scheduler, the number of consecutive exceptions, and the stretch time in microseconds.

Example 5-4 Function to Set `STRETCH` Exception Policy

```
int
setStretchFrameMode(frs_t *frs,int consecErrs,uint microSecs)
{
  frs_recv_info_t work;
  bzero((void*)&work,sizeof(work));
  work.rmode = MFBERM_EXTENDFRAME_STRETCH;
  work.tmode = EFT_FIXED; /* only choice available */
  work.maxcerr = consecErrs;
  work.xtime = microSecs;
  return frs_setattr(frs,0,0,FRS_ATTR_RECOVERY,(void*)&work);
}
```

Querying Counts of Exceptions

When you set a policy that permits exceptions, the frame scheduler controller thread can query for counts of exceptions. This is done with a call to `frs_pthread_getattr()` or `frs_getattr()`, passing the handle to the frame scheduler, the number of the minor frame and the thread ID of the thread within that frame.

The values returned in a structure of type `frs_overrun_info_t` are the counts of overrun and underrun exceptions incurred by that thread in that minor frame. In order to find the count of all overruns in a given minor frame, you must sum the counts for all threads queued to that frame. If a thread is queued to more than one minor frame, separate counts are kept for it in each frame.

The function in Example 5-5 takes a frame scheduler handle and a minor frame number. It gets the list of thread IDs queued to that minor frame, and returns the sum of all exceptions for all of them.

Example 5-5 Function to Return a Sum of Exception Counts (pthread Model)

```
#define THE_MOST_TIDS 250
int
totalExcepts(frs_t * theFRS, int theMinor)
{
    int numTids = frs_getqueuelen(theFRS, theMinor);
    int j, sum;
    pthread_t allTids[THE_MOST_TIDS];
    if ( (numTids <= 0) || (numTids > THE_MOST_TIDS) )
        return 0; /* invalid minor #, or no threads queued? */

    if (frs_pthread_readqueue(theFRS, theMinor, allTids) == -1)
        return 0; /* unexpected problem with reading IDs */

    for (sum = j = 0; j<numTids; ++j)
    {
        frs_overrun_info_t work;
        frs_pthread_getattr(theFRS     /* the scheduler */
                    theMinor,          /* the minor frame */
                    allTids[j],        /* the threads */
                    FRS_ATTR_OVERRUNS, /* want counts */
                    &work);            /* put them here */
        sum += (work.overruns + work.underruns);
    }
    return sum;
}
```

Note: The frame scheduler read queue functions return the number of threads present on the queue at the time of the read. Applications can use this returned value to eliminate calls to `frs_getqueuelen()`.

Using Signals Under the Frame Scheduler

The frame scheduler itself sends signals to the threads using it. Threads can communicate by sending signals to each other. In brief, a frame scheduler sends signals to indicate the following:

- The frame scheduler has been terminated
- An overrun or underrun has been detected
- A thread has been dequeued

The rest of this section describes how to specify the signal numbers and how to handle the signals:

- "Handling Signals in the Frame Scheduler Controller" on page 89
- "Handling Signals in an Activity Thread" on page 90
- "Setting Frame Scheduler Signals" on page 90
- "Handling a Sequence Error" on page 91

Handling Signals in the Frame Scheduler Controller

When a frame scheduler detects an overrun or underrun exception from which it cannot recover, and when it is ready to terminate, it sends a signal to the frame scheduler controller.

Tip: Child processes inherit signal handlers from the parent, so a parent should not set up handlers prior to `fork()` unless they are meant to be inherited.

The frame scheduler controller for a synchronized frame scheduler should have handlers for underrun and overrun signals. The handler could report the error and issue `frs_destroy()` to shut down its scheduler. A frame scheduler controller for a synchronized scheduler should use the default action for `SIGHUP` (exit) so that completion of the `frs_destroy()` quietly terminates the frame scheduler controller.

The frame scheduler controller for the master (or only) frame scheduler should catch underrun and overrun exceptions, report them, and shut down its scheduler.

When a frame scheduler is terminated with `frs_destroy()`, it sends `SIGKILL` to its frame scheduler controller. This cannot be changed and `SIGKILL` cannot be handled.

Hence `frs_destroy()` is equivalent to termination for the frame scheduler controller.

Handling Signals in an Activity Thread

A frame scheduler can send a signal to an activity thread when the thread is removed from any queue using `frs_pthread_remove()` or `frs_premove()`. The scheduler can also send a signal to an activity thread when it is removed from the last or only minor frame to which it was queued (at which time it is scheduled only by Linux). For more information, see "Managing Activity Threads" on page 71.

In order to have these signals sent, the frame scheduler controller must set nonzero signal numbers for them, as discussed in "Setting Frame Scheduler Signals".

Setting Frame Scheduler Signals

The frame scheduler sends signals to the frame scheduler controller.

The signal numbers used for most events can be modified. Signal numbers can be queried using `frs_pthread_getattr(FRS_ATTR_SIGNALS)` or `frs_getattr(FRS_ATTR_SIGNALS)` and changed using `frs_pthread_setattr(FRS_ATTR_SIGNALS)` or `frs_setattr(FRS_ATTR_SIGNALS)`, in each case passing an `frs_signal_info` structure. This structure contains room for four signal numbers, as shown in Table 5-2.

Table 5-2 Signal Numbers Passed in `frs_signal_info_t`

Field Name	Signal Purpose	Default Signal Number
`sig_underrun`	Notify frame scheduler controller of underrun	`SIGUSR1`
`sig_overrun`	Notify frame scheduler controller of the overrun	`SIGUSR2`
`sig_dequeue`	Notify an activity thread that it has been dequeued with `frs_pthread_remove()` or `frs_premove()`	0 (do not send)
`sig_unframesched`	Notify an activity thread that it has been removed from the last or only queue in which it was queued	`SIGRTMIN`

Signal numbers must be changed before the frame scheduler is started. All the numbers must be specified to `frs_pthread_setattr()` or `frs_setattr()`, so the proper way to set any number is to first file the `frs_signal_info_t` using `frs_pthread_getattr()` or `frs_getattr()`. The function in Example 5-6 sets the signal numbers for overrun and underrun from its arguments.

Example 5-6 Function to Set Frame Scheduler Signals

```
int
setUnderOverSignals(frs_t *frs, int underSig, int overSig)
{
  int error;
  frs_signal_info_t work;
  error = frs_pthread_getattr(frs,0,0,FRS_ATTR_SIGNALS,(void*)&work);
  if (!error)
  {
    work.sig_underrun = underSig;
    work.sig_overrun = overSig;
    error = frs_pthread_setattr(frs,0,0,FRS_ATTR_SIGNALS,(void*)&work);
  }
  return error;
}
```

Handling a Sequence Error

When `frs_create_vmaster()` is used to create a frame scheduler triggered by multiple interrupt sources, a sequence error signal is dispatched to the controller thread if the interrupts come in out of order. For example, if the first and second minor frame interrupt sources are different, and the second minor frame's interrupt source is triggered before the first minor frame's interrupt source, then a sequence error has occurred.

This type of error condition is indicative of unrealistic time constraints defined by the interrupt information template.

The signal code that represents the occurrence of a sequence error is `SIGRTMIN+1`. This signal cannot be reset or disabled using the `frs_setattr()` interface.

Using Timers with the Frame Scheduler

Frame scheduler applications cannot use POSIX high-resolution timers. With other interval timers, signal delivery to an activity thread can be delayed, so timer latency is unpredictable.

If the frame scheduler controller is using timers, it should run on a node outside of those containing CPUs running frame scheduler worker threads.

Example 5-7 Minimal Activity Process as a Timer

```
frs_join(scheduler-handle)
do {
   usvsema(frs-controller-wait-semaphore);
   frs_yield();
} while(1);
_exit();
```

Disk I/O Optimization

A real-time program sometimes must perform disk I/O under tight time constraints and without affecting the timing of other activities such as data collection. This chapter covers techniques that can help you meet these performance goals:

- "Memory-Mapped I/O" on page 93
- "Asynchronous I/O" on page 93

Memory-Mapped I/O

When an input file has a fixed size, the simplest as well as the fastest access method is to map the file into memory. A file that represents a database (such as a file containing a precalculated table of operating parameters for simulated hardware) is best mapped into memory and accessed as a memory array. A mapped file of reasonable size can be locked into memory so that access to it is always fast.

You can also perform output on a memory-mapped file by storing into the memory image. When the mapped segment is also locked in memory, you control when the actual write takes place. Output happens only when the program calls `msync()` or changes the mapping of the file at the time that the modified pages are written. The time-consuming call to `msync()` can be made from an asynchronous process. For more information, see the `msync`(2) man page.

Asynchronous I/O

You can use asynchronous I/O to isolate the real-time processes in your program from the unpredictable delays caused by I/O. Asynchronous I/O in Linux strives to conform with the POSIX real-time specification 1003.1-2003.

This section discusses the following:

- "Conventional Synchronous I/O" on page 94
- "Asynchronous I/O Basics" on page 94

Conventional Synchronous I/O

Conventional I/O in Linux is synchronous; that is, the process that requests the I/O is blocked until the I/O has completed. The effects are different for input and for output.

For disk files, the process that calls `write()` is normally delayed only as long as it takes to copy the output data to a buffer in kernel address space. The device driver schedules the device write and returns. The actual disk output is asynchronous. As a result, most output requests are blocked for only a short time. However, since a number of disk writes could be pending, the true state of a file on disk is unknown until the file is closed.

In order to make sure that all data has been written to disk successfully, a process can call `fsync()` for a conventional file or `msync()` for a memory-mapped file. The process that calls these functions is blocked until all buffered data has been written. For more information, see the `fsync`(2) and `msync`(2) man pages.

Devices other than disks may block the calling process until the output is complete. It is the device driver logic that determines whether a call to `write()` blocks the caller, and for how long.

Asynchronous I/O Basics

A real-time process must read or write a device, but it cannot tolerate an unpredictable delay. One obvious solution can be summarized as "call `read()` or `write()` from a different process, and run that process in a different CPU." This is the essence of asynchronous I/O. You could implement an asynchronous I/O scheme of your own design, and you may wish to do so in order to integrate the I/O closely with your own configuration of processes and data structures. However, a standard solution is available.

Linux supports asynchronous I/O library calls that strive to conform with the POSIX real-time specification 1003.1-2003. You use relatively simple calls to initiate input or output.

For more information, see the `aio_read`(3) and `aio_write`(3) man pages.

PCI Devices

To perform programmed I/O on PCI devices on an SGI Altix system, do the following to determine the resource filename (`resource`*N*) and create an appropriate program to open the file and memory-map it:

1. Examine the output of the `lspci`(8) command to determine which device you want to map:

   a. Record the domain, bus, slot, and function for the device (this information will help you locate the appropriate resource address file).

      For example, for an SGI Altix server:

```
# lspci
0001:00:01.0 SCSI storage controller: LSI Logic / Symbios Logic SAS1064 PCI-X Fusion-MPT SAS (rev 02)
0001:00:02.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet (rev 10)
0001:00:02.1 Ethernet controller: Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet (rev 10)
0001:00:03.0 PCI bridge: IBM PCI-X to PCI-X Bridge (rev 03)
0001:01:01.0 USB Controller: NEC Corporation USB (rev 43)
0001:01:01.1 USB Controller: NEC Corporation USB (rev 43)
0001:01:01.2 USB Controller: NEC Corporation USB 2.0 (rev 04)
0001:01:02.0 IDE interface: Silicon Image, Inc. PCI0680 Ultra ATA-133 Host Controller (rev 02)
0002:00:01.0 Co-processor: Silicon Graphics, Inc. IOC4 I/O controller (rev 3e)
```

      The first field gives the information that is required to map the PCI registers into memory. The format is:

      Domain`:`*Bus*`:`*Slot*`.`*Function*

      The above example, the highlighted output of `0001:00:02.1` for the Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet card equates to domain 1, bus 0, slot 2, and function 1.

   b. Determine the `resource`*N* numbers from the `Region` numbers in the `lspci -vv` output. The `Region` value corresponds directly to each `resource`*N* value.

      In the following example, the `Region 0` output (highlighted) indicates that there is one `resource`*N* value, `resource0`:

```
# lspci -n -s 0001:00:02.1  -vv
0001:00:02.1 Class 0200: 14e4:1648 (rev 10)
 Subsystem: 10a9:801a
 Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B-
 Status: Cap+ 66MHz+ UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort- <TAbort- <MAbort- >SERR- <PERR-
 Latency: 64 (16000ns min), Cache Line Size: 128 bytes
 Interrupt: pin B routed to IRQ 62
 Region 0: Memory at 81c0730000 (64-bit, non-prefetchable) [size=64K]
 Capabilities: [40] PCI-X non-bridge device
  Command: DPERE- ERO- RBC=2048 OST=1
  Status: Dev=00:02.1 64bit+ 133MHz+ SCD- USC- DC=simple DMMRBC=2048 DMOST=1 DMCRS=16 RSCEM- 266MHz- 533MHz-
 Capabilities: [48] Power Management version 2
  Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot+,D3cold-)
  Status: D0 PME-Enable- DSel=0 DScale=1 PME-
 Capabilities: [50] Vital Product Data
 Capabilities: [58] Message Signalled Interrupts: Mask- 64bit+ Queue=0/3 Enable-
  Address: ac039016642c0034  Data: 248a
```

A device can have both 32-bit and 64-bit base address registers (BARs). If a BAR is mapping a 64-bit address space, then two 32-bit BARs are used to map that 64-bit `Region`. As a result, `Region` numbers may not be consecutive. For example, in the following `lspci` output, there are three `Region` values (`Region 0`, `Region 1` and `Region 3`):

```
# lspci -n -s 0001:00:01.0  -vv
0001:00:01.0 Class 0100: 1000:0050 (rev 02)
  Subsystem: 1000:1000
  Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B-
  Status: Cap+ 66MHz+ UDF- FastB2B- ParErr- DEVSEL=medium >TAbort- <TAbort- <MAbort- >SERR- <PERR-
  Region 0: I/O ports at 1001000 [disabled] [size=256]
  Region 1: Memory at 81c0700000 (64-bit, non-prefetchable) [size=16K]
  Region 3: Memory at 81c0710000 (64-bit, non-prefetchable) [size=64K]
  Expansion ROM at 81c0800000 [disabled] [size=1M]
...
```

There is no `Region 2` because the card's second BAR is mapping a 64-bit region and thus uses two 32-bit BARs to do so. In this example. there would be three corresponding resource numbers (`resource0`, `resource1`, and `resource3`) that would be used to memory-map the PCI registers.

Note: Only memory base-address registers (not I/O base-address registers) can be memory mapped. The base address must be page aligned.

2. Based on the information in step 1, determine the resource address file that you want to open:

   `/sys/bus/pci/devices/`*domain*`:`*bus*`:`*slot*`.`*function*`/resource`*N*

   For the Broadcom example above, the resource address file is:

   ```
   /sys/bus/pci/devices/0001:00:02.1/resource0
   ```

   In the case of the LSI Logic card example showing multiple `Region` values, there would be three `resource` files:

   ```
   /sys/bus/pci/devices/0001:00:01.0/resource0
   /sys/bus/pci/devices/0001:00:01.0/resource1
   /sys/bus/pci/devices/0001:00:01.0/resource3
   ```

3. Create a program that opens the appropriate resource file for the domain, bus, slot, function, and resource in which you are interested. For example, C the program for the Broadcom card could include the following lines:

```
sprintf(path, "/sys/bus/pci/devices/%04x:%02x:%02x.%x/%s",
       (unsigned)domain, (unsigned)bus, (unsigned)slot, (unsigned)function,
       "resource0");
if ((fd = open(path, O_RDWR)) == -1) {
  perror("Couldn't open resource file");
  exit(1);
}
```

4. Add a line to the program that will memory-map the opened file from offset 0. For example, in C:

```
ptr = mmap( NULL, getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

For a complete example, see Appendix D, "Reading MAC Addresses Sample Program" on page 175.

For details about kernel-level PCI device drivers, see the *Linux Device Driver Programmer's Guide-Porting to SGI Altix Systems*.

User-Level Interrupts

The user-level interrupt (ULI) facility allows a hardware interrupt to be handled by a user process.

A user process may register a function with the kernel, linked into the process in the normal fashion, to be called when a particular interrupt is received. The process, referred to as a *ULI process*, effectively becomes multithreaded, with the main process thread possibly running simultaneously with the interrupt handler thread. The interrupt handler is called asynchronously and has access only to the process's address space.

The ULI facility is intended to simplify the creation of device drivers for unsupported devices. ULIs can be written to respond to interrupts initiated from external interrupt ports. A programming error in the driver will result in nothing more serious than the termination of a process rather than crashing the entire system, and the developer need not know anything about interfacing a driver into the kernel.

The ULI feature may also be used for high-performance I/O applications when combined with memory-mapped device I/O. Applications can make all device accesses in user space. This is useful for high-performance I/O applications such as hardware-in-the-loop simulators.

ULIs are essentially *interrupt service routines (ISRs)* that reside in the address space of a user process. As shown in Figure 8-1, when an interrupt is received that has been registered to a ULI, it triggers the user function. For function prototypes and other details, see the `uli`(3) man page.

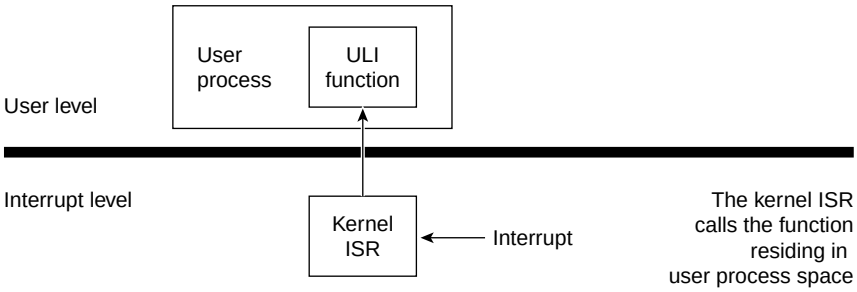

Figure 8-1 ULI Functional Overview

Note: The `uli`(3) man page and the `libuli` library are installed as part of the REACT package.

This chapter discusses the following:

- "Overview of ULI" on page 100
- "Setting Up ULI" on page 103

Overview of ULI

All registration functions return an opaque identifier for the ULI, which is passed as an argument to various other ULI functions. Table 8-1 lists the arguments that are common to all registration functions.

Table 8-1 Common Arguments for Registration Functions

Function	Description
`func`	Points to the function that will handle the interrupt.
`ULI_register_irq`	Requests that an interrupt be handled as a ULI. Once a registration function has been called, the handler function may be called asynchronously any time the associated hardware sees fit to generate an interrupt. Any state needed by the handler function must have been initialized before ULI registration. The process will continue to receive the ULI until it exits or the ULI is destroyed (see `ULI_destroy` below), at which time the system reverts to handling the interrupt in the kernel. The CPU that executes the ULI handler is the CPU that would execute the equivalent kernel-based interrupt handler if the ULI were not registered (that is, the CPU to which the device sends the interrupt).
`ULI_destroy`	Destroys a ULI. When this function returns, the identifier will no longer be valid for use with any ULI function and the handler function used with it will no longer be called.

Function	Description
`ULI_block_intr`	Blocks a ULI. If the handler is currently running on another CPU in a multiprocessing environment, `ULI_block_intr` will spin until the handler has completed.
`ULI_unblock_intr`	Unblocks a ULI. Interrupts posted while the ULI was blocked will be handled at this time. If multiple interrupts occur while blocked, the handler function will be called only once when the interrupt is unblocked.
`ULI_sleep`	Blocks the calling thread on a semaphore associated with a particular ULI. The registration function initializes the ULI with a caller-specified number of semaphores. `ULI_sleep` may return before the event being awaited has occurred, thus it should be called within a `while` loop.
`ULI_wakeup`	Wakes up the next thread sleeping on a semaphore associated with a particular ULI. If `ULI_wakeup` is called before the corresponding `ULI_sleep`, the call to `ULI_sleep` will return immediately without blocking.

For more details, see the `uli`(3) man page.

This section discusses the following:

- "Restrictions on the ULI Handler" on page 101
- "Planning for Concurrency: Declaring Global Variables" on page 103
- "Using Multiple Devices" on page 103

Restrictions on the ULI Handler

Of the ULI library functions listed above, only `ULI_wakeup` may be called by the handler function.

Each ULI handler function runs within its own POSIX thread running at a priority in the range 80 through 89. Threads that run at a higher priority should not attempt to block ULI execution with `ULI_block()` because deadlock may occur.

If a ULI handler function does any of the following, its behavior is undefined:

- Causes a page fault
- Uses the floating point unit (FPU)
- Makes a system call
- Executes an illegal instruction

Note: To avoid page faults, use the `mlock()` or `mlockall()` function prior to creating the ULI.

You can only use the `ULI_sleep` and `ULI_wakeup` functions inside of a share group. These functions cannot wake up arbitrary processes.

In essence, the ULI handler should do only the following things, as shown in Figure 8-2:

- Store data in program variables in locked pages, to record the interrupt event. (For example, a ring buffer is a data structure that is suitable for concurrent access.)
- Program the device as required to clear the interrupt or acknowledge it. The ULI handler has access to the whole program address space, including any mapped-in devices, so it can perform PIO loads and stores.
- Post a semaphore to wake up the main process. This must be done using a ULI function.

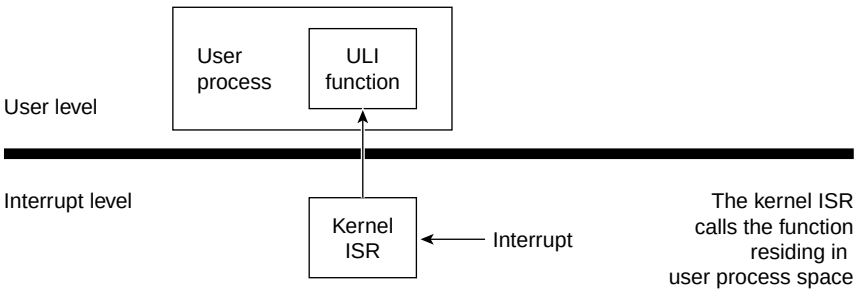

Figure 8-2 ULI Handler Functions

Planning for Concurrency: Declaring Global Variables

Because the ULI handler can interrupt the program at any point, or run concurrently with it, the program must be prepared for concurrent execution. This is done by declaring global variables. When variables can be modified by both the main process and the ULI handler, you must take special care to avoid race conditions.

An important step is to specify `-D_SGI_REENTRANT_FUNCTIONS` to the compiler in order to get the reentrant versions of the C library functions. This ensures that, if the main process and the ULI handler both enter the C library, there is no collision over global variables.

You can declare the global variables that are shared with the ULI handler with the keyword `volatile` so that the compiler generates code to load the variables from memory on each reference. However, the compiler never holds global values in registers over a function call, and you almost always have a function call such as `ULI_block_intr()` preceding a test of a shared global variable.

Using Multiple Devices

The ULI feature allows a program to open more than one interrupting device. You register a handler for each device. However, the program can only wait for a specific interrupt to occur; that is, the `ULI_sleep()` function specifies the handle of one particular ULI handler. This does not mean that the main program must sleep until that particular interrupt handler is entered, however. Any ULI handler can waken the main program, as discussed under "Interacting With the Handler" on page 106.

Setting Up ULI

A program initializes for ULI in the following major steps:

1. Load the `uli` kernel module:

   ```
   [root@linux root]# modprobe uli
   ```

2. For a PCI, map the device addresses into process memory.
3. Lock the program address space in memory.
4. Initialize any data structures used by the interrupt handler.

5. Register the interrupt handler.
6. Interact with the device and the interrupt handler.

An interrupt can occur any time after the handler has been registered, causing entry to the ULI handler.

This section discusses the following:

- "Opening the Device Special File" on page 104
- "Locking the Program Address Space" on page 104
- "Registering the Interrupt Handler" on page 105
- "Registering a Per-IRQ Handler" on page 106
- "Interacting With the Handler" on page 106
- "Achieving Mutual Exclusion" on page 107

Opening the Device Special File

Devices are represented by device special files. In order to gain access to a device, you open the device special file that represents it. If the appropriate loadable kernel modules have been loaded (that is, the `extint` and `ioc4_extint` modules), the device file `/dev/extint#` should be created automatically for you, where # is replaced by a system-assigned number, one for each of the IOC4 devices present in the system.

Locking the Program Address Space

The ULI handler must not reference a page of program text or data that is not present in memory. You prevent this by locking the pages of the program address space in memory. The simplest way to do this is to call the `mlockall()` system function:

```
if (mlockall(MCL_CURRENT|MCL_FUTURE)<0) perror ("mlockall");
```

The `mlockall()` function has the following possible difficulties:

- The calling process must have either superuser privilege or `CAP_MEMORY_MGT` capability. This may not pose a problem if the program needs superuser privilege in any case (for example, to open a device special file). For more information, see the `mlockall`(3C) man page.
- The `mlockall()` function locks all text and data pages. In a very large program, this may be so much memory that system performance is harmed.

In order to use `mlock()`, you must specify the exact address ranges to be locked. Provided that the ULI handler refers only to global data and its own code, it is relatively simple to derive address ranges that encompass the needed pages. If the ULI handler calls any library functions, the library DSO must be locked as well. The smaller and simpler the code of the ULI handler, the easier it is to use `mlock()`.

Registering the Interrupt Handler

When the program is ready to start operations, it registers its ULI handler. The ULI handler is a function that matches the following prototype:

```
void function_name(void *arg);
```

The registration function takes arguments with the following purposes:

- The address of the handler function.
- An argument value to be passed to the handler on each interrupt. This is typically a pointer to a work area that is unique to the interrupting device (supposing the program is using more than one device).
- A count of semaphores to be allocated for use with this interrupt.

The semaphores are allocated and maintained by the ULI support. They are used to coordinate between the program process and the interrupt handler, as discussed in "Interacting With the Handler" on page 106. You should specify one semaphore for each independent process that can wait for interrupts from this handler. Normally, one semaphore is sufficient.

The value returned by the registration function is a handle that is used to identify this interrupt in other functions. Once registered, the ULI handler remains registered until the program terminates or `ULI_destroy()` is called.

Registering a Per-IRQ Handler

`ULI_register_irq()` takes two additional arguments to those already described:

- The CPU where the interrupt is occurring
- The number of the interrupt line to attach to

Interacting With the Handler

The program process and the ULI handler synchronize their actions using the following functions:

- `ULI_sleep()`
- `ULI_wakeup()`

When the program cannot proceed without an interrupt, it calls `ULI_sleep()`, specifying the following:

- The handle of the interrupt for which to wait
- The number of the semaphore to use for waiting

Typically, only one process ever calls `ULI_sleep()` and it specifies waiting on semaphore 0. However, it is possible to have two or more processes that wait. For example, if the device can produce two distinct kinds of interrupts (such as normal and high-priority), you could set up an independent process for each interrupt type. One would sleep on semaphore 0, the other on semaphore 1.

When a ULI handler is entered, it wakes up a program process by calling `ULI_wakeup()`, specifying the semaphore number to be posted. The handler must know which semaphore to post, based on the values it can read from the device or from program variables.

The `ULI_sleep()` call can terminate early, for example if a signal is sent to the process. The process that calls `ULI_sleep()` must test to find the reason the call returned. It is not necessarily because of an interrupt.

The `ULI_wakeup()` function can be called from normal code as well as from a ULI handler function. It could be used within any type of asynchronous callback function to wake up the program process.

The `ULI_wakeup()` call also specifies the handle of the interrupt. When you have multiple interrupting devices, you have the following design choices:

- You can have one child process waiting on the handler for each device. In this case, each ULI handler specifies its own handle to `ULI_wakeup()`.
- You can have a single process that waits on any interrupt. In this case, the main program specifies the handle of one particular interrupt to `ULI_sleep()`, and every ULI handler specifies that same handle to `ULI_wakeup()`.

Achieving Mutual Exclusion

The program can gain exclusive use of global variables with a call to `ULI_block_intr()`. This function does not block receipt of the hardware interrupt, but does block the call to the ULI handler. Until the program process calls `ULI_unblock_intr()`, it can test and update global variables without danger of a race condition. This period of time should be as short as possible, because it extends the interrupt latency time. If more than one hardware interrupt occurs while the ULI handler is blocked, it is called for only the last-received interrupt.

There are other techniques for safe handling of shared global variables besides blocking interrupts. One important, and little-known, set of tools is the `test_and_set()` group of functions documented in the `test_and_set_bit`(9) and `test_and_clear_bit`(9) man pages and defined in the `/usr/include/asm/bitops.h` file.

REACT System Configuration

This chapter explains how to configure restricted and isolated CPUs on a system running the REACT real-time for Linux product:

- "Configuration Overview" on page 109
- "`reactconf.pl` Command-Line Summary" on page 111
- "Generating a REACT System Configuration" on page 113
- "Example `reactconf.pl` Interactive-Mode Output" on page 119
- "Running with REACT Enabled" on page 123
- "Running a Process on a Restricted CPU" on page 123
- "Dynamically Changing the Configuration" on page 124
- "Granting Users Access to Select REACT Features" on page 125
- "Linux Capabilities and the SGI ProPack Captools Package" on page 126
- "Viewing the Current REACT Configuration" on page 135
- "Disabling and Enabling REACT" on page 135

For information about creating an external interrupt character special device file, see "Opening the Device Special File" on page 104. For information about potential problems, see Chapter 11, "Troubleshooting" on page 151.

Configuration Overview

Initial configuration of the REACT real-time system requires that you reboot the system with certain kernel command line options. After the system is rebooted with the real-time configuration, REACT is automatically enabled and you can make changes to the real-time and bootcpusets dynamically, without additional reboots. You can disable and reenable the REACT real-time configuration.

Configurable items include:

- The configured real-time CPUs (the `rtcpu` devices) and their memory nodes(`mems`)

  Note: Memory is not distributed in Altix XE systems (as opposed to Altix NUMA-based systems), therefore all memory on a single system image is always on node 0.

- The bootcpuset (`/boot`) and its memory nodes
- Interrupts, which can be redirected

Note: By default, the system is booted with the most restrictive bootcpuset possible (see `/etc/bootcpuset.conf`). When the system comes up, all but a few CPU-specific threads will be running within the bootcpuset.

Timers that were created during the boot sequence are also restricted to CPUs in the bootcpuset. If these timers have their base on a potential `rtcpu`, it could cause significant delays in interrupt response times.

Usually, you cannot change a timer's base CPU. When configuring a system, be careful of creating timers when the system is not configured for real-time. (For example, inserting a module when the bootcpuset is not enabled can create a timer on a CPU that could later be used for an `rtcpu`.) However, you can move tasks and threads.

The configuration procedure uses the following:

- `/usr/sbin/reactconf.pl` is a Perl script that generates a REACT configuration. It creates the `/etc/bootcpuset.conf` and `/etc/dynamic_cpuset.conf` files for saving information about the cpusets and modifies the file `/etc/elilo.conf` (Altix) or `/boot/grub/menu.lst` (Altix XE) appropriately .

  Note: The script places backup files in `/etc/elilo.conf.rtbak` or `/etc/menu.lst.rtbak` and `/etc/bootcpuset.conf.rtbak` before making any changes to the original files.

  REACT is enabled by default when you run the `reactconf.pl` script and then reboot. When the configuration is complete and REACT is enabled, the system

will always come up with the same REACT CPU configuration without operator intervention.

- `/etc/init.d/sgi_react` is a startup script that enables or disables REACT across reboots. By default, `sgi_react` is set to `off` until you first run the `reactconf.pl` script.
- `/sbin/react-init.sh` is a `bash` script that determines if REACT is enabled:
  - If `sgi_react` is set to `on`, it will boot with `init=bootcpuset` and set the CPUs allowed for the `init` process
  - If `sgi_react` is set to `off`, it will still boot with `init=bootcpuset` but it will display the following error message to the boot screen:

```
ERROR! ERROR! ERROR! ERROR! ERROR!
Run 'reactconf.pl --enable' or
'reactconf.pl --disable'  and reboot.
Do not manipulate React through chkconfig.
```

`reactconf.pl` Command-Line Summary

The `reactconf.pl` script has the following command-line attributes (you may abbreviate options as shown below):

```
reactconf.pl cpulist
reactconf.pl [--disable | --enable | --non-interactive cpulist] [--no-output | --warnings]
reactconf.pl --show
reactconf.pl --help
```

cpulist

Specifies the real-time CPUs to be configured, where *cpulist* takes one of the following formats:

- A list of CPUs (you cannot specify CPU 0):

  cpu,cpu,...

- A range of CPUs (you cannot specify CPU 0 or a descending range):

  cpu-cpu

	• A mixture of the above: `reactconf.pl` *cpu,...cpu-cpu,cpu,...*
`--disable` (`-d`)	Disables the real-time CPUs specified in the `/etc/dynamic_cpuset.conf` file. See "Disabling REACT" on page 135.
`--enable` (`-e`)	Enables the real-time CPUs specified in the `/etc/dynamic_cpuset.conf` file. See "Enabling REACT" on page 136.
`--help` (`-h`)	Displays the usage statement.
`--non-interactive` (`-n`)	Configures the specified real-time CPUs from *cpulist* and automatically selects default options for memory and IRQ configuration. The default is to enter interactive mode, in which you are prompted for information. By default, all output is suppressed when you specify `--non-interactive` (so there is no need to specify `--no-output`). If you want to see warning and error messages, also specify `--warnings`. See "Noninteractive Mode" on page 117
`--no-output` (`-o`)	Suppress all output, including warnings and error messages. By default, informational, warning, and error messages are displayed to standard output.
`--show` (`-s`)	Shows the current configuration. See "Viewing the Current REACT Configuration" on page 135.
`--warnings` (`-w`)	Suppress informational messages but prints errors and warnings. By default, informational, warning, and error messages are displayed to standard output. If you specify this option with `-no-output`, the `--warnings` option is ignored and all output will be suppressed.

Generating a REACT System Configuration

You can use `reactconf.pl` to generate a REACT system configuration in one of the following ways:

- "Interactive Mode" on page 113
- "Noninteractive Mode" on page 117

Interactive Mode

Do the following to use `reactconf.pl` in interactive mode:

1. Log in as `root`.
2. Verify that the `RUN_PARALLEL` option in `/etc/sysconfig/boot` is set to `no` (which prevents REACT from starting too early).
3. Decide which CPUs you want to restrict for real-time use.

   Examining `/proc/interrupts` can aid in determining these CPUs. You should choose CPUs that are not already servicing any regular interrupts beyond the per-CPU interrupts, such as for the timer and the interprocessor interrupts (IPI). You cannot use CPU 0 or the CPU on the node where the console is attached for real time.

   Note: The `/usr/sbin/reactconf.pl` script attempts to prevent using incorrect CPUs and redirecting certain interrupts.

   ---

4. Use the `/usr/sbin/reactconf.pl` script to configure one or more CPUs as follows using interactive mode:

   `reactconf.pl` *cpulist*

   For details about *cpulist* and other options, see "`reactconf.pl` Command-Line Summary" on page 111.

The `reactconf.pl` script will set up the following files as appropriate, based on the CPUs listed and the number of CPUs on your system:

`/etc/bootcpuset.conf`
`/etc/elilo.conf` (Altix) or `/boot/grub/menu.lst` (Altix XE)
`/etc/dynamic_cpuset.conf`

For example, on an 8-processor system, the following command line will use interactive mode to configure CPUs 0, 1, 4, 5, and 6 in the bootcpuset, with CPUs 2, 3, and 7 restricted and isolated for real-time use:

```
[root@linux root]# reactconf.pl 2-3,7
```

The `reactconf.pl` script will append `init=/sbin/react-init.sh` to the default section in `/etc/elilo.conf` (Altix) or `/boot/grub/menu.lst` (Altix XE), which will be displayed. (See Example 9-1 on page 119 for Altix output; Altix XE output will be similar). If it is correct, answer `y` and it will be written at the end of the `reactconf.pl` script execution.

5. Configure memory nodes when prompted by the `reactconf.pl` script. (For a complete example of `reactconf.pl` output, see "Example `reactconf.pl` Interactive-Mode Output" on page 119.)

Note: Memory is not distributed in Altix XE systems (as opposed to Altix NUMA-based systems), therefore all memory on a single system image is always on node 0.

SGI recommends that you use the default setup, because using a different configuration could negatively impact performance. To choose the default setup, enter `n`:

```
Would you like to configure memory nodes for the boot
and/or real-time cpusets in a manner different then the default setup?
[y/n] n
```

By entering `n`, nodes will automatically be configured according to the `/var/run/cpunodemap`.

However, if you understand the performance implications and you would like to manually configure memory nodes, enter `y` and answer the questions that follow. For example:

```
Enter nodes for the /boot cpuset (non-rt), to allocate memory.:

List of nodes that is not in descending order.  0-1

Would you like to configure memory nodes for the rtcpuN cpusets?[y/n] y

Enter node for rtcpu1 cpuset to allocate memory on? 1

Enter node for rtcpu2 cpuset to allocate memory on? 0
   ...
```

6. Choose whether or not to redirect interrupts:

```
Do you want to move any of the IRQs? [y/n]
```

Entering `n` will not change anything on the system. Entering `y` will give you the following options:

```
1) Move IRQs round-robined into /boot cpuset.
2) Move IRQs on a case-by-case basis.
3) Do not redirect IRQs.
```

Option `1` causes IRQs that can be moved to be evenly dispersed to CPUs in the `/boot` cpuset . The `reactconf.pl` script attempts to prevent redirecting certain interrupts, including the console interrupt.

Option `2` displays a list of individual IRQs. You will be asked to specify if the IRQ is a normal or a real-time interrupt and to select the exact CPU in the respective cpusets to which the interrupt will be redirected.

For example, for a regular IRQ:

```
IRQ list: 28 29 30 31 49 51 56 62 63 65 66 67 232 233 238 239 240 254

Enter the IRQ that you would like to redirect or 'd' when done: 28

Would you like to direct IRQ 28 to a:
                  1) CPU in the /boot cpuset
                  2) CPU in a /rtcpu cpuset
                  Selection [1 | 2]1
```

```
Your cpus configured for the /boot cpuset are: 0
Select one to redirect IRQ 28: 0

Enter the IRQ that you would like to redirect or 'd' when done: d
```

For example, for a real-time IRQ:

```
IRQ list: 28 29 30 31 49 51 56 62 63 65 66 67 232 233 238 239 240 254

Enter the IRQ that you would like to redirect or 'd' when done: 28

Would you like to direct IRQ 28 to a:
                  1) CPU in the /boot cpuset
                  2) CPU in a /rtcpu cpuset
                  Selection [1 | 2]2

Your cpus configured for rtcpus are: 1 2 3
Select one to redirect IRQ 28:1

Enter the IRQ that you would like to redirect or 'd' when done: d
```

If you elect to redirect interrupts, the settings will take place when the `reactconf.pl` script is automatically run again at startup.

7. Verify that the displayed configuration information is correct and (if so) enter `y` to confirm its use. For example:

```
/etc/dynamic_cpuset.conf
bootcpus 0
bootmems 0
rtcpus 1,2,3
rtmems 0,1,1
IRQs 28,29,30,31,49,51,62,63,65,66,67,232,233,238,239,240,254
changed_mask 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

Does the above /etc/dynamic_cpuset.conf look OK? (y=yes)y
```

After you enter `y`, the `/etc/dynamic_cpuset.conf` file is written.

8. Reduce the Altix system flush duration if appropriate:

Note: This step does not apply to Altix XE systems.

System flush creates delays in interrupt response times of approximately 23 usec. Use this procedure only for systems running real-time applications that are sensitive to delays of this length. (There is a slight chance that heavily subscribed systems running with extremely heavy NUMAlink traffic could experience system hangs.) This setting is static across system boots.

a. Reboot the machine but interrupt the boot process when the `EFI Boot Manager` menu comes up. Then select the following:

```
EFI Shell [Built-in]
```

b. Enter power-on diagnostic (POD) mode by using the `pod` command:

```
Shell> pod
```

c. Enter the following (with quotes) to turn the system flush duration down to a single clock tick (the default is 8):

```
 0 000: POD SysCt (RT) Cac> setallenv SysFlushDur "1"
```

d. Exit POD mode:

```
0 000: POD SysCt (RT) Cac> exit
```

e. Reset the system to allow the new `SysFlushDur` setting to take effect and allow the system to reboot.

Noninteractive Mode

Do the following to use `reactconf.pl` in noninteractive mode:

1. Log in as `root`.
2. Verify that the `RUN_PARALLEL` option in `/etc/sysconfig/boot` is set to `no` (which prevents REACT from starting too early).
3. Decide which CPUs you want to restrict for real-time use.

   Examining `/proc/interrupts` can aid in determining these CPUs. You should choose CPUs that are not already servicing any regular interrupts beyond the per-CPU interrupts, such as for the timer and the interprocessor interrupts (IPI).

You cannot use CPU 0 or the CPU on the node where the console is attached for real time.

Note: The `/usr/sbin/reactconf.pl` script attempts to prevent using incorrect CPUs and redirecting certain interrupts.

---

4. Use the `/usr/sbin/reactconf.pl` script to configure one or more CPUs as follows using noninteractive mode:

   `reactconf.pl --non-interactive` *cpulist*

   By default, all output is suppressed when you use noninteractive mode. To display warning and error messages, add the `--warnings` option:

   `reactconf.pl --non-interactive` *cpulist* `--warnings`

   For details about *cpulist* and other options, see "`reactconf.pl` Command-Line Summary" on page 111.

   The following example will automatically set up the system with CPUs 1-3 as real-time, with the CPUs configured to local memory nodes, and with IRQs redirected in a round-robin manner to non-real-time CPUs:

   ```
   [root@linux root]# reactconf.pl --non-interactive 1-3
   ```

   The `reactconf.pl` script will set up the following files as appropriate, based on the CPUs listed and the number of CPUs on your system:

   `/etc/bootcpuset.conf`
   `/etc/elilo.conf` (Altix) or `/boot/grub/menu.lst` (Altix XE)
   `/etc/dynamic_cpuset.conf`

5. Reduce the Altix system flush duration if appropriate:

   Note: This step does not apply to Altix XE systems.

   System flush creates delays in interrupt response times of approximately 23 usec. Use this procedure only for systems running real-time applications that are sensitive to delays of this length. (There is a slight chance that heavily subscribed systems running with extremely heavy NUMAlink traffic could experience system hangs.) This setting is static across system boots.

   ---

a. Reboot the machine but interrupt the boot process when the `EFI Boot Manager` menu comes up. Then select the following:

```
EFI Shell [Built-in]
```

b. Enter power-on diagnostic (POD) mode by using the `pod` command:

```
Shell> pod
```

c. Enter the following (with quotes) to turn the system flush duration down to a single clock tick (the default is 8):

```
 0 000: POD SysCt (RT) Cac> setallenv SysFlushDur "1"
```

d. Exit POD mode:

```
0 000: POD SysCt (RT) Cac> exit
```

e. Reset the system to allow the new `SysFlushDur` setting to take effect and allow the system to reboot.

Example `reactconf.pl` Interactive-Mode Output

Following is an example `reactconf.pl` output in interactive mode. The script modifies the bootcpusets based on the CPUs provided and changes the append value for the default entry in `/etc/elilo.conf` on Altix systems and `/boot/grub/menu.lst` on Altix XE systems. (The script output also contains information currently contained in the `/etc/elilo.conf` file, such as comments.)

Note: POD mode does not apply to Altix XE systems.

Example 9-1 `reactconf.pl` Output

```
[root@linux root]# /usr/sbin/reactconf.pl 1-3
cpuset: attached one pid to cpuset </>
/boot cpuset already removed.
SAL Console driver CPU 0
System not booted with bootcpuset configured.

Continuing with realtime configuration...

*********** Warning manual configuration should be done by someone ***********
```

```
********** with a knowledge of the system configuration.          **********

The preferred method is to have cpus allocate memory on a local node.
By default the script will configure the boot and realtime cpusets in this manner.

Would you like to configure memory nodes for the boot and/or realtime cpusets
in a manner different then the default setup? [y/n] n


/etc/elilo.conf:
# Modified by YaST2. Last modification on Mon Jan 22 09:13:08 CST 2007
timeout = 80
read-only
relocatable
default = Linux
prompt


image = /boot/vmlinuz
###Don't change this comment - YaST2 identifier: Original name: linux###

    label = Linux
    append = "  console=ttySG0 splash=silent thash_entries=2097152 init=/sbin/react-init.sh isolcpus=1,2,3"
    initrd = /boot/initrd
    root = /dev/sda11


image = /boot/vmlinuz
###Don't change this comment - YaST2 identifier: Original name: failsafe###

    label = Failsafe
    append = "ide=nodma nohalt noresume 3"
    initrd = /boot/initrd
    root = /dev/sda11


image = /boot/vmlinuz.save
###Don't change this comment - YaST2 identifier: Original name: linux###

    label = Linux-sgi-engr
    append = "  console=ttySG0 splash=silent thash_entries=2097152"
```

```
    initrd = /boot/initrd.save
    root = /dev/sda11


image = /boot/vmlinuz.save
###Don't change this comment - YaST2 identifier: Original name: failsafe###

    label = Failsafe-sgi-engr
    append = "ide=nodma nohalt noresume 3"
    initrd = /boot/initrd.save
    root = /dev/sda11


cpuset: attached one pid to cpuset
/boot cpuset already removed.
SAL Console driver CPU 0
System not booted with bootcpuset configured.

Does the above elilo.conf look OK? (y=yes overwrites /etc/elilo.conf)y

bootcpuset cpus 0
bootcpuset mems 0
realtime cpus   1 2 3
realtime mems   0 1 1


To minimize latency of real-time interrupts, it is often necessary to direct interrupts
to specific real-time processors. It is also necessary to direct other interrupts
away from specific real-time processors. Redirected irq's seem to have higher latency,
so only redirect irqs if you must move them away from CPUs that must be real-time.
It is preferable to select CPUs for real-time in such a way as to not require interrupt
redirection.
Do you want to move any of the IRQs? [y/n] y
           CPU0       CPU1       CPU2       CPU3
 28:          0          0          0          0       LSAPIC  cpe_poll
 29:          0          0          0          0       LSAPIC  cmc_poll
 30:          0          0          0          0       SN hub  cpe_hndlr
 31:          0          0          0          0       LSAPIC  cmc_hndlr
 49:          0          0          0          0       SN hub  SN_hub_error, SN_hub_error
 51:          0          0          0          0       SN hub  PCIBR error, PCIBR error, PCIBR error, PCIBR error
 56:          0          0          0          0       SN hub  mmtimer
```

```
 61:      10691          0          0          0       SN hub  ide0
 62:          0       8599          0          0       SN hub  libata
 63:      12947          0          0          0       SN hub  eth0
232:          0          0          0          0       LSAPIC  mca_rdzv
233:       2374          0          0          0       SN hub  system controller events, SAL console driver
238:          0          0          0          0       LSAPIC  perfmon
239:     301620     301334     301373     301367       LSAPIC  timer
240:          0          0          0          0       LSAPIC  mca_wkup
254:        421       1351       1681       1647       LSAPIC  IPI
ERR:          0
Here is your current irq configuration.


Options:
     1) Move IRQs round-robined into /boot cpuset.
     2) Move IRQs on a case-by-case basis.
     3) Do not redirect IRQs.
Selection [1 | 2 | 3] 1
Not moving the mmtimer IRQ 56


/etc/dynamic_cpuset.conf
bootcpus 0
bootmems 0
rtcpus 1,2,3
rtmems 0,1,1
IRQs 28,29,30,31,49,51,61,62,63,232,233,238,239,240,254
changed_mask 1,1,1,1,1,1,1,1,1,1,1,1,1,1

Does the above /etc/dynamic_cpuset.conf look OK? (y=yes)y


Backup files are /etc/elilo.conf.rtbak and /etc/bootcpuset.conf.rtbak
Remember to change the SysFlushDur during reboot if not done already,
from pod mode, run 'setallenv SysFlushDur "1"',
see the REACT Realtime for Linux documentation for details.


Please reboot your system to restrict and isolate cpus.
```

Running with REACT Enabled

When the system comes back up, `reactconf.pl --enable` is automatically run from `/etc/init.d/sgi_react`. The `--enable` option does the following:

- Creates a container cpuset named `rtcpus` and cpusets labeled `rtcpu`*N* for each CPU that is not part of the bootcpuset (`/rtcpus/rtcpu`*N*). You can use these cpusets to run your real-time threads. You will find these cpusets in `/dev/cpuset`, along with the bootcpuset set up by `reactconf.pl` *cpulist* in the previous steps and stored in `/etc/dynamic_cpuset.conf`.
- Configures the cpuset's memory nodes by setting the values in `/dev/cpuset/rtcpus/rtcpu`*N*`/mems` and `/dev/cpuset/boot/mems`.

  Note: Memory is not distributed in Altix XE systems (as opposed to Altix NUMA-based systems), therefore all memory on a single system image is always on node 0.

  In step 5 above in "Interactive Mode" on page 113, you either specified that the memory nodes would be set up in a default manner that is optimal for performance or selected specific nodes and cpusets yourself.
- Redirects interrupts from user specification. The proper mask values are echoed in to `/proc/irq/interrupt/smp_affinity`. For more information, see "Redirect Interrupts" on page 48.

Running a Process on a Restricted CPU

To run a process on a restricted CPU, you must invoke or attach it to a real-time cpuset (that is, a cpuset containing a CPU that does not exist in the bootcpuset, such as the `/dev/cpuset/rtcpus/rtcpu`*N* cpusets created above). For example:

```
[root@linux root]# cpuset --invoke /rtcpus/rtcpu4 -I ./foo
```

or:

```
[root@linux root]# echo $$ | cpuset -a /rtcpus
[root@linux root]# dplace -c 1 ./foo
```

Note: The `dplace` command example will attach the process to the second real-time CPU, not the second CPU on the system.

To attach an existing process to a restricted CPU, you can use `cpuset --attach`. For example, to attach your current process to CPU 2:

```
[root@linux root]# echo $$ | cpuset --attach /rtcpus/rtcpu2
```

You can also assign a process by using the following `libsgirt` function:

```
sysmp(MP_MUSTRUN_PID, cpu, pid)
```

For more information, see the `cpuset`(1), `dplace`(1), `libsgirt`(3), and `libcpuset`(3) man pages.

Dynamically Changing the Configuration

After the system has been initially booted with a real-time configuration, that configuration is the configuration that will be in place across reboots and saved into `/etc/dynamic_cpuset.conf`. You can thereafter make changes dynamically without rebooting the system. Use the following command to form and implement a new configuration (that is, rewrite the contents of `/etc/dynamic_cpuset.conf`; no reboot is required):

`reactconf.pl` *cpulist*

where *cpulist* is one of the varieties explained in "Generating a REACT System Configuration" on page 113.

Note: If CPUs that previously were not reserved for real-time are later added to the list of CPUs reserved for real-time, holdoff situations could be encountered until the next reboot due to timers having been previously set on those CPUs.

Granting Users Access to Select REACT Features

The cpusets, devices, and control files associated with REACT are normally accessible only by the `root` user.

You can use the `react-permissions.pl` script to specify a group of users that have access to the following REACT features:

- Cpusets created by the `react.conf.pl` script
- User-level interrupts (ULI)
- The frame scheduler
- The `kbar`(3) kernel barrier facility
- External interrupts

The script generates the `/etc/udev/rules.d/99-sgi-react.rules` file and and a new `/etc/sysconfig/sgi-react.conf` configuration file, which initially holds the group ID and permissions. It changes the group ownership and file mode permissions for REACT `/dev`, `/sys/class/extint`, and `/dev/cpuset` files, both immediately and across reboots.

After running `react-permissions.pl`, the specified users can run REACT applications without having the ability to overwrite any file on the system. (That is, the specified users do not have `CAP_DAC_OVERRIDE` authority.)

Note: The script does not give the specified users access to native system calls that require specific capabilities, such as `sched_setscheduler()`. To directly use those system calls, a user must have the required process capabilities set.

For example, the following will change the group ownership to `rtgroup` and the file mode to the default of `664` on the above mentioned files, giving users that are members of `rtgroup` the ability to use the select REACT feature set:

```
[root@linux root]# react-permissions.pl rtgroup
```

Linux Capabilities and the SGI ProPack Captools Package

This section discusses the following:

- "Capabilities Provided with Standard Linux" on page 126
- "Enhanced Capabilities Provided with the SGI ProPack Captools Package" on page 127
- "Captools Policy Specification and Usage" on page 132
- "Captools Usage Example for REACT" on page 134

Capabilities Provided with Standard Linux

The Linux capabilities security mechanism (often referred to as *POSIX capabilities*) provides relatively fine-grained control over the privileges of a process. As a process attribute, a capability allows the process to perform a specific set of restricted operations without granting general override of the system's protection scheme.

A process can possess multiple capabilities. Collectively, all defined capabilities comprise the set of abilities that are traditionally associated with the `root` user. As of the 2.6.14 Linux kernel, there are 31 capabilities implemented, each defined by a bit within the capabilities bitmap. For specifics on implemented capabilities and the individual capability definitions, refer to the `capabilities`(7) manpage. Each process has three capability sets containing up to 31 capabilities:

- *Effective set*: the capabilities used by the kernel to perform permission checks for the process.
- *Inheritable set*: the capabilities preserved across an `execve`(2).
- *Permitted set*: the capabilities that the process may assume (that is, a limiting superset for the effective and inheritable sets). If a process drops a capability from its permitted set, it can never reacquire that capability unless it uses `exec`(2) to execute the program that sets the user ID to `root`.

A child created via `fork`(2) inherits copies of its parent's capability sets.

The standard Linux implementation of capabilities is limited to processes and does not have the ability to attach capabilities to an executable file, such that a process could inherit capabilities when the file is executed with `exec()`. Consequently, the following are true during an `exec()`:

- All three file capability sets are initially assumed to be cleared.
- If a program that sets the user ID to `root` is being executed or if the real user ID of the process is `0` (meaning `root`), then the file inheritable and permitted set capabilities are enabled (that is, each capability is defined to be `1`).
- If a program that sets the user ID to `root` is being executed, then the file effective set is enabled (that is, each capability is defined to be `1`).

Traditionally, access and manipulation of process capabilities is facilitated via the `capget`(2) and `capset`(2) system calls, respectively.

Enhanced Capabilities Provided with the SGI ProPack Captools Package

In order to enhance the standard Linux capabilities mechanism, SGI has included the captools package with SGI ProPack. This toolset is particularly useful when used in conjunction with REACT. The captools package facilitates capability access and manipulation, going beyond that of standard Linux.

The `capabilities`(7) manpage provides a detailed list and descriptions of individual capabilities. Specific uses of capabilities in the context of real-time applications may involve the following individual capabilities:

Capability	Description
`CAP_IPC_LOCK`	Permits memory locking. See the `mlock`(2), `mlockall`(2), `mmap`(2), and `shmctl`(2) man pages.
`CAP_SYS_NICE`	Allows the following: • Raising the process nice value. See the `nice`(2) and `setpriority`(2) man pages. • Setting of real-time scheduling policies for calling process and setting scheduling policies and priorities for arbitrary processes. See the `sched_setscheduler`(2) and `sched_setparam`(2) man pages. • Using FIFO and round-robin (real-time) scheduling on processes owned by this process and setting the scheduling algorithm used by another process.

CAP_SYS_PTRACE	Allows arbitrary processes to be traced using ptrace(2).
CAP_SYS_RESOURCE	Allows for greater than 64-Hz interrupts from the real-time clock.
CAP_SYS_TIME	Allows modification of the real-time (hardware) clock and the modification of the system clock. See the settimeofday(2), stime(2), and adjtimex(2) man pages.

The captools package consists primarily of a kernel module and a perl script that together facilitate assignment of capabilities to particular users, groups, and/or processes. The kernel module implements the kernel space portion as part of the Linux Security Module (LSM) framework, while the policy script uses the user space functionality to gain access to the cap_over capabilities override kernel module.

The cap_over(3) (capabilities override) kernel module is an LSM that allows certain programs to gain one or more POSIX.1e capabilities(7). The capabilities granted can be based on the process' user ID (UID), group ID (GID), and/or executable pathname. The cap_over module enables the application of relatively complex capabilities policies. For example, it is possible to give a particular program a single capability and further restrict it by allowing execution by a single specific user. The cap_over module is the kernel-space part of the captools package that enables manipulation of individual capabilities and lets the system administrator adjust capabilities on a per-user, per-group, and/or per-path basis. The user-space part consist of a perl script in /usr/sbin/policy.pl that is used to access the cap_over structure inside the kernel.

The capabilities available to cap_over are seen as elements in the pseudo-filesystem subdirectory /proc/sys/kernel/cap_over/:

```
cap_audit_control
cap_audit_write
cap_chown
cap_dac_override
cap_dac_read_search
cap_fowner
cap_fsetid
cap_ipc_lock
cap_ipc_owner
cap_kill
cap_linux_immutable
cap_mknod cap_lease
```

```
cap_net_admin
cap_net_bind_service
cap_net_broadcast
cap_net_raw
cap_setgid
cap_setuid
cap_setpcap
cap_sys_admin
cap_sys_boot
cap_sys_chroot
cap_sys_module
cap_sys_nice
cap_sys_pacct
cap_sys_ptrace
cap_sys_rawio
cap_sys_resource
cap_sys_time
cap_sys_tty_config
```

The `cap_over` module populates a structure within the constraints of the LSM framework and is a part of the system kernel parameters structure under `proc/sys`. Because the structure is a part of the kernel parameters structure, you can access its contents via the `sysctl`(8) command. For example, to examine the values of the individual capabilities (all capabilities are null at the outset):

```
[root@linux root]# sysctl -a |grep cap_over
        kernel.cap_over.lease =
        kernel.cap_over.mknod =
        kernel.cap_over.sys_tty_config =
        kernel.cap_over.sys_time =
        kernel.cap_over.sys_resource =
        kernel.cap_over.sys_nice =
        kernel.cap_over.sys_boot =
        kernel.cap_over.sys_admin =
        kernel.cap_over.sys_pacct =
        kernel.cap_over.sys_ptrace =
        kernel.cap_over.sys_chroot =
        kernel.cap_over.sys_rawio =
        kernel.cap_over.sys_module =
        kernel.cap_over.ipc_owner =
        kernel.cap_over.ipc_lock =
        kernel.cap_over.net_raw =
```

```
kernel.cap_over.net_admin =
kernel.cap_over.net_broadcast =
kernel.cap_over.net_bind =
kernel.cap_over.fs_immutable =
kernel.cap_over.setuid =
kernel.cap_over.setgid =
kernel.cap_over.kill =
kernel.cap_over.fsetid =
kernel.cap_over.fowner =
kernel.cap_over.dac_read_search =
kernel.cap_over.dac_override =
kernel.cap_over.chown =
```

Although it would be possible to assign individual capabilities using the `sysctl` command, SGI recommends that you use the user space policy compiler `/usr/sbin/policy.pl` as a part of the captools package in order to parse a predefined policy configuration file. For an example policy configuration file, see `/usr/share/captools/examples/example.conf`.

The policy configuration file is a textual representation of the capabilities policy. The basic format of the file is:

capname1`[,`*capname2*`,...] {`
`[user` *username1*`[,`*username2*`,[...]]]`
`[group` *groupname1*`[,`*groupname2*`,[...]]]`
`[audit (on|1|off|0)]`
`[path` *pathname1*`]`
`[path` *pathname2*`]`
`[...] }`

where:

capnameN Lists one or more capability names to which the rule applies. Each name is a lowercase version of a particular capability; see the `capabilities`(7) manpage.

Note: The prefix `CAP_` is removed. In a few cases, the name is further abbreviated; for example, `CAP_NET_BIND_SERVICE` is just `net_bind` and `CAP_LINUX_IMMUTABLE` is `fs_immutable`.

`user` Lists one or more user names. A process running with a UID or effective UID (EUID) that is the same as one of the users on the list is

given access to the capability represented by `cap_name` (assuming all other conditions hold).

`group` Lists one or more group names. A process running with a GID or effective GID (EGID) that is the same as one of the groups on the list is given access to the capability represented by `cap_name` (assuming all other conditions hold).

`audit` Specifies whether or not processes getting extra capabilities under this rule should be audited and logged in `/var/log`. The default (if you do not use the `audit` command in a particular rule), is to use the value of `default_audit`, which takes the same values as `audit`. You can set this at any point in the file (do not put it inside a rule). By default, `default_audit` is set to `on`, meaning that everything is audited unless it is explicitly marked as not audited. You should always leave `default_audit` set to `on`, and turn off auditing just for those rules that you are absolutely sure do not require audit information.

`path` Specifies that if the process is not included by one of the named paths, then it does not get the capability. If you do not specify `path`, the default is `path any`, which means that any process matching the UID/GID requirements gets the capability.

You must specify at least one of `user`, `group`, and `path`; if you do not specify any of these, the rule will be ignored.

Note: The capability is granted if and only if **all of the conditions hold**. For example, the following rule says to give `/some/binary` the capability `cap_name` if it is run by `someuser` AND the GID/EGID is `somegrp`:

```
cap_name {
          user someuser
          group somegrp
          path /some/binary }
```

If you want to do an OR between users and groups, you must write two separate rules, one for each. For example, to allow `cap_name` if it is run by `someuser` or by anyone in `somegrp`, you would need:

```
cap_name {
          user someuser
          path /some/binary }

cap_name {
          group somegrp
          path /some/binary }
```

SGI suggests the following when writing the policy configuration file:

- Use `/usr/share/captools/examples/example.conf` as a starting point.
- Try to limit the number of alternate users, groups, and paths.

The method that the `cap_over` module employs to store rules is such that the number of rule combinations quickly expands. For example, a rule that specifies 2 possible users, 2 possible groups, and 2 possible paths, causes 8 rules to be stored by the module. However, there is no fixed upper limit, and the searches are only done when a program is loaded, so the overhead will likely be minimal.

Captools Policy Specification and Usage

Once the capabilities policy is committed to the policy configuration file, it can then be parsed by the `/usr/sbin/policy.pl` policy compiler perl script.

The output of the `policy.pl` script is either errors, or formatted text, suitable for passing to the `cap_over`(3) structure, conforming to the LSM framework. In many cases, the policy compiler also prints various warnings to standard output.

Caution: The warnings produced by `policy.pl` are security caveats.

You must ensure that you understand the implications of these warnings and consider when implementing a capability set. In most cases, you must be extremely cautious about applying rules that do not provide specific path names. SGI highly recommends that you enable audits for such rules.

The capabilities that can be used to immediately (or at least very easily) obtain `root` privileges are (at a minimum)

```
chown
dac_override
fowner
fsetid
mknod
setgid
setuid
sys_admin
sys_module
sys_ptrace
sys_rawio
```

The most useful case for giving access to users is when they already have `root` privilege, for the purpose of privilege separation (for the same reason that the system administrator would not use `root` as his or her user account). For example, the `network` administrator account might have a rule such as the following:

```
net_bind,net_broadcast,net_admin,net_raw {
         user network # special network admin account
         audit on
         path any }
```

In this example, where `network` is granted extra capabilities, the grant is not done for security reasons (the odds are fairly high that a sufficiently motivated person could gain `euid == 0` with the above rule), but for privilege separation, which is desirable.

To make assignments within the structure, you can use the `sysctl` command to load settings from a file that contains the outputs of the `policy.pl` policy compiler. For example:

```
[root@linux root]# /usr/sbin/policy.pl > policy.out
[root@linux root]# sysctl -p policy.out
```

Note: Writing something new to the `sysctl` or `/proc` file erases anything currently in standard input.

Captools Usage Example for REACT

The cpusets, devices and control files associated with REACT are normally accessible only by the `root` user. One mechanism to grant users some of the REACT features is via the `react-permissions.pl` script. However, the script is limited in that it does not give the specified users access to native system calls that require specific capabilities, such as `sched_setscheduler()`. To directly use those system calls, a user must have the required process capabilities set. You can provide this by using captools for REACT.

For example, suppose you want to grant the following capabilities to the `react-users` group, along with an additional user `janedoe`:

```
ipc_lock
sys_nice
sys_ptrace
```

You could do the following:

1. As `root`, write a new policy configuration file named `react-caps.conf` (using `/usr/share/captools/examples/example.conf` as a reference) that contains the following:

   ```
   # let anyone from react-users group along with user janedoe
   # lock memory, and execute nice and ptrace calls
   ipc_lock,sys_nice,sys_ptrace {
      group react-users
      user janedoe
      audit off
      # implicit 'path any'
   }
   ```

2. Run the `react-caps.conf` file through the policy compiler:

```
[root@linux root]# /usr/sbin/policy.pl ./react-caps.conf > policy.out
```

3. Apply the policy to the `cap_over` security module in `/proc/sys/kernel`:

```
[root@linux root]# sysctl -p policy.out
```

After the configuration is written and applied as a policy, the users in group `react-users` along with the user `janedoe` will have capabilities to execute `ipc_lock`, `sys_nice`, and `sys_ptrace`. Moreover, these capabilities will apply through `exec` and `fork` executions and therefore conform to multithreaded environments.

Note: You must reapply the policy if the system is rebooted.

Viewing the Current REACT Configuration

To view your current REACT configuration:

```
[root@linux root]# reactconf.pl --show
```

Disabling and Enabling REACT

You can disable and enable REACT by using the `reactconf.pl` script:

- "Disabling REACT" on page 135
- "Enabling REACT" on page 136

Disabling REACT

To return the system to normal, do the following:

1. Stop real-time processes.
2. Enter the following:

```
[root@linux root]# reactconf.pl --disable
```

To suppress all output, use the `--no-output` option. For example:

```
[root@linux root]# reactconf.pl --disable --no-output
```

To display only warning and error messages, use the `--warnings` option:

```
[root@linux root]# reactconf.pl --disable --warnings
```

For more information about these options, see "`reactconf.pl` Command-Line Summary" on page 111.

The `--disable` option does the following:

- Removes the created `/boot` and `rtcpu`*N* cpusets
- Moves any changed IRQs back to their original CPUs

Note: If timers are created while the system is not configured for real-time, they could run on any CPU, potentially creating significant holdoffs when the system is reconfigured for real-time (unless a reboot is done first).

Enabling REACT

If you disable REACT using the `reactconf.pl --disable` option ("Disabling REACT" on page 135), you can reenable it by using the following command:

```
[root@linux root]# reactconf.pl --enable
```

To suppress all output, use the `--no-output` option. For example:

```
[root@linux root]# reactconf.pl --enable --no-output
```

To display only warning and error messages, use the `--warnings` option:

```
[root@linux root]# reactconf.pl --enable --warnings
```

For more information about these options, see "`reactconf.pl` Command-Line Summary" on page 111.

No reboot is necessary. The system will use the configuration saved in `/etc/bootcpuset.conf` and `/etc/dynamic_cpuset.conf`.

On Altix XE systems, if the previous IRQ configuration was manual, the configuration will default to `round-robin` after `--enable` is run.

SGI Linux Trace

This chapter discusses the following:

- "Overview of SGI Linux Trace" on page 137
- "Installing SGI Linux Trace" on page 138
- "Gathering Trace Data" on page 139
- "Monitoring Trace Events" on page 146
- "Exiting from the `tracevisualizer` GUI" on page 148
- "Removing SGI Linux Trace" on page 148

Overview of SGI Linux Trace

The SGI Linux Trace feature generates traces for kernel events such as interrupt handling, scheduling, and system calls. You can use the SGI Linux Trace tools to record and view trace events and analyze how kernel behavior impacts the execution of applications.

SGI Linux Trace consists of the following:

- A debug kernel with traces inserted
- The `tracevisualizer`(1) graphical user interface (GUI)
- The `tracedaemon`(1) command, which is available from within the GUI or directly from the command line
- Sample platform-specific data files gathered with the frame scheduler enabled and running the `simple_pt` example program:
  - Altix:

    ```
    /var/SLT-DataFiles/ia64/Default-example.proc
    /var/SLT-DataFiles/ia64/slt-cpu.example-all
    ```

– Altix XE:

```
/var/SLT-DataFiles/x86_64/Default-example.proc
/var/SLT-DataFiles/x86_64/slt-cpu.example-all
```

You can view these files using `tracevisualizer`.

For additional details, see the man pages and the `tracevisualizer` GUI help text in the following file:

```
/usr/share/doc/TraceToolkit-0.9.5-1/Help.tracevisualizer
```

Note: SGI Linux Trace is based on the open-source Linux Trace Toolkit and has been enhanced for use on SGI ProPack systems.

Installing SGI Linux Trace

To install the `sltdebug` kernel and SGI Linux Trace, do the following:

1. Log in as the superuser.
2. Install the `kernel-sltdebug` RPM:

   ```
   [root@linux root]# rpm -Uvh kernel-sltdebug-*.rpm
   ```

3. Install the `TraceToolkit` RPM:

   ```
   [root@linux root]# rpm -Uvh TraceToolkit-*.rpm
   ```

4. Do one of the following:

   a. If you want to use the `slt` service, set it to start after a reboot and then reboot:

      ```
      [root@linux root]# chkconfig slt on
      [root@linux root]# reboot
      ```

   b. If you do not want to use the `slt` service, you must enter the following the commands manually while running the `slt` kernel. (Without these steps, the `tracedaemon` will not function.)

      i. Create the `/mnt/debug` directory if it does not already exist:

         ```
         [root@linux root]# mkdir /mnt/debug
         ```

ii. Mount the `debugfs` filesystem:

```
[root@linux root]# mount -t debugfs debugfs /mnt/debug
```

iii. Load the `slt` module:

```
[root@linux root]# modprobe slt
```

Note: This procedure installs the `sltdebug` kernel as the default kernel. When you are done with the `sltdebug` kernel, you should remove the `kernel-sltdebug` RPM or change the symbolic links in `/boot` back to the default kernel. If you reboot into a non-`sltdebug` kernel without removing both the `kernel-sltdebug` and `TraceToolkit` RPMs, you must remove the `slt` module. See "Removing SGI Linux Trace" on page 148.

Gathering Trace Data

The `tracedaemon`(1) command reads buffers of trace data provided by the kernel and writes that data to a file. You can run `tracedaemon` from within the `tracevisualizer` GUI or from the command line.

This section discusses the following:

- "Invoking the `tracevisualizer` GUI" on page 139
- "Recording Events" on page 140
- "Trace Files and Event Types" on page 141
- "Exiting from the `tracedaemon` Command Line" on page 146

Invoking the `tracevisualizer` GUI

To gather data, you must run the `tracevisualizer` GUI or the `tracedaemon` command as `root`. To allow non-`root` users to use the command, you can do one of the following:

- Configure `sudo` to allow execution of `tracedaemon` by specific users.
- Configure the command to set `setuid root`.

To invoke the `tracevisualizer` GUI, enter the following:

```
[user@linux user]# tracevisualizer
```

To write the event data in ASCII format to the specified output file, enter the following:

`tracevisualizer` *trace_input_file proc_file output_file*

For example:

```
[user@linux user]# tracevisualizer slt-cpu.1 Default.proc asciitraceoutput
```

For information about options that filter out the information written to *output_file*, see the `tracevisualizer`(1) man page.

Recording Events

When you want to start recording events, click the stoplight toolbar icon. You can then select options to control the following:

- The time duration for which the trace is to be recorded. You can click **Start** to start recording and **Stop** to stop recording, or you can enter a specific duration in seconds. The default is 120 seconds.

Caution: You do not want to collect events indefinitely because you would end up with huge trace files that would consume all disk space.

- The CPUs in which threads will be run. Select **Bootcpuset: On** to run threads in every CPU in the bootcpuset (or CPU 0 if no bootcpuset is present) or select **Off** to specify specific CPUs in **CPUs To Run Threads** in one of the following formats:
  - A list of CPUs:

    cpu,cpu,...

  - A range of CPUs (you cannot specify a descending range):

    cpu-cpu

  - A mixture of the above:

    cpu,...cpu-cpu,cpu,...

- The sub-buffer size and number. (A *sub-buffer* is a portion of a CPU buffer. The size of the CPU buffer equals the number of sub-buffers multiplied by the sub-buffer size.) If you experience data being overwritten or dropped, you may need to increase the default values.

You can also use the `tracedaemon` command line to specify the information for recording events:

```
tracedaemon [-h]
tracedaemon [-b] [-n n_subbufs] [-s subbuf_size] [-t seconds]
tracedaemon [-n n_subbufs] [-r cpulist] [-s subbuf_size] [-t seconds]
```

`-b`	Runs the `tracedaemon` command and threads on all CPUs listed in the bootcpuset. The default is CPU 0.
`-h`	Displays the usage statement.
`-n` *n_subbufs*	Specifies the number of sub-buffers. The default is 4.
`-r` *cpulist*	Specifies on which CPUs the `tracedaemon` process and threads can run, where *cpulist* takes one of the following formats described above. This is useful for keeping traffic off of certain CPUs. By default, per-cpu threads run locally on the CPU in which they are collecting data and the `tracedaemon` process runs wherever the scheduler puts it.
`-s` *subbuf_size*	Specifies the sub-buffer size in bytes. The default is 262144.
`-t` *seconds*	Specifies the total run time in seconds. The default is 120 seconds.

For example, to record trace events for 200 seconds and run `tracedaemon` and threads in the bootcpuset (or CPU 0 if there is no bootcpuset), enter the following:

```
[root@linux root]# tracedaemon -t 200 -b
```

Trace Files and Event Types

The event information for each CPU is recorded in a separate file that can be read by `tracevisualizer` and displayed graphically. The files are located in the same directory from which the `tracevisualizer` GUI or the `tracedaemon` command is run.

The trace files are named as follows:

- `slt-cpu.`*N*, with *N* corresponding to the CPU number.
- `slt-cpu.all`, which combines information from all of the individual `slt-cpu.`*N* files. This file is only created when you run `tracedaemon` from inside the GUI.

  Note: If you invoked `tracedaemon` from the command line, the `slt-cpu.all` file is not created.

- `Default.proc` process/IRQ information file.

For example, suppose you have 4 CPUs. If you use the default options in the GUI, the following files would be output:

```
slt-cpu.0
slt-cpu.1
slt-cpu.2
slt-cpu.3
slt-cpu.all
Default.proc
```

Table 10-1 summarizes the types of events that are recorded.

Table 10-1 Trace Events that are Recorded

Event Type	Raw Trace Output Name	Event Graph Representation	Description
Advanced programmable interrupt controller (APIC)	`APIC Intr Timer`	`APIC Timer`	The thread has entered a local APIC timer interrupt.
	`APIC Call Func`	`APIC CF`	The thread has entered the APIC `Interrupt for SMP Call` function. The passed function will be run on the CPUs.
	`APIC Intr Exit`	(no representation)	The function execution is finished and the thread is exiting the APIC interrupt processing.
Frame scheduler	`FRS Yield`	`FRSYL`	A frame scheduler application worker thread has called `frs_yield()` to indicate the end of its processing for the current minor frame.
	`FRS Intr Entry`	`FRSINTENT`	The frame scheduler interrupt/event processing has started.
	`FRS Intr Exit`	`FRSINTEX`	The frame scheduler interrupt/event processing has finished.
Interrupt	`Badbreak`	`BBRK`	The thread is entering privileged mode to handle a bad system call.
	`Break`	`BRK`	The thread is entering privileged mode to handle a system call.

Event Type	Raw Trace Output Name	Event Graph Representation	Description
	`Fault`	`FAULT`	The thread is entering privileged mode to handle a system fault.
	`Kernel exit`	(a change will appear in the graph)	The interrupt processing is complete and thread is returning to the previous user-mode processing.
	`Interrupt return`	`INTRT`	The interrupt processing is complete and the thread is returning to the previous kernel-mode processing.
	`IRQ entry`	`IRQ`	The running thread has been preempted to handle the top-half of an interrupt (IRQ) event.
	`Lightweight`	`LTW`	The thread is entering privileged mode via a lightweight mechanism such as a fastpath system call.
	`Opfault`	`OPF`	The thread is entering privileged mode to handle an illegal operation.
	`Pagefault`	`PGF`	The thread is entering privileged mode to handle a fault in the requested page.
	`Soft IRQ`	`SIRQ:`*IRQ number*	Soft-IRQ execution for previous IRQ event.
	`Tasklet action`	`TA:`*function address*	Tasklet execution for previous IRQ event.
	`Tasklet hiaction`	`THA:`*function address*	High-priority tasklet execution for previous IRQ event.

Event Type	Raw Trace Output Name	Event Graph Representation	Description
	`Unaligned`	`UNA`	The thread is entering privileged mode to handle an unaligned memory access.
Scheduler	`Sched activate`	`Ac:`*PID*	The thread has been moved onto the CPU run queue. The thread is in the ready-to-run state.
	`Sched deactivate`	`De:`*PID*	The thread has been moved off the CPU run queue. The thread is in the wait/sleeping state.
	`Sched switch`	`Sw:`*PID*	The CPU has been allocated to a new thread. The new thread's register state, stack, and memory mappings are switched onto the CPU. The new thread is in the running state. The previous thread will likely have been deactivated prior to the switch.
System call	`Syscall entry`	*system call name*	The thread is entering a system call. System calls can be invoked from user-mode and kernel-mode on Linux.
	`Syscall exit`	(no representation)	The thread has exited a system call handler.

For more information, see the `tracedaemon`(1) man page.

Exiting from the `tracedaemon` Command Line

If you run `tracedaemon` from the command line, do one of the following to exit:

- Press `Ctrl-c`
- Enter the following, using the process ID (PID) for the `tracedaemon` process:

  `[root@linux root]# kill -9` *tracedaemon_PID*

Monitoring Trace Events

This section discusses the following:

- "Opening a Trace File" on page 146
- "Zooming In On An Event" on page 147
- "Changing the Time Frame and Display" on page 147
- "Seeing Process Details" on page 148
- "Seeing All Event Trace Details" on page 148
- "Filtering Events Based on CPU" on page 148

For more details, see the GUI help text.

Opening a Trace File

To monitor events, you must open a trace file and the `Default.proc` process/IRQ information file. You must have permission to read the files.

Use the following menu selection to invoke the **Open Trace** window:

File
> Open Trace

Note: You could also click on the left-most icon in the icon bar to open a new trace. For more information about the shortcuts in the icon bar, see the GUI help.

Enter the path to a trace file and process/IRQ information file, or click the **Browse** button to open the **Select Trace File** or **> Select Proc File**, which lets you select a filename.

By default, the trace is shown in the **Event Graph** output, zoomed out to a great distance.

Zooming In On An Event

The graph displays the current start time, the end time, the resulting span of time, and the format of the time ruler in either microseconds (us) or nanoseconds (ns).

In most cases, the graph will be most useful if you zoom in to a smaller time span. You may also wish to resize the window. To zoom in, select the following from the menu bar or use the + spyglass icon:

Tools
> Zoom In

Changing the Time Frame and Display

You can use several methods to change the time frame and display:

- Use the scroll bar at the bottom of the graph for slight changes
- Left-click the mouse button to zoom in and right-click to zoom out
- Use the following menu selection to set the start and end time:

  Tools
  > View Time Frame ...

- Use the **Time Display** menu to select one of the following units of time that will be used to display the trace:
  - **Actual Time** displays the actual time that was recorded in the trace file (initialized to zero)
  - **Clock Ticks** displays the actual time normalized and displayed in clock ticks. (A *clock tick* is determined by the resolution of the real-time clock.) This option only applies to Altix ia64 systems.
- Display the time difference between the two points on the graph by clicking the middle mouse button at the first point (which will display a vertical line drawn as dashes) and at the second point (which will cause a second vertical line to appear), which then displays the time difference in the left of the bottom status bar.

Seeing Process Details

To see a particular process, click on the **Process Analysis** tab and select the specific process ID number on the left. The right side of the screen will display characteristics of the process and system call accounting. The `Kernel` process (process 0) summarizes the system.

Seeing All Event Trace Details

To see details about all trace entries, click on the **Raw Trace** tab.

Filtering Events Based on CPU

To turn on or off events based on CPU both in the **Event Graph** and **Raw Trace** output, use the **Filter CPU's** menu. Enter the CPUs that you want to display.

Exiting from the `tracevisualizer` GUI

To exit from the `tracevisualizer` GUI, select:

File
> Exit

Removing SGI Linux Trace

To remove the `sltdebug` kernel and SGI Linux Trace, do the following:

1. Log in as the superuser.
2. Remove the `kernel-sltdebug` RPM:

   ```
   [root@linux root]# rpm -ev kernel-sltdebug-*
   ```

3. Remove the `TraceToolkit` RPM:

   ```
   [root@linux root]# rpm -ev TraceToolkit-*
   ```

4. Reboot the system (so that it uses the default kernel).

Note: If you remove the `kernel-sltdebug` RPM but not the `TraceToolkit` RPM, or if you reboot into a non-`sltdebug` kernel without removing either RPM, you must do the following to stop the `slt` service and prevent it from starting after a reboot

```
[root@linux root]# /etc/init.d/slt stop
[root@linux root]# chkconfig slt off
```

Troubleshooting

This chapter discusses the following:

- "Diagnostic Tools" on page 151
- "Problem Removing `/rtcpus`" on page 154
- "Problem Removing `/boot` Cpuset" on page 155

Diagnostic Tools

You can use the following diagnostic tools:

- Use the `cat`(1) command to view the `/proc/interrupts` file in order to determine where your interrupts are going:

  ```
  cat /proc/interrupts
  ```

  For an example, see Appendix A, "Example Application" on page 157.

- Use the `profile.pl`(1) Perl script to do procedure-level profiling of a program and discover latencies. For more information, see the `profile.pl`(1) man page.

- Use the following `ps`(1) command to see where your threads are running:

  ```
  ps -FC processname
  ```

  For an example, see Appendix A, "Example Application" on page 157.

  To see the scheduling policy, real-time priority, and current processor of all threads on the system, use the following command:

  ```
  ps -eLo pid,tid,class,rtprio,psr,cmd
  ```

  For more information, see the `ps`(1) man page.

- Use the `top`(1) command to display the largest processes on the system. For more information, see the `top`(1) man page.

- Use the `strace`(1) command to determine where an application is spending most of its time and where there may be large latencies. The `strace` command is a

very flexible tool for tracing application activities and can be used for tracking down latencies in an application. Following are several simple examples:

– To see the amount of time being used by system calls in the form of histogram data for a program named `hello_world`, use the following:

```
[root@linux root]# strace -c hello_world
execve("./hello_world", ["hello_world"], [/* 80 vars */]) = 0
Hello World
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 27.69    0.000139          28         5         3 open
 20.92    0.000105          15         7           mmap
 10.76    0.000054          54         1           write
  7.57    0.000038          13         3           fstat
  6.57    0.000033          17         2         1 stat
  5.98    0.000030          15         2           munmap
  4.58    0.000023          12         2           close
  4.38    0.000022          22         1           mprotect
  4.18    0.000021          21         1           madvise
  2.99    0.000015          15         1           read
  2.39    0.000012          12         1           brk
  1.99    0.000010          10         1           uname
------ ----------- ----------- --------- --------- ----------------
100.00    0.000502                    27         4 total
```

– You can record the actual chronological progression through a program with the following command (line breaks added for readability):

```
[root@linux root]# strace -ttT hello_world
14:21:03.974181 execve("./hello_world", ["hello_world"], [/* 80 vars */]) = 0
..
14:21:03.976992 mmap(NULL, 65536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
   = 0x2000000000040000 <0.000007>
14:21:03.977053 write(1, "Hello World\n", 12Hello World
) = 12 <0.000008>
14:21:03.977109 munmap(0x2000000000040000, 65536) = 0 <0.000009>
14:21:03.977158 exit_group(0)           = ?
```

The time stamps are displayed in the following format:

hour:*minute*:*second*.*microsecond*

The execution time of each system call is displayed in the following format:

<*second*>

Note: You can use the `-p` option to attach to another already running process.

For more information, see the `strace`(1) man page.

- Use the `tracevisualizer` command. See Chapter 10, "SGI Linux Trace" on page 137.
- To find the CPU-to-core numbering scheme, examine the following fields in the `/proc/cpuinfo` file:

```
processor
physical id
core id
```

For example, the following for an Altix XE x86_64 system shows that logical CPU 0 and CPU 2 are cores sharing the same socket:

```
processor       : 0
...
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2


processor       : 2
...
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
```

The following output for an Altix ia64 system shows that two logical CPUs share the same socket:

```
processor  : 0
vendor     : GenuineIntel
arch       : IA-64
family     : 32
```

```
model      : 1
revision   : 1
archrev    : 0
features   : branchlong, 16-byte atomic ops
cpu number : 0
cpu regs   : 4
cpu MHz    : 1669.000503
itc MHz    : 416.875000
BogoMIPS   : 3325.95
siblings   : 2
physical id: 0
core id    : 0
thread id  : 0

processor  : 1
vendor     : GenuineIntel
arch       : IA-64
family     : 32
model      : 1
revision   : 1
archrev    : 0
features   : branchlong, 16-byte atomic ops
cpu number : 0
cpu regs   : 4
cpu MHz    : 1669.000503
itc MHz    : 416.875000
BogoMIPS   : 3325.95
siblings   : 2
physical id: 0
core id    : 1
thread id  : 0
```

Problem Removing `/rtcpus`

You should stop real-time processes before using the `--disable` option. However, the script will attempt to remove the process from the real-time CPUs and display the following failure message if it was unable to move them:

```
"*** Problem removing /rtcpus/rtcpu3. cpuset***
 Try again.  If that doesn't work check /dev/cpuset/rtcpus/rtcpu3/tasks
 for potential problem PIDS;
```

Problem Removing `/boot` Cpuset

Removing the REACT configuration by using `reactconf.pl --disable` can sometimes fail due to a large number of sleeping processes in the `/boot` cpuset. The following error message indicates this problem:

```
*** Problem removing /boot cpuset. ***
Try again, or check /dev/cpuset/boot/tasks for potential problem PIDS
and manually move them to root cpuset.
# echo  | cpuset --attach=/
```

To resolve the problem, run the `reactconf.pl --disable` command again.

Example Application

This appendix discusses an example of a multithreaded application that demonstrates using external interrupts and other aspects of REACT. It uses netlink sockets to communicate from kernel space to user space. You can use it as a performance benchmark to compare between machines or settings within REACT, such as for external interrupts, cpusets, and CPU isolation.

The application is composed of the following:

- Example kernel module, which shows examples of the following concepts:
  - Creating and building a driver with a standard miscellaneous device interface
  - Setting up and registering a external interrupt handler
  - Creating and binding a kernel thread
  - Using netlink sockets to communicate with a user application
- Example user-space application, which shows examples of the following concepts :
  - Assigning threads to cpusets, thereby changing thread/CPU affinity
  - Changing thread/CPU affinity without cpusets
  - Creating, destroying, and signaling threads
  - Changing a thread's scheduling policies and priorities
  - Locking memory
  - Setting up a netlink socket to communicate with a kernel thread

This example puts the data into a matrix and multiplies two matrices together. The worker thread displays the multiplication and calculates how long it takes to multiply the two matrices together. You can modify the size of the matrix to see how it effects the time to calculate the multiplication. For example, you could use a field-programmable gate array (FPGA) to implement the multiply function in order to show how much faster it is under these circumstances than under normal calculation. You could also run on two different platforms to compare the speed of integer multiplication.

This program runs as a multithreaded process. The main process launches the following threads, sets each thread's scheduling policy and priority, and displays the thread policy and priority information:

- The receiving thread (`netlink_receive`) does the following:
  1. Tells the kernel to start the processing of interrupts (a one-time event)
  2. Locks its current and future memory (if requested).
  3. Uses the example kernel module driver to do the following:
     a. Waits for messages from the kernel netlink socket.
     b. Signals the worker thread with the data from the driver.
- The worker thread (`worker_routine`) does the following:
  1. Waits to be signaled by the receive thread for data.
  2. Fills two matrices with the data and multiplies them together. The output will be printed to the console.
  3. Calculates the time it takes for the matrices to by multiplied together.
- The interrupt handler (`extint_run`) runs when a hardware external interrupt is received. It wakes up the `bench_extintd` thread.
- The kernel thread (`bench_extintd`) gets data, sends messages with the data to the receiving thread (`netlink_receive`), and then sleeps until another interrupt occurs.

`netlink_receive` is set at a higher priority than the time-consuming `worker_routine`.

Figure A-1 describes the example. Step 1 occurs once, but steps 2 through 4 are repeated for each external interrupt.

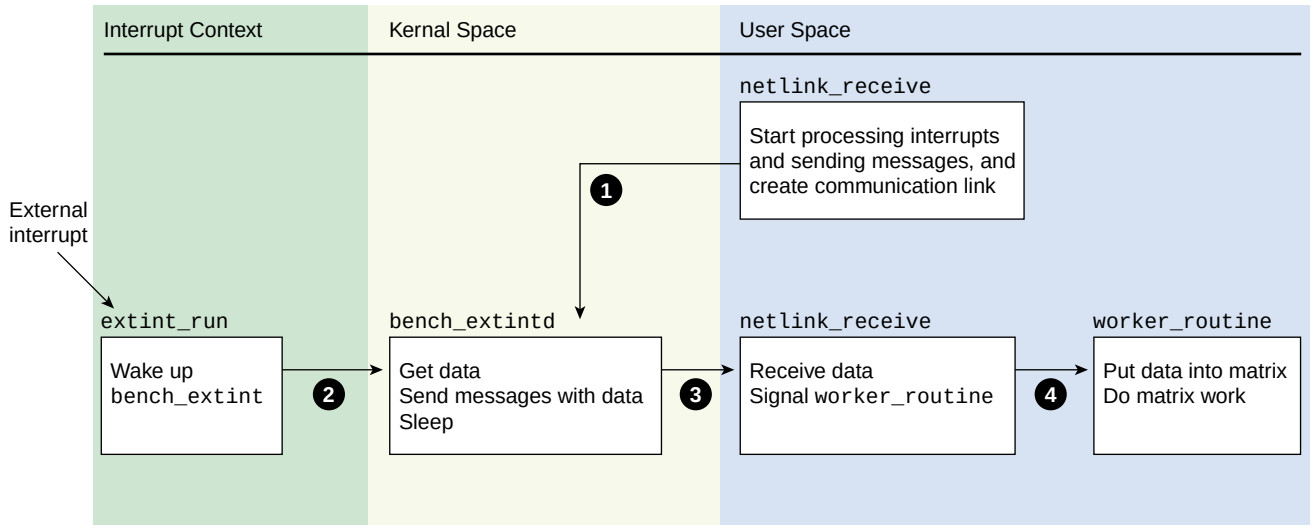

Figure A-1 Example Work Flow

The rest of this section discusses the following:

- "Setting Up External Interrupts" on page 159
- "Building and Loading the Kernel Module" on page 160
- "Building the User-Space Application" on page 161
- "Running the Sample Application" on page 161
- "`set_affinity` code" on page 164

Setting Up External Interrupts

To set up external interrupts, do the following:

1. Log in to the target system as `root`.
2. Load the `ioc4_extint` module:

```
[root@linux root]# modprobe ioc4_extint
```

3. Insert the required information into the `source`, `mode`, and `period` files in the `/sys/class/extint/extint0/` directory. For example:

```
[root@linux root]# echo loopback >/sys/class/extint/extint0/source
[root@linux root]# echo toggle >/sys/class/extint/extint0/mode
[root@linux root]# echo 1000000 >/sys/class/extint/extint0/period
```

For more information about external interrupts see Chapter 3, "External Interrupts" on page 19..

Building and Loading the Kernel Module

To build the `bench_extint_mod` application kernel module, do the following on the target system:

1. Log in to the target system as `root`.
2. Ensure that the `kernel-source-*.rpm` RPM is installed.
3. Change to the `kernel` directory:

   ```
   [root@linux root]# cd /usr/share/react/samples/bench/kernel
   ```

4. Build the `bench_extint_mod.ko` file:

```
[root@linux kernel]# make -C /lib/modules/‘uname -r‘/build SUBDIRS=$PWD modules
```

   For more information, see the `uname`(1) man page.

5. Copy the `bench_extint_mod.ko` file to the directory defined by the `uname -r` output:

   ```
   [root@linux kernel]# cp bench_extint_mod.ko /lib/modules/‘uname -r‘
   ```

6. Make a dependency file:

   ```
   [root@linux kernel]# depmod
   ```

   For more information, see the `depmod`(8) man page.

7. Load the `bench_extint_mod` module:

   ```
   [root@linux kernel]# modprobe bench_extint_mod
   ```

   For more information, see the `modprobe`(8) man page.

8. Use the `bench_extint_mod` kernel module with the `bench_example` application.

Note: You must load the `ioc4_extint` module before the `bench_extint_mod` module.

Building the User-Space Application

To build the user-space module, do the following:

1. Change to the user directory:

```
[root@linux root]# cd /usr/share/react/samples/bench/user
```

2. Build the module:

```
[root@linux user]$ make
```

Running the Sample Application

You can run the `bench_example` application in the following modes:

- *Matrix multiply mode* receives data from the kernel module and puts that data into a matrix. After two matrices are full, it multiplies them together and calculates the amount of time taken for the calculation. See "Matrix Multiply Mode Examples" on page 162.
- *Netlink socket bench mode* causes the application to send multiple messages from kernel space to user space during one iteration. The number of messages sent per iteration depends upon notification from the user application to start sending messages. See "Netlink Socket Benchmark Mode Examples" on page 163.

Do the following:

- Ensure that you have the `bench_extint_mod` module loaded by using the `lsmod`(1) command, which should show it in the module list.

For example:

```
[root@linux root]# lsmod
          Module                  Size  Used by
          bench_extint_mod      546232  0
          ioc4_extint            27272  0
          ioc4                   24704  1 ioc4_extint
          extint                 32008  2 bench_extint_mod,ioc4_extint
```

If the output does not include `bench_extint_mod`, follow the instructions in "Building and Loading the Kernel Module" on page 160.

- Execute the `bench` command as desired.

  The `bench` command has the following options:

`-b`*messages*	Runs the application in benchmark mode with the specified number of messages in each send. *messages* is an integer in the range `1` through `100`. (If you enter an invalid number, the default is `100`.)
`-h`	Prints usage instructions.
`-k`*cpu*	Specifies the CPU where the kthread will run.
`-m`	Locks memory.
`-p`*cpu*	Specifies the CPU where the bench process will run.
`-r`*cpu*	Specifies the CPU where the receive thread will run.
`-s`*size*	Specifies the size of buffers in bytes for network socket bench mode. The default is `1024`. You can vary the size of the buffers to see the impact on performance.
`-t`*sec*	Specifies the total run time in seconds, with a maximum of 30 seconds. The default is `30`.
`-w`*cpu*	Specifies the CPU where the worker thread will run.

Matrix Multiply Mode Examples

To run in Matrix Multiply for 30 seconds:

```
[root@linux root]# ./bench -t30
```

To run with memory locked and bench processes running on CPU 2 (real-time or non real-time):

```
[root@linux root]# ./bench -m -p2 -t30
```

To run the bench process on CPU 3 and the worker and receive threads on CPU 2:

```
[root@linux root]# ./bench -m -p3 -r2 -w2 -t30
```

See also "`set_affinity` code" on page 164.

Netlink Socket Benchmark Mode Examples

The following shows an example in bench mode that runs for 30 seconds with memory locked and a buffer size of 512 bytes. There are 50 messages in each send. The process is running on CPU 1, the receive thread running on CPU 2, the worker thread is running on CPU 3, and the kernel thread is running on CPU 1:

```
[root@linux root]# ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
```

If you have multiple terminals open, you can run the following `tail`(1) and `ps`(1) commands to see where things are running:

```
[root@linux root]# tail -f /var/log/messages
Feb 16 08:54:05 dewberry kernel: bench_extint init
Feb 16 08:54:40 dewberry kernel: bench_extint ran 14958, thread ran 14958 dropped msgs 0
Feb 16 08:54:40 dewberry kernel: ioctl unregister bench_extint

[root@linux root]# ps -eLF
UID     PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY        TIME CMD
root  10076  6747 10076  0    3  5951 18696   1 11:34 pts/0  00:00:00 ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
root  10076  6747 10078 11    3  5951 18696   2 11:34 pts/0  00:00:00 ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
root  10076  6747 10079 99    3  5951 18696   3 11:34 pts/0  00:00:04 ./bench -m -t30 -p1 -r2 -w3 -k1 -b50 -s512
root  10077    15 10077 10    1     0     0   1 11:34 ?      00:00:00 [bench_exintd]
```

`set_affinity` code

You can use the following functions to set process and thread affinity for real-time and non real-time CPUs. You can compile this file as part of another application, but you must link it against the `libcpuset`.

```
#include <sys/syscall.h>
#include <unistd.h>
#include "errors.h"

#define CPUSET_ROOT "/dev/cpuset"
#define BITS_PER_LONG (sizeof(unsigned long) * 8)


pid_t _gettid(){
  return syscall(__NR_gettid);
}

void do_pthread_affinity(int cpu) {

        int nrcpus = cpuset_cpus_nbits();
        int bitmask_size = (nrcpus/BITS_PER_LONG);
        unsigned long cpus[bitmask_size];
        pid_t tid = _gettid();

        cpus[cpu/64] = 1 << (cpu % 64);

        if (sched_setaffinity(tid, sizeof(cpus), cpus)) {
                perror("set_affinity");
                exit(1);
        }

}
void set_thread_affinity(int cpu) {

        char path[50],fullpath[50];

        sprintf(path, "/rtcpus/rtcpu%d", cpu);
        sprintf (fullpath, CPUSET_ROOT "/rtcpus/rtcpu%d",cpu);

        if (access(fullpath, F_OK) != 0) {
                /* no cpuset, so try moving it without */
```

```
                do_pthread_affinity(cpu);
                return;
        }

        /* Move the process into the cpuset */
        if (cpuset_move(_gettid(), path) == -1) {
                perror("cpuset_move");
                exit(1);
        }
}


/* Set the current proc to run on cpu . */
void set_process_affinity(int cpu) {

        int nrcpus = cpuset_cpus_nbits();
        int bitmask_size = (nrcpus/BITS_PER_LONG);
        unsigned long cpus[bitmask_size];
        char path[50],fullpath[50];
        unsigned long mask;

        cpus[cpu/64] = 1 << (cpu % 64);

        sprintf(path, "/rtcpus/rtcpu%d", cpu);
        sprintf (fullpath, CPUSET_ROOT "/rtcpus/rtcpu%d",cpu);

        if (access(fullpath, F_OK) != 0) {
                /* no cpuset, so try moving it without */
        if (sched_setaffinity(getpid(), sizeof(cpus), cpus)) {
                        perror("set_process_affinity");
                }
                return;
        }
        /* Move the process into the cpuset */
        if (cpuset_move(getpid(), path) == -1)
                perror("cpuset_move");


}
```

High-Resolution Timer Example for Altix ia64 Servers

Note: This appendix does not apply to Altix XE servers.

Example B-1 demonstrates the use of SGI high-resolution timers. It will run both high-resolution and normal-resolution POSIX timers in both relative mode and absolute mode.

Example B-1 High-Resolution Timer

```
/****************************************************************************
 *                                                                          *
 * This sample program demonstrates the use of SGI high resolution timers   *
 * in SGI REACT.  It will run both high resolution and normal resolution     *
 * POSIX timers in both relative mode and absolute mode.                    *
 *                                                                          *
 * This sample program requires the REACT rtgfx kernel, which supports high  *
 * resolution timers via clock id CLOCK_SGI_CYCLE.                          *
 *                                                                          *
 * Note that the clock id CLOCK_REALTIME is RTC (high) resolution with       *
 * clock_gettime() on SGI systems, but is jiffies resolution with           *
 * timer_create(), clock_getres() and other POSIX time calls.               *
 *                                                                          *
 *                                                                          *
 * A simple way to build this sample program is:                            *
 *   cc -o timer_sample timer_sample.c -lrt                                 *
 *                                                                          *
 * Invocation example (500 usec timer):                                     *
 *   ./timer_sample 500                                                     *
 *                                                                          *
 * Invocation example (500 usec timer on realtime cpu 2):                   *
 *   cpuset --invoke=/rtcpu2 --invokecmd=./timer_sample 500                 *
 *                                                                          *
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <signal.h>
#include <time.h>
#include <errno.h>
#include <asm/unistd.h>
#include <pthread.h>
#include <strings.h>
#include <sys/time.h>
#include <getopt.h>
#include <libgen.h>

#define CLOCK_SGI_CYCLE 10      /* value of CLOCK_SGI_CYCLE from kernel */

struct timespec time1;
int flag;

/* Timer has triggered, get current time and indicate completion */
void sigalarm(int signo)
{
        clock_gettime(CLOCK_REALTIME,&time1);
        flag = 1;
}


int timer_test(int clock_id, long nanosec) {
        struct itimerspec ts;
        struct sigevent se;
        struct sigaction act;
        sigset_t sigmask;
        struct timespec sleeptime, time0;
        timer_t timer_id;
        long i;
        int signum = SIGRTMAX;
        int status;

        /* Set up sleep time for loops: */
        sleeptime.tv_sec = 1;
        sleeptime.tv_nsec = 0;

        /* Set up signal handler: */
        sigfillset(&act.sa_mask);
```

```
act.sa_flags = 0;
act.sa_handler = sigalarm;
sigaction(signum, &act, NULL);

/* Set up timer: */
memset(&se, 0, sizeof(se));
se.sigev_notify = SIGEV_SIGNAL;
se.sigev_signo = signum;
se.sigev_value.sival_int = 0;
status = timer_create(clock_id, &se, &timer_id);
if (status < 0) {
        perror("timer_create");
        return -1;
}


/* Start relative timer: */
ts.it_value.tv_sec = nanosec / 1000000000;
ts.it_value.tv_nsec = (nanosec % 1000000000);
ts.it_interval.tv_sec = 0;
ts.it_interval.tv_nsec = 0;

printf("Waiting for timeout of relative timer: ");
fflush(stdout);
flag = 0;
/* Get current time for reference */
clock_gettime(CLOCK_REALTIME,&time0);
/*
 * There will be some latency between getting the start time above,
 * and setting the relative time in timer_settime.
 */
status = timer_settime(timer_id, 0, &ts, NULL);
if (status < 0) {
        perror("timer_settime");
        return -1;
}

/* Loop waiting for timer to go off */
while (!flag) nanosleep(&sleeptime, NULL);
if (time1.tv_nsec < time0.tv_nsec)
        printf("Total time=%luns\n",
```

```
                1000000000LL - (time0.tv_nsec - time1.tv_nsec) +
                ((time1.tv_sec - time0.tv_sec -1)*1000000000LL));
else
        printf("Total time=%luns\n",
                time1.tv_nsec - time0.tv_nsec +
                ((time1.tv_sec - time0.tv_sec)*1000000000LL));


/* Start absolute timer: */
printf("Waiting for timeout of absolute timer: ");
fflush(stdout);
flag = 0;
/* Get current time and add timeout to that for absolute time */
clock_gettime(CLOCK_REALTIME,&time0);
i = time0.tv_nsec + (nanosec % 1000000000);
ts.it_value.tv_nsec = i % 1000000000;
ts.it_value.tv_sec = (time0.tv_sec + (nanosec / 1000000000)) +
                (i / 1000000000);
/* There should be less latency than what we saw above */
status = timer_settime(timer_id, TIMER_ABSTIME, &ts, NULL);
if (status < 0) {
        perror("timer_settime");
        return -1;
}

/* Loop waiting for timer to go off */
while (!flag) nanosleep(&sleeptime, NULL);
if (time1.tv_nsec < time0.tv_nsec)
        printf("Total time=%luns\n",
                1000000000LL - (time0.tv_nsec - time1.tv_nsec) +
                ((time1.tv_sec - time0.tv_sec -1)*1000000000LL));
else
        printf("Total time=%luns\n",
                time1.tv_nsec - time0.tv_nsec +
                ((time1.tv_sec - time0.tv_sec)*1000000000LL));


/* Cleanup */
timer_delete(timer_id);

return 0;
```

```
}

int main(int argc, char *argv[])
{
        long timeout;

        if (argc < 2) {
                printf("usage: %s <timeout usec>\n", basename(argv[0]));
                return -1;
        }

        timeout = atol(argv[1]);
        if (timeout <= 0) {
                printf("Timeout negative or 0 specified\n");
                printf("usage: %s <timeout usec>\n", basename(argv[0]));
                return -1;
        }

        /* Run timer_test with normal (jiffies) resolution timer. */
        printf("\nRunning with CLOCK_REALTIME (normal resolution)..\n");
        if (timer_test(CLOCK_REALTIME, timeout * 1000)) {
                return -1;
        }


        /* Now run timer_test with high resolution timer. */
        printf("\nRunning with CLOCK_SGI_CYCLE (high resolution)..\n");
        return timer_test(CLOCK_SGI_CYCLE, timeout*1000);
}
```

Sample User-Level Interrupt Programs

The following applications demonstrate some of the user-level interrupt (ULI) interface:

- "`uli_sample` Sample Program" on page 173
- "`uli_ei` Sample Program" on page 174

The applications are installed with the ULI RPM and are located in:

`/usr/share/react/uli/examples/`

`uli_sample` Sample Program

The `uli_sample` program registers for notification on CPU 0 for occurrences of a specified interrupt number. To use `uli_sample`, do the following:

1. Load the ULI feature kernel module:

   ```
   [root@linux root]# modprobe uli
   ```

2. Change to the directory containing `uli_sample`:

   ```
   [root@linux root]# cd /usr/share/react/uli/examples/
   ```

3. Run `uli_sample`, where *interrupt#* is the interrupt number:

   ```
   [root@linux root]# ./uli_sample interrupt#
   ```

   For example, to register for notification on CPU 0 for occurrences of the interrupt number 34, enter the following:

   ```
   [root@linux root]# ./uli_sample 34
   ```

`uli_ei` Sample Program

The `uli_ei` program requires the external interrupt to run and prints a message every time the external interrupt line is toggled. To use `uli_ei`, do the following:

1. Load the ULI feature kernel module, if not already done:

   ```
   [root@linux root]# modprobe uli
   ```

2. Load the external interrupt kernel module:

   ```
   [root@linux root]# modprobe ioc4_extint
   ```

3. Set the external interrupt mode to `toggle`:

   ```
   [root@linux root]# echo toggle > /sys/class/extint/extint0/mode
   ```

4. Change to the directory containing `uli_ei`:

   ```
   [root@linux root]# cd /usr/share/react/uli/examples/
   ```

5. Run `uli_ei`:

   ```
   [root@linux root]#  ./uli_ei
   ```

Reading MAC Addresses Sample Program

Example D-1 reads the MAC address from the Broadcom Corporation NetXtreme BCM5704 Gigabyte Ethernet card on an SGI Altix ia64 system. It demonstrates how to memory map and interact with hardware devices from user space.

Note: The following example is for an Altix ia64 architecture server. It will not correctly read the MAC address on Altix XE x86_64 servers because the offset to the MAC address is different (that is, it is not 0x410 on Altix XE servers).

Example D-1 Reading MAC Addresses on an Altix Server

```
/*
 * Sample code to map in PCI memory for a specified device and display
 * the contents of a (hard coded) register.
 */

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <errno.h>

/*
 * The MAC address is located in the first bar of the Broadcom Corporation
 * NetXtreme BCM5704 Gigabyte ethernet controller, so we open the first
 * resource file, which is resource0.
 */
#define RESOURCE "resource0"
/* Offset from the first base address to the MAC address */
#define MAC_OFFSET 0x410

int main(int argc, char **argv)
{
  char path[128];
  unsigned long domain, bus, slot, function;
  char *ptr;
  char *endptr;
```

```
unsigned int *data;
int fd;

if(argc != 5)
{
  printf("Must supply domain, bus, slot and function in hex\n");
  printf("Using the lspci command is one way to find this information\n");
  exit(1);
}

domain = strtoul(argv[1], &endptr, 16);
if (*endptr != '\0') {
  printf("domain out of range\n");
  exit(1);
}
bus = strtoul(argv[2], &endptr, 16);
if (*endptr != '\0') {
  printf("bus out of range\n");
  exit(1);
}
slot = strtoul(argv[3], &endptr, 16);
if (*endptr != '\0') {
  printf("slot out of range\n");
  exit(1);
}
function = strtoul(argv[4], &endptr, 16);
if (*endptr != '\0') {
  printf("function out of range\n");
  exit(1);
}

/*
 * Get the pathname to the proper resource file for the first bar on the
 * specified device, then open the file and memory map it.
 */
sprintf(path, "/sys/bus/pci/devices/%04x:%02x:%02x.%x/%s",
        (unsigned)domain, (unsigned)bus, (unsigned)slot, (unsigned)function,
        RESOURCE);
printf("resource file path is %s\n", path);

if ((fd = open(path, O_RDWR)) == -1) {
```

```
    perror("Couldn't open resource file");
    exit(1);
  }
  ptr = mmap( NULL, getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if (ptr == MAP_FAILED)
  {
    perror("mmap");
    exit(1);
  }

  data = (unsigned int*) (ptr + MAC_OFFSET);
  printf( "ptr is %p, data is %p\n", ptr, data );
  printf( "MAC is %04hx%08x\n", *(unsigned short *)data, *(data+1));
  return 0;
}
```

IRIX Differences

This appendix summarizes some differences between REACT for IRIX and REACT for Linux:

- `sysfs` attribute files are not compatible with the IRIX `ei`(7) driver. The IRIX driver uses `ioctl`(2) to interact with user space, which is generally not preferred in Linux and does not perform well due to kernel locking issues. An advantage of the attribute file interface is that it is easy to use from the command line, rather than requiring specially written and compiled applications.
- IRIX has the ability to attach capabilities to an executable file, so that a process inherits those capabilities when the file is run via `exec()`. The Linux implementation of capabilities is limited to processes.
- The Linux `capget`(2) and `capset`(2) system calls are used for getting and setting capabilities and are essentially equivalent to the IRIX `cap_get_proc`(3C) and `cap_set_proc`(3C) system calls. For details, see the man pages for the system calls and the Linux `capabilities`(7) and IRIX `capabilities`(4) man pages.

Glossary

activity

When using the frame scheduler, the basic design unit: a piece of work that can be done by one thread or process without interruption. You partition the real-time program into activities and use the frame scheduler to invoke them in sequence within each frame interval.

address space

The set of memory addresses that a process may legally access. The potential address space in Linux is 2^{64}; however, only addresses that have been mapped by the kernel are legally accessible.

arena

A segment of memory used as a pool for allocation of objects of a particular type.

asynchronous I/O

I/O performed in a separate process so that the process requesting the I/O is not blocked waiting for the I/O to complete.

average data rate

The rate at which data arrives at a data collection system, averaged over a given period of time (seconds or minutes, depending on the application). The system must be able to write data at the average rate, and it must have enough memory to buffer bursts at the *peak data rate*.

clock tick

A measure of time determined by the resolution of the real-time clock.

control law processor

A type of stimulator provides the effects of laws of physics to a machine.

controller thread

A top-level process that handles startup and termination.

CPU

Central Processing Unit refers to cores (not sockets).

device driver

Code that operates a specific hardware device and handles interrupts from that device.

device service time

The time the device driver spends processing the interrupt and dispatching a user thread.

device special file

The symbolic name of a device that appears as a filename in the `/dev` directory hierarchy. The file entry contains the *device numbers* that associate the name with a *device driver*.

external interrupt

A hardware signal from an I/O device, such as the SGI IOC4 chip, that is generated in response to a voltage change on an externally accessible hardware port.

fastcall

A version of a function call that has been optimized in assembler in order to bypass the context switch typically necessary for a full system call.

file descriptor

A number returned by `open()` and other system functions to represent the state of an open file. The number is used with system calls such as `read()` to access the opened file or device.

firm real-time program

A program that experiences a significant error if it misses a deadline but can recover from the error and can continue to execute. See also *hard real-time program* and *soft real-time program*.

frame interval

The amount of time that a program has to prepare the next display frame. A frame rate of 60 Hz equals a frame interval of 16.67 milliseconds.

frame rate

The frequency with which a simulator updates its display, in cycles per second (Hz). Typical frame rates range from 15 to 60 Hz.

frame scheduler

A process execution manager that schedules activities on one or more CPUs in a predefined, cyclic order.

frame scheduler controller

The thread or process that creates a frame scheduler. Its thread or process ID is used to identify the frame scheduler internally, so a thread or process can only be identified with one scheduler.

frame scheduler controller thread

The thread that creates a frame scheduler.

guaranteed rate

A rate of data transfer, in bytes per second, that definitely is available through a particular file descriptor.

hard real-time program

A program that experiences a catastrophic error if it misses a deadline. See also *firm real-time program* and *soft real-time program*.

hardware latency

The time required to make a CPU respond to an interrupt signal.

hardware-in-the-loop (HWIL) simulator

A simulator in which the role of operator is played by another computer.

interrupt

A hardware signal from an I/O device that causes the computer to divert execution to a device driver.

interrupt information template

An array of `frs_intr_info_t` data structures, where each element in the array represents a minor frame.

interrupt propagation delay

See *hardware latency*.

interrupt redirection

The process of directing certain interrupts to specific real-time processors and directing other interrupts away from specific real-time processors in order to minimize the latency of those interrupts.

interrupt response time

The total time from the arrival of an interrupt until the user process is executing again. Its main components are *hardware latency*, *software latency*, *device service time*, and *mode switch*.

interrupt service routine (ISR)

A routine that is called each time an interrupt occurs to handle the event.

interval time counter (ITC)

A 64–bit counter that is scaled from the CPU frequency and is intended to allow an accounting for CPU cycles.

interval timer match (ITM) register

A register that allows the generation of an interval timer when a certain ITC value has been reached.

isolate

To remove the Linux CPU from load balancing considerations, a time-consuming scheduler operation.

IPI

Interprocessor interrupt.

jitter

Numerous short interruptions in process execution.

locks

Memory objects that represent the exclusive right to use a shared resource. A process that wants to use the resource requests the lock that (by agreement) stands for that resource. The process releases the lock when it is finished using the resource. See *semaphore*.

major frame

The basic frame rate of a program running under the frame scheduler.

master scheduler

The first frame scheduler, which provides the time base for the others. See also *slaves* and *sync group*.

minor frame

The scheduling unit of the frame scheduler, the period of time in which any scheduled thread or process must do its work.

mode switch

The time it takes for a thread to switch from kernel mode to user mode.

new pthreads library (NPTL)

The Linux pthreads library shipped with 2.6 Linux.

overrun

When incoming data arrives faster than a data collection system can accept it and therefore data is lost.

overrun exception

When a thread or process scheduled by the frame scheduler should have yielded before the end of the minor frame but did not.

page fault

The hardware event that results when a process attempts to access a page of virtual memory that is not present in physical memory.

pages

The units of real memory managed by the kernel. Memory is always allocated in page units on page-boundary addresses. Virtual memory is read and written from the swap device in page units.

peak data rate

The instantaneous maximum rate of input to a data collection system. The system must be able to accept data at this rate to avoid overrun. See also *average data rate*.

process

The entity that executes instructions in a Linux system. A process has access to an *address space* containing its instructions and data.

pthread

A thread defined by the POSIX standard. Pthreads within a process use the same global address space. Also see *thread*.

rate-monotonic analysis

A technique for analyzing a program based on the periodicities and deadlines of its threads and events.

rate-monotonic scheduling

A technique for choosing scheduling priorities for programs and threads based on the results of *rate-monotonic analysis*.

restrict

To prevent a CPU from running scheduled processes

scheduling discipline

The rules under which an activity thread or process is dispatched by a frame scheduler, including whether or not the thread or process is allowed to cause *overrun* or *underrun exceptions*.

segment

Any contiguous range of memory addresses. Segments as allocated by Linux always start on a page boundary and contain an integral number of pages.

semaphore

A memory object that represents the availability of a shared resource. A process that needs the resource executes a `p` operation on the semaphore to reserve the resource, blocking if necessary until the resource is free. The resource is released by a `v` operation on the semaphore. See also *locks*.

shield

To switch off the timer (scheduler) interrupts that would normally be scheduled on a CPU.

simulator

An application that maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model as visual output.

slaves

The other schedulers that take their time base interrupts from the *master scheduler*. See also *sync group*.

soft real-time program

A program that can occasionally miss a deadline with only minor adverse effects. See also *firm real-time program* and *hard real-time program*.

software latency

The time required to dispatch an interrupt thread.

spraying interrupts

The distribution of I/O interrupts across all available processors as a means of balancing the load.

stimulator

An application that maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and outputs the changed model as nonvisual output.

sub-buffer

A portion of a CPU buffer. The size of the CPU buffer equals the number of sub-buffers multiplied by the sub-buffer size.

sync group

The combination of a *master scheduler* and *slaves*.

thread

An independent flow of execution that consists of a set of registers (including a program counter and a stack). Also see *pthread*.

TLB

Translation lookaside buffer, which translates CPU virtual memory addresses to bus physical memory addresses.

transport delay

The time it takes for a simulator to reflect a control input in its output display. Too long a transport delay makes the simulation inaccurate or unpleasant to use.

ULI process

A user process that has registered a function with the kernel, linked into the process in the normal fashion, to be called when a particular interrupt is received.

underrun exception

When a thread or process scheduled by the frame scheduler should have started in a given minor frame but did not (owing to being blocked), an underrun exception is signaled. See *overrun exception*.

unsynchronized drifty ITCs

Systems with processors that run at the same speed but do not have the same clock source and therefore their ITC values may experience drift relative to one another.

user-level interrupt (ULI)

A facility that allows a hardware interrupt to be handled by a user process.

Index

A

abstraction layer, 19
access to select REACT features, 125
activity thread management, 71
actual time, 147
address space (locking in memory), 104
aircraft simulator, 3
application example, 157
asynchronous I/O, 93
average data rate, 5

B

backup configuration files, 110
barrier facility, 43
/boot, 110
/boot/grub/menu.lst, 110
bootcpuset, 46, 110

C

C language, 6
cache warming, 66
callout deregistration, 33
callout mechanism, 31
callout registration, 32
CAP_DAC_OVERRIDE authority, 125
cap_get_proc, 181
CAP_IPC_LOCK, 127
cap_over kernel module, 128
cap_set_proc, 181
CAP_SYS_NICE, 127
CAP_SYS_PTRACE, 128
CAP_SYS_RESOURCE, 128
CAP_SYS_TIME, 128
capabilities, 126
capabilities and executable file, 179
captools package, 127
cat, 151
chkconfig, 111
clock processor, 47
clock tick, 147
clock_getres(), 13
clock_gettime, 13, 14
CLOCK_MONOTONIC, 13
CLOCK_REALTIME, 13
clock_settime, 13
CLOCK_SGI_CYCLE, 14
clocks, 12
configuration, 109
configuration scripts, 109
console driver, 49
console interrupts, 11
console node, 49
control law process stimulator, 4
controller thread, 80
CPU
  isolating, 50
  restricting, 10, 50
  workload control, 41
CPU 0, 47, 49
CPU-bound, 9
CPU-to-core numbering scheme, 153
cpuset, 46, 123
cpusets, 58
cycles per second, 2

D

data collection system, 4

deregistration of callout, 33
dev attribute file, 20
/dev/cpuset/boot/mems, 123
/dev/cpuset/rtcpus/rtcpuN/mems, 123
/dev/extint#, 104
device service time, 53, 55
device special file, 104
device-driver time base, 59
diagnostic tools, 151
direct RTC access, 14
disabling REACT, 135
disciplines, 9
disk I/O optimization, 93
distributed applications, 17
dplace, 124
driver creation and building, 157
driver deregistration, 31
driver interface, 26
driver registration, 26
driver template, 34
dynamic configuration changes, 124

E

earnings-based scheduler, 10
effective set of capabilities, 126
ei driver, 179
enabling REACT, 110
enhanced capabilities, 127
/etc/bootcpuset.conf.rtbak, 110
/etc/bootcpuset.conf, 110
/etc/dynamic_cpuset.conf, 110
/etc/elilo.conf.rtbak, 110
/etc/elilo.conf, 110
/etc/init.d, 111
/etc/menu.lst.rtbak, 110
events recorded by tracedaemon, 142
example work flow, 159
exception types, 84
external interrupt ingest, 38
external interrupt setup and registration, 157
external interrupt with frame scheduler, 73
external interrupts, 19
extint, 20, 104
extint_device, 26
extint_properties, 26

F

fastcall, 13
feedback loop, 2
filter tracedaemon events based on CPU, 148
first-in-first-out, 10
flush duration, 48
fork(), 81
FPGA, 157
frame interval, 2
frame rate, 2
frame scheduler, 12, 57, 59
  advantages, 12
  API, 62
  background discipline, 75
  basics, 58
  continuable discipline, 76
  current frame extension, 85
  design process, 78
  exception counts, 87
  exception handling, 84
  exception policies, 86
  exception types, 84
  external interrupt, 73
  frame scheduler controller, 61, 69
  frs_run flag, 67
  frs_yield flag, 67
  high-resolution timer, 73
  interval timers not used with, 92
  library interface for C programs, 64
  major frame, 60
  managing activity threads, 71
  minor frame, 60
  multiple exceptions, 86

multiple synchronized, 69
overrun exception, 74, 84
overrunnable discipline, 76
pausing, 71
preparing the system, 79
process outline for single, 80
real-time discipline, 74
repeat frame, 85
scheduling disciplines, 74
scheduling rules of, 67
sequence error handling, 91
signal use under, 89
signals in an activity thread, 90
signals produced by, 89, 90
starting up a single scheduler, 69
starting up multiple schedulers, 70
synchronized schedulers, 81
thread structure, 66
time base selection, 59, 72
underrun exception, 74, 84
underrunable discipline, 76
using consecutive minor frames, 77
warming up cache, 66
frame scheduler controller, 61, 69
receives signals, 90
frs
See "frame scheduler", 57
frs_create(), 64, 80
frs_create_master(), 64, 80, 82
frs_create_slave(), 64, 83
frs_create_vmaster(), 64, 80, 82
frs_destroy(), 65, 81, 83
frs_enqueue(), 64, 70, 81
frs_fsched_info_t, 62
frs_getattr(), 65, 87
frs_getqueuelen(), 65, 71
frs_intr_info_t, 62, 63
frs_join, 64
frs_join(), , 66, 70, 81, 83
frs_overrun_info_t(), 87
frs_pinsert(), 64, 71
frs_premove(), 65, 71, 90
frs_pthread_enqueue(), 64, 67, 74, 81–83
frs_pthread_getattr(), 65, 87
frs_pthread_insert, 64
frs_pthread_insert(), 71
frs_pthread_readqueue(), 65, 71
frs_pthread_register(), 65
frs_pthread_remove(), 65, 71, 90
frs_pthread_setattr(), 64, 86
example code, 86, 87
frs_queue_info_t, 62
frs_readqueue(), 65, 71
frs_recv_info_t, 62
frs_resume(), 65, 71
frs_run, 68
frs_setattr(), 64, 86
frs_start, 65
frs_start(), 70, 81–83
frs_stop, 65
frs_stop(), 71
frs_t, 62
frs_userintr(), 65
frs_yield, , 65, 66, 68
with overrunable discipline, 76
fsync, 94

G

generating a REACT system configuration, 109
global variables and ULI, 103
ground vehicle simulator, 3

H

hard real-time program, 1
hardware latency, 53, 54
hardware–in–the–loop simulator, 4
high output modes, 36
high-priority tasklet execution event, 144
high-resolution POSIX timers, 14

high-resolution timer, 73, 167
Hz (hertz, cycles per second), 2

I

I/O interrupts, 11
I/O-bound, 9
IDE driver, 35
illegal operation event, 144
implementation functions, 27
ingest section for external interrupts, 38
inheritable set of capabilities, 126
interchassis communication, 17
internal driver circuit I/O connectors, 39
interprocessor interrupt (IPI), 113, 118
interrupt
  group. See interrupt group, 72
  See also user-level interrupt (ULI), 99
interrupt control, 11
interrupt events, 144
interrupt group, 72
interrupt information template, 63
interrupt notification interface, 31
interrupt propagation delay, 54
interrupt redirection, 48
interrupt response time
  components, 53
  definition of, 52
  minimizing, 56
interrupt return event, 144
interrupt service routines (ISRs), 55, 99
interval
  See "frame interval", 2
interval time counter, 14
interval timer, 92
introduction, 1
IOC4 chip, 19
IOC4 driver, 34
ioc4_extint, 104
IOC4–specific character special device and class, 34
ioctl, 179
isolating a CPU, 50
ITC, 14
ITC register access, 14

K

kbar, 43
kernel barrier facility, 43
kernel critical section, 54
kernel facilities for real-time, 9
kernel module insertion/removal, 52
kernel scheduling, 41
kernel thread control, 46
kernel thread creating and binding, 157

L

L2 system controller, 49
latency, 53, 54
libc reentrant version, 103
libcpuset, 124
libsgirt, 124
libuli, 100
Linux capabilities and SGI ProPack captools, 126
Linux Security Module (LSM), 128
Linux Trace, 137
Linux Trace Toolkit, 138
load balancing, 50
locking memory, 104
locking virtual memory, 10
low output modes, 36
low-level driver interface, 26
low-level driver template, 34
lspci, 95

M

MAC address sample program, 175

major frame, 60
master scheduler, 82
maximum response time guarantee, 52
mechanism for callout, 31
memory locking, 157
memory locking (virtual), 10
memory-mapped I/O, 93
mems, 110
Message-Passing Interface (MPI), 17
minor frame, 60, 67
mlock(), 10, 102
mlockall(), 10, 102
mode attribute file, 21
mode switch, 53, 55
modelist attribute file, 21
monitoring trace events, 146
MPI, 17
ms (milliseconds), 2
msync, 93, 94
multiple devices and ULI, 103
multiple independent drivers, 34
multiprocessor architecture
  and frame scheduler, 69

N

netlink socket use, 157
new pthreads library (NPTL), 46
nice value, 9
normal-time program, 1
NPTL, 46

O

operator, 2
output modes, 36
overhead work, 47
overrun, 5
overrun exception, 68
overrun in frame scheduler, 74

P

page fault, 10
page fault event, 144
param.h, 42
PCI devices and programmed I/O, 95
peak data rate, 5
period attribute file, 21
permitted set of capabilities, 126
physical interfaces, 38
physical memory requirements, 10
policy compiler script, 132
policy.pl, 128
POSIX capabilities, 126
POSIX real-time policies, 10
POSIX real-time specification 1003.1-2003, 94
POSIX timers, 14
power plant simulator, 3
priorities, 41
priority band, 42
problem removing /boot, 155
problem removing /rtcpus, 154
/proc manipulation, 11
/proc/cpuinfo, 153
/proc/interrupts, 48, 113, 118, 151
/proc/irq/interrupt/smp_affinity, 123
/proc/sys/kernel/cap_over/, 128
process control, 5
process details in tracevisualizer, 148
process mapping to CPU, 10
process running on restricted CPU, 123
profile.pl, 151
programmed I/O and PCI devices, 95
programming language for REACT, 6
propagation delay, 54
provider attribute file, 21
ps, 43, 151
pthread priority, 46
pthread_attr_setinheritsched(), 46
pthread_attr_setschedparam(), 46
pthread_attr_setschedpolicy(), 46

pthread_attr_t, 46
pthread_attr_t(), 62
pthread_create(), 46, 80
PTHREAD_EXPLICIT_SCHED, 46
PTHREAD_INHERIT_SCHED, 46
pthread_setschedparam(), 46
pthread_t, 62
pulse output modes, 36

Q

quantum attribute file, 21

R

rate
  See "frame rate", 2
raw trace, 148
react-init.sh, 111
react-permissions.pl, 125
reactconf.pl, 47, 109, 110, 157
real-time applications, 1
real-time clock (RTC), 12
real-time priority band, 42
real-time program, 1
  and frame scheduler, 12
reentrant C library, 103
register access, 14
register format, 37
registration of callout, 32
repeat frame, 85
response time, 52
response time guarantee, 52
restricted CPU and running a process, 123
restricting a CPU, 50
round-robin, 10
RTC, 12
RTC access, 14
rtcpu, 58
rtcpu devices, 110

S

SAL console driver, 49
sample system configuration scripts, 109
sched_setparam(), 42
sched_setscheduler(), 10, 42
scheduler events, 143, 145
scheduling, 41
scheduling discipline
  See also frame scheduler scheduling disciplines, 74
scheduling disciplines, 9
scheduling policy, 157
scripts, 109
SGI high-resolution POSIX timers, 14
SGI Linux Trace, 137
SGI ProPack captools package, 127
sgi_react, 111
sgiioc4 driver, 35
sig_dequeue, 90
sig_overrun, 90
sig_underrun, 90
sig_unframesched, 90
signal, 89
signal handler, 82, 83
SIGRTMIN, 90
SIGUSR1, 90
SIGUSR2, 90
simulator, 2
single frame scheduler start, 69
slave scheduler, 82
SLT, 137
slt-cpu.all, 142
slt-cpu.N, 142
SLT-DataFiles, 137
SN hub device interrupts, 56
socket programming, 17
soft real-time program, 1
soft-IRQ execution event, 144
software latency, 53, 54
source attribute file, 22

sourcelist attribute file, 22
special scheduling disciplines, 9
stimulator, 2
strace, 43, 151
strobe output modes, 36
sub-buffer size, 141
sync group, 82
synchronous I/O, 94
/sys/class/extint/extint#/ , 20
/sys/class/ioc4_intout/intout#/dev, 36
sysconf(_SC_CLK_TCK), 13
sysfs attribute files, 20, 179
system abstraction layer (SAL) console driver, 49
system call event, 145
system call exit, 145
system configuration generation, 109
system configuration scripts, 109
system controllers, 49
system fault event, 144
system flush duration, 48
system-call time base, 59

T

tasklet action event, 144
tasklet hiaction, 144
test_and_clear_bit, 109
test_and_set_bit, 109
thread, 61
thread control, 46
thread creation, destruction, and signals, 157
time base for frame scheduler, 72
time base support, 59
time difference, 147
time estimation, 68
time frame and display in tracevisualizer, 147
time slices, 42
time-share applications, 10
timer interrupts, 11, 42
timer_create, 13
timer_create(), 16
timer_settime(), 16
timer_t, 16
timers, 14
toggle output modes, 36
top, 151
trace events, 142
tracedaemon, 137
tracevisualizer, 137
transport delay, 2
troubleshooting, 151

U

ULI
  See "User-level interrupt (ULI) facility", 99
uli, 100
ULI_block_intr, 101
ULI_destroy, 100
ULI_register_irq(), 100, 106
ULI_sleep(), 101
ULI_unblock_intr, 101
ULI_wakeup(), 101
unaligned access event, 145
underrun exception, 68
underrun, in frame scheduler, 74
unsupported hardware device capabilities, 33
usecs (microseconds), 52
user access, 125
user application communication, 157
user thread control, 46
user thread dispatch, 55
user-level interrupt (ULI), 99
  concurrency, 103
  global variables, 103
  handler interaction, 106
  initializing, 103
  interrupt handler registration, 105
  multiple devices, 103
  mutual exclusion, 107
  overview, 99

per-IRQ handler, 106
program address space locking, 104
restrictions on handler, 101
ULI_block_intr(), 107
ULI_sleep (), 106
ULI_sleep () function, 103
ULI_wakeup () function, 106
user-level interrupts (ULI), 173
/usr/include/asm/bitops.h, 109
/usr/include/asm/param.h, 42
/usr/include/sn/timer.h, 14
/usr/include/sys/pthread.h, 62
/usr/sbin/policy.pl, 128, 132
/usr/sbin/reactconf.pl, 109
/usr/share/captools/examples/example.conf, 130
/usr/share/src/react/examples, 63

V

virtual memory locking, 10
virtual reality simulator, 3
volatile keyword, 103
Vsync time base, 59

W

wave stimulator, 4

Z

zooming in tracevisualizer, 147