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

007–4746–001

COPYRIGHT

© 2005 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

The software described in this document is "commercial computer software" provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics, SGI, the SGI logo, and Altix, are registered trademarks and NUMAlink, SGI ProPack, and REACT are trademarks of Silicon Graphics, Inc., in the United States and/or other countries worldwide.

Linux is a registered trademark of Linus Torvalds, used with permission by Silicon Graphics, Inc. SUSE LINUX and the SUSE logo are registered trademarks of Novell, Inc. All other trademarks mentioned herein are the property of their respective owners.

Record of Revision

Version	Description
001	February 2005 Original publication

Contents

About This Guide . . . . **ix**
Audience ix
What This Guide Contains ix
Related Publications and Sites x
Conventions xi
Obtaining Publications xi
Reader Comments xii

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

2. How Linux and REACT Support Real–Time Programs **7**
Kernel Facilities 7

Special Scheduling Disciplines 7
Virtual Memory Locking 8
Processes Mapping and CPUs 8
Interrupt Distribution Control 9
Clocks 10
ITC and ITM Registers 10
SHub Real-Time Clock (RTC) 11
Interchassis Communication 11
Socket Programming 12
Message-Passing Interface (MPI) 12

3. Controlling CPU Workload 13

Using Priorities and Scheduling Queues 13
Scheduling Concepts 13
Timer Interrupts 14
Real-Time Priority Band 14
Controlling Kernel and User Threads 15
Minimizing Overhead Work 15
Avoid the Clock Processor (CPU 0) 16
Reduce the System Flush Duration 16
Redirect Interrupts 16
Select the Console Node for SAL Console Driver Interrupt 17
Restrict and Isolate CPUs 17
Restricting a CPU from Scheduled Work 18
Isolating a CPU from Scheduler Load Balancing 19
Understanding Interrupt Response Time 19
Maximum Response Time Guarantee 20

Components of Interrupt Response Time 20
Hardware Latency 21
Software Latency 21
Kernel Critical Sections 22
Interrupt Threads Dispatch 22
Device Service Time 23
Interrupt Service Routines 23
User Threads Dispatch 23
Mode Switch 23
Minimizing Interrupt Response Time 23

4. Optimizing Disk I/O 25
Memory-Mapped I/O 25
Asynchronous I/O 25
Conventional Synchronous I/O 26
Asynchronous I/O Basics 26

5. PCI Devices 27

6. Installation Overview 29

7. Generating a REACT System Configuration 31
Procedure 32
Example `reactcfg.pl` Output 34
Example `reactboot.pl` Output 36

8. Troubleshooting 37

Appendix A. Example Application 39
Building and Loading a Kernel Module 40

Makefile 41
rt_samplemod.h 42
rt_samplemod.c 42
ipi.c 46
Building and Loading a User-Space Application 48
Makefile 48
rt_sample.c 49
common.h 54
common.c 54
mmtimer.c 57
Running the Sample Application 59

Appendix B. Reading MAC Addresses Example Program 63

Glossary 67

Index 73

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", describes the important classes of real-time programs, emphasizing their performance requirements
- Chapter 2, "How Linux and REACT Support Real–Time Programs", provides an overview of the real-time support for programs in Linux and REACT
- Chapter 3, "Controlling CPU Workload", describes how you can isolate a CPU and dedicate almost all of its cycles to your program's use
- Chapter 4, "Optimizing Disk I/O", describes how to set up disk I/O to meet real-time constraints, including the use of asynchronous I/O
- Chapter 5, "PCI Devices" on page 27, discusses the Linux PCI interface
- Chapter 6, "Installation Overview", lists the RPMs required to run REACT

- Chapter 7, "Generating a REACT System Configuration", discusses the `reactcfg.pl` and `reactboot.pl` configuration scripts that are provided to assist you in setting up REACT
- Chapter 8, "Troubleshooting" on page 37, discusses diagnostic tools that apply to real-time applications
- Appendix A, "Example Application" on page 39, provides excerpts of application modules to be used with REACT
- Appendix B, "Reading MAC Addresses Example Program" on page 63, provides a sample program for reading the MAC address from an ethernet card

Related Publications and Sites

The following books may be useful:

- Available from the online SGI Technical Publications Library:
  - *Linux Configuration and Operations Guide*
  - *Linux Device Driver Programmer's Guide*
  - *SGI Altix 3000 User's Guide*
  - *SGI Altix 350 System User's Guide*
  - *SGI L1 and L2 Controller Software User's Guide*
  - *SGI ProPack 4 for Linux 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*, 2nd Edition, Alessandro Rubini and Jonathan Corbet, O'Reilly, c June 2001, ISBN: 0-596-00008-1

For more information about the SGI Altix series, see the following sites:

- 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 release notes on your system by accessing the `README` file(s) for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.

- 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

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.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:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

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

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 incorrect and unusable if it fails to meet its performance requirements and therefore falls out of step with the external device. A *soft real-time program* is one that can tolerate occasional periods of slow response time but as long as the answer is accurate and is delivered within a given threshold of time, then the computation is successful.

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:

- A SUSE LINUX Enterprise Server 9, Service Pack 1 (SLES9 SP1) kernel built by SGI with several added enhancements to enable low-latency interrupt response
- System configuration scripts `reactcfg.pl` and `reactboot.pl` to help you easily generate a real-time system
- IRIX to Linux REACT compatibility library

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

How Linux and REACT Support Real–Time Programs

This chapter provides an overview of the real-time support for programs in Linux:

- "Kernel Facilities" on page 7
- "Clocks" on page 10
- "Interchassis Communication" on page 11

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 7
- "Virtual Memory Locking" on page 8
- "Processes Mapping and CPUs" on page 8
- "Interrupt Distribution Control" on page 9

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 the earnings-based scheduler is effective at scheduling time-share 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. Tasks scheduled using the POSIX real-time policies are not subject to “earnings” controls. For more information about scheduling, see "Real-Time Priority Band" on page 14 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 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 16.

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:

```
# cat /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.

Clocks

A real-time program sometimes needs a way to create a high-precision time stamp. For time stamps with a precision of less than 1 microsecond, you can use one of the following:

- On SGI systems running Linux, the `mmtimer` driver provides a nonstandard, nonportable, memory-mapped interface to the unadjusted, system-wide real-time clock (RTC). For an example, see "`rt_sample.c`" on page 49 or the `/usr/include/sn/mmtimer.h` file (which is installed with the SGI `mmtimer-devel` RPM).
- The POSIX `clock_gettime()` function with the `CLOCK_REALTIME` value provides a standard interface to the memory-mapped RTC. Although `CLOCK_REALTIME` only reports a resolution of approximately 1 millisecond, the resolution is much higher and is equivalent to that provided by the `mmtimer` driver. Function call overhead has also been all but eliminated for `clock_gettime()` with the `CLOCK_REALTIME` value on SGI systems running Linux. Unlike direct RTC access, however, `CLOCK_REALTIME` is fine-tuned based on clock synchronization to an outside time source via the network time protocol (NTP) or other means. For more information, see the `clock_gettime`(3) man page.
- The *interval time counter (ITC) register* provides a free-running, 64–bit counter that counts up to a fixed relationship to the processor clock. For more information, see "ITC and ITM Registers".

ITC and ITM Registers

SGI systems for Linux have two application registers of importance for time keeping:

- The *interval time counter (ITC) register* provides a 64–bit counter that is scaled from the CPU frequency and is intended to allow an accounting for CPU cycles
- The *interval timer match (ITM) register* allows the generation of an interval timer when a certain ITC value has been reached.

ITCs in an SGI system for Linux are not necessarily synchronized. These systems can contain processors with different frequencies. The ITCs may therefore also run at different frequencies, meaning that the values obtained from the ITC registers cannot be easily related to one another. Even if some processors run at the same speed, they may not have the same clock source and therefore their ITC values may experience drift relative to one another. Systems are then said to have *unsynchronized drifty ITCs*.

This configuration is discoverable through the system abstraction layer (SAL), which provides a standardized way to determine hardware characteristics. To detect this situation, set the `/proc/sal/itc_drift` value to `1` in the `proc` filesystem.

SGI systems for Linux always use the ITC and ITM to generate the timer tick which in turn moves the system clock forward. Tick processing is only performed using CPU 0 and therefore regular time processing is not affected by the differences in ITC values that may exist between CPUs.

For more information, see the information about the interval timer counter in the *Linux Device Driver Programmer's Guide*.

SHub Real-Time Clock (RTC)

Scalable hub (SHub) ASICs are core components to interconnect multiple CPUs in SGI systems for Linux. They have the capability to maintain a synchronized time clock called a *global clock* or the *real-time clock (RTC)*, which is 55 bits wide. At 20Mhz, the RTC will take 57 years to wrap around to zero and provides a resolution of around 40 nanoseconds.

A special clock pin connects all SHubs in a system. One node is selected to be the source for the global clock and all other systems then synchronize to that clock source. Elaborate mechanisms have been designed to ensure that synchronization is maintained even if the timing signal becomes distorted or silent for a time period.

The RTC is guaranteed to be a monotonic time source.

Interchassis Communication

SGI systems support standard network interfaces that let you send packets or streams of data over a local network or the Internet.

This section discusses the following:

- "Socket Programming" on page 12
- "Message-Passing Interface (MPI)" on page 12

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 for Linux.

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.

Controlling 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 13
- "Minimizing Overhead Work" on page 15
- "Understanding Interrupt Response Time" on page 19

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 13
- "Controlling Kernel and User Threads" on page 15

Scheduling Concepts

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

- "Timer Interrupts" on page 14
- "Real-Time Priority Band" on page 14

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 1 millisecond (ms) in every CPU. The frequency of this interval is defined in the `/usr/include/asm/param.h` file. 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 simply 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.

Priority 99, the highest real-time priority, should not be used by applications. This priority should be reserved for any high-priority kernel threads.

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

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 for Linux 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 can use the `reactcfg.pl` configuration script to simplify this procedure; see Chapter 7, "Generating a REACT System Configuration" on page 31.

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 16
- "Reduce the System Flush Duration" on page 16
- "Redirect Interrupts" on page 16
- "Select the Console Node for SAL Console Driver Interrupt" on page 17
- "Restrict and Isolate CPUs" on page 17

Note: The steps are independent of each other, but each needs to 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 4 in Chapter 7, "Generating a REACT System Configuration" on page 31.

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 `/proc/irq/`*interrupt_number*`/smp_affinity` file shows a bitmask of the CPUs that are allowed to receive a given interrupt. By writing a bitmask to this file, you can indicate which CPU is allowed to receive that 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:

```
# echo 2 > /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 `reactboot.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 7, "Generating a REACT System Configuration" on page 31.

Select the Console Node for SAL Console Driver Interrupt

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 and Isolate 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
- Isolated

Note: The term *isolated*, when referring to a CPU in Linux, means to remove the CPU from load balancing considerations, a time-consuming scheduler operation. It does not refer to the IRIX definition for isolated CPUs.

You can use the `reactcfg.pl` configuration script to perform the step required to restrict or isolate a CPU. For more information, see Chapter 7, "Generating a REACT System Configuration" on page 31.

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, do the following (or use the `reactcfg.pl` configuration script documented in Chapter 7, "Generating a REACT System Configuration" on page 31):

1. Configure a bootcpuset as described in "Controlling Kernel and User Threads" on page 15 and reboot the system.
2. Do one of the following:
   - Make a cpuset that encompasses just the CPU that you want to restrict. See the `cpuset`(1) man page for more information.
   - Run the SGI-provided `sysmp(MP_RESTRICT,cpu)` call from program control found in `libsgirt`. For more information, see the `libsgirt`(3) man page.

To remove the restriction, allowing the CPUs to execute any scheduled process, you must reboot the system without the configured bootcpuset.

After restricting a CPU, you can assign processes to it using the SGI `cpuset` command.

For example, to run a program on the `cpuset` named `rtcpu3` (which was set up to include only CPU 3), do the following:

```
# cpuset --invoke=/rtcpu3 -I ~rt/bin/rtapp
```

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) and libsgirt(3) man pages.

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, you must specify them at system boot time. Do the following (or use the reactcfg.pl configuration script documented in Chapter 7, "Generating a REACT System Configuration" on page 31):

1. Include the following string in the append argument for the kernel you are booting, or in the /etc/elilo.conf file:

   ```
   isolcpus=cpu,..
   ```

   For example, suppose you have the following existing append argument:

```
append = "selinux=0 console=ttyS0,115200n8 init=/sbin/bootcpuset"
```

   To isolate CPUs 2 and 3, change the append argument to the following:

```
append = "selinux=0 console=ttyS0,115200n8 init=/sbin/bootcpuset isolcpus=2,3"
```

2. If you edited the /etc/elilo.conf file in step 1, run the elilo command to place a copy of the updated elilo.conf file in the appropriate directory:

   ```
   # elilo
   ```

3. Reboot the system. After the reboot completes, CPUs 2 and 3 will be isolated.

Normally, you would also want to restrict the isolated CPUs. See "Restricting a CPU from Scheduled Work" on page 18.

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. Linux guarantees a maximum interrupt response time on certain systems, but you must configure the system properly 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 3-1 diagrams the parts discussed in the following sections.

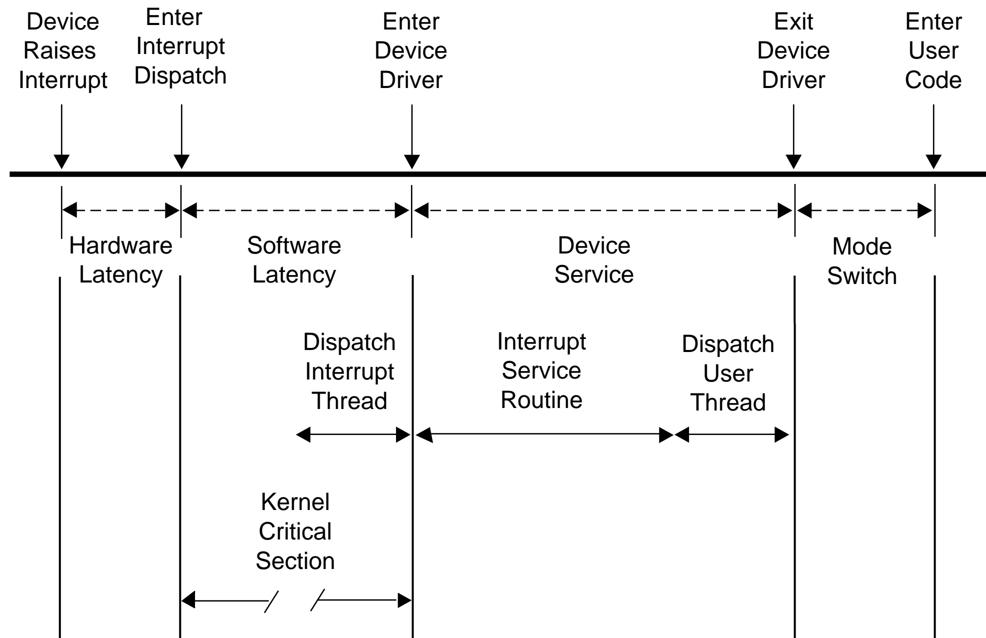

Figure 3-1 Components of Interrupt Response Time

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 5, "PCI Devices" on page 27

Software Latency

Software latency is affected by the following:

- "Kernel Critical Sections" on page 22
- "Interrupt Threads Dispatch" on page 22

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 18 and "Isolating a CPU from Scheduler Load Balancing" on page 19.

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 16.

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.

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

Optimizing Disk I/O

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 25
- "Asynchronous I/O" on page 25

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 simply 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 26
- "Asynchronous I/O Basics" on page 26

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

The Linux `pci` interface provides a mechanism to access the PCI bus address spaces from user programs.

Performing programmed I/O on `pci` devices on SGI Linux systems involves the following steps:

1. Examine `/proc/bus/pci/devices` and obtain offsets for the base address registers (BARs) of the devices that you want to map. The format of the lines is as follows: (with field widths in the number of hexadecimal characters shown, and line breaks added for readability):

bus:2x (slotnumber<<3_|_fn):2x vend:8x bar0 bar1 bar2 bar3 . . .

For example:

```
if ((fptr = fopen( "/proc/bus/pci/devices", "r")) == NULL) {
                printf( "Unable to open /proc/bus/pci/devices\n" );
        }

        while(fgets(buf, sizeof(buf) - 1, fptr)) {
                sscanf( buf, "%2x%2x %8x %*x %lx %*lx %*lx %*lx %*lx %*lx %*lx %*lx
                          %*lx %*lx %*lx %*lx %*lx %*lx",
                &sbus, &sdevfn, &svend, &sbar[0], &sbar[1], &sbar[2], &sbar[3],
                &sbar[4], &sbar[5], &sbar[6], &sbar[7]);
                if(( sbus == bus ) && (sdevfn == devfn)) {
                        /* This is the bus, slot and function we're looking for,
                         * so save the base address register offset information.  */
                        for(int i=0; i> 16;
                        device = svend & 0xffff;
                }
        }
        fclose( fptr );
```

2. Open the appropriate device file for the bus, slot, and function in which you are interested. The device files are named as follows:

   `/proc/bus/pci/`*bus*`/`*slot*`.`*function*

For example:

```
memfile = (char*) malloc( 32 );
        sprintf( memfile, "/proc/bus/pci/%02d/%02d.%d", bus, slot, function );
        fd = open( memfile, O_RDWR );
```

3. Set the memory map state for the file to `MEM` space using the `PCIIOC_MMAP_IS_MEM` request to the `ioctl()` system call.

   For example:

   ```
   ioctl(fd, PCIIOC_MMAP_IS_MEM);
   ```

4. Map the opened file, using the offset obtained in step 1 as the *offset* parameter.

   For example:

   ```
   tmpPtr = (char *) mmap( NULL, (size_t) len, PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, (off_t) offset[bar]);
   ```

For a complete example, see Appendix B, "Reading MAC Addresses Example Program" on page 63.

Installation Overview

To install REACT, do the following:

1. Install SUSE LINUX Enterprise Server 9, Service Pack 1 (SLES9 SP1) as outlined in the *SGI ProPack 4 for Linux Start Here*.

   Note: Do not install a kernel other than the default Linux kernel.

   ---

2. Install the appropriate SGI ProPack 4 for Linux RPMs for your site, including at least the following:

   `cpuset-*.rpm`

   The following RPM is optional but recommended:

   `mmtimer-devel`

   Note: Do not install a kernel and kernel module RPMs other than the defaults. In step 3 below, the REACT kernel and modules will be installed over the defaults.

   For more information, see *SGI ProPack 4 for Linux Start Here*.

3. Install the following RPMs from the REACT CD:

   - REACT kernel (required):

     `kernel-rt-*.rpm`

   - REACT sample system configuration scripts (required to run `reactcfg.pl` and `reactboot.pl`):

     `react-configuration-*.rpm`

   - REACT kernel source (required to build `rt_sample_mod` discussed in Appendix A, "Example Application" on page 39):

     `kernel-rt-source-*.rpm`

   - IRIX to Linux REACT compatibility library:

     `libsgirt-*.rpm`

- REACT documentation:

  ```
  react-docs-*.rpm
  ```

- SLES9 SP1 modules built for the REACT kernel:

  ```
  km-rt-arsess-*.rpm
  km-rt-csa-*.rpm
  km-rt-job-*rpm
  km-rt-numatools-*.rpm
  km-rt-xpmem-*.rpm
  km-rt-xvm-standalone-*.rpm
  ```

For more information, see the following file on the REACT CD:

```
/usr/share/doc/sgi-react-1.0/README.relnotes
```

Generating a REACT System Configuration

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

This procedure uses the following configuration scripts to assist in generating a simple REACT configuration:

- `reactcfg.pl` is a Perl script that edits the `/etc/bootcpuset.conf` and `/etc/elilo.conf` scripts. "Example `reactcfg.pl` Output" on page 34 shows the script workflow.

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

- `reactboot.pl` is a Perl script that sets the cpusets and redirects interrupts. See "Example `reactboot.pl` Output" on page 36.

These scripts as released make assumptions about how your system is configured. You should use them in their default state as an aid to learning the configuration procedure. No configuration changes are made to your system until you confirm the action to overwrite the `/etc/elilo.conf` file, at which point both `/etc/bootcpuset.conf` and `/etc/elilo.conf` will be overwritten.

After you understand the overall procedure and the REACT system configuration, you can modify the scripts to meet your specific needs and generate your final production configuration. If you only have a single system to configure, you may prefer to edit the `/etc/elilo.conf` and `/etc/bootcpuset.conf` files directly rather than modify the `reactcfg.pl` script. However, if you have multiple systems to modify, it may be more efficient to modify the script and then reuse it on the other systems. The `reactboot.pl` script is meant to be run after every reboot; therefore, you should modify the script as necessary or write a new script.

Note: This procedure assumes that, at a minimum, you have installed the following RPMs from the SGI ProPack for Linux and REACT CDs:

```
cpuset-*.rpm
kernel-rt-*.rpm
react-configuration-*.rpm
```

You must have at least these RPMs installed before beginning this procedure.

Procedure

Do the following:

1. Log in as `root`.
2. 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 `timer` and `IPI`). You cannot use CPU 0 for real time. Although the `reactboot.pl` script attempts to redirect all interrupts to CPUs that are not used for real time, certain interrupts cannot be moved, including the console interrupt.

   Note: Interrupt affinity changes are temporary and are not retained across system reboots. SGI recommends that you run a set-up script equivalent to `reactboot.pl` after every reboot to set up your real-time cpusets and set the affinity of device interrupts.

   ---

3. Run the `reactcfg.pl` script, supplying a comma-separated list of individual real-time CPUs, a range of real-time CPUs, or a mixture of both:

   `reactcfg.pl` *real-time-CPU-list*

   The script will set up the appropriate `/etc/bootcpuset.conf` and `/etc/elilo.conf` files based on the CPUs listed and the number of CPUs on your system. As an example, on an 8–processor system, the following command

line will set up CPUs 0, 1, 4, 5, and 6 in the bootcpuset, with CPUs 2, 3 and 7 restricted and isolated for real-time use:

```
# reactcfg.pl 2-3,7
```

It will also create a new section labeled `rt` in your `/etc/elilo.conf` file and make this the default boot section for your system.

4. Reduce the Altix system flush duration:

   a. Reboot the machine but interrupt the boot process when the **EFI Boot Manager** menu comes up and 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"
      ```

      Note: Use this procedure only for systems running time-critical real-time applications. (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.

      ---

   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. All but a few CPU-specific threads will be running within the bootcpuset. The real-time CPUs will be isolated from the effects of load balancing.

5. When the system comes back up, run the `reactboot.pl` script to determine how to do the run-time set up of your system:

   ```
   # reactboot.pl
   ```

The reactboot.pl script does the following:

- Creates cpusets labeled rtcpu*N* for each CPU that is not part of the bootcpuset. 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 reactcfg.pl.
- Redirects interrupts to the CPUs contained in the bootcpuset. This is just a working approximation; someone familiar with your site's hardware configuration and system requirements should set up the interrupt redirection. To set up the interrupt redirection, echo the correct values to the /proc/irq/*interrupt*/smp_affinity cpumasks. For more information, see "Redirect Interrupts" on page 16.

SGI recommends that you run a set-up script equivalent to reactboot.pl after every reboot to set up your real-time cpusets and set the affinity of device interrupts. You can modify reactboot.pl to configure interrupt affinity and cpusets according to your own needs.

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/rtcpu*N* cpusets created above).

For more information, see the cpuset(1) and libcpuset(3) man pages and the set_affinity_cpuset() function in the rt_sample application in Appendix A, "Example Application" on page 39.

Example reactcfg.pl Output

Following is an example of the output generated by reactcfg.pl (line breaks added here for readability). The script modifies the bootcpusets based on the CPUs you provide, adds the rt label, and changes the append values. (The script output also contains information currently contained in the /etc/elilo.conf file, such as comments.)

```
# reactcfg.pl 2-3
realtime cpus   2 3
bootcpuset cpus 0 1
bootcpuset mems 0


/etc/bootcpuset.conf
```

```
cpus 0,1
mems 0

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


/etc/elilo.conf:
# This file has been transformed by /sbin/elilo.
# Please do NOT edit here -- edit /etc/elilo.conf instead!
# Otherwise your changes will be lost e.g. during kernel-update.
#
# Modified by YaST2. Last modification on Fri Nov  5 16:32:33 2004


prompt
timeout = 80
read-only
relocatable
default = rt
append = "selinux=0 console=ttyS0,115200n8 kdb=on splash=silent elevator=cfq
  thash_entries=2097152"

image = vmlinuz
    ###Don't change this comment - YaST2 identifier: Original name: linux###
    label = Linux
    initrd = initrd
    root = /dev/sdb6

image = vmlinuz
    ###Don't change this comment - YaST2 identifier: Original name: failsafe###
    label = Failsafe
    initrd = initrd
    root = /dev/sdb6
    append = "ide=nodma nohalt noresume selinux=0 barrier=off"

image = vmlinuz
    label = rt
    initrd = initrd
    root = /dev/sdb6
append = "selinux=0 console=ttyS0,115200n8 kdb=on splash=silent elevator=cfq
  thash_entries=2097152 init=/sbin/bootcpuset isolcpus=2,3"
```

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


Backup files are /etc/elilo.conf.rtbak and /etc/bootcpuset.conf.rtbak
Please reboot your system to restrict and isolate cpus
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.
```

Example `reactboot.pl` Output

Following is an example of the output generated by `reactboot.pl`:

```
# reactboot.pl
CPUSET /dev/cpuset/rtcpu2 created
CPUSET /dev/cpuset/rtcpu3 created
```

Troubleshooting

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 "Running the Sample Application" on page 59.

- 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 "Running the Sample Application" on page 59. 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:

```
$ 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):

```
$ 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 timestamps 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.

Example Application

This appendix provides excerpts from a sample real-time application to be used with REACT. The application is composed of two different parts:

- *Example kernel module*, which shows a simple example of creating and building a driver with a standard miscellaneous device interface. It is provided primarily to service the example user-space application.

  The example kernel module provides two modes of operation:

  - A thread can wait for an interrupt to be sent to the CPU on which it last ran
  - A thread can send an interrupt to a specific CPU

  Upon receiving an interrupt, the waiting process will simply read the system real-time clock (RTC) and return that value to user-space.

- *Example user-space application*, which shows examples of the following concepts:

  - Creating cpusets and assigning threads to them, thereby changing thread/CPU affinity
  - Changing thread/CPU affinity without cpusets
  - Changing scheduling policies and priorities
  - Reading the system RTC via the memory-mapped `mmtimer` driver
  - Locking memory

  This program runs as a dual process:

  - A *sending process* (`IPI_send`) that does the following:

    1. Sets its CPU affinity either via cpusets or via the `sched_setaffinity()` call. For more information, see the `sched_setaffinity`(2) man page.
    2. Sets its scheduling policy and priority.
    3. Uses the example kernel module driver to repeatedly send interrupts to a given destination CPU.

- A *receiving process* (main) that does the following:
    1. Locks its current and future memory (if requested).
    2. Sets its CPU affinity via cpusets.
    3. Sets its scheduling policy and priority.
    4. Uses the example kernel module driver to do the following:
        - Wait for interrupts
        - Retrieve kernel RTC readings
        - Read the memory-mapped RTC for comparison readings

Note: Header information has been truncated from the examples for readability.

This example application requires that the SGI ProPack for Linux mmtimer-devel RPM is installed on the system.

Building and Loading a Kernel Module

To build the rt_sample_mod application kernel module, do the following on the target system:

1. Log in to the target system as root.
2. Set up a kernel module development environment:
    a. Ensure that the kernel-rt-source-*.rpm RPM is installed.
    b. Ensure that the kernel-rt kernel is in use.
    c. Change to the source directory:

    ```
    # cd /lib/modules/`uname -r`/source
    ```

    d. Create the configuration file:

    ```
    # make cloneconfig
    ```

    e. Build the kernel:

    ```
    # make compressed
    ```

3. Make the rt_sample_mod directory:

   ```
   # mkdir /root/rt_sample_mod
   ```

4. Change to the rt_sample_mod directory:

   ```
   # cd /root/rt_sample_mod
   ```

5. Copy the rt_mod_sample source files to the target system. For example:

   ```
   # scp bob@server:~/rt_sample/rt_sample_mod/* .
   ```

6. Build the rt_sample_mod.ko file:

   ```
   # make -C /lib/modules/‘uname -r‘/source SUBDIRS=$PWD modules
   ```

7. Copy the rt_sample_mod.ko file to the ‘uname -r‘ directory:

   ```
   # cp rt_sample_mod.ko /lib/modules/‘uname -r‘
   ```

   For more information, see the uname(1) man page.

8. Make a dependency file:

   ```
   # depmod
   ```

   For more information, see the depmod(8) man page.

9. Load the rt_sample_mod module:

   ```
   # modprobe rt_sample_mod
   ```

   For more information, see the modprobe(8) man page.

You can then use the rt_sample_mod kernel module with the rt_sample application.

Makefile

Following is a portion of the kernel-space Makefile:

```
CFLAGS += -D__KERNEL__ -DMODULE -g

obj-m += rt_sample_mod.o
rt_sample_mod-objs := rt_samplemod.o ipi.o
```

rt_samplemod.h

Following is a portion of the `rt_samplemod.h` file:

```
#define MAXCPUS 64

/* We might store more information here if we were measuring response
 * times at different points.
 */
typedef struct rttask_struct {
        unsigned long long rtctime;      /* Time waiting while thread is woken */
        struct semaphore sysrt_sema;     /* Mutex to wait on */
        int ready;                       /* Waiting for interprocessor interrupt (IPI)*/
} rttask_t;

extern int rt_register_percpu_irq(void);
extern void rt_send_IPI_single (int);
extern void rt_free_percpu_irq(void);

extern rttask_t * rttasks[MAXCPUS];

#define IA64_IPI_RT     0xfc    /* Interprocessor interrupt for real-time test */
```

rt_samplemod.c

Following is a portion of the `rt_samplemod.c` file:

```
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/sn/addrs.h>
#include <asm/sn/shub_mmr.h>
#include <asm/sn/clksupport.h>

#include "rt_samplemod.h"

MODULE_LICENSE("GPL");

#define RTMOD_NAME      "rt_sample_mod"

static ssize_t rt_samplemod_run(struct file * file, const char * buf, size_t count,
```

```
                loff_t *f_pos);
/*
 * Avoid the big kernel lock (BKL) by making rt_samplemod_run() a 'write' call rather
 *  than an ioctl(), to avoid introducing latency.
 */
static struct file_operations rt_samplemod_fops = {
        owner:  THIS_MODULE,
        write:  rt_samplemod_run,
};
static struct miscdevice rt_samplemod_miscdev = {
        152,
        RTMOD_NAME,
        &rt_samplemod_fops
};

#define RT_INTR_WAIT    1
#define RT_SEND_IPI     2

rttask_t * rttasks[MAXCPUS];

int rt_samplemod_init(void) {

        memset(rttasks, 0, sizeof(rttasks));

        /* Register this device */
        strcpy(rt_samplemod_miscdev.devfs_name, RTMOD_NAME);
        if (misc_register(&rt_samplemod_miscdev)) {
                printk(KERN_ERR "%s: failed to register device\n", RTMOD_NAME);
                return -1;
        }
        /* Set up the interrupt request */
        rt_register_percpu_irq();

        printk(KERN_INFO "%s has been initialized\n",RTMOD_NAME);

        return 0;
}

void rt_samplemod_exit(void) {
        int i;
```

```
        rt_free_percpu_irq();
        for (i=0; i<MAXCPUS; i++) {
                if (rttasks[i] != NULL) kfree(rttasks[i]);
        }
        misc_deregister(&rt_samplemod_miscdev);

        printk(KERN_INFO "%s exit\n",RTMOD_NAME);
}

/*
 * Avoid the BKL by making rt_samplemod_run() a
 * 'write' call rather than an ioctl(), to avoid introducing latency.
 * Pass everything in as 'buf', ignoring everything else.
 */
static ssize_t
rt_samplemod_run(struct file * file, const char * buf, size_t count,
                loff_t *f_pos)
{
        int error = 0;
        unsigned long long cmd;
        unsigned long long arg1;
        unsigned long long * argp = (unsigned long long *)(buf+sizeof(cmd));

        if (copy_from_user(&cmd, (void *)buf, sizeof(cmd))) {
                return -EFAULT;
        }
        if (__get_user(arg1, argp)) {
                return -EFAULT;
        }

        switch (cmd) {
                case RT_INTR_WAIT:
                        {
                        rttask_t * curr;
                        unsigned long cpu = smp_processor_id();


                        /* First time here on this CPU */
                        if (rttasks[cpu] == NULL) {
                                /* Allocate and initialize the per-cpu data area */
                                curr = kmalloc(sizeof(rttask_t),GFP_KERNEL);
```

```
                        if (curr == NULL) {
                                error = -ENOMEM;
                                break;
                        }
                        curr->ready = 0;
                        init_MUTEX_LOCKED(&curr->sysrt_sema);
                        rttasks[cpu] = curr;
                } else {
                        curr = rttasks[cpu];
                }
                /* The send will not happen until 'ready' is set and
                 * the semaphore is down, so there is no race.
                 */
                curr->ready = 1;
                down_interruptible(&curr->sysrt_sema);

                /* This is how we read the RTC time in the kernel.
                 * Note that we are only measuring the time to go from
                 * kernel space to user space.  An actual interrupt response
                 * measurement would take RTC time readings at other
                 * points within this driver.
                 */
                curr->rtctime = rtc_time();
                argp = (unsigned long long *)(buf+sizeof(cmd));
                /* Copy RTC reading to user space */
                error = __put_user(curr->rtctime, argp);
                }
                break;
        case RT_SEND_IPI:
                {
                int cpuid = arg1; /* Destination cpu */
                rttask_t * tsk = rttasks[cpuid];

                if ((tsk == NULL) || (!tsk->ready) ||
                        (tsk->sysrt_sema.count.counter != -1)) {
                        error = -ENOMEM;
                        break;
                }
                /*
                 * Only single threaded here per CPU, so no race with
                 * above.
```

```
                         */
                        tsk->ready = 0;
                        rt_send_IPI_single(cpuid);
                        }
                        break;
                default:
                        error = -EINVAL;
                        break;
        }
        return error;
}

module_init(rt_samplemod_init);
module_exit(rt_samplemod_exit);
```

ipi.c

Following is a portion of the `ipi.c` file:

```
#include <linux/irq.h>
#include <asm/sn/clksupport.h>
#include <asm/sn/intr.h>
#include <asm/hw_irq.h>
#include "rt_samplemod.h"


/*
 * This is the actual interrupt handler.  We could get an RTC reading
 * here if we were measuring response times
 */
static irqreturn_t
rt_handle_IPI (int irq, void *dev_id, struct pt_regs *regs)
{
        int this_cpu = get_cpu(); /* Disable preemption and get CPU number */
        rttask_t * tsk = rttasks[this_cpu];

        mb();

        up(&tsk->sysrt_sema); /* Raise the semaphore */
```

```
        put_cpu(); /* Re-enable preemption */

        return IRQ_HANDLED;
}


static inline void
rt_send_IPI(int cpuid, int vector, int delivery_mode)
{
        long    physid;

        physid = cpu_physical_id(cpuid);

        sn_send_IPI_phys(physid, vector, delivery_mode);
}

/*
 * Replaces send_IPI_single, but no per-CPU operation bit is set.
 * It also disables preemption and interrupt requests itself, and calls
 * rt_send_IPI rather than the nonexported platform_send_ipi
 * (which becomes sn2_send_IPI).
 * Caller must ensure the existence of the rttasks element.
 */
void
rt_send_IPI_single (int dest_cpu)
{
        unsigned long s;

        get_cpu(); /* Prevent preemption. */
        /* For performance, make sure we are not interrupted until the IPI is sent */
        local_irq_save(s);
        rt_send_IPI(dest_cpu, IA64_IPI_RT, IA64_IPI_DM_INT);
        local_irq_restore(s);
        put_cpu();
}

/* Register our interrupt request */
int
rt_register_percpu_irq(void)
{
        /* With SA_PERCPU_IRQ, this becomes similar to register_percpu_irq */
```

```
        return request_irq(IA64_IPI_RT, rt_handle_IPI, SA_PERCPU_IRQ,
                                "Realtime IPI", NULL);
}

/* Free our interrupt request */
void
rt_free_percpu_irq(void)
{
        free_irq(IA64_IPI_RT,NULL);
}
```

Building and Loading a User-Space Application

To build and load the user-space module, enter the following:

```
[user@linux user]$ make
```

Makefile

Following is a portion of the user-space `Makefile`:

```
TARGETS=rt_sample
COMFILES=common.c mmtimer.c
COMOBJECTS=$(COMFILES)
SFILES=rt_sample.c
SOBJECTS=$(SFILES)

default: $(TARGETS)

rt_sample: $(SOBJECTS) $(COMOBJECTS) -lcpuset
```

`rt_sample.c`

Following is a portion of the `rt_sample.c` file:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <signal.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sn/mmtimer.h>
#include "common.h"

#define SLEEPTIME       3000            /* usecs to sleep between interrupts */

extern unsigned long * open_mmtimer(void);


/* name of the device, usually in /dev */
#define RTMOD_NAME "rt_sample_mod"
#define DEFAULT_RUN_TIME        60              /* Run for 60 seconds default */
#define RT_INTR_WAIT     1
#define RT_SEND_IPI      2

typedef signed short    cpuid_t;        /* cpuid */
typedef struct parm_s {
        unsigned long cmd;
        unsigned long arg1;
} parm_t;


cpuid_t cpu = 2;
cpuid_t ipi_cpu = 3;
int memlock = 0;

/* These externs get set in mmtimer.c */
```

```
extern unsigned long period;
extern unsigned long mask;
int rt_fd;


char * usage =
"usage: rt_sample [-chm] [-p rcv_processor] [-o snd_processor] [-t seconds]\n"
"       -c  IPI src processor is in cpuset\n"
"       -h  print usage instructions\n"
"       -m  lock memory\n"
"       -p# receiving processor\n"
"       -o# IPI src processor\n"
"       -t# total run time (secs)\n";


/*
 * Print out the command line options and quit.
 */
static void
usage_exit(void)
{
        fprintf(stderr, "%s", usage);
        exit(0);
}

/* This routine sends IPIs to the receiving CPU. */
void
IPI_send(int cpuset)
{
        parm_t rti;
        struct timespec req;

        if (cpuset)
                set_affinity_cpuset(ipi_cpu);
        else
                set_affinity(ipi_cpu);

        set_scheduling(10, SCHED_RR);
        sleep(5);  /* Give other IPI_send processes a chance to start if
                        they are on the same CPU */
```

```
        req.tv_sec = 0; req.tv_nsec = SLEEPTIME * 1000;
        rti.cmd = RT_SEND_IPI;
        rti.arg1 = cpu;
        while (1) {
                /* Send interrupt */
                if (write(rt_fd, &rti, 0) != 0) {
                        /* printf("Send not successful\n"); */
                        continue;
                }
                nanosleep(&req, NULL);
        }
}

int
main(int argc, char **argv)
{
        unsigned long max_delta = 0;
        unsigned long min_delta = 0x7fffffffffffff;
        unsigned long total_time = DEFAULT_RUN_TIME; /* sec */
        unsigned long start;
        unsigned long run_time;
        unsigned long * ptimer;
        int cid = 0;    /* default to no cpusets for IPI source CPU */
        pid_t child;
        int opt;

        static char* opt_string = "cmp:o:t:h";
        while ((opt = getopt(argc, argv, opt_string)) >= 0) {
                switch (opt){
                case 'c':
                        cid = 1;
                        break;
                case 'm':
                        memlock = 1;
                        break;
                case 'p':
                        cpu = atoi(optarg);
                        break;
                case 'o':
                        ipi_cpu = atoi(optarg);
                        break;
```

```
                case 't':
                        total_time = atoi(optarg);
                        break;
                case 'h':
                default:
                        usage_exit();
                }
        }
        /* Open rt_sample_mod sample driver */
        if ((rt_fd = open("/dev/"RTMOD_NAME,O_RDWR)) == -1) {
                printf("Failed to open /dev/%s\n",RTMOD_NAME);
                exit(1);
        }
        /* Open mmtimer driver and get memory mapped RTC address */
        if (!(ptimer = open_mmtimer())) {
                exit(1);
        }

        /* Fork off the IPI sender */
        child = fork();
        if (child==0) {
                IPI_send(cid);
                exit(0);
        }

        /* Do not allow paging if selected */
        if (memlock &&
            (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)) {
                perror("memlock");
                exit(1);
        }

        /* Create and attach to a single CPU cpuset for this CPU. */
        set_affinity_cpuset(cpu);

        /* Change the scheduling to priority and policy. */
        set_scheduling(30, SCHED_FIFO);

        /* Read the RTC via mmtimer */
        start = *ptimer;
```

```
/* Loop for total_time receiving IPIs from IPI_send */
do {
        time_t ct;
        struct timespec thrwake_ts, current_ts;
        unsigned long current;
        parm_t rti;

        unsigned long delta;

        rti.cmd = RT_INTR_WAIT;
        rti.arg1 = 0;

        /* Wait to be signaled that an interrupt has arrived */
        if (write(rt_fd, &rti, 0) != 0) {
                perror("write to rt_fd failed");
                continue;
        }

        /* The rest of this loop represents whatever processing
         * must be done on the data from the driver.
         *
         * This simple example just takes the RTC time difference
         * between the RTC value measured in the kernel and the
         * current RTC time.
         */

        /* Read the RTC via mmtimer */
        current = *ptimer;

        /* delta is in nsec */
        delta = ((current & mask) - (rti.arg1 & mask)) * period;
        if (delta > max_delta) {
                max_delta = delta;
        }
        if (delta < min_delta) {
                min_delta = delta;
        }
        /* elapsed time so far is in sec */
        run_time = (((current & mask) - (start & mask)) * period) / 1e9;
} while (run_time < total_time);
```

```
        if (kill(child, SIGTERM)) {
                perror("Unable to kill child process\n");
        }
        wait(child, NULL, 0);
        printf("Minimum delta %ld nsec\n",min_delta);
        printf("Maximum delta %ld nsec\n",max_delta);
        return 0;
}
```

common.h

Following is a portion of the `common.h` file:

```
extern void set_affinity(int);
extern void set_affinity_cpuset(int);
extern void set_scheduling(int, int);
```

common.c

Following is a portion of the `common.c` file:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sched.h>
#include <bitmask.h>
#include <cpuset.h>
/* NR_CPUS is size kernel is configured for */
#define NR_CPUS 512
#define BITMASK_SIZE NR_CPUS/64

/* Set the current process to run on CPU <cpu>.
 * Note that this CPU must not be part of a cpuset other than the
 * one the current process is in.
 */
void set_affinity(int cpu) {
        unsigned long cpus[BITMASK_SIZE];

        if (cpu > (NR_CPUS-1)) {
                printf("set_affinity: Invalid cpu %d\n",cpu);
```

```
                exit(1);
        }
        cpus[cpu/64] = 1 << (cpu % 64);

        if (sched_setaffinity(0, sizeof(cpus), cpus)) {
                perror("set_affinity");
                exit(1);
        }
}

/*
 * Set up a cpuset with cpus==<cpu> and mems==<cpu>/2.
 * Then set the current process to run on CPU <cpu> in cpuset 'rtcpu<cpu>'
 * For more information about mems, see the cpuset(1) man page.
 */
void set_affinity_cpuset(int cpu) {
        char path[50];
        struct cpuset *cp;
        struct bitmask *cpus;
        struct bitmask *mems;

        sprintf(path, "/rtcpu%d", cpu);
        cp = cpuset_alloc();
        if (cp == NULL) {
                printf("cpuset_alloc failed\n");
                exit(1);
        }
        cpus = bitmask_alloc(cpuset_cpus_nbits());
        mems = bitmask_alloc(cpuset_mems_nbits());

        /* Set up bitmasks for cpus and mems */
        bitmask_setbit(cpus, cpu);
        bitmask_setbit(mems, cpu/2);

        /* Set the 'cpus' value in the cpuset structure */
        if (cpuset_setcpus(cp, cpus) == -1) {
                perror("cpuset_setcpus");
                exit(1);
        }
        /* Set the 'mems' value in the cpuset structure */
        if (cpuset_setmems(cp, mems) == -1) {
```

```
                perror("cpuset_setmems");
                exit(1);
        }
        /* Create the actual cpuset in /dev/cpuset */
        if (cpuset_create(path, cp) == -1) {
                if (errno != EEXIST) {
                        perror("cpuset_create");
                        exit(1);
                }
        }

        cpuset_free(cp);
        bitmask_free(cpus);
        bitmask_free(mems);

        /* Move the process into the cpuset */
        if (cpuset_move(getpid(), path) == -1) {
                perror("cpuset_move");
                exit(1);
        }

}

/* Change the scheduling policy and priority <pri>
   A priority of -1 passed in selects the maximum priority.
 */
void set_scheduling(int pri, int policy) {
        struct sched_param param;
        int maxpri;

        if (policy != SCHED_FIFO && policy != SCHED_RR) {
                printf("Bad policy %d set!\n", policy);
                exit(1);
        }
        /* Get the maximum priority value for this policy */
        maxpri = sched_get_priority_max(policy);

        if (pri==-1)
                param.sched_priority = maxpri;
        else
                param.sched_priority = pri > maxpri ? maxpri : pri;
```

```
        /* Change the priority and policy if pri != 0 */
        if ((pri==0) || (sched_setscheduler(0, policy, &param)==-1)) {
                perror("Unable to set sched policy");
        } else {
                printf("Sched policy set to %d, pri=%d\n",
                        policy, param.sched_priority);
        }
}
```

mmtimer.c

Following is a portion of the mmtimer.c file:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <cntl.h>
#include <n/mmtimer.h>
#include "common.h"


unsigned long period;
unsigned long mask = 0xffffffffffffffff;

/*
 * Open RTC interface and get parameters.
 */
unsigned long * open_mmtimer(void) {
        int fd, result, offset;
        unsigned long * iotimer_addr64;
        unsigned long val = 0;
```

```
if((fd = open("/dev/"MMTIMER_NAME, O_RDONLY)) == -1) {
        printf("failed to open /dev/%s", MMTIMER_NAME);
        return NULL;
}

/*
 * Can we mmap in the counter?
 */
if ((result = ioctl(fd, MMTIMER_MMAPAVAIL, 0)) != 1) {
        printf("mmap unavailable\n");
        return NULL;
}

/*
 * Get the offset for each clock
 */
if ((offset = ioctl(fd, MMTIMER_GETOFFSET, 0)) == -ENOSYS) {
        printf("offset unavailable for clock\n");
        return NULL;
}

/*
 * Get the frequency in Hz and calculate the period in nsec
 */
if((result = ioctl(fd, MMTIMER_GETFREQ, &val)) != -ENOSYS)
        if(val < 10000000) /* less than 10 MHz? */
                printf("ERROR: frequency only %ld MHz, should be >= 10 MHz\n", val/1000000);
        else {
                printf("frequency: %ld MHz\n", val/1000000);
        }
else
        printf("ERROR: failed to get frequency\n");

/*
 * Get the resolution in femtoseconds (1e-15) and save period in nsec.
 */
if((result = ioctl(fd, MMTIMER_GETRES, &val)) == -ENOSYS) {
        printf("ERROR: failed to get resolution\n");
        return NULL;
}
period = (val/1e+6);
```

```
        printf("period: %lld nsec\n",period);

        /*
         * Get the number of valid bits and compute the mask.
         */
        if((result = ioctl(fd, MMTIMER_GETBITS, 0)) == -ENOSYS) {
                printf("ERROR: can't get number of bits in counter\n");
                return NULL;
        }
        mask = ~(0xffffffffffffffff << result);

        /*
         * Map the clock.
         */
        iotimer_addr64 = (unsigned long *)mmap(0, 0x4000, PROT_READ,
                                                MAP_PRIVATE, fd, 0);
        if (iotimer_addr64 <= 0) {
                printf("failed to mmap /dev/%s",MMTIMER_NAME);
                return NULL;
        }
        iotimer_addr64 += offset;
        close(fd);

        return iotimer_addr64;
}
```

Running the Sample Application

To run the rt_sample user-space application, do the following:

1. Ensure that you have the rt_sample_mod module loaded by using the lsmod(1) command, which should show it in the module list:

   ```
   # lsmod
           Module                  Size  Used by
           rt_sample_mod          72784  0
           ..
   ```

   If the output does not include rt_sample_mod, follow the instructions in "Building and Loading a Kernel Module" on page 40.

2. Execute the `rt_sample` command as desired.

The `rt_sample` command has the following options:

`-c`	Specifies that the CPU sending the interrupt must be in the cpuset. By default, the CPU does not have to be in a cpuset.
`-h`	Prints the usage instructions.
`-m`	Locks memory. By default, memory is not locked.
`-p` *receiving_CPU*	Specifies the number of the CPU that is receiving; this CPU must be in a cpuset. The default is CPU 2.
`-o` *sending_CPU*	Specifies the CPU sending the interrupt. The default is CPU 3.
`-t` *seconds*	Specifies the total run time in seconds. The default is 60 seconds.

Therefore, to run `rt_sample` for 60 seconds (the default) with CPU 3 as the sender in a cpuset and CPU 2 as the receiver (which always must be in a cpuset), enter the following:

```
rt_sample -c -p 2 -o 3 -m
```

You can monitor the IPI interrupts produced and received by examining the `/proc/interrupts` file. Doing so for the above example should result in output similar to the following:

```
# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
 28:         10         10         10         10          LSAPIC  cpe_poll
..
252:          0          0       6839          0          LSAPIC  Realtime IPI
..
```

To see where the threads are running, use the `-F` option to the `ps` command (the seventh column, `PSR`, shows the last processor on which the thread ran). For example:

```
# ps -FC rt_sample
UID        PID  PPID  C    SZ  RSS PSR STIME TTY          TIME CMD
root      6407  6151  0   185 2880   2 15:04 pts/0    00:00:00 rt_sample -c -p 2 -o 3 -m
root      6408  6407  0   185 1248   3 15:04 pts/0    00:00:00 rt_sample -c -p 2 -o 3 -m
                                     ^
                                     ^
```

displays the last CPU on which the thread ran

Reading MAC Addresses Example Program

This appendix provides a sample program for reading the MAC address from an ethernet card on an SGI system for Linux (line breaks added for readability). It demonstrates how to memory map and interact with hardware devices from user space.

```
/* 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 <unistd.h>
#include <fcntl.h>
#include <linux/pci.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <errno.h>


int main( int argc, char **argv )
{
  char path[128];
  char buf[1024];
  int fd;
  FILE *fptr;
  unsigned int bus, device, function, devfn;
  unsigned int sbus=0, sdevfn=0, svend;
  unsigned long bar, sbar =0;
  char *ptr;
  unsigned int *data;

  if( argc != 4 )
  {
    printf( "Must supply bus slot and function\n" );
    exit( 1 );
  }
```

```
bus = atoi( argv[1] );
device = atoi( argv[2] );
function = atoi( argv[3] );
devfn = PCI_DEVFN( device, function );

errno = 0;
if( (fptr = fopen( "/proc/bus/pci/devices", "r" )) == NULL )
{
  perror( "Unable to open /proc/bus/pci/devices" );
  exit( 1 );
}

while( fgets( buf, sizeof(buf) - 1, fptr ) )
{
 sscanf( buf, "%2x%2x %8x %*x %lx %*lx %*lx %*lx %*lx %*lx %*lx %*lx
        %*lx %*lx %*lx %*lx %*lx %*lx",
        &sbus, &sdevfn, &svend, &sbar);
  if( ( sbus == bus ) && (sdevfn == devfn) )
  {
    bar = sbar & ~0xf;
    printf( "Found matching bar %lx\n", bar );
  }
  sbar = 0;
}
fclose( fptr );
if( !bar ) exit(1); /* did not find */

sprintf( path, "/proc/bus/pci/%02d/%02d.%d", bus, device, function );
printf( "path is :%s\n", path );

fd = open( path, O_RDWR );
if( fd == -1 ) perror( "Couldn't open device file" );

ioctl( fd, PCIIOC_MMAP_IS_MEM );
ptr = mmap( NULL, getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED, fd, (off_t) bar);
if( ptr == MAP_FAILED )
{
  perror( "mmap failed!" );
  exit( 1 );
}
```

```
  data = (unsigned int*) (ptr + 0x414);
  printf( "ptr is %p, data is %p\n", ptr, data );
  printf( "MAC is %08x\n", *data );
}
```

Glossary

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*.

control law processor

A type of stimulator provides the effects of laws of physics to a machine.

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*.

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.

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 interval

The inverse of *frame rate*, that is, 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.

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 is incorrect and unusable if it fails to meet its performance requirements, and so falls out of step with the external device.

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 propagation delay

See *hardware latency*.

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*.

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.

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*.

mode switch

The time it takes for a thread to switch from kernel mode to user mode.

overrun

When incoming data arrives faster than a data collection system can accept it and therefore data is lost.

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.

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.

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.

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*.

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.

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.

soft real-time program

A program that can tolerate occasional periods during which response time may be an order of magnitude longer than the average time. If the answer is accurate and is delivered within a threshold of time, then the computation is successful.

software latency

The time required to dispatch an interrupt thread.

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.

Index

A

aircraft simulator, 3
append argument, 19
asynchronous I/O, 25
average data rate, 5

B

bootcpuset, 15, 33
bootcpuset.conf, 33

C

C language, 6
clock pin, 11
clock processor, 16
clocks, 10
compatibility library (IRIX to Linux) RPM, 29
configuration, 31
configuration script RPMs, 29
configuration scripts, 31
console driver, 17
console interrupts, 9
console node, 17
control law process stimulator, 4
CPU
  isolating, 19
  restricting, 8, 18
  workload control, 13
CPU 0, 16, 17
CPU-bound, 7
cpumasks, 34
cpuset, 15, 34
cycles per second, 2

D

data collection system, 4
device service time, 20, 23
disciplines, 7
disk I/O optimization, 25
distributed applications, 12
documentation RPM, 30
drifty ITCs, 11

E

earnings-based scheduler, 7
elilo, 19
/etc/elilo.conf, 19, 33

F

feedback loop, 2
first-in-first-out, 7
flush duration, 16
frame interval, 2
frame rate, 2
fsync, 26

G

generating a REACT system configuration, 31
ground vehicle simulator, 3

H

hard real-time program, 1

hardware latency, 20, 21
hardware–in–the–loop simulator, 4
Hz (hertz, cycles per second), 2

I

I/O interrupts, 9
I/O-bound, 7
include file, 10
installation, 29
interchassis communication, 11
interrupt control, 9
interrupt propagation delay, 21
interrupt redirection, 16
interrupt response time
  components, 20
  definition of, 19
  minimizing, 23
interval
  See "frame interval", 2
interval time counter (ITC), 10
interval timer match (ITM), 10
introduction, 1
IRIX compatibility library, 29
isolating a CPU, 19
isolcpus, 19

K

kernel critical section, 22
kernel facilities for real-time, 7
kernel RPM, 29
kernel scheduling, 13
kernel thread control, 15

L

L2 system controller, 17
latency, 20, 21
libcpuset, 34
libsgirt, 18
load balancing, 19
locking virtual memory, 8

M

maximum response time guarantee, 20
memory locking (virtual), 8
memory-mapped I/O, 25
Message-Passing Interface (MPI), 12
mlock(), 8
mlockall(), 8
mmtimer, 10
mmtimer_devel, 10
mode switch, 20, 23
monotonic time source, 11
MPI, 12
ms (milliseconds), 2
msync, 25, 26

N

nice value, 7
normal-time program, 1

O

operator, 2
overhead work, 15
overrun, 5

P

page fault, 8
param.h, 14
peak data rate, 5

physical memory requirements, 8
POSIX real-time policies, 7
POSIX real-time specification 1003.1-2003, 26
power plant simulator, 3
priorities, 13
priority 99, 14
priority band, 14
/proc manipulation, 9
/proc/interrupts, 17, 32
/proc/sal/itc_drift, 11
process control, 5
process mapping to CPU, 8
programming language for REACT, 6
propagation delay, 21
ps, 15

R

rate
  See "frame rate", 2
reactboot.pl, 17
reactcfg.pl, 15–19, 31
README.relnotes, 30
real-time applications, 1
real-time clock (RTC), 11
real-time priority band, 14
real-time program, 1
release notes, 30
response time, 19
response time guarantee, 20
restricting a CPU, 18
round-robin, 8
RPMs, 29

S

SAL console driver, 17
sample system configuration script RPMs, 29
sample system configuration scripts, 31
sched_setparam(), 14
sched_setscheduler(), 8, 14
scheduling, 13
scheduling disciplines, 7
script RPMs, 29
scripts, 31
SHub, 11
simulator, 2
SLES9, 29
SLES9 SP1 modules, 30
SN hub device interrupts, 23
socket programming, 12
soft real-time program, 1
software latency, 20, 21
special scheduling disciplines, 7
stimulator, 2
strace, 15, 37
SUSE LINUX, 29
system abstraction layer (SAL) console driver, 17
system configuration generation, 31
system configuration script RPMs, 29
system configuration scripts, 31
system controllers, 17
system flush duration, 16

T

thread control, 15
time slices, 14
time stamp creation, 10
time-share applications, 7
timer interrupts, 9, 14
transport delay, 2
troubleshooting, 37

U

unsynchronized drifty ITCs, 11
usecs (microseconds), 20
user thread control, 15

user thread dispatch, 23
/usr/include/asm/param.h, 14

V

virtual memory locking, 8
virtual reality simulator, 3

W

wave stimulator, 4