REACT™ Real-Time Programmer's Guide

Document Number 007-2499-001

CONTRIBUTORS

Written by David Cortesi
Illustrated by Gloria Ackley
Edited by Christina Cary
Production by Gloria Ackley
Engineering contributions by Rich Altmaier, Jeffrey Heller, Ralph Humphries, and Luis Stevens
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

REACT™ Real-Time Programmer's Guide
Document Number 007-2499-001

Contents

List of Examples

List of Figures

List of Tables

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 support that IRIX™ and the Silicon Graphics CHALLENGE™, Onyx™, and POWERCHALLENGE computers provide to real-time programs. The support bundled with all versions of IRIX is called REACT™. A set of extra-cost features is called REACT/Pro™. This guide covers both REACT and REACT/Pro.

This guide is designed to be read online, using IRIS InSight™. You are encouraged to read it in non-linear order using all the navigation tools that Insight provides.

Who This Guide Is For

This guide is written for real-time programmers. You, a real-time programmer, are assumed to be

- an expert in the use of your programming language, which must be either C, Ada, or FORTRAN to use the features described here
- 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 UNIX® system programming, although you do need to be familiar with UNIX as an environment for developing software.

What the Book Contains

Here is a summary of what you will find in the following chapters.

Chapter 1, “Real-Time Programs,” describes the important classes of real-time programs, emphasizing the different kinds of performance requirements they have.

Chapter 2, “Basic Features of the CHALLENGE and IRIX™ Architectures,” contains an overview of how IRIX manages the resources of a fully symmetric multiprocessor like the Challenge / Onyx for the benefit of normal, interactive UNIX applications; and points out how these methods often conflict with the needs of real-time programs.

Chapter 3, “How IRIX™ and REACT / Pro™ Support Real-Time Programs,” gives an overview of the real-time features of IRIX. From these overview topics you can jump to the detailed topics that interest you most.

Chapter 4, “Managing Virtual Memory in a Real-Time Program,” covers the management of your virtual address space: locking it to real memory; mapping devices and files into it; and sharing segments of it between processes.

Chapter 5, “Managing Time and Time Intervals,” covers the use of timers and clocks in the Challenge / Onyx architecture.

Chapter 6, “Controlling CPU Workload,” describes how you can isolate a CPU and dedicate almost all of its cycles to your program’s use.

Chapter 7, “Using the Frame Scheduler,” describes the REACT / Pro Frame Scheduler, which gives you a simple, direct way to structure your real-time program as a group of cooperating processes, efficiently scheduled on one or more isolated CPUs.

Chapter 8, “Optimizing Disk I / O for a Real-Time Program,” describes how to set up disk I / O to meet real-time constraints, including the use of asynchronous I / O and guaranteed-rate I / O.

Chapter 9, “Managing Device Interactions,” summarizes the software interfaces to external hardware, including and user-level programming of external interrupts and VME and SCSI devices.

Other Useful Books

The following books contain more information that can be useful to a real-time programmer.

- For a survey of all IRIX facilities and manuals, *Programming on Silicon Graphics Systems: An Overview*. This useful manual, part of the IRIX Developer Option, is new in version 5.3; part number 007-2476-001.
- For administration of a multiprocessing server, including system generation and the use of the XFS file system, *IRIX Advanced Site and Server Administration Guide*. The version that covers the XFS file system is version 5.3, part number 007-0603-100.
- For details of the architecture of the CPU, processor cache, processor bus, and virtual memory, *MIPS R4000 Microprocessor User's Manual* by Joseph Heinrich, Prentice-Hall, 1993 (ISBN 0-13-105925-4) and the *MIPS R10000 Microprocessor User's Manual*, available in 1995.
- For details of many IRIX system facilities not covered in this book, *Topics in IRIX Programming*, part number 007-2478-001 and *MIPS Compiling and Performance Tuning Guide*, 007-2479-001 (both are available with the IRIX Developer's Option).
- For the design and construction of device interrupt handlers and on programming the SCSI interface, *IRIX Device Driver Programming Guide*, part number 007-0911-050; and *IRIX Device Driver Reference Pages*, part number 007-2183-003.
- For programming inter-computer connections using sockets, *IRIX Network Programming Guide*, part number 007-0810-050.
- For coding functions in assembly language, *MIPSpro Assembly Language Programmer's Guide*, part number 007-2418-001.

In addition, Silicon Graphics offers training courses in Real-Time Programming and in Parallel Programming.

Chapter 1

Real-Time Programs

This chapter surveys the categories of real-time programs, and indicates which types can best be supported by REACT and REACT/Pro. As an experienced programmer of real-time applications, you might want to read the chapter to verify that this book uses terminology that you know; or you might want to proceed directly to Chapter 2, “Basic Features of the CHALLENGE and IRIX™ Architectures”.

Defining Real-Time Programs

A real-time program is any program that must maintain a fixed, absolute timing relationship with an external hardware device.

Normal-time programs do not require a fixed timing relationship to external devices. A normal-time program is a correct program when it produces the correct output, no matter how long that takes. 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 not incorrect, merely slow.

A real-time program is one that is incorrect and unusable if it fails to meet its performance requirements, and so falls out of step with the external device.

Major Types of Real-Time Programs

There are three major types of real-time programs: simulators, data collection systems, and process control systems. This section describes each type briefly; simulators and data collection systems are described in more detail in following sections.

- A simulator maintains an internal model of the world. It receives control inputs, updates the model to reflect them, and displays the changed model. It must process inputs in real time in order to maintain an accurate simulation, and it must generate output in real time to keep up with the display hardware.

  Silicon Graphics systems are well suited to programming many kinds of simulators.

- A data collection system receives input from reporting devices, for example telemetry receivers, and stores the data. It may be required to process, reduce, analyze or compress the data before storing it. It must react in real time in order to avoid losing data.

  Silicon Graphics systems are suited to many data collection tasks.

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

  Although Silicon Graphics systems can be used for process control, they are not usually the most economical choice. Dedicated process-control computers are generally better for these uses.

Simulators

All simulators have the same four components,

- An internal model of the world or part of it; for example a model of a vehicle travelling through a model geography, or a model of the physical state of a nuclear power plant.
- An external device through which the state of the model is displayed; for example one or more video displays, audio speakers, or a simulated instrument panel.
- An external devices to supply control inputs; for example a steering wheel, a joystick, or simulated knobs and dials.
- An operator (or hardware under test) that “closes the loop” by viewing the display and moving the controls in response.

Requirements on Simulators

The real-time requirements on a simulator vary depending on the nature of these four components. Two key performance requirements on a simulator are *frame rate* and *transport delay.*

Frame Rate

A crucial measure of simulator performance is the rate at which its display is updated. This rate is called the *frame rate*, whether or not the simulator displays its model on a video screen.

Frame rate is given in cycles per second (abbreviated Hz). Typical frame rates run from 15 Hz to 60 Hz, although both higher and lower rates are used in special situations.

The inverse of frame rate is *frame interval*. For example, a frame rate of 60 Hz allows a frame interval of 1 / 60 second, or 16.67 milliseconds. To maintain a frame rate of 60 Hz, a simulator must update its model and prepare a new display in at most 16.67 ms.

The REACT / Pro Frame Scheduler helps you organize a multi-process application so that it can achieve a specified frame rate. (See Chapter 7, “Using the Frame Scheduler.”)

Transport Delay

Transport delay is the term for the number of frames that elapses before a control motion is reflected in the display. When the transport delay is too long, the operator will perceive the simulation as sluggish or unrealistic. If a visual display is slow to react, a human operator can become physically ill.

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 will be several analogue control inputs or 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.

Silicon Graphics systems with REACT/Pro are well suited to building aircraft simulators.

Ground Vehicle Simulators

Simulators for automobiles, tanks, and heavy equipment have been built with Silicon Graphics 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 an electric power generation plant (nuclear or conventional). Power-plant simulators have been built using Silicon Graphics 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 connect the control inputs and multiplex them onto the VME bus. Also, the number of display outputs (simulated gauges, charts, warning lights, and so on) can be very large and may also require special, 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. (So also does a vehicle simulator. One difference is that a vehicle simulator strives for an exact model of the laws of physics, which a virtual reality simulator typically does not need to do.)

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, and 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.

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

Hardware-in-the-loop (HITL) Simulators

The operator of a simulator need not be a person. In a hardware-in-the-loop simulator, the role of operator is played by another computer, such as an aircraft autopilot or the control and guidance computer of a missile. The simulator's display output is a set of input signals to the computer under test. The simulator's control inputs are the output signals of the computer under test.

Depending on the hardware being exercised, the simulator may have to maintain a very high frame rate, up to 1000 Hz. Silicon Graphics systems can be used for some hardware simulators. Special-purpose systems may be more practical or more economical for very demanding frame rates.

Data Collection Systems

A data collection system has either three or four major parts:

1. Sources of data, for example telemetry. Typically the source or sources are interfaced to the VME bus.
2. A repository for the data. This can be a raw device such as a tape, or it can be a disk file or even a database system.
3. 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.

4. 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.

Requirements on Data Collection Systems

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.

Achieving High Transfer Rates to Devices

The Challenge / Onyx systems support a variety of I / O types with different bandwidth and latency characteristics:

- VME device registers can be mapped directly into the program's address space, where they can be read and written as memory variables. This is implemented as *programmed I/O (PIO)*.

  Memory-mapping makes I / O programming simple, especially when large numbers of devices or complex device protocols are involved. Memory-mapped, programmed I / O can transfer data from 250 KB / second to 1 MB / second. (See "PIO Access" on page 190.)

- When transferring 32 or more consecutive bytes, transfer rate can be increased using *direct memory access (DMA)* to devices on the VME bus. The Challenge / Onyx systems allow DMA access to VME devices that do not normally support DMA, through unique DMA engine (see "Program Access to the VME Bus" on page 189.)

- Maximum transfer rates on the VME bus are achieved with a VME device that supports block mode transfer as a bus master. When the device supports it, Challenge/Onyx systems can achieve VME transfer rates greater than 50 MB/second.
- Multiple SCSI controllers can be attached to all Silicon Graphics systems. SCSI transfer rates can reach 14 MB/second on each channel for 16-bit SCSI-II controllers (see "SCSI Hardware on CHALLENGE and Onyx Systems" on page 182).

Achieving High Transfer Rates to Disk

A data collection system can exploit two features to achieve a high rate of data transfer to disk,

- asynchronous disk I/O
- Guaranteed-rate I/O (GRIO), part of XFS

Asynchronous I/O that conforms to POSIX 1003.1b-1993 is a standard feature of IRIX 5.3. You use asynchronous I/O library calls to initiate disk I/O in a separate process, while your real-time process continues to work with the input data. (In fact you can start asynchronous I/O to any device, not just to disk files.) You can ensure that the asynchronous process performing the I/O executes on a different CPU than the one used by the real-time process.

Using GRIO, your real-time program can claim a specified portion of the bandwidth of a device. I/O requested by other processes is deferred, if necessary, to ensure that your process achieves the promised data rate.

For details of both these features, see Chapter 8, "Optimizing Disk I/O for a Real-Time Program."

Real-Time Programming Languages

The majority of real-time programs are written in C, which is the most common language for system programming on UNIX. All of the examples in this book are in C syntax.

The second most common real-time language is Ada, which is used for many defense-related projects. SGI sells the MP/Ada product. Ada programs can call any function that is available to a C program, so all the facilities described in this book are available, although the syntax may vary slightly. Ada offers additional features that are useful in real-time programming; for example, MP/Ada includes a partial implementation of POSIX threads which are used to implement Ada tasking.

SGI will be supplying a new Ada implementation, and a separate book addressed to real-time programming in Ada, in 1995.

Some real-time programs are written in FORTRAN. A program in FORTRAN can access any IRIX system function, that is, any facility that is specified in volume 2 of the reference pages. For example, all the facilities of the REACT/Pro Frame Scheduler are accessible through the IRIX system function **schedctl()**, and hence can be accessed from a FORTRAN program (see “The Frame Scheduler API” on page 118).

A FORTRAN program cannot directly call C library functions, so any facility that is documented in volume 3 of the reference pages is not directly available in FORTRAN. Thus the mmap() function, a system function, is available (see “The Segment Mapping Function mmap()” on page 54), but the **usinit()** library function, which is basic to SGI semaphores and locks, is not available. However, it is possible to link subroutines in C to FORTRAN programs, so you can write interface subroutines to encapsulate C library functions and make them available to a FORTRAN program.

Chapter 2

Basic Features of the CHALLENGE and IRIX™ Architectures

The architecture of the CHALLENGE, Onyx, and POWERCHALLENGE computers provides multiple CPUs, a large real memory, a high-speed system bus, and fast I/O channels. (For brevity, the phrase Challenge/Onyx is used to refer to these machines as a single type.)

The IRIX operating system normally manages the hardware resources so as to optimize the throughput of a large number of UNIX* applications, both batch and interactive.

This chapter gives a high-level summary of IRIX methods, pointing out how they can sometimes conflict with the needs of a real-time program.

If you already know IRIX and the Challenge/Onyx architecture, you can skip to Chapter 3, “How IRIX™ and REACT/Pro™ Support Real-Time Programs,”, which introduces the features you can use to create fully deterministic system behavior for real-time programs.

Symmetric Multiprocessor Architecture

Figure 2-1 shows a simplified, high-level view of the Challenge / Onyx architecture.

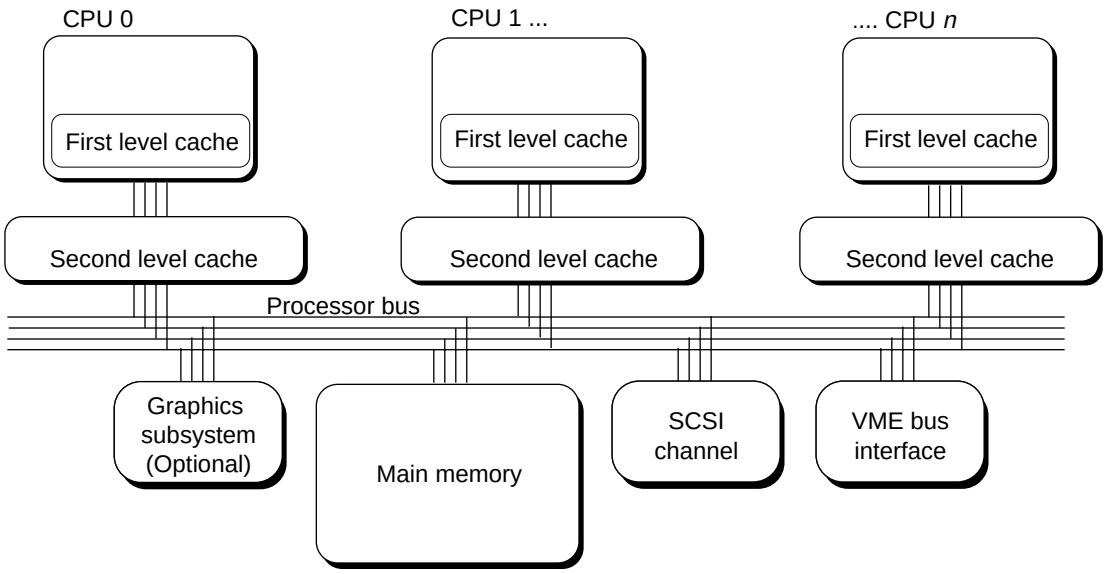

Figure 2-1 Symmetric Multiprocessor Architecture

CPUs, Memory, and the System Bus

A Challenge / Onyx system contains from 2 to as many as 36 CPUs. All are functionally identical. The CPUs are connected to each other and to a single memory by the processor bus. The processor bus carries 128-bit parallel packets at a data rate of 1.2 Gigabytes / second. An important feature of the bus design is that it is "fair," that is, there is a very low probability of any CPU on it starving for access. This helps to make real-time program timings determinate and repeatable.

There is a single physical memory (shown as "main memory" in Figure 2-1) that is accessed equally by all CPUs. For example, there is a single image of the UNIX kernel in memory, and any of the CPUs could be executing instructions from it, in any combination, at any time.

Concurrent Execution

The Challenge / Onyx computers permit true concurrency—two or more CPUs executing the same program at the same instant. However, most ordinary UNIX programs execute in only one CPU at a time.

Two or more CPUs, executing on behalf of different processes, can enter the IRIX kernel simultaneously. The kernel is written to optimize concurrent use. It uses *semaphores* and *locks* to serialize the use of the data structures that can be used by two or more processes at the same time.

A real-time program may need to use two or more CPUs concurrently in order to finish the work it needs to do in each frame interval. You can structure your real-time program as multiple processes. You can force the processes to run on multiple CPUs concurrently, and you can use semaphores and locks to protect the common resources. Process creation is discussed later in this chapter, under “Process Management” on page 14.

Memory Hierarchy

Each CPU in a Challenge / Onyx system accesses memory through a four-level hierarchy:

- First-level instruction and data caches within the CPU chip provide the fastest access to recently-used data (the cache size depends on the microprocessor model).
- A larger second-level cache on each CPU board stores recently-used instructions and data (this cache size depends on the CPU board model).
- Main memory contains the current state of swapped-in processes.
- Swapped-out virtual pages are kept in the swap partition on disk.

There is a ratio of roughly 100:1 in access speeds between each level of this hierarchy. There is a large reward of execution speed for a program that maintains *locality of reference*, and so executes mostly out of cache. This is examined in more detail under “Reducing Cache Misses” on page 68. At the other extreme, there is a large penalty of lost time for any program that causes pages to be swapped in and out of memory.

Cache Coherency Updates

Each CPU has two levels of cache that hold copies of memory data. Multiple copies of the same data can exist in different caches. When a CPU writes to its cache memory, it broadcasts the fact on the processor bus. Other CPUs that have cached the same location invalidate their cached copies, so that if they need to refer to it again, they will reload the modified data.

This is a greatly oversimplified summary of a complicated protocol that ensures consistent, correct behavior of the multiple CPUs, even when they use the same memory areas. (For details on the subject, refer to one of the architecture books listed in “Other Useful Books” on page xxiii.) Cache management is built into the hardware at a low level.

Virtual Memory

In general, each UNIX process has its own *address space*. The process sees the address space as a contiguous 2 gigabytes of memory locations containing the process’s code, data, and other resources.

The composition of the address space, and the methods by which a process can share it with other processes, are covered in Chapter 4, “Managing Virtual Memory in a Real-Time Program.”

The IRIX kernel manages each process’s address space as a set of *pages*. All pages are the same size in one implementation of IRIX. (The page size is 4 KB in 32-bit systems, larger in 64-bit systems. Programs should always determine the page size dynamically by calling the **getpagesize()** function.)

Some or all of the pages that represent a process’s address space may be stored on disk. When the process is executing and attempts to access a page not in memory, it causes a *page fault* interrupt. The kernel suspends the process until it can provide the page contents. If the page has defined contents, the kernel schedules a disk I/O operation to load it. If this is the first use of a stack or heap page, the kernel simply creates a page of zeros. In order to make room for the needed page, the kernel may have to invalidate some other page, and may have to save the contents of the other page to the swap disk.

A page fault causes an unpredictable and possibly lengthy pause in the execution of a process. A real-time program cannot tolerate such delays. However, you can have part or all of your program's address space locked into memory, so that a page fault cannot occur.

Translation Lookaside Buffer Updates

Virtual addresses are mapped to real memory locations using translation tables kept in memory. For speed, each CPU has a cache of recently-used page addresses, called the *translation lookaside buffer (TLB)*.

Under certain conditions, kernel code executing in one CPU can change the address space mapping in a way that could invalidate TLB entries held by other CPUs. In order to synchronize the TLBs, the kernel broadcasts an interrupt to all CPUs. The interrupt service routine in each CPU purges the TLB for that CPU so it will be reloaded with accurate values. Memory accesses immediately after a TLB purge are slow, while the TLB contents are reconstructed. The TLB update interrupt comes at unpredictable times. A real-time program with tight timing constraints cannot tolerate being delayed this way.

However, when you dedicate one or more CPUs to executing your real-time program, you can isolate your dedicated CPUs from TLB interrupts. (For details, see "Isolating a CPU From TLB Interrupts" on page 104).

Device Interrupts

When a device needs attention, it requests an interrupt, which forces a CPU to enter the code of a *device driver* to service the interrupt. The device driver will mask off (block) other interrupts while it works. The kernel also masks interrupts during some critical sections.

Interrupts from the VME bus are grouped into 7 priority levels. Each device on the bus interrupts on a particular level. Higher numbered levels have superior priority (IRQ7 is superior to IRQ1).

By default, interrupts are "sprayed" (dynamically distributed, in rotation) to all CPUs in order to equalize the load of handling interrupts. You can control this in two ways:

- Designate CPUs that are not to receive sprayed interrupts. You would do this to protect real-time processes in those CPUs from being interrupted by devices not related to real-time work.
- Specify that interrupts of a specified VME interrupt level are to be directed to a specified CPU. You would do this either to group all non-real-time interrupts on a designated CPU, or to direct real-time interrupts to a CPU that is dedicated to handling them.

For details on these actions, see "Minimizing Overhead Work" in Chapter 6.

Interrupt Latency

When interrupts come from the real-time input and output devices, you are concerned about *interrupt latency*, the amount of time that elapses between the hardware signal and the start of the IRIX kernel's response to it. Interrupt latency has several sources, some of which you can control. (See "Components of Interrupt Response Time" in Chapter 6.)

Interrupt Response Time

The time that elapses from the arrival of an interrupt until the system returns to executing user code is *interrupt response time*. It includes interrupt latency, plus the time spent in the device driver (called *device service time*), plus the time IRIX needs to switch program contexts, and other factors. When you take full advantage of the features of IRIX and REACT/Pro and configure the system properly, you can guarantee a maximum 200 microsecond interrupt response time. See "Minimizing Interrupt Response Time" in Chapter 6.

Process Management

A *process* is one executable instance of a program. The IRIX kernel creates new processes, and by default it attempts to schedule their shared use of the hardware in a fair and effective way. You can alter the default scheduling to favor a real-time program in several different ways.

Process Composition

A process consists of an address space containing the program text and data, and a number of *process attributes* managed by the IRIX kernel. A few examples of process attributes are

- a unique process ID number
- machine register contents, representing the current instruction and stack level as well as working data
- UNIX user and group identities
- current working directory for file searches
- signal-handling status

For a more complete list, refer to the fork(2) reference page and read the list of attributes that a new process does and does not inherit from its parent.

Process Creation

There are two system calls that create a process. They differ in that one creates a new address space and the other does not.

Normal Process Creation With fork()

The conventional method of creating a new process in UNIX is to issue the **fork()** system call. It creates a "child" process, which is a copy of the "parent" process that issued the call. The address space of the child is a duplicate of the parent's address space, as are most of its attributes, including its machine register contents. Only the return value of **fork()** differs. The use of **fork()** is shown in Example 2-1.

Example 2-1 Schematic of Using fork()

```
int childProcId;
switch(childProcId = fork())
{
case 0:
   { /* this is executed by the child process */ }
   break;
case -1:
   { /* parent process, no child process created */ }
   break;
default:
   { /* parent process, child process exists */ }
}
```

IRIX does not physically duplicate all the pages of the parent's address space. That would waste a great deal of time. Instead, the page translation table that defines the child's address space initially refers to the physical pages of the parent's address space. However, the table designates these pages as "copy on write."

Whenever the child process writes into a page, it causes a hardware trap. The kernel then makes a duplicate of that one page so that the child has a unique copy into which it can write. Thus only the pages that are written are copied, and then only when the child uses them.

Address Space Replacement With exec()

The **exec()** system call is the means by which UNIX "loads a program" (see the exec(2) reference page for details.) This call replaces the entire address space with a new one based on a program image loaded from an executable file. The **exec()** call also initializes many of the process attributes (refer to the exec(2) reference page for details).

The combination of **fork()** and **exec()** suits the needs of a command shell. The way a UNIX command shell launches a program is to **fork()**, creating a new process. In the new process it calls **exec()**, replacing the new address space. In the great majority of **fork()** calls, the child's address space is completely replaced before more than one or two of its pages have been copied.

However, **fork()** is not well-suited to building a program designed as a number of small, cooperating processes—the kind of design that your real-time application needs if it is to exploit multiple CPUs.

Lightweight Process Creation With sproc()

The **sproc()** system call is unique to IRIX. It creates a new process that shares its parent's address space. The new process has its own machine registers and its own memory region for its stack. Otherwise, both processes execute concurrently using the same program text, data, and many process attributes. A parent process and its children by **sproc()** constitute a *process group*.

For several reasons, you should use **sproc()** if you structure your real-time application as multiple, cooperating processes:

- The kernel does much less work to create a process with **sproc()**. For example, it does not have to build a page table to describe a new address space.
- The parent process can initialize disk files, device files, global data structures, memory-mapped I/O, and other objects, and all these are automatically available to the child processes.
- The parent and all child processes have write access to global data, and can use high-performance semaphores and locks to regulate access.
- There is only one address space to lock into memory, no matter how many processes use it.

Process Scheduling

By default, the IRIX kernel manages time-sharing processes under these assumptions:

- There are far more processes (dozens to hundreds) than there are CPUs to execute them.
- The system's resources should be shared among all processes as equitably as possible.
- Most processes spend most of their time waiting for input or output.

- As long as a process makes some progress (is not blocked indefinitely), its exact rate of progress is not crucial (“the system is busy” is always a valid excuse for slow response).

When managing a mix of time-sharing programs, the IRIX kernel attempts to keep all CPUs busy and all processes advancing, and is generally successful at this. (For details, see “Using Priorities and Scheduling Queues” on page 85.)

However, IRIX also supports real-time programs. When a real-time program is running, the assumptions for scheduling must change: There is typically only one real-time program in a system. You are prepared to give it all of the system’s resources if necessary. It spends little time waiting for input. Most important, its precise rate of progress is an integral part of its design.

Your real-time program can give itself a high scheduling priority or, if it cannot tolerate time-sharing at all, it can seize one or more CPUs and dedicate them to its exclusive use. The specific calls used are surveyed in Chapter 3, “How IRIX™ and REACT/Pro™ Support Real-Time Programs” and covered in detail in Chapter 6, “Controlling CPU Workload”.

I/O Scheduling

When a process initiates I/O, IRIX usually suspends the process until data transfer is complete. By understanding the I/O system, and by using the Asynchronous I/O feature, you can make sure that a real-time process is not blocked in this way.

Disk I/O

When a process requests disk input, it is blocked until the data has been read and copied into the designated buffer. When a process requests disk output, it is blocked until the data has been copied into a kernel buffer or until the disk write is complete, depending on the options used when the file was opened.

VME Bus I/O

Your program can perform I/O to the VME bus in three ways: programmed I/O (PIO), direct memory access (DMA) from VME Bus Master devices, and a unique form of DMA from VME Bus Slave devices.

When it uses programmed I/O, your program polls the device registers or memory as if they were variables in memory, and does not block. Your real-time program can do PIO in a time-critical process.

VME-bus I/O using either form of DMA generally does delay the requesting process until the DMA transfer is complete. All of these methods are discussed under "Program Access to the VME Bus" on page 189.

Other I/O

In general, UNIX allows your process to open any device for I/O with the **open()** call. You specify a pathname designating one of the special device files found in the */dev* directory. The **open()** call returns a file descriptor which you can pass to the **read()** or **write()** functions. For device files, these functions are routed directly to the device driver for the device. Through this means your program can read or write serial devices, SCSI devices, and (in SGI systems other than Challenge/Onyx, devices on the GIO or EISA bus.

A call to a device driver for input or output normally blocks the calling process until the data has been transferred.

Asynchronous I/O

Typically, a real-time process cannot allow itself to be blocked for I/O. *Asynchronous I/O* is a feature of IRIX 5.x which gives you the ability to schedule I/O to be done in a separate process. This process—created automatically for you—requests the I/O while your real-time process continues to execute. For details on asynchronous I/O, see Chapter 8, "Optimizing Disk I/O for a Real-Time Program."

Chapter 3

How IRIX™ and REACT/Pro™ Support Real-Time Programs

This chapter provides an overview of the real-time support in IRIX and REACT/Pro. The discussion uses terms that are defined in Chapter 1, “Real-Time Programs” and Chapter 2, “Basic Features of the CHALLENGE and IRIX™ Architectures”.

Some of the features mentioned here are discussed in more detail in the following chapters of this guide. For details on other features you are referred to reference pages or to other manuals. The main topics surveyed are

- “Kernel Facilities for Real-Time Programs,” including special scheduling disciplines, isolated CPUs, and locked memory pages
- “REACT/Pro Frame Scheduler,” which takes care of the details of scheduling multiple processes on multiple CPUs at guaranteed rates
- “Interprocess Communication,” reviewing the ways that a concurrent, multiprocess program can coordinate its work
- “Timers and Clocks,” reviewing your options for time-stamping and interval timing
- “Interchassis Communication,” reviewing two ways of connecting multiple chassis

Kernel Facilities for Real-Time Programs

The IRIX kernel has a number of features that are valuable when you are designing your real-time program.

Kernel Optimizations

The IRIX kernel has been carefully optimized for performance in a multiprocessor environment. Some of the optimizations are as follows:

- Instruction paths to system calls and traps are optimized, including some hand coding, to maximize cache utilization.
- In the real-time dispatch class (described further in “Using Priorities and Scheduling Queues” on page 85), the run queue is kept in priority-sorted order for fast dispatching.
- Floating point registers are saved only if the next process needs them, and restored only if saved.
- Paging I/O is prioritized with the process priority.
- The kernel tries to redispatch a process on the same CPU where it most recently ran, in hopes of finding some of its data remaining in cache (see “Understanding Affinity Scheduling” on page 90).

Special Scheduling Disciplines

The default IRIX scheduling algorithm employs “degrading” priorities. Processes are ranked by a priority value, the one with the lowest priority number running first. But the priority number of a process grows steadily while it runs. The longer a process runs without suspending itself, the lower its priority, and the more likely it is that another process will preempt it.

Nondegrading Priorities

A real-time process needs an unchanging priority. The kernel allows you to apply a nondegrading priority to a specified process. When this priority is in the range of real-time priorities (smaller than normal priorities), the process is scheduled from a real-time scheduling queue, which is tested before the normal dispatch queue. For more information, see “Setting a Nondegrading Batch Priority” on page 88 and “Setting a Nondegrading Real-Time Priority” on page 89.)

Deadline Scheduling

The kernel also supports a deadline scheduling discipline. Under deadline scheduling, a process can request a certain amount of processing time in every interval of a specified length—for example, 30 milliseconds in every 100 milliseconds. For more information, see “Using Deadline Scheduling” on page 92.

Gang Scheduling

When your program is structured as a process group (see “Lightweight Process Creation With sproc()” on page 17), you can request that all the processes of the group be scheduled as a “gang.” The kernel runs all the members of the gang concurrently, provided there are enough CPUs available to do so. This helps to ensure that, when members of the process group coordinate through the use of locks, a lock will usually be released in a timely manner. Without gang scheduling, the process that holds a lock might not be scheduled in the same interval as another process that is waiting on that lock.

For more information, see “Using Gang Scheduling” on page 91.

Processor Sets

IRIX 5.2 and above support the concept of *processor sets*. You can partition the CPUs of a system into multiple, possibly overlapping sets. Then you can

- assign a set of processors to work on a specific scheduling queue, for example the real-time queue, or the gang-scheduling queue
- assign certain processes to run on a specified processor set
- run a UNIX command on a specified processor set (if the command is a shell, commands started from that shell run on the same processor set)

The use of kernel scheduling queues, priorities, and processor sets is covered in more detail in Chapter 6, “Controlling CPU Workload,”. When a real-time application requires only a fraction of the system’s power, these tools may be sufficient to ensure the needed performance. For more critical applications, you need to replace the kernel scheduler with the Frame Scheduler (see “REACT / Pro Frame Scheduler” on page 26).

Locking Virtual Memory

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

This allows you to protect a process from the unpredictable delays caused by paging. Of course the locked memory is not available for the address spaces of other processes. The system must have enough physical memory to hold the real-time address space plus space for a minimum of other activities.

The system calls used to lock memory are discussed in detail in Chapter 4, “Managing Virtual Memory in a Real-Time Program.”

Mapping Processes and CPUs

Normally IRIX tries to keep all CPUs busy, dispatching the next ready process to the next available CPU. (This simple picture is complicated by the needs of affinity scheduling, deadline scheduling, and gang scheduling). Since the number of ready processes changes all the time, dispatching is a random process. A 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.

Real-time processes cannot tolerate this unpredictability. To reduce it, you can dedicate one or more CPUs to real-time work. There are two steps:

- Restrict one or more CPUs from normal scheduling, so that they can run only the processes that are specifically assigned to them.
- 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 (if scheduling is enabled on that CPU). For details, see “Assigning Work to a Restricted CPU” on page 102. The REACT/Pro Frame Scheduler takes care of both steps automatically; see “REACT/Pro Frame Scheduler” on page 26.

Controlling Interrupt Distribution

In normal operations, CPUs receive frequent interrupts:

- I/O interrupts are “sprayed” to different CPUs to equalize workload.
- A scheduling clock causes an interrupt to every CPU every time-slice interval of 10 milliseconds.
- Whenever interval timers are in use (“Timers and Clocks” on page 40), a CPU handling timers receives frequent timer interrupts.
- When the map of virtual to physical memory changes, a TLB interrupt is broadcast to all CPUs.

These interrupts can make the execution time of a process unpredictable. However, you can designate one or more CPUs for real-time use, and keep interrupts of these kinds away from those CPUs. The system calls for interrupt control are discussed at more length under “Minimizing Overhead Work” on page 97. The REACT / Pro Frame Scheduler also takes care of interrupt isolation.

REACT/Pro Frame Scheduler

The REACT / Pro Frame Scheduler is a process execution manager that schedules processes on one or more CPUs in a predefined, cyclic order. The scheduling interval is determined by a repetitive time base, usually a hardware interrupt.

Many real-time programs must sustain a fixed frame rate. In such programs your central design problem is that the program must complete certain activities in every frame interval. When there is more to do in a frame than can be done on one CPU, some activities must run concurrently on multiple CPUs.

Besides designing the activities themselves, you must design a way to schedule and initiate activities in sequence, once per frame, on multiple CPUs. This is what the REACT / Pro Frame Scheduler does: executes the multiple processes of your real-time program one or more CPUs.

How Frames Are Defined

The Frame Scheduler divides time into successive frames, each of the same length. You specify the time base as one of

- a specific interval in microseconds
- the Vsync (vertical retrace) interrupt from the graphics subsystem
- an external interrupt (see “External Interrupts” on page 44)
- a device interrupt from a specially-modified device driver
- a software call (normally used for debugging)

The interrupts from the time base define *minor frames*. You choose the fixed number of minor frames that make a *major frame*, as shown in Figure 3-1.

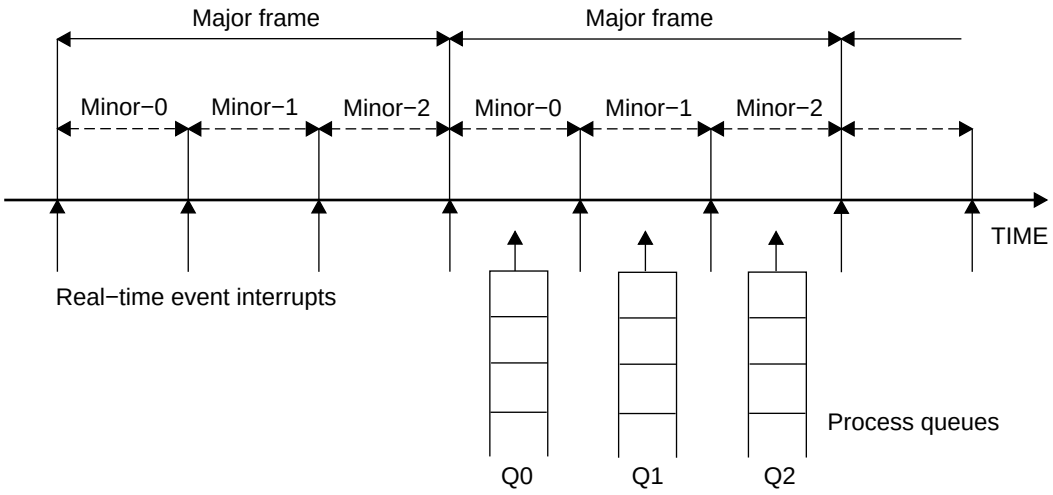

Figure 3-1 Major and Minor Frames

The Frame Scheduler keeps a queue of processes for each minor frame. It dispatches each process once in its scheduled turn. The process runs until it finishes its work; then it yields.

In the simplest case, you have a single frame rate, such as 60 Hz, and every activity your program does must be done once per frame. In this case, the major and minor frame rates are the same. In other cases, you have activities that must be done in every minor frame, but you also have activities that are done less often, in every other, or every third, minor frame. In these cases you define the major frame so that its rate is the rate of the least-frequent activity. Sometimes what is called a “major frame” here is called a “process cycle.”

Advantages of the Frame Scheduler

The Frame Scheduler makes it easy for you to organize a real-time program as a set of independent, cooperating processes. The Frame Scheduler manages the housekeeping details of reserving and isolating CPUs. You concentrate on designing the activities and implementing them as processes in a clean, structured way. It is relatively easy to change the number of activities, or their sequence, or the number of CPUs, even late in the project.

Designing With the Frame Scheduler

To use the Frame Scheduler, you approach the design of your real-time program in the following steps.

1. Partition the program into activities, where each activity is an independent piece of work that can be done without interruption.

   For example, in a simple vehicle simulator, activities might include "poll the joystick," "update the positions of moving objects," "cull the set of visible objects," and so forth.

2. Decide the relationships among the activities:

   - Some must be done once per minor frame, others less frequently.
   - Some must be done before or after others.
   - Some may be conditional. For example, an activity could poll a semaphore and do nothing unless an event had completed.

3. Estimate the worst-case time required to execute each activity. Some activities may need more than one minor frame interval (the Frame Scheduler allows for this).

4. Schedule the activities: If all are executed sequentially, will they complete in one major frame? If not, choose activities that can execute concurrently on two or more CPUs, and estimate again. You may have to change the design in order to get greater concurrency.

When the design is complete, implement each activity as an independent process that communicates with the others using shared memory, semaphores, and locks (see "Interprocess Communication" on page 30).

The main process that initiates the program will contain these steps:

1. Open, create, andinitialize all the shared files and memory resources.
2. Initiate a Frame Scheduler on each of the CPUs that you need (a single library call for each CPU).
3. Initiate each activity as a process using **sproc()** or **fork()**.

   Each process initializes itself and then issues a library call to “join” its assigned Frame Scheduler.
4. Enqueue each process to the Frame Scheduler that will dispatch it.

   Specify the minor frame or frames in which the process should run, and specify a scheduling discipline (described further in Chapter 7, “Using the Frame Scheduler”).
5. Start the Frame Schedulers going (a library call for each one).
6. Wait for a signal indicating it is time to shut down.
7. Terminate the Frame Schedulers.

Each Frame Scheduler seizes its assigned CPU, isolates it, and takes over process scheduling on it. It waits for all enqueued processes to initialize themselves and join it. Then it begins dispatching the processes in the specified sequence during each frame interval. It monitors errors, such as a process that fails to complete its work within its frame, and takes a specified action when an error occurs.

The Frame Scheduler is discussed in more detail in Chapter 7, “Using the Frame Scheduler”. Sample programs that illustrate the Frame Scheduler are described under “Frame Scheduler Examples” on page 226.

Interprocess Communication

In a program organized as multiple, cooperating processes, the processes need to share data and coordinate their actions in well-defined ways. IRIX with REACT provides the following mechanisms, which are surveyed in the topics that follow:

- Shared memory allows a single segment of memory to appear in the address spaces of multiple processes. The Silicon Graphics implementation is also the basis for implementing interprocess semaphores, locks, and barriers.
- Semaphores are used to coordinate access from multiple processes to resources that they share.
- Locks provide a low-overhead, high-speed method of mutual exclusion.
- Barriers make it easy for multiple processes to synchronize the start of a common activity.
- Signals provide asynchronous notification of special events or errors. IRIX supports signal semantics from all major UNIX heritages, but POSIX-standard signals are recommended for real-time programs.

Shared Memory Segments

IRIX allows you to map a segment of memory into the address spaces of two or more processes at once. The block of shared memory can be read concurrently, and possibly written, by all the processes that share it. There are two interfaces, one compatible with SVR4 UNIX and one unique to IRIX.

IRIX Shared Memory Arenas

IRIX supports a unique system of shared memory allocation. The purpose is to create a memory *arena* designed as the basis for high-speed, low-overhead communication between concurrent processes.

You create a shared memory segment with a call to **usinit()**. The argument to **usinit()** is a file pathname string. The file is created (if necessary) and mapped into a segment of memory in the calling process (for a description of mapping files into memory, see Chapter 4). The file, and hence the segment, may or may not continue to exist after the creating process ends. This, and many other options, can be set by calling **usconfig()** before calling **usinit()**.

Once the memory segment exists, any other process can access it by calling **usinit()** with the same pathname string. If that process has access privileges to the specified file, the memory segment is made part of its address space and it, too, can read the memory space, and optionally write in it.

There is a set of memory-allocation library calls that you can use to suballocate memory within a shared arena allocated by **usinit()**. Equally important, IRIX support for semaphores, locks, and barriers is based on the use of arenas allocated with **usinit()**.

For more information on **usinit()** and arenas, refer to *Topics in IRIX Programming* manual, and to the usinit(3p), usconfig(3p) and usmalloc(3p) reference pages. See also the sample code on page 215 and page 222 of "Interprocess Communication" in Appendix A. In addition, some of the special cases of **usinit()** are covered in Chapter 4 of this book.

SVR4-Compatible Shared Memory

IRIX supports shared memory library calls compatible with those in AT&T SVR4 UNIX. In this scheme, one process calls **shmget()** to create a segment of shared memory. In some ways the segment resembles a file more than it resembles memory, for example

- the segment has an owner and group ID, as a file does
- the segment has read and write access permissions for user, group and public, similar to those of a file
- the segment, with its contents intact, continues to exist until it is explicitly deleted using **shmctl()** or until the system is rebooted

A shared segment has an associated integer key. Any other process can present the key to **shmat()**. If the user and group ID of the calling process have access permission, the segment becomes part of the address space of the process. Its virtual address is returned, and the process can use it as memory. If the process has write access, it can update the segment as well as read it.

The SVR4 shared memory facility is useful between processes created by **fork()**, since they have separate address spaces. Processes created by **sproc()** share their entire address space by default.

For sample code and more information on SVR4-compatible shared memory, refer to *Topics in IRIX Programming*, and to the ipcst(1), shmget(2), shmctl(2), and stdipc(3) reference pages.

There is a family of memory-allocation library calls that you can use to suballocate memory within a shared segment (or within any other segment of memory). Refer to the amalloc(3p) reference page for details.

Tip: Use an SVR4-compatible shared memory segment if you require portability. Otherwise, the IRIX implementation is faster and more flexible for a real-time program.

Semaphores

A *semaphore* is a memory object that represents the state of a shared resource. The content of a semaphore is an integer count, representing the number of resource units now available. Typically the count is 1, and the semaphore represents the availability of a single object such as a table or file.

A process that needs to use the resource executes a “P” operation on the semaphore. This operation tests and decrements the count in the semaphore. If the count is nonzero before the operation, at least one resource unit is available. The count is reduced by 1 and the process continues executing. Otherwise the process is blocked until a resource unit is available; then it continues. In either case, following a P operation, the process knows that it has exclusive use of a resource unit.

When it finishes its work, the process releases the resource by executing a "V" operation on the semaphore. This operation increments the count. It also unblocks any process that might be blocked in a P operation, waiting for the resource. If more than one process is waiting, the one that has waited longest is released first (FIFO order).

Tip: Useful mnemonics for P and V: P de***p***letes the resource. V re***v***i***v***es it.

IRIX supports two forms of semaphore: SVR4-compatible, and Silicon Graphics.

IRIX Semaphores

IRIX supports a set of semaphore operations designed for low-overhead coordination between multiple concurrent processes. You create these semaphores within a shared arena created with **usinit()** (see "IRIX Shared Memory Arenas" on page 30). The **usnewsema()** call creates a semaphore. You specify the arena handle and the initial value for the semaphore (that is, the count of resources that it represents, typically 1).

To acquire a resource, blocking if it is not available, a process applies the **uspsema()** call to the semaphore. To test the resource, acquiring it if it is available but not blocking when it is in use, a process can call **uscpsema()**. To release the resource, a process calls **usvsema()**.

IRIX also supports a parallel set of "pollable" semaphores. The P operation on a pollable semaphores does not block when the resource is in use. Instead, it returns a flag value, and the process must use the **poll()** system call to find out when a V operation has made the resource available.

IRIX semaphores support "metering" (use counts) and debug tracing. You can turn either facility on and off dynamically. By metering a semaphore, you can find out how often processes actually block in a P operation. This can reveal whether or not a resource is a bottleneck to performance.

For more information on semaphores, refer to *Topics in IRIX Programming*, and to the usnewsema(3), usnewpollsema(3), uspsems(3), usvsema(3), and poll(2) reference pages. The sample program shown in "Interprocess Communication" on page 212 uses IRIX semaphores, and demonstrates the use of metering information.

SVR4-Compatible Semaphores

SVR4-compatible semaphores are created in sets of one or more—typically a set contains all the semaphores that one application needs. A set is created by a **semget()** call, which specifies an integer key to identify the set and access permissions for the set.

Like a shared memory segment (see "SVR4-Compatible Shared Memory" on page 31), a set of semaphores is somewhat like a file in that it

- has a user and group ID from the process that created it
- has read and write access permissions for owner, group and world
- continues to exist after its creating process ends.

Once a set of semaphores exists, any other process can issue **semget()** with the same key. If the user and group ID of the calling process have access permission, the process can use the semaphores in the set.

SVR4-compatible semaphores do not support the conventional P and V operations. Instead, the **semop()** system call supplies a wider range of operations, including incrementing and decrementing counts by more than 1. The **semop()** call supports concurrent operations on multiple semaphores at once. This is convenient in some cases because it allows you to claim more than one resource simultaneously, without danger of *deadlock*.

For sample code and more information on SVR4-compatible semaphores, refer to *Topics in IRIX Programming*, and to the ipcst(1), semget(2), semctl(2), and semop(2) reference pages. The administration of SVR4-compatible semaphores is covered in the *IRIX Advanced Site and Server Administrator Guide*.

Tip: If you require portability, use SVR4-compatible semaphores. Otherwise, the IRIX semaphore implementation is faster, has more features, and works with the IRIX shared-memory implementation.

Locks

A lock is a memory object that represents the exclusive right to use a shared resource. A process that wants to use the resource sets the lock. The process releases the lock when it is finished with the resource.

A lock is functionally the same as a semaphore with a count of 1. The set operation on a lock and the P operation on a semaphore with a count of 1 both acquire exclusive use of a resource. In a multiprocessor, the important difference between a lock and semaphore is that, when the resource is not immediately available, a semaphore always suspends the process, while a lock does not.

A lock, in a multiprocessor system, is set by "spinning." The program enters a tight loop using the test-and-set machine instruction to test the lock's value and to set it as soon as the lock is clear. In practice the lock is often already available, and the first execution of test-and-set acquires the lock. In this case, setting the lock takes a trivial amount of time.

When the lock is already set, the process spins on the test a certain number of times. If the process that holds the lock is executing concurrently in another CPU, and if it releases the lock during this time, the spinning process acquires the lock instantly. There is zero latency between release and acquisition, and no overhead from entering the kernel for a system call.

If the process has not acquired the lock after a certain number of spins, it defers to other processes by calling **sginap()**. When the lock is released, the process resumes execution.

You create a lock in an arena created by **usinit()**. The lock is allocated by **usnewlock()**. You set a lock with **ussetlock()** and release it with **usunsetlock()**.

Like IRIX semaphores, locks can collect metering (use-count) information and/or debugging trace data. You can use the metering information to find out how many times a lock was used and how often a process had to spin or block at a lock.

For more information on locks, refer to *Topics in IRIX Programming*, and to the usnewlock(3), ussetlock(3) and usunsetlock(3) reference pages. See also

the sample code on page 217 of “Interprocess Communication” in Appendix A.

Barriers

A barrier is a memory object that represents a point of rendezvous between multiple processes. You use a barrier to ensure that processes do not advance until some necessary preparation has been done.

A barrier is created by **newbarrier()** in an arena built by **usinit()**. The barrier is used by some fixed number (*N*) of processes. When each process is ready to rendezvous with the others, it issues **barrier(***N***)**. As each process arrives at the barrier, it is suspended. When the *N*th process calls **barrier()**, all the processes resume execution. This is the computing equivalent of *N* coworkers who agree to go to lunch together. As each one realizes it is lunch time, he or she goes to the lobby. When the *N*th coworker reaches the lobby, all of them depart for lunch.

As an example of the use of a barrier, imagine that you discover that a nested loop to take the sum of a large matrix is a bottleneck in your program. To speed up the calculation you divide it between two processes. (Presumably they will run in different CPUs.) The first process is the one that requires the matrix sum, and which originally calculated the sum by itself. The second process is a new one, whose only purpose is to assist in the matrix sum calculation. You create a barrier named *matsum* to coordinate the two.

The logic of the second, helper, process would be as follows:

1. Call **barrier(***matsum*,2**)** to wait until it is time to take the sum.
2. Calculate the sum over all even-numbered rows of the matrix.
3. Store the sum in global *evensum*.
4. Call **barrier(***matsum*,2**)** to wait until the first process is finished.
5. Return to step 1.

The logic of the first, main process would be as follows:

1. Perform other work as required until the matrix sum is needed.
2. Call **barrier(***matsum*,2**)** to release the helper process.

3. Calculate the sum over all odd-numbered rows of the matrix.
4. Call **barrier(***matsum*,2**)** to wait until the second process has finished its calculation.
5. Add *evensum* to the odd total to get the grand total.
6. Return to step 1.

The example can be generalized to more processes, and to any other calculation that can be partitioned in this way.

Mutual Exclusion Primitives

IRIX supports library functions that perform atomic (uninterruptable) sample-and-set operations on words of memory. For example, **test_and_set()** copies the value of a word and stores a new value into the word in a single operation; while **test_then_add()** samples a word and then replaces it with the sum of the sampled value and a new value.

These primitive operations can be used as the basis of mutual-exclusion protocols using words of shared memory. For details, see the test_and_set(3p) reference page.

The **test_and_set()** and related functions are based on the MIPS R4000 instructions Load Linked and Store Conditional. Load Linked retrieves a word from memory and tags the processor data cache “line” from which it comes. The following Store Conditional tests the cache line. If any other processor or device has modified that cache line since the Load Linked was executed, the store is not done. The implementation of **test_then_add()** is comparable to the following assembly-language loop:

```
1:
    ll    retreg, offset(targreg)
    add   tmpreg, retreg, valreg
    sc    tmpreg, offset(targreg)
    beq   tmpreg, 0, b1
```

The loop continues trying to load, augment, and store the target word until it succeeds. Then it returns the value retrieved. For more details on the R4000 machine language, see one of the books listed in “Other Useful Books” on page xxiii.

The Load Linked and Store Conditional instructions only operate on memory locations that can be cached. Uncached pages (for example, pages implemented as reflective shared memory, see “Reflective Shared Memory” on page 43) cannot be set by the **test_and_set()** functions.

Signals

A signal is an urgent notification of an event, sent asynchronously to a process. Some signals originate from the kernel: for example, the SIGFPE signal that notifies of an arithmetic overflow; or SIGALRM that notifies of the expiration of a timer interval (for the complete list, see the signal(5) reference page). The Frame Scheduler issues signals to notify your program of errors or termination. Other signals can originate within your own program.

In order to receive a signal, a process must establish a signal handler, a function that will be entered when the signal arrives.

There are three UNIX traditions for signals, and IRIX supports all three. They differ in the library calls used, in the range of signals allowed, and in the details of signal delivery (see Table 3-1). Your real-time program should use the POSIX interface for signals.

Table 3-1 Signal Handling Interfaces

Function	SVR4-compatible Calls	BSD 4.2 Calls	POSIX Calls
set and / or query signal handler	**sigset(**2**)** **signal(**2**)**	**sigvec(**3**)** **signal(**3**)**	**sigaction(**2**)** **sigsetops(**3**)** **sigaltstack(**2**)**
send a signal	**sigsend(**2**)** **kill(**2**)**	**kill(**3**)** **killpg(**3**)**	**sigqueue(**2**)**
temporarily block specified signals	**sighold(**2**)** **sigrelse(**2**)**	**sigblock(**3**)** **sigsetmask(**3**)**	**sigprocmask(**2**)**

Table 3-1 Signal Handling Interfaces

Function	SVR4-compatible Calls	BSD 4.2 Calls	POSIX Calls
query pending signals			**sigpending(**2**)**
wait for a signal	**sigpause(**2**)**	**sigpause(**3**)**	**sigsuspend(**2**)** **sigwait(**2**)** **sigwaitinfo(**2**)** **sigtimedwait(**2**)**

The POSIX interface supports the following 64 signal types:

1-31	Same as BSD
32	Reserved by IRIX kernel
33-48	Reserved by the POSIX standard for system use
49-64	Reserved by POSIX for real-time programming

Signals with smaller numbers have priority for delivery. The low-numbered BSD-compatible signals, which include all kernel-produced signals, are delivered ahead of real-time signals; and signal 49 takes precedence over signal 64. (The BSD-compatible interface supports only signals 1-31. This set includes two user-defined signals.)

IRIX 5.3 supports POSIX signal handling as specified in document 1003.1b-1993. This includes FIFO queueing new signals when a signal type is held, up to a system maximum of queued signals. (The maximum can be adjusted using *systune*; see the systune(1) reference page.)

For more information on the POSIX interface to signal handling, refer to *Topics in IRIX Programming* and to the signal(5), sigaction(2), and sigqueue(2) reference pages. Some POSIX signal-handling functions are used in sample code in “Interprocess Communication” in Appendix A on page 220 and following.

Signal Latency

The time that elapses from the moment a signal is generated until your signal handler begins to execute is the *signal latency*. Signal latency can be long, as real-time programs measure time, and signal latency has a high variability. (Some of the factors are discussed under “Signal Delivery and Latency” on page 141.) In general, you should use signals to deliver infrequent messages of high priority. You should not use the exchange of signals as the basis for scheduling in a real-time program.

Note: Signals are delivered at particular times when using the Frame Scheduler. See “Using Signals Under the Frame Scheduler” on page 141.

Timers and Clocks

A real-time program sometimes needs a source of timer interrupts, and some need a way to create a high-precision timestamp. Both of these are provided by IRIX.

Timer Interrupts (Itimers)

IRIX supports the BSD UNIX feature of interval timers or “itimers.” An itimer is a request to have a signal sent at the expiration of a specified interval. In order to use an itimer, you establish a signal handler, then issue the **setitimer()** call. The timer can be a one-shot or it can repeat at a regular interval.

There are three itimers (see Table 3-2), only one of which is of interest to a real-time programmer.

Table 3-2 Types of itimer

Kind of itimer	Interval Measured	Resolution	Signal Sent
ITIMER_REAL	Elapsed clock time	1 millisecond or less	SIGALRM
ITIMER_VIRTUAL	User time (process execution time)	1 second	SIGVTALRM
ITIMER_PROF	User+system time	1 second	SIGPROF

The ITIMER_VIRTUAL and ITIMER_PROF timers are not useful to a real-time program because of their coarse precision and because their intervals vary depending on when and how often the process is dispatched. The ITIMER_REAL type measures absolute time, and on the Challenge/Onyx, its resolution can be 500 microseconds or less.

Timers and the resolution of the real-time timer are discussed further in Chapter 5, “Managing Time and Time Intervals.” Sample code that sets up an itimer can be located near page 220 (see “Interprocess Communication” in Appendix A).

Note: Interval timers are usually not necessary, and should not be used, under the Frame Scheduler. See “Using Timers with the Frame Scheduler” on page 143.

Timestamps

The IRIX operating system and Silicon Graphics hardware provide two forms of free-running clock that you can use as a timestamp; that is, as a value establishes the relative time difference between two events. One clock is returned by a standard system call; the other is a hardware device you map into process address space.

Time of Day Timestamp

The BSD-compatible function **gettimeofday()** returns the time of day as two long integers which together give the time since 1/1/1970 to the microsecond. The resolution of this value is at least 10 milliseconds—that is, it is guaranteed to change at least 100 times a second. The actual resolution depends on the system.

The time of day timestamp is discussed further in Chapter 5, “Managing Time and Time Intervals.” The sample program under “Getting the Time of Day Stamp” on page 211 tests the time-of-day clock to find out its true precision.

Hardware Cycle Counter

The cycle counter is a high-precision hardware counter that is updated continuously. In a Challenge / Onyx machine it is a 64-bit value. In other Silicon Graphics architectures the cycle counter has less precision; for example, in the Indy it is a 32-bit counter.

In the Challenge / Onyx, the cycle counter is incremented every 21 nanoseconds. In other architectures the frequency is lower, although it is always comparable to the instruction execution time. (For example, in the Indy it is incremented every 40 nanoseconds.) Because of the high frequency, the cycle counter is certain to contain a different value every time it is sampled.

Note: Considered as a time standard, the Challenge / Onyx cycle counter is accurate to 1 part in 10,000. If you use it to measure intervals between events, be aware that it can drift by as much as 100 microseconds per second.

You sample the cycle counter by mapping it into the process's address space, then reading it as if it were a memory variable. The method is covered in Chapter 5, "Managing Time and Time Intervals." The sample program under "Mapping and Reading the Cycle Counter" on page 202 also demonstrates its use.

Interchassis Communication

Silicon Graphics systems support three methods by which you can connect multiple computers:

- Standard network interfaces let you send packets or streams of data over a local network or the Internet.
- Reflective shared memory (provided by third-party manufacturers) lets you share segments of memory between computers, so that programs running on different chassis can access the same variables.
- External interrupts let one Challenge / Onyx signal another.

Socket Programming

The 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.

For more information about socket programming, refer to one of the networking books listed in “Other Useful Books” on page xxiii.

Reflective Shared Memory

Reflective shared memory consists of hardware that makes a segment of memory appear to be accessible from two or more computer chassis. Actually the Challenge/Onyx implementation consists of VME bus devices in each computer, connected by a very high-speed, point-to-point network.

The VME bus address space of the memory card is mapped into process address space. Firmware on the card handles communication across the network, so as to keep the memory contents of all connected cards consistent. Reflective shared memory is slower than real main memory but faster than socket I/O. Its performance is essentially that of programmed I/O to the VME bus, which is discussed under “PIO Access” on page 190.

Reflective shared memory systems are available for Silicon Graphics equipment from several third-party vendors. The details of the software interface differ with each vendor. However, in most cases you use **mmap()** to map the shared segment into your process’s address space (see Chapter 4, “Managing Virtual Memory in a Real-Time Program” as well as the usrvme(7) reference page).

External Interrupts

The Challenge / Onyx systems (only) support external interrupt lines for both incoming and outgoing external interrupts. Software support for these lines is provided in IRIX version 5.3.

Four outgoing external interrupt lines appear on the back panel of the computer. You can control them individually, creating pulses or simply asserting and deasserting the lines.

Two input jacks for external interrupts are provided. Either of these jacks can cause an interrupt, but you cannot distinguish which jack caused a given interrupt. The interrupt is level-triggered, not edge-triggered.

For details of the use and programming of external interrupts, see “External Interrupts” on page 195. You can use the external interrupt as the time base for the Frame Scheduler. In that case, the Frame Scheduler manages the external interrupts for you. (See “Selecting a Time Base” on page 125.)

Chapter 4

Managing Virtual Memory in a Real-Time Program

When planning a real-time program you must understand how IRIX creates the virtual address space of a process, and how you can modify the normal behavior of the address space. The major topics covered are:

- “Defining the Address Space” on page 45 tells what the address space is and how it is created.
- “Interrogating the Memory System” on page 52 summarizes the ways your program can get information about the address space.
- “Mapping Segments of Memory” on page 53 documents the different ways that you can create new memory segments with predefined contents.
- “Locking Pages in Memory” on page 65 discusses when and how to lock pages of virtual memory to avoid page faults.
- “Reducing Cache Misses” on page 68 documents techniques for avoiding performance problems due to poor cache use.
- “Additional Memory Features” on page 71 summarizes functions for address space management.

Defining the Address Space

Each process has a *virtual address space*; in other words, a set of memory addresses that the process can use. When 32-bit addressing is in use, the addresses can range from 0 to 0x7fffffff; that is, 2^{31} numbers, for a total theoretical size of 2 gigabytes. When 64-bit addressing is used, a process’s address space can encompass 2^{40} numbers. (The numbers greater than 2^{31} or 2^{40} are reserved for kernel and supervisor address spaces.) In practice, most programs use a much smaller range of addresses.

A *segment* of the address space is any range of contiguous addresses. Certain segments are created or reserved for certain uses.

Addresses are called “virtual” because they are not directly related to the physical RAM addresses where the data actually exists. The mapping from a virtual address to a real memory location is kept in tables that IRIX creates and the hardware maintains (see “Virtual Memory” on page 12).

Address Space Boundaries

A process has at least 3 segments of usable addresses:

- A text segment contains the executable image of the program. The text segment is always read-only.
- A data segment contains the “heap” of allocated data space.
- A stack segment contains the function-call stack.

Another text segment is created for each dynamic shared object (DSO) with which a process is linked. A process can create additional data segments in various ways described later in the chapter.

Although the address space begins at location 0, by convention the lowest segment is allocated at 0x00400000 (4 megabytes). Addresses less than this are left undefined so that an attempt to reference them (for example, through an uninitialized pointer variable) causes a hardware exception.

Typically, the text segments are at smaller virtual addresses and stack and data segments at larger ones, although you should not write code that depends on this. The sample program shown in “Probing the Address Space” on page 223 finds and displays some standard segment base addresses.

Page Numbers and Offsets

IRIX manages memory in units of a page. The size of a page can differ from one system to another. The size when 32-bit addressing is used is 4,096 bytes. In each 32-bit virtual address,

- the least-significant 12 bits specify an offset from 0 to 0x0fff within a page
- the most-significant 20 bits specify a virtual page number (VPN)

The page size when 64-bit addressing is used is greater than 4,096 bytes but the principle is the same. The less-significant bits of an address specify an offset within a page, while the more-significant bits specify the VPN.

Page tables, built by IRIX during a **fork()** or **exec()** call, specify which VPNs are defined. The tables are consulted by the hardware. Recently-used table entries are cached in the processor chip (see "Translation Lookaside Buffer Updates" on page 13).

The actual size of a page in the present system can be learned with **getpagesize()** as noted under "Interrogating the Memory System" on page 52.

Address Definition

Most of the possible addresses in an address space are undefined, that is, not entered in the page tables, not related to contents of any kind, and not available for use. A reference to an undefined address causes a SIGBUS error.

Addresses are defined, that is, made available for potential use, in one of four ways:

Fork — When a process is created using **fork()**, any addresses that were defined in the parent's address space are defined in the address space of the new process (see "Normal Process Creation With fork()" on page 15).

Stack — The call stack is created and extended automatically. When a function is entered and more stack space is needed, IRIX makes the stack segment larger (to the limit set by *rlimit_stack_max*), defining new addresses if required.

Mapping — Your program can ask IRIX to *map* (associate byte-for-byte) a segment of address space to one of a number of special objects, for example, the contents of a file. This is covered further under "Mapping Segments of Memory" on page 53.

Allocation — The **brk()** function extends the segment devoted to data (the *heap*) to a specific virtual address, subject to *rlimit_data_max*. The **malloc()** function allocates memory for use, calling **brk()** as required. (See the brk(2) and malloc(3) reference pages). The more commonly used library version of **malloc()** calls the underlying **malloc()** (see the malloc(3x) reference page).

When an address is defined, it is entered in the page tables and related to a *backing store*, a source from which its contents can be retrieved. A page in the data or stack segment is related to a page in the swap partition on disk.

The total size of the defined pages in an address space is its *virtual size*, displayed by the *ps* command under the heading SZ (see the ps(1) reference page).

Once addresses have been defined in the address space, there is no way to undefine them except to terminate the process. To free allocated memory makes the freed memory available for reuse within the process, but the pages are still defined in the page tables and the swap space is still allocated.

Address Space Limits

The maximum size for the address space is set as a resource limit. The possible range of any resource limit is established in the kernel tuning parameters. To see the kernel limits, use

```
fgrep rlimit /var/sysgen/mtune/kernel
```

Hard limits on the size of the address space are set by:

rlimit_vmem_max — Total size of the address space of a process

rlimit_data_max — Size of the portion of the address space used for data

rlimit_stack_max — Size of the portion of the address space used for stack

The same limits can be displayed using the C-shell command *limits*.

Your program can query resource limits with **getrlimit()** and can change the current limits with **setrlimit()** (see the getrlimit(2) reference page).

Tip: These limits interact in the following way: each time your program creates a process with **sproc()** (see "Lightweight Process Creation With sproc()" on page 17), space equal to *rlimit_stack_max* is dedicated to the stack of the new process. When *rlimit_stack_max* is set high, a program that creates many processes can quickly run into the *rlimit_vmem_max* boundary.

Delayed Definition

It is important to note that, beginning with IRIX 5.2, **brk()** and **malloc()** merely test the new size of the data segment against the resource limits. They do not actually define new addresses, and they do not cause swap disk space to be allocated. Addresses are *reserved* with **brk()** or **malloc()**, but they are only *defined* and allocated in swap when your program references them.

This is not the conventional behavior of UNIX systems. Conventionally, space defined with **malloc()** is defined immediately, including the allocation of swap space. Three results follow from the conventional method:

- A program can detect immediately when swap space is exhausted. A call to **malloc()** returns NULL when memory cannot be allocated. A program can probe the limits of swap space by repeated calls to **malloc()**.
- A large memory allocation by one program can fill swap, causing other programs to see out-of-memory errors—whether the program ever uses its allocated memory or not.
- A **fork()** or **exec()** call fails unless there is free space in swap equal to the data and stack sizes of the new process.

The IRIX use of delayed definition has the opposite effects:

- A program cannot detect the limits of swap space using **malloc()**, which never returns NULL until the program exceeds its resource limit.

  Instead, when finally a program accesses a new page and there is no room in the swap partition, the program receives a SIGKILL signal.

- A large memory allocation by one program cannot monopolize the swap disk unless the program actually uses the allocated memory.
- Much less swap is required for a successful **fork()**.

As you write a new program, you should assume that delayed definition may be used. Allocate no more memory than your program needs, and use the memory immediately after allocating it.

If you are porting a program written for a conventional UNIX system, you might discover that it tests the limits of allocatable memory by calling **malloc()** until **malloc()** returns a NULL, and then does not use the memory. In this case you have three choices:

- Recode this part of the program.
- Using **setrlimit()**, set a lower maximum for *rlimit_data_max*, so that **malloc()** returns NULL at a reasonable allocation size.
- Restore the conventional UNIX behavior for the whole system. Use *chkconfig* to turn variable *vswap* off, and reboot (see the chkconfig(1) reference page).

For a more detailed discussion of "virtual swap," refer to the swap(1) reference page. In IRIX 5.2, *vswap* is on by default. In IRIX 5.3 it is off by default (but can easily be turned on using *chkconfig*).

Note: The function **calloc()** touches all allocated pages in the course of filling them with zeros. Hence memory allocated by **calloc()** is defined as soon as it is allocated (through IRIX 5.3). However you should not rely on this behavior. It is possible to implement **calloc()** in such a way that it, like **malloc()**, does not define allocated pages until they are used, and this might be done in a future version of IRIX.

Page Validation

Although an address is defined, the corresponding page is not necessarily loaded in physical memory. The sum of the address spaces of all processes is normally far larger than available real memory. IRIX keeps selected pages in real memory. A page that is not present in real memory is marked as "invalid" in the page tables. Invalid pages can be any of the following:

Text	Pages of program text—executable code of programs and dynamically-linked libraries—can be retrieved on demand from the program or library files on disk.
Data	Pages of data from the heap and stack can be retrieved from the system swap partition on disk.
Never used	Pages that have been defined but never used can be created as pages of binary zero when needed.

When your process refers to a VPN that is defined but invalid, a hardware interrupt occurs. The interrupt handler chooses a page of physical memory to hold your page. In order to acquire a memory page, it might have to invalidate some other page belonging to your process or to another process. The contents of the needed page are retrieved from the appropriate backing store, and your process continues to execute.

Page validation takes from 10 to 50 milliseconds, a delay that a real-time program normally cannot tolerate.

The total size of all the valid pages in an address space is the *resident set size*, displayed by the *ps* command under the heading RSS.

Read-Only Pages

A page of memory can be marked as valid for reading but invalid for writing. Program text is marked this way because program text is read-only; it is never changed. If a process attempts to modify a read-only page, a hardware interrupt occurs. When the page is truly read-only, the kernel turns this into a SIGSEGV signal to the program. Unless the program is handling this signal (see "Signals" on page 38) the result is to terminate the program with a segmentation fault.

Copy-on-Write Pages

When **fork()** is executed, the new process shares the pages of the parent process under a rule of copy-on-write. The pages in the new address space are marked read-only. When the new process attempts to modify a page, a hardware interrupt occurs. The kernel makes a copy of that page, and changes the new address space to point to the copied page. Then the process continues to execute, modifying the page of which it now has a unique copy.

You can apply the copy-on-write discipline to the pages of an arena shared with other processes (see “Mapping a File for Shared Memory” on page 60).

Interrogating the Memory System

You can get information about the state of the memory system with the system calls shown in Table 4-1.

Table 4-1 Memory System Calls

Memory Information	System Call Invocation
Size of a page	uiPageSize = getpagesize(); ulPageSize = sysconf(_SC_PAGESIZE);
Virtual and resident sizes of a process	syssgi(SGI_PROCSZ, pid, &uiSZ, &uiRSS);
Maximum stack size of a process	uiStackSize = prctl(PR_GETSTACKSIZE)
Free swap space in 512-byte units	swapctl(SC_GETFREESWAP, &uiBlocks);
Total physical swap space in 512-byte units	swapctl(SC_GETSWAPTOT, &uiBlocks);
Total real memory	sysmp(MP_KERNADDR, MPSA_RMINFO, &rmstruct);
Free real memory	sysmp(MP_KERNADDR, MPSA_RMINFO, &rmstruct);
Total real+swap space	sysmp(MP_KERNADDR, MPSA_RMINFO, &rmstruct);

The structure used with the **sysmp()** call shown above has this form (a more detailed layout is in *sys/sysmp.h*):

```
struct rminfo {
   long freemem; /* pages of free memory */
   long availsmem; /* total real+swap memory space */
   long availrmem; /* available real memory space */
   long bufmem; /* not useful */
   long physmem; /* total real memory space */
};
```

A sample program that applies **swapctl()** and **sysmp()** to display these numbers is shipped in the 4DGifts example directory. See *~4Dgifts/examples/unix/irix/freevmen.c*

Mapping Segments of Memory

Your process can create new segments within the address space. Such a "mapped" segment can represent

- the contents of a file
- the registers of a VME device
- a segment filled with binary zero
- a view of the kernel's private address space or of physical memory

A mapped segment can be private to one address space, or it can be shared between address spaces. When shared, it can be

- read-only to all processes
- read-write to the creating process and read-only to others
- read-write to all sharing processes
- copy-on-write, so that any sharing process that modifies a page is given its own unique copy of the page

The Segment Mapping Function mmap()

The **mmap()** function (see the mmap(2) reference page) creates shared or unshared segments of memory. The basic functions of **mmap()** are supported by other UNIX versions including System V release 4, so many uses of it are portable beyond IRIX. Some features of **mmap()**, however, are unique to IRIX.

The **mmap()** function performs many kinds of mappings based on six parameters. The function prototype is:

```
void * mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off)
```

The result of the function call is the base address of the new segment, or else -1 to indicate that no segment was created. The size of the new segment is *len*, rounded up to a page. An attempt to access data beyond that point causes a SIGBUS signal.

Describing the Source Object

Three of the parameters describe the source of the data to be mapped into memory:

fd — A file descriptor returned by **open()** (see the open(2) reference page). All **mmap()** calls require a file descriptor, which specifies the backing store for the mapped segment. The descriptor can be represent a file, or it can be based on a pseudo-file that represents memory or a device.

off — The offset into the object represented by *fd* where the mapped data begins. When *fd* describes a disk file, *off* is an offset into the file. When *fd* describes memory, *off* is an address in that memory. *off* must be an integral multiple of the system page size (see “Interrogating the Memory System” on page 52).

len — The number of bytes of data from *fd* to be mapped. The initial size of the segment will be *len*, rounded up to a multiple of whole pages.

Describing the New Segment

Three parameters of **mmap()** describe the segment to be created.

addr — Normally 0 to indicate that IRIX should pick a convenient base address, *addr* can specify a virtual address to be the base of the segment. See “Choosing a Segment Address” on page 62.

prot — Specifies access control on the new segment. You use constants to specify a combination of read, write, and execute permission. The access control can be changed later (see “Changing Memory Protection” on page 71).

flags — Specifies details of how the new segment is to be managed.

One element of *flags* modifies the meaning of *addr*. Discussion of this is under “Choosing a Segment Address” on page 62. The remaining elements determine the way the segment behaves.

One element of *flags* specifies what should happen when a process stores data past the current end of the segment (provided storing is allowed by *prot*). If *flags* contains MAP_AUTOGROW, the segment will be extended with zero-filled space. Otherwise the initial *len* value is a permanent limit, and an attempt to store more than *len* bytes from the base address causes a SIGSEGV signal.

Two elements of *flags* specify the rules for sharing the segment between two address spaces when the segment is writable:

- MAP_SHARED specifies that changes made to the common pages are visible to other processes sharing the segment.

  When a shared segment is writeable, any changes to the segment in memory are also written to the file that is mapped. The mapped file is the backing store for the segment.

  When MAP_AUTOGROW is specified also, a store beyond the end of the segment lengthens the segment and also the file it is mapped to.

- MAP_PRIVATE specifies that changes to shared pages are private to the process that makes the changes.

  The pages of a private segment are shared on a copy-on-write basis—there is only one copy as long as they are unmodified. When the process that specifies MAP_PRIVATE stores into the segment, that page is copied. The process has a private copy of the modified page from then on. The backing store for unmodified pages is the file; the backing store for modified pages is the system swap space.

  When MAP_AUTOGROW is specified also, a store beyond the end of the segment lengthens only the private copy of the segment; the file is unchanged.

The difference between MAP_SHARED and MAP_PRIVATE is important only when the segment can be modified. When the *prot* argument does not include PROT_WRITE, there is no question of modifying or extending the segment, so backing store is always the mapped object. However, the choice of MAP_SHARED or MAP_PRIVATE does affect how you lock the mapped segment into memory, if you do; see “Locking Program Text and Data” on page 66.

Processes created with **sproc()** normally share a single address space, including mapped segments (see “Lightweight Process Creation With sproc()” on page 17). However, if *flags* contains MAP_LOCAL, each new process created with **sproc()** receives a private copy of the mapped segment, on a copy-on-write basis.

When the segment is based on a file or on */dev/zero* (see “Mapping a Segment of Zeros” on page 61), **mmap()** normally defines all the pages in the segment. This includes allocating swap space for the pages of a segment based on */dev/zero*. However, if *flags* contains MAP_AUTOGROW, the pages are not defined until they are accessed (see “Delayed Definition” on page 49).

Note: The MAP_LOCAL and MAP_AUTOGROW flag elements are IRIX features that are not portable to System V.

Mapping a File for I/O

You can use **mmap()** as a simple, low-overhead way of reading and writing a disk file. Open the file using **open()**, but then instead of passing the file descriptor to **read()** or **write()**, use it to map the file. Read or write the file by accessing it as memory. Memory accesses are translated into direct calls to the device driver:

- An attempt to access a mapped page, when the page is not resident in memory, is translated into a call on the read entry point of the device driver to read that page of data.
- When the kernel needs to reclaim a page of physical memory occupied by a page of a mapped file, and the page has been modified, the kernel calls the write entry point of the device driver to write the page. It also writes any modified pages when the file mapping is changed by **munmap()** or another **mmap()** call, when the program **msync()** for the segment, or when the program ends.

When mapping a file for input only (when the *prot* argument of mmap() does not contain PROT_WRITE), you can use either MAP_SHARED or MAP_PRIVATE. When writing is allowed, you must use MAP_SHARED, or changes will not be reflected in the file.

Memory mapping provides an excellent way to read a file containing pre-calculated, constant data. For example, a visual simulator could map a portion of a file containing precalculated scenery elements. Time-consuming calculation of the scenery elements could be done off-line by another program which also mapped the file in order to fill it with data.

Mapped File Sizes

Since the potential address space is more than 2000 megabytes, you can in theory map very large files into memory. To map an entire file:

1. Open the file to get a file descriptor.
2. Use **lseek(***fd*,0,SEEK_END**)** to discover the size of the file.
3. Map the file with an *off* of 0 and *len* of the file size.

You can lock a mapped file into memory. This is discussed further under “Locking Pages in Memory” on page 65.

Apparent Process Size

When you map a large file into memory, the space is counted as part of the virtual size of the process. This can lead to very large apparent sizes. For example, under IRIX 5.3, the ObjectServer maps a large database into memory, with the result that a typical result of *ps -l* looks like this:

```
70 S 0 566 1 0 26 20 * 33481:225 80272230 ? 0:45 objectse
```

The virtual size of 33481 certainly gets one's attention! However, note the more modest real storage size of 225. Most of the mapped pages are not in physical memory. Also realize that the backing store for pages of a mapped file is the file itself—no swap space is used.

Mapping Portions of a File

You do not have to map the entire file; you can map any portion of it, from one page-size to the file size. The *off* parameter of **mmap()** can be used as the logical equivalent of **lseek()**. That is, to map a different segment of the file, you would specify

- the same file descriptor
- the desired offset in *off*
- the current segment base address as *addr*
- MAP_FIXED in *flags* to force the use of *addr*

The old segment is replaced with a new segment at the same address, now containing data from a different offset in the file.

Each time you replace a segment with **mmap()**, the previous segment is discarded. The new segment is not locked in memory, even if the old segment was locked.

File Permissions

Access to a file for mapping is controlled by the same file permissions that control I/O to the file. The protection in *prot* must agree with the file permissions. For example, if the file is read-only to the process, **mmap()** will not allow *prot* to specify write or execute access.

Note: Since real-time programs often need to run with root privilege for other reasons, file permissions are no protection against accidental updates.

NFS Considerations

The file that is mapped can be local to the machine, or can be mounted by NFS. In either case, be aware that changes to the file will be buffered and will not be immediately reflected on disk. Use **msync()** to force modified pages of a segment to be written to disk (see "Synchronizing the Backing Store" on page 72).

If IRIX needs to read a page of a mapped, NFS-mounted file, and an NFS error occurs (for example, because the file server has gone down), the error is reflected to your program as a SIGBUS exception.

Warning: When two or more processes in the *same* system map an NFS-mounted file, their image of the file will be consistent. But when two or more processes in *different* systems map the same NFS-mounted file, there is no way to coordinate their updates, and the file can be corrupted.

File Integrity

Any change to a file is immediately visible in the mapped segment. This is always true when *flags* contains MAP_SHARED, and initially true when flags contains MAP_PRIVATE. A change to the file can be made by another process that has mapped the same file. A change can also be made by a process that opens the file for output and then applies either **write()** to update the file or **ftruncate()** to shorten it (see the write(2) and ftruncate(3) reference pages). In particular, if the mapped file is shortened, an attempt to access a memory page that corresponds to a now-deleted portion of the file will cause a bus error signal (SIGBUS) to be sent.

When MAP_PRIVATE is specified, a private copy of a page of memory is created whenever the process stores into the page (copy-on-write). This prevents the change from being seen by any other process that uses or maps the same file, and it protects the process from detecting any change made to that page by another process. However, this applies only to pages that have been written into.

Frequently you cannot use MAP_PRIVATE because it is important to see data changes and to share them with other processes that map the same file. However, it is also important to prevent an unrelated process from truncating the file and so causing SIGBUS exceptions.

The one sure way to block changes to the file is to install a mandatory file lock. You place a file lock with the **lockf()** function (see the lockf(3) reference page). However, a file lock is normally "advisory"; that is, it is effective only when every process that uses the file also calls **lockf()** before changing it.

You create a mandatory file lock by changing the protection mode of the file, using the **chmod()** function to set the mandatory file lock protection bit (see the chmod(2) reference page). When this is done, a lock placed with **lockf()** is recognized and enforced by **open()**.

Mapping a File for Shared Memory

You can use **mmap()** simply to create a segment of memory that can be shared among unrelated processes:

- In one process, create a file to represent the segment.

  Typically the file will be located in */usr/tmp*, but it can be anywhere. The file permissions determine the access permitted to other processes.
- In one of the other process, use **open()** specifying the file path.
- In that other process, use **mmap()** specifying the file descriptor of the file.

Tip: The **usinit()** function is a more convenient and general way of creating a shared arena. It is based on **mmap()** but adds other services. See the usinit(3) reference page and the manual *Topics in IRIX Programming*.

You can also create a shared segment of memory using SVR4-compatible shared memory functions. See "SVR4-Compatible Shared Memory" on page 65.

Mapping a Segment of Zeros

You can use **mmap()** to create a segment of zero-filled memory. Create a file descriptor by applying **open()** to the special device file */dev/zero*. Map this descriptor with *addr* and *off* of 0 and *len* set to the segment size you want. A segment created this way cannot be shared between unrelated processes. However, it can be shared among any processes that share access to the original file descriptor. For more information about */dev/zero*, see the zero(7) reference page.

The difference between using **mmap()** and **calloc()** is that **calloc()** defines all pages of the segment immediately. When you specify MAP_AUTOGROW, **mmap()** does not actually define a page of the segment until the page is accessed. You can create a very large segment and yet consume swap space in proportion to the pages actually used.

You can find an example of code using a mapping of */dev/zero* on “Probing the Address Space” on page 222.

Mapping Physical Memory

You can use **mmap()** to create a segment that is a window on physical memory. To do so you would create a file descriptor by opening the special file */dev/mem*. For more information, see the mem(7) reference page.

Obviously the use of such a segment is dangerous as well as being both hardware-dependent and release-dependent.

Mapping Kernel Virtual Memory

You can use **mmap()** to create a segment that is a window on the kernel’s virtual address space. To do so you would create a file descriptor by opening the special file */dev/mmem* (note the double “m”). For more information, see the mem(7) (one “m”) reference page.

The possible *off* and *len* values you can use when mapping */dev/mmem* are constrained by the contents of */var/sysgen/master.d/mem*. Normally this file restricts possible mappings to specific hardware registers such as the

high-precision clock. For an example of mapping */dev/mmem*, see the code shown in “Mapping and Reading the Cycle Counter” on page 206.

Mapping a VME Device

You can use **mmap()** to create a segment that is a window on the VME bus address space. This allows you to do programmed I/O (PIO) to VME devices. (For DMA access to VME devices, see “DMA Access to Master Devices” on page 191.)

To do PIO, you create a file descriptor by opening one of the special devices in */dev/vme*. These files correspond to VME devices. For details on the naming of these files, see the uservme(7) reference page.

The name of the device that you open and pass as the file descriptor determines the bus address space (A16, A24, or A32). The values you specify in *off* and *len* must agree with accessible locations in that VME bus space. A read or write to a location in the mapped segment causes a call to the read or write entry of the kernel device driver for VME PIO. An attempt to read or write an invalid location in the bus address space causes a SIGBUS exception to all processes that have mapped the device.

Note: On the Challenge/Onyx hardware, PIO reads and writes are asynchronous. Following an invalid read or write, as much as 10 milliseconds can elapse before the SIGBUS signal is raised.

For a discussion of the bandwidth available, see “PIO Access” on page 190.

Choosing a Segment Address

Normally there is no need to map a segment to any particular virtual address. You specify *addr* as 0 and IRIX picks an unused virtual address. This is the usual method and the recommended one.

You can specify a nonzero value in *addr* to request a particular base address for the new segment. You specify MAP_FIXED in *flags* to say that *addr* is an absolute requirement, and that the segment must begin at *addr* or not be created. If you omit MAP_FIXED, **mmap()** takes a nonzero *addr* as a suggestion only.

Segments at Fixed Offsets

In rare cases you may need to create two or more mapped segments with a fixed relationship between their base addresses. This would be the case when there are offset values in one segment that refer to the other segment, as diagrammed in Figure 4-1.

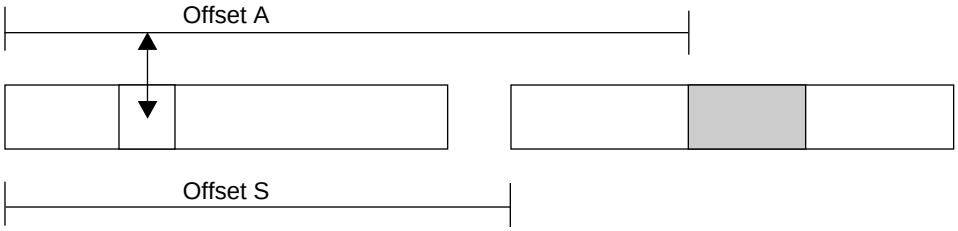

Figure 4-1 Segments With a Fixed Offset Relationship

In Figure 4-1, a word in one segment contains an offset value *A* giving the distance in bytes to an object in a different mapped segment. Offset *A* is accurate only when the two segments are separated by a known distance, offset *S*.

You can create segments in such a relationship using the following procedure.

1. Map a single segment large enough to encompass the lengths of all segments that need fixed offsets. Use 0 for *addr*, allowing IRIX to pick the base address. Let this base address be *B*.
2. Map the smaller segments over the larger one. For the first (the one at the lowest relative position), specify *B* for *addr* and MAP_FIXED in *flags*.
3. For the remaining segments, specify *B*+offset *S* for *addr* and MAP_FIXED in *flags*.

The initial, large segment establishes a known base address and reserves enough address space to hold the other segments. The later mappings replace the first one, which cannot be used for its own sake.

Segments at a Fixed Address

You can specify any value for *addr*. IRIX will create the mapping if there is no conflict with an existing segment, or return an error if necessary. However, you cannot tell what virtual addresses will be available for mapping in any particular installation or version of the operating system.

There are three exceptions. First, after IRIX has chosen an address for you, you can always map a new segment of the same or shorter length at the same address. This allows you to map different parts of a file into the same segment at different times (see "Mapping Portions of a File" on page 58).

Second, the low 4 MB of the address space are unused (see "Address Space Boundaries" on page 46). It is a very bad idea to map anything into the 0 page since that makes it hard to trap the use of uninitialized pointers. But you could use other parts of the initial 4 MB for mapping.

Third, the MIPS Application Binary Interface (ABI) specification (an extension of the System V ABI published by AT&T) states that addresses from 0x30000000 through 0x3ffc0000 are reserved for user-defined segment base addresses.

You may specify values in this range as *addr* with MAP_FIXED in *flags*. When you map two or more segments into this region, no two segments can occupy the same 256 KB unit. This rule ensures that segments will always start in different pages, even when the maximum possible page size is in use. For example, if you wanted to create two segments each of 4096 bytes, you could place them at 0x30000000 through 0x30000fff and at 0x30040000 through 0x30040fff. (256 KB is 0x00040000.)

If you follow these rules you can place segments at a known address and your program will run on any system that complies with the MIPS ABI. For an example of mapping a segment in this range, see "Probing the Address Space" on page 223.

Note: If two programs in the same system attempt to map different objects to the same absolute address, the second attempt will fail.

SVR4-Compatible Shared Memory

A different way to create a shared segment of memory is with the SVR4-compatible shared memory functions such as **shmget()** (see “SVR4-Compatible Shared Memory” on page 31). These functions enable you to create one or more unique segments of memory, each named by a numeric key, and to map them into the address space of any process.

When your aim is to share a block of memory between processes, these functions allow you to do it in a portable way. They differ from **mmap()** in the following ways:

- They identify a shared segment by an integer key rather than through a file descriptor.
- The number of segments that can be allocated controlled by system configuration parameters.
- Segments can only represent memory, not files or devices.

Locking Pages in Memory

A page fault interrupts a process for many milliseconds. Not only are page faults lengthy, their occurrence and frequency are unpredictable. If your real-time frame rate exceeds a few hertz, your program cannot tolerate such interruptions. The solution is to lock some or all of the pages of your address space into memory. A page fault cannot occur on a locked page.

Locking Functions

There are two functions that you use to lock segments into physical memory.

mpin()	Locks a specified range of pages into memory.
plock()	Locks all program text, or all data, or the entire address space.

The two functions have the same effect. They differ only in how you specify the pages to be locked. (Refer to the mpin(2) and plock(2) reference pages.)

Using **mpin()** you have to calculate the starting address and the length of the segment to be locked. It is relatively easy to calculate the starting address and length of global data or of a mapped segment, but it can be awkward to learn the starting address and length of program text or of stack space. The best use of **mpin()** is to lock mapped memory segments, since you know their starting addresses and lengths immediately after creating them.

Both **plock()** and **mpin()** define all pages of the specified segments before locking them. It is possible to receive a SIGKILL exception while locking because there was not enough swap space to define all pages (see “Delayed Definition” on page 49).

Locking pages in memory of course reduces the memory that is available for all other programs in the system. Locking a large program will increase the rate of page faults for other programs.

You use either **munpin()** or **punlock()** to unlock pages, allowing the kernel to reclaim them when necessary. Locked pages of an address space are unlocked when the last process using the address space terminates.

Locking Program Text and Data

Using **plock()** you specify whether to lock text, data, or both.

When you specify the text option, the function locks all executable text as loaded for the program, including shared objects (DSOs). (It does not lock segments created with **mmap()** even when you specify PROT_EXEC. Use mpin() to lock executable, mapped segments.)

When you specify the data option, the function locks the default data (heap) and stack segments, and any mapped segments made with MAP_PRIVATE, as they are defined at the time of the call. If you extend these segments after locking them, the newly-defined pages are also locked as they are defined.

Although new pages are locked when they are defined, you still should extend these segments to their maximum size while initializing the program. The reason is that it takes time to extend a segment: the kernel must process a page fault and create a new page frame, possibly writing other pages to backing store to make space.

One way to ensure that the full stack is created before it is locked is to call **plock()** from a function like this one:

```
#define MAX_STACK_DEPTH 100000 /* your best guess */
int call_plock()
{
   char dummy[MAX_STACK_DEPTH];
   return plock(PROCLOCK);
}
```

The large local variable forces the call stack to what you expect will be its maximum size before **plock()** is entered.

The **plock()** function does not lock mapped segments you create with MAP_SHARED. You must lock them individually using **mpin()**. You only need to do this from one of the processes that shares the segment.

Locking Mapped Files Into Memory

If you map a file before you lock the data segment into memory (see “Mapping a File for I/O” on page 57), the mapped file is read into the locked pages. If you map a file after locking the data segment, the new mapped segment is not locked. Pages of file data are read on demand, as the program accesses them. From these facts you can conclude that:

- You should map small files before locking memory, thus getting fast access to their contents without paging delays.
- Conversely, if you map a file after locking memory, your program could be delayed for input on any access to the mapped segment.
- However, if you map a large file and then try to lock memory, the attempt to lock could fail because there is not enough physical memory to hold the entire address space, including the file.

In a real-time program you cannot tolerate a delay to read a file page. However, a very large file can easily exceed the capacity of physical memory.

One alternative for a large file is to not map it, but to use conventional read and write access to it. However, this alternative forfeits the convenience of referring to the file as if it were an array in memory.

Another alternative is to map the entire file, perhaps hundreds of megabytes, into the address space, but to lock only the portion or portions that are of interest at any moment. For example, a vehicle simulator could lock the parts of a scenery file that the vehicle is approaching. When the vehicle moves away from a segment of scenery, the simulator could unlock those parts of the file, and possibly use **madvise()** to release them.

You can use **mpin()** to lock any portion of a mapped segment, and **munpin()** to unlock portions that are not needed. A call to **mpin()** implies a wait while the contents of that portion of the file are read, so this call should be made in an asynchronous process.

Reducing Cache Misses

When the frame rate is high you become concerned, not with the loss of milliseconds to a page fault, but with the loss of microseconds to a cache miss. When your program accesses instructions or data that are not in cache memory (see Figure 2-1 on “Symmetric Multiprocessor Architecture” on page 10 and “Memory Hierarchy” on page 11), the CPU requests a load of a “cache line” of 128 bytes from main memory. Possibly hundreds of CPU clock cycles pass while the cache is being loaded. Due to the pipeline architecture of the CPU, it can often continue to work during this delay. However, multiple successive cache misses can bring effective work to a halt for tens of microseconds.

In a normal program, delays due to cache misses are not noticeable because the overall average speed of the program is satisfactory. However, for a real-time program with a frame rate above 50 Hz, a cache miss can cause the unpredictable loss of a useful fraction of one frame interval.

There are three general steps you can take to improve cache use. For an example of a program in which good cache use was a top design objective, see *~4Dgifts/examples/grafix/skywriter*.

Note: In addition to the following guidelines, the IRIX kernel assists you in maintaining good cache use with special scheduling rules. See “Understanding Affinity Scheduling” on page 90.

Locality of Reference

The key to good cache performance is to maintain strong locality of reference. This can be restated as a rule of thumb: “Keep things that are used together, close together.” Or, “Extract the greatest possible use from any 128-byte cache line before touching another.” You must decide how to apply these principles in the context of your program design. Some possible techniques:

- When designing a large data structure, group small fields together at one end of the structure. Do not mix small and large fields.
- Consolidate frequently-tested switches, flags, and pointers into a single record so they will tend to stay in cache.
- Avoid searching linked lists of structures. Each time a process visits a link merely to find the address of the next link, it is likely to incur a cache miss. Worse, a search over a long list fills the cache with unneeded links, driving out important data.
- Avoid striding through a large array of structures (such as an array of graphics library objects), visiting only one or two fields in each structure. Whenever possible, arrange the data so that any sequential scan will visit and use every byte before moving on.
- Use inline function definitions for functions that are called within innermost loops. Do not use inline definitions indiscriminately, however, because they increase the total size of the binary, potentially causing more cache misses in non-looping code.
- Use **memalign()** to allocate important structures on 128-byte boundaries, so as to ensure the structures fit in the smallest number of cache lines (see the memalign(3) reference page).

Cache Mapping in Challenge/Onyx

The cache design in the Challenge / Onyx line uses a simple algorithm to assign a memory location to a cache line: the address of a byte of data is taken modulo the cache size to generate the cache address. This means that two words that are separated in main memory by an exact multiple of the cache size are always loaded to the same cache location.

Only one of the words can occupy the cache at a time, so if your program alternates between words, it will have a cache miss on each reference. It is surprisingly easy to create this situation. The following code fragment causes bad performance in a Challenge/Onyx with a 1 MB cache.

```
float part1[262144]; /* 1 MB */
float part2[262144]; /* adjacent 1 MB */
for (j=0;j<262144;++j) part1[j] = part2[j];
```

In that code fragment, the words of each array hash to the identical cache lines, so each assignment in the loop incurs two cache misses. (Some Challenge/Onyx systems have caches of different sizes, but the same principle applies.)

Note: The cache in the R8000-based POWER Challenge *does not* use simple modulus mapping; it is an associative memory that is much more resistant to cache conflicts.

Multiprocessor Cache Conflicts

As described under “Memory Hierarchy” on page 11, when one CPU modifies cached data, it broadcasts the fact on the bus. Any other CPU holding that same cache line marks it invalid. If another CPU then needs to refer to the so-called “dirty” cache line, it has to fetch the modified version from the first CPU. This takes even longer than reloading the cache line from main memory.

These conflicts can cause cache delays when the processes in two or more CPUs are working on the same data concurrently. There is no conflict so long as all CPUs are *reading* the data. Each works from its own cache copy in that case. But whenever one CPU modifies the data, all other CPUs suffer a cache miss on the same data.

In general the only way to avoid such conflicts is to separate the readers and writers in time. Arrange the program so that data is updated occasionally in a burst, then used for a longer period. When using the REACT/Pro Frame Scheduler, plan the schedule so the process that updates the data runs in a different minor frame from processes that read the data.

Detecting Cache Problems

There are relatively few tools for detecting or fixing cache problems in code. You can combine the two IRIX profiling tools, *pixie* and *prof* (see the pixie(1) and prof(1) reference pages), to arrive at a tentative diagnosis.

The *pixie* tool modifies the executable of a program so that every basic block is counted during execution. Its output ranks functions by the absolute count of instructions they executed.

The *prof* tool samples the instruction counter of the program while the program is executing. Its output ranks functions by the amount of time that the CPU spent in their code.

Normally the output of these tools should agree on the location of the hot spots in a program. However, if *prof* shows that a function is taking more time than is justified by its *pixie* execution count, that function may be running slowly due to cache-miss problems.

Additional Memory Features

Your program can work with the IRIX memory manager to change the handling of the address space.

Changing Memory Protection

You can change the memory protection of specified pages using **mprotect()** (see the mprotect(2) reference page). For a segment that contains a whole number of pages you can specify protection of:

Read-only — By making pages read-only, you cause a SIGSEGV signal to be generated in any process that tries to modify them. You could do this as a debugging measure, to trap an intermittent program error.

Read-write	You can change read-only pages back to read-write. You can put read-write protection on pages of program text, but this is bad idea except in unusual cases. For example, a debugging tool will make text pages read-write in order to set breakpoints.
Executable	Normal data pages cannot be executed. This is a protection against program errors—wild branches into data are trapped quickly. If your program constructs executable code, or reads it from a file, the protection must be changed to executable before the code can be executed.
No access	You can make pages inaccessible while retaining them as part of the address space.

Note: The **mprotect()** function changes the access rights only to memory image of a mapped file. You can apply it to the pages of a mapped file in order to control access to the file image in memory. However, **mprotect()** does not affect the access rights to the file itself, nor does it prevent other processes from opening and using the file as a file.

Synchronizing the Backing Store

IRIX writes modified pages to the backing store as infrequently as possible, in order to save time. When pages are locked, they are never written to backing store. This does not matter when the pages are ordinary data.

When the pages represent a file mapped into memory, you may want to force IRIX to write any modifications into the file. This creates a checkpoint, a known-good file state from which the program could resume.

The **msync()** function (see the msync(2) reference page) asks IRIX to write a specified segment to backing store. The segment must be a whole multiple of pages. You can optionally request

- synchronous writes, so the call does not return until the disk I/O is complete—ensuring that the data has been written
- page invalidation, so that the memory pages are released and will have to be reloaded from backing store if they are referenced again

Releasing Unneeded Pages

Using the **madvise()** function (see the madvise(2) reference page) you can tell IRIX that a range of pages is not needed by your process. The pages remain defined in the address space, so this is not a means of reducing the need for swap space. However, IRIX puts the pages at the top of its list of pages to be reclaimed when some other process (or the calling process) suffers a page fault.

The **madvise()** function is rarely needed by real-time programs, which are usually more concerned with keeping pages in memory than with letting them leave memory. However, there could be a use for it in special cases.

Chapter 5

Managing Time and Time Intervals

The topics in this chapter cover the details of using the timer and time-of-day facilities of IRIX. The emphasis is on high-precision timing facilities used by real-time programs in Challenge / Onyx systems.

Using Interval Timers

As described under “Timer Interrupts (Itimers)” on page 40, IRIX supports software interval timers of three kinds. All three are used the same way:

1. The program calls **setitimer()**, specifying a time duration and an optional repeat duration.
2. The kernel counts down the time interval against some time base.
3. When the interval has elapsed, the kernel generates a signal to the process, and optionally starts a new interval of the repeat duration.

The three kinds differ in the time base they use, the precision with which you can specify the intervals, and in the signals they send (see Table 3-2 on page 40).

Of the three, only the ITIMER_REAL, which measures elapsed time, is useful for real-time processes. The signal it generates, SIGALRM, is always delivered as soon as possible. (Other signals are not delivered until a scheduling interval occurs; see “Signal Delivery and Latency” on page 141.)

Note: Interval timers are not normally used with the Frame Scheduler. See “Using Timers with the Frame Scheduler” on page 143

Using an Itimer

Example 5-1 shows an outline of code to initialize a repeating timer signal.

Example 5-1 Timer Initialization

```
usema_t * alarmSema; /* initialized elsewhere */
void uponSigalrm()
{
   usvsema(alarmSema); /* count a period */
}
int setUpTimer(long periodInMicrosec)
{
   struct sigaction alarmActor = {SA_RESTART,uponSigalrm,0};
   struct itimerval period = {{0, 0}, {0, 0}};
   if (sigaction(SIGALRM, & alarmActor, NULL))
   {
      perror("sigaction");
      return -1;
   }
   period.it_interval.tv_usec = periodInMicrosec;
   period.it_value.tv_usec = periodInMicrosec;
   if (setitimer(ITIMER_REAL, &period, NULL))
   {
      perror("setitimer");
      return -1;
   }
   return 0; /* indicate success */
}
```

The example begins by defining a signal-handling function, **uponSigalrm()**. This handler performs the V (count-up, re*vive*) operation on a semaphore. When the main process has completed its work for one interval and is ready to wait until the next interval, it will perform the P (count-down, de*p*lete) operation on the same semaphore. (This is one of many possibilities for interaction between the signal handler and the main process. For another method based on sigsuspend(2), see “Interprocess Communication” in Appendix A on page 220.)

The periodic timer is initialized in the function **setUpTimer()**. It establishes **uponSigalrm()** as the signal handler. Then it initializes the itimer, using a period passed as an argument.

Time Signal Latency

It takes time for the kernel to deliver the SIGALRM that notifies your program at the end of the interval. (The issue of signal latency in general is discussed under “Signal Delivery and Latency” on page 141.) The signal latency is less for SIGALRM than for other signals, since the kernel initiates a scheduling cycle immediately after the timer interrupt, without waiting for the end of a fixed time slice. When the program is running or ready to run, in a CPU that has been restricted and isolated (as discussed in Chapter 6), the latency is fairly short and consistent from one signal to the next (however, it is not advisable to use a repeating itimer as the time base for a real-time program). Under less favorable conditions, signal latency can be variable and sometimes lengthy relative to a fast timer frequency.

How Timers Are Managed

The IRIX kernel can be asked to implement itimers for many processes at once, each interval having a different length and starting at a different time. The kernel’s method differs depending on the hardware architecture:

- In earlier multiprocessor architectures, there is no hardware support for interval timers, and the kernel has to rely on frequent, periodic interrupts as a time base.
- In the Challenge / Onyx and POWER-Challenge architecture, each CPU has a clock comparator that the kernel can program to cause an interrupt after a specific interval has elapsed.

Timer Management Without a Clock Comparator

In systems without a hardware clock comparator, the kernel manages interval timers using a periodic interrupt, as follows.

- The kernel keeps active *itimerval* structures in a list.
- The kernel arranges to be interrupted at a regular interval of length *T*.
- On each timer interrupt, *T* is deducted from the *it_value* field of each active *itimerval* structure.
- If the result is negative, the kernel processes the timer event: it sends the signal, and either removes the timer from the list or restarts it with a new interval, depending on *itimerval.it_interval*.

The key point is the value of the interval *T* at which the kernel updates timers. No timer interval can be shorter than *T*. The smaller the value of *T*, the more frequently the kernel must inspect all timers.

By default, *T* is one second divided by the value *HZ* defined in */usr/include/sys/param.h*. In all recent versions of IRIX, *HZ*=100, so *T*, the minimum itimer interval, is 10 milliseconds. For normal processes—that is, processes not running at a nondegrading priority in the real-time band—no interval shorter than 10 milliseconds can be scheduled. The requested interval is rounded up to whole multiples of 10 milliseconds.

Timer Management in Challenge, Onyx, and POWER-Challenge

In the Challenge / Onyx and POWER-Challenge architectures, each CPU has a hardware-operated cycle counter and a hardware comparator that generates an interrupt when the comparator register matches the cycle counter. In these systems, the kernel can manage interval timers with the minimum number of interrupts.

- The kernel keeps active *itimerval* structures in a list, sorted by ascending time until expiration.
- The kernel calculates the cycle counter value at which the next interval timer will expire, and sets this value in the comparator register.
- When the interval expires, an interrupt occurs.
- The kernel processes the timer event: it sends the signal, and either removes the timer from the list or restarts it with a new interval, depending on *itimerval.it_interval*.

In the Challenge / Onyx systems, the number of timer interrupts the kernel must handle depends only on the number of active timer requests and their repetition rates. If there are many timers, or if there are repeating timers with very short intervals, there will be many interrupts. Normally there are fewer interrupts than in a system without a clock comparator.

Using a Fast Timer Frequency

A process running at a nondegrading, real-time priority (see “Setting a Nondegrading Real-Time Priority” on page 89) can make use of an interval that is shorter than the *HZ* frequency (10 milliseconds). The actual interval that elapses is different in different versions of IRIX.

Timer requests from processes that are not running under a nondegrading real-time priority are, in all systems, rounded up to the *HZ* interval.

Fast Timers in IRIX 5.3 and After

In systems with a clock comparator, beginning with IRIX version 5.3, the *fasthz* tuning parameter has no effect. Timer requests from real-time processes are rounded up to the nearest timer unit—21 nanoseconds in the Challenge / Onyx system.

Fast Timers Prior to IRIX 5.3

In IRIX 6.0, IRIX 5.2, and earlier versions, the minimum effective timer resolution for real-time processes is set by the *fasthz* system tuning parameter. Any timer request is rounded up to the nearest multiple of the *fasthz* interval.

You can inspect the current value of *fasthz* using the command

```
grep fasthz /var/sysgen/mtune/kernel
```

The default value for a Challenge / Onyx system is 1000. That is, the minimum effective timer interval for a real-time process is 1 millisecond.

The value of *fasthz* can be set, within limits, using the *systune* command (see the systune(1) reference page). The higher it is set, the more precise your timer intervals can be.

Selecting the fasthz Value

You may need to experiment with different *fasthz* values in order to produce a repeatable, short interval. On a Challenge / Onyx system the hardware clock interval is 21 nanoseconds, which does not divide evenly into most *fasthz* intervals. The fast timer support code uses integer division, and it has some difficult problems with truncation.

For example, suppose you want to set an itimer to define a 60 Hz frame rate. You would set an itimer with a value of 16,667 microseconds.

Experience has shown that the default fasthz value of 1000 does not give good results with a 16.67 millisecond itimer. In IRIX version 5.2, this combination "jitters" on either side of the target interval. In version 5.3, itimers are guaranteed always to delay at least the specified time. While in 5.3 the interval is not shorter than 16.67 milliseconds, it is sometimes longer. Changing to a fasthz of 2400 (a multiple of the desired interval rate) does not solve the problem, which was due to integer truncation in the timer routine. However, a fasthz frequency slightly less than a multiple of the desired frame rate, 2390, does produce a dependable 16.7 millisecond interval.

Note: Itimers are *not* recommended as a way to schedule for a high frame rate. When designing a real-time program for a high frame rate the REACT / Pro Frame Scheduler offers a much more reliable and accurate way to schedule repetitive processes.

Where Timer Interrupts are Taken

Normally, interval timer interrupts are taken by the CPU in which the itimer was initialized. Beginning with IRIX 5.3, when you isolate a particular CPU (see "Isolating a CPU From TLB Interrupts" on page 104), all itimers pending for that CPU are retargeted to the assigned Clock CPU (see "Assigning the Clock Processor" on page 98). However, any new itimer requests made in an isolated CPU after it has been isolated are taken on that CPU.

For a continuously-running real-time process, it is generally best to take interval timer interrupts on the CPU where the process runs. This helps to reduce the impact of timer handling on other CPUs, and helps to reduce the time to deliver the SIGALRM.

There is a kernel tuning parameter, *itimer_on_clkcpu* (you can find it in */var/sysgen/mtune/kernel*), which if set forces all timer interrupts to be taken on the CPU that is the clock (dispatching) CPU. This parameter can hurt the performance of timer signal delivery. It should only be set when it is crucial that a process see an exact, consistent relationship between itimer intervals and time of day stamps.

Fast Timers in Older Architectures

In systems that have no clock comparator, the kernel implements fast timers by shortening the periodic timer interval to the *fasthz* frequency when necessary. When a real-time process sets a timer at an interval that is not a multiple of HZ, the kernel initiates periodic timer interrupts at the *fasthz* rate, for example 1000 times per second.

The frequent interrupts needed to support fast timers cause an overhead load of several percent of the power of one CPU. In these systems you should try to design your real-time application to use intervals that are multiples of 10 milliseconds, so that the fast interrupt rate is not needed. When that is not feasible, you can assign the interrupt to a particular CPU (see “Assigning the fasthz Processor” on page 98).

Avoiding Timer Interrupts Before IRIX 5.3

Beginning with IRIX 5.3, when you isolate a particular CPU, all pending itimers for that CPU are retargeted to the assigned Clock CPU. However, in IRIX 6.0, 5.2, and earlier, existing interval timers are not moved; they remain on the original CPUs, which continued to receive interrupts for those timers. If the itimers were repeating, the interrupts continued indefinitely.

Certain daemons set up repeating interval timers when they are initialized. These timers continue to interrupt the CPUs on which the daemons were running when they set the timers. In releases before 5.3, the only way to be sure that a CPU will never receive fast timer interrupts is to restrict that CPU early in the boot process (see “Restricting a CPU From Scheduled Work” on page 101). For example, a shell script named *S05restrict* could be placed in */etc/rc2.d* containing the line

```
mpadmin -r 1 #set CPU 1 to restricted state
```

and additional lines as required to isolate all CPUs used for real-time work.

Using Timestamps

There are two sources of timestamps, as described earlier in “Timestamps” on page 41. They differ in their accessibility, precision, and accuracy.

Using the Time of Day

The **gettimeofday()** function returns the time as two long integers in a *timeval* structure. (For details of the call, refer to the gettimeofday(3) reference page.)

```
struct timeval {
   long tv_sec; /* seconds since Jan. 1, 1970 */
   long tv_usec; /* additional microseconds */
}
```

The nominal resolution of this timestamp is 1 microsecond. However, it is not practical for the kernel to update an internal timestamp with this frequency. The actual timestamp value is updated at intervals that are convenient to the kernel. The intervals are not regular; however they are never greater than 10 milliseconds.

This does not mean that successive calls to **gettimeofday()** return the same value. On the contrary, experimentation reveals that successive calls always return different values. Usually the difference is 1 in the *tv_usec* field. However, when the kernel updates the timestamp between two successive calls, the difference is greater. See the sample program under “Getting the Time of Day Stamp” on page 211. A similar program, using two child processes getting the time alternately, showed that it was impossible to get the identical timestamp value even in different processes.

Normally an IRIX system uses one of the time-synchronization daemons, either *timed* or *timeslave*, to keep the local clock accurate. These daemons use **adjtime()** to adjust the time of day when necessary. (See the timed(1), timeslave(1), and adjtime(2)reference pages.) Thus the timestamp returned by **gettimeofday()** should reflect the true local time, with an inaccuracy of at worst -10 milliseconds.

Either the *date* command or the time daemon can adjust the current time by a negative increment. As a result, **gettimeofday()** can in rare circumstances return duplicate values (see the date(1) reference page).

Using the Cycle Counter

All Silicon Graphics systems have a free-running counter that is updated by hardware at a high frequency. You can map the image of this counter into the process address space, then sample its value as an integer. (For a discussion on mapping segments of memory, see “The Segment Mapping Function mmap()” on page 54 and the following topics.)

The precision of the cycle counter depends on the hardware system. In the Challenge / Onyx line it is a 64-bit integer. The frequency that it counts also varies with the system. In the Challenge / Onyx, it is 21 nanoseconds (47.6 MHz).

You obtain the virtual address of the cycle counter (in the kernel’s virtual address space) using **syssgi()**, in a call that also returns the clock precision in picoseconds (1e-12 seconds, millionths of a microsecond). See the syssgi(2) reference page. For an example program, see “Mapping and Reading the Cycle Counter” on page 202.

Since the update frequency of the cycle counter is close to the maximum CPU instruction rate, it is not possible to read the same value from it twice.

However, the cycle counter is simply a free-running hardware device; it is not synchronized with any corrected time base. Its drift rate can be as high as 1 part in 10,000—100 microseconds per second, or approximately 8 seconds per day.

Comparing the Timestamps

The two timestamp sources can be compared on several different attributes, listed in Table 5-1.

Table 5-1 Comparison of Timestamp Functions

Timestamp	Overhead	Precision	Accuracy	Drift
gettimeofday()	System call (100s of instructions)	1 microsecond	+0, -10 milliseconds	1 part in 10,000 short-term; corrected to negligible amount over long periods
cycle counter	negligible	21 nanoseconds (Challenge)	instruction cycle time	1 part in 10,000, varying

Because **gettimeofday()** is synchronized to a time standard, you must use it when you want to record the actual time of an event, and when times recorded in one machine will be compared to times recorded in another. You should use it to measure durations of minutes and longer; in those cases its possible error of up to -10 milliseconds becomes less important than its resistance to long-term drift.

Because the cycle counter can be sampled with negligible overhead and has very high precision, you should use it whenever you want to measure durations of seconds or less, and whenever you simply want a source of unduplicated, monotonically-increasing, unsigned numbers to provide unique key values.

Chapter 6

Controlling CPU Workload

This chapter describes how to use IRIX 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. The main topics covered are:

- "Using Priorities and Scheduling Queues" on page 85 covers scheduling concepts, tells how to set nondegrading priorities, and explains affinity scheduling, gang scheduling, and deadline scheduling.
- "Using Processor Sets" on page 94 describes how to define sets of CPUs and how to assign them to specific kinds of work.
- "Minimizing Overhead Work" on page 97 discusses how to remove all unnecessary interrupts and overhead work from the CPUs that you want to use for real-time programs.
- "Minimizing Interrupt Response Time" on page 107 discusses the components of interrupt response time and how to minimize them.

Using Priorities and Scheduling Queues

The default IRIX scheduling algorithm is designed for a conventional time-sharing system, in which the best results are obtained by favoring I/O-bound processes and discouraging CPU-bound processes. However, the latest versions of IRIX for the Challenge/Onyx systems support a variety of scheduling disciplines that are optimized for parallel processes. You can take advantage of these in different ways to suit the needs of different programs.

Note: You can use the methods discussed here to make a real-time program more predictable. However, to reliably achieve a high frame rate, you should plan to use the REACT/Pro Frame Scheduler described in Chapter 7.

Scheduling Concepts

In order to understand the differences between scheduling methods you need to know some basic concepts.

Tick Interrupts

In normal operation, the kernel pauses to make scheduling decisions every 10 milliseconds in every CPU. The duration of this interval, which is called the “tick” because it is the metronomic beat of the scheduler, is defined in *sys/param.h*. Every CPU is normally interrupted by a timer every tick interval. The kernel updates accounting values, checks for superior-priority processes that have become ready to run, and does other housekeeping work. Before it returns to the interrupted process, the kernel checks for pending signals, and may divert the process into a signal handler.

You can stop the tick interrupt in selected CPUs in order to keep these interruptions from interfering with real-time programs—see “Making a CPU Nonpreemptive” on page 106.

Time Slices

Each process has a guaranteed time slice, which is the amount of time it is normally allowed to execute without being preempted. By default the time slice is 3 ticks, or 30 ms. Often, a typical process will be blocked for I/O before reaching the end of its time slice.

At the end of a time slice, the kernel chooses which process to run next on the same CPU based on process priorities. When runnable processes have the same priority, the kernel runs them in turn.

Priorities

Every process that is ready to run (not blocked on I/O or a semaphore) is listed in a queue of processes. (There are actually multiple queues, as described in a later topic.) Every process has a priority and a “nice” value. When a CPU needs a process to run, it normally takes the one with the lowest sum of priority and nice value. Thus a *lower-numbered* priority value gives a process a *superior priority* to run.

The specific priority values are shown in Table 6-1. The constant identifiers are defined in *sys/schedctl.h*.

Table 6-1 Priority Ranges

Numeric Range	Purpose	Identifiers
30 … 39	Real-time and other high-priority processes	NDPHIMAX … NDPHIMIN
40 … 127	Normal user processes with degrading priorities	NDPNORMAX … NDPNORMIN
40 … 127	Processes with assigned, nondegrading priorities	NDPNORMAX … NDPNORMIN
128 … 254	Batch jobs and other low-priority processes	NDPLOMAX … NDPLOMIN

Note that the names ending in MAX correspond to the lowest numbers. This reflects the fact that processes with *lower* priority values have *superior* priority for use of the system; while those with *higher* values have *inferior* priority.

Aging Priorities

In order to favor I/O bound processes and to penalize CPU-bound processes, IRIX "ages" or "degrades" the priority of any normal process as it runs. The longer a process runs without blocking, the worse its priority becomes. When the process finally suspends voluntarily (to wait for I/O or some event), its priority is restored.

Scheduler Queues

The kernel maintains not one but several different scheduling queues, each containing processes that are scheduled under a different set of rules. These rules are covered in the following topics. The queues are listed in Table 6-2.

Table 6-2 Scheduler Queues

Queue	Processes and Discipline
Kernel	Kernel code
Real-time	Processes with fixed priorities between 30 and 39
Time-sharing	Processes with priorities between 40 and 127 (priorities in this range can be either degrading or nondegrading)
Batch	Batch processes with priorities between 128 and 254
Deadline	Processes under the deadline scheduling rules
Gang	Processes under the gang scheduling rules with priorities less than 128
Gang-batch	Processes under the gang scheduling rules with priorities of 128 or greater

You can list the names of the queues and their associated priority-range numbers using

```
pset -q
```

Setting a Nondegrading Batch Priority

Any user can give create a process with a nondegrading priority in the batch range. This is done with the *npri* command (see the npri(1) reference page).

```
npri -h 129 echo hello from the Batch queue
```

The specified command executes with fixed priority 129 (in this example). The same priority change could be performed within the program using **schedctl()** (see the schedctl(2) reference page), as shown in Example 6-1.

Example 6-1 Setting a Nondegrading Batch Priority

```
if (-1 == schedctl(NDPRI,0,129))
   { perror("most unlikely error"); }
```

The smallest numerical value a regular user can set in these ways is established by the system tuning parameter *ndpri_hilim*. To see its value use

```
fgrep ndrpi_hilim /var/sysgen/mtune/disp
```

Typically *ndpri_hilim* is set to 128, the superior priority within the batch range. The system administrator could change the limit to a smaller number, allowing ordinary users to set nondegrading priorities that compete with interactive processes, or even with real-time processes.

Setting a Nondegrading Real-Time Priority

With superuser privilege a user can create a process that executes in the real-time band of priorities.

```
npri -h 38 sh ~rtuser/bin/realtime.sh
```

The same change can be effected from within a process using **schedctl()**, as shown in Example 6-2.

Example 6-2 Setting a Real-Time Priority

```
if (-1 == schedctl(NDPRI,0,38))
{
   if (EPERM == errno)
      fprintf(stderr,"You forget to suid again\n");
   else
      perror("schedctl");
}
```

The real-time priorities are those numerically less than or equal to the system tuning parameter *ndpri_lolim*, which is normally 39.

The kernel guarantees that a runnable process with one of these priorities will never sit idle waiting for a process with a lower priority.

The preemptible network daemon, *rtnetd*, which is used by default on multiprocessor systems, normally runs at a nondegrading priority of 39. If you give your real-time process a superior (numerically lower) priority value, it cannot be preempted by network I/O.

Caution: If a process with a real-time priority goes into a loop, it can monopolize its CPU, excluding all other processes.

On a multiprocessor system a runaway real-time process is not the disaster it would be on a uniprocessor. You can kill the looping process with a command executed on another CPU. However, if you have isolated all but one CPU, for example by running the Frame Scheduler on all other CPUs, a high-priority process on the remaining CPU can lock the entire system. A looping process with priority 30 can lock out all other processes, including network and NFS daemons and the X-server, making the system unusable.

Understanding Affinity Scheduling

Affinity scheduling is a special scheduling discipline used in the Challenge/Onyx systems. You do not have to take action to benefit from affinity scheduling, but you should know that it is done.

As a process executes, it causes more and more of its data and instruction text to be loaded into the processor cache (see “Reducing Cache Misses” on page 68). This creates an “affinity” between the process and the CPU. No other process can use that CPU as effectively, and the process cannot execute as fast on any other CPU.

The IRIX kernel notes the CPU on which a process last ran, and notes the amount of the affinity between them. Affinity is measured as the amount of time the process used the CPU, with 300 microseconds or less having zero affinity, and 10 milliseconds or more having 100% affinity.

When the process gives up the CPU—either because its time slice is up or because it is blocked—one of three things will happen to the CPU:

- The CPU runs the same process again immediately.
- The CPU spins idle waiting for work.

- The CPU runs a different process.

The first two actions do not reduce the process's affinity. But when the CPU runs a different process, that process begins to build up an affinity while simultaneously reducing the affinity of the earlier process.

As long as a process has any affinity for a CPU, it is dispatched only on that CPU if possible. When its affinity has declined to zero, the process can be dispatched on any available CPU. The result of the affinity scheduling policy is that:

- I/O-bound processes, which execute for short periods and build up little affinity, are quickly dispatched whenever they become ready.
- CPU-bound processes, which build up a strong affinity, are not dispatched as quickly because they have to wait for "their" CPU to be free. However, they do not suffer the serious delays of repeatedly "warming up" a cache.

Using Gang Scheduling

You have been advised to design a real-time program as a family of cooperating, lightweight processes sharing an address space (see, for example, "Lightweight Process Creation With sproc()" on page 17). These processes typically coordinate their actions using locks or semaphores ("Interprocess Communication" on page 30).

When process A attempts to seize a lock that is held by process B, one of two things will happen, depending on whether or not process is B is running concurrently in another CPU.

- If process B is not currently active, process A spends a short time in a "spin loop" and then is suspended. The kernel selects a new process to run. Time passes. Eventually process B runs and releases the lock. More time passes. Finally process A runs and now can seize the lock.
- When process B is concurrently active on another CPU, it typically releases the lock while process A is still in the spin loop. The delay to process A is negligible, and the overhead of multiple passes into the kernel and out again is avoided.

In a system with many processes, the first scenario is common even when processes A, B, and their siblings have real-time priorities. Clearly it would be better if processes A and B were always dispatched concurrently.

Gang scheduling achieves this. Any process in a share group can initiate gang scheduling. Then all the processes that share that address space are scheduled as a unit. IRIX tries to ensure that all the members of the share group are dispatched when any one of them is dispatched.

You initiate gang scheduling with a call to **schedctl()**, as sketched in Example 6-3

Example 6-3 Initiating Gang Scheduling

```
if (-1 == schedctl(SCHEDMODE,SGS_GANG))
{
   if (EPERM == errno)
      fprintf(stderr,"You forget to suid again\n");
   else
      perror("schedctl");
}
```

You can turn gang scheduling off again with another call, passing SGS_FREE in place of SGS_GANG.

Using Deadline Scheduling

You can apply the deadline scheduling discipline to any process that must be assured of receiving a certain amount of execution time out of every interval, regardless of what other processes are running.

A process with normal or batch priority might enjoy a lot of execution time under light system load, but might be held idle for long periods under heavy system load. A process with a high, nondegrading priority is assured of getting all the execution time it can use, but it can monopolize resources. Deadline scheduling is best for a process that must have a certain minimum amount of time, but which should use little or none of the remainder of the time.

It requires no special privilege to assign deadline scheduling to a process. You can do it with the *npri* command. The following command schedules a shell script to execute at least 20% of each 100-millisecond interval.

```
npri -d 100,20 /usr/local/bin/deadline.prog
```

If the system cannot dedicate the requested amount of time, the command returns an error. Otherwise, the process is guaranteed the specified amount of time per interval.

Note: Execution time can be given at any point within the interval, and need not be continuous. Thus deadline scheduling cannot be used as the basis for a reliable real-time frame rate.

A deadline guarantee is not inherited by processes created by **fork()** or sproc(). Each process must have deadline scheduling set for it.

The **schedctl()** call is used to set deadline scheduling within a process, and to choose a rule for what the process should do in the balance of the interval, after it has achieved its target percentage:

DL_ONLY	Process should be idle the rest of the period.
DL_ANY	Process should execute under the normal rules for its priority and nice value.

For an example of using **schedctl()** to set deadline scheduling, see “Deadline Scheduling Subroutines” on page 224.

Note: The kernel uses a high-precision interval timer to measure usage under deadline scheduling. When deadline scheduling is in use, more frequent timer interrupts are generated. In some architectures this causes frequent kernel interrupts (see “Fast Timers in Older Architectures” on page 81 and “Assigning the fasthz Processor” on page 98).

Deadline scheduling was first supported in IRIX 5.0.

Changing the Time Slice Duration

You can change the length of the time slice for all processes from its default 30ms using the *systune* command (see the systune(1) reference page). The kernel variable is *slice_length*; its value is the number of tick intervals that comprise a slice. There is probably no good reason to make a global change of the time-slice length.

You can change the length of the time slice for one particular process using the **schectl()** function (see the schedctl(2) reference page). The code would resemble Example 6-4.

Example 6-4 Setting the Time-Slice Length

```
#include <sys/schedctl.h>
int setMyTimeSliceInTicks(const int ticks)
{
   int ret = schedctl(SLICE,0,ticks)
   if (-1 == ret)
      { perror("schedctl(SLICE)"); }
   return ret;
}
```

You might lengthen the time slice for the parent of a process group that will be gang-scheduled (see “Using Gang Scheduling” on page 91). This will keep members of the gang executing concurrently longer.

Using Processor Sets

A processor set is a group of 1 or more designated CPUs. You define a processor set and apply it using *pset* (see the pset(1) reference page). A processor set is identified by an integer that you assign. For example, to create set 1357 containing all odd-numbered CPUs in an 8-CPU system, use:

```
pset -s 1357 1,3,5,7
```

You can also define processor sets in a file, */etc/psettab*, so they are defined at all times. With root privilege, you can create any number of processor sets. Sets can be disjoint or overlapping.

With root privilege, you can use processor sets in several ways to partition the system workload.

Tip: Most of the variants of the *pset* command have a functional equivalent in the **sysmp(**MP_PSET**)** function. For details, refer to the sysmp(2) reference page.

Assigning a Process to a Processor Set

Using *pset* you can assign a designated process or command to execute on a specified processor set only. For example, you can run a shell script on CPUs 2 and 3 this way:

```
pset -s 10023 2,3
pset -c 10023 /bin/csh ~/runreal.csh
```

The created process (and any processes it might create) runs only on the CPUs in that group. Those CPUs are available to run other processes as well.

A user with administrator privilege can call the **schedctl()** function to associate a process with a specified processor set. This assignment is inherited over a **fork()**—so if it is applied to a shell process, all the commands run from that shell are also assigned to the processor set. This gives somewhat more control over the assignment than does *pset*. Example 6-5 shows the absolute minimum command code.

Example 6-5 Command to Assign Process to Processor Set

```
#include <limits.h>
#include <sys/types.h>
#include <sys/prctl.h>
#include <sys/schedctl.h>
main(int argc, char **argv)
{
    if (argc != 3) exit(-1);
    if (-1 == schedctl(SETPSET, atoi(argv[1]), atoi(v[2])))
        perror(“schedctl:”);
}
```

The code in Example 6-5 can clearly be extended and improved with more detailed diagnostics and with security checks. An additional function, to invoke **schedctl(**UNSETPSET**)** when the second argument is -1, would be useful. However, a command of this sort can be used by a system operator, or, with setuid permission, in login scripts.

Tip: Keep in mind that affinity scheduling tends to keep a CPU-bound process on one CPU in any case (see “Understanding Affinity Scheduling” on page 90). In general, the dynamic operation of the IRIX scheduler, guided by a nondegrading priority or deadline scheduling, can do a better job of allocating CPUs to processes than you can do with a static assignment through *pset*.

Assigning a Processor Set to a Queue

Using *pset* you can assign a set of CPUs to service a particular scheduling queue (see “Scheduler Queues” on page 88). Only those CPUs will take processes from that queue, and they will take processes from no other queue. For example, to assign a single CPU to service the batch queue, use:

```
pset -s 1009 9
pset -q bt 1009
```

Only CPU 9 will work on processes with a batch-level priority, and CPU 9 will work on no other processes.

If you assign a processor set to the gang or gang-batch queue, the set should contain enough CPUs to match the size of the largest gang. (Assigning a single CPU to the gang queue would rather defeat the purpose of gang scheduling.)

Assigning a Discipline to a Processor Set

The kernel recognizes the concept of a scheduling discipline apart from the queues and methods mentioned already. At present only one special discipline is defined: the Graphics discipline, which includes all processes that open a graphics pipe.

You can use *pset* to assign a processor group to service only processes that use a particular discipline, without regard for the queue they are in.

Processor Set Contradictions

You can create contradictions using the *pset* command. For example, you can assign a processor set to the gang-scheduled queue, and also assign a normal or real-time process to that same processor set. Since the assigned process is not gang-scheduled, it will never appear in the queue that the processor group can service. Since the process is assigned to that group, it can run on no other CPUs. Accordingly, the process never runs at all. You have to change one of the assignments before the process can even terminate.

It is also possible to create an empty set, one with no CPUs assigned to it. Processes or queues that depend on that set simply do not execute. Some users consider this to be a feature, not an problem. For example, if the processor set servicing the batch queue is made empty, batch-queue work—even active, half-completed programs—simply sit and do not execute. At some later time, *pset* is used to take CPUs from some other processor set and reassign them to the batch queue set, at which time the unserviced jobs begin to execute again.

Minimizing Overhead Work

A certain amount of CPU time must be spent on general housekeeping. Since 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 (regardless of whether it will be scheduled normally, or as a gang, or by the Frame Scheduler). Then apply the following steps to isolate and restrict those CPUs. The steps are independent of each other. Each needs to be done to completely free a CPU.

Assigning the Clock Processor

Every CPU that uses normal IRIX scheduling takes a “tick” interrupt that is the basis of process scheduling. However, one CPU does additional housekeeping work for the whole system, on each of its tick interrupts. You can specify which CPU has these additional duties using the privileged *mpadmin* command (see the mpadmin(1) reference page). For example, to make CPU 0 the clock CPU (a common choice), use

```
mpadmin -c 0
```

The equivalent operation from within a program uses **sysmp()** as shown in Example 6-6 (see also the sysmp(2) reference page).

Example 6-6 Setting the Clock CPU

```
#include <sys/sysmp.h>
int setClockTo(int cpu)
{
   int ret = sysmp(MP_CLOCK,cpu);
   if (-1 == ret) perror("sysmp(MP_CLOCK)");
   return ret;
}
```

Assigning the fasthz Processor

When high precision timers are used, timer interrupts occur more frequently. In machines that lack a clock comparator, fast timer interrupts cause a lot of overhead processing (see “Fast Timers in Older Architectures” on page 81). A particular CPU can be designated to handle this work. You can use the -f parameter of *mpadmin* to find out which CPU has responsibility:

```
% mpadmin -f
0
```

With root privilege, *mpadmin* can be used to specify the CPU to handle the fast timer.

```
mpadmin -f 1
```

The equivalent operation from software uses **sysmp()**, as shown in Example 6-7.

Example 6-7 Setting the fasthz CPU

```
#include <sys/sysmp.h>
int setFasthzTo(int cpu)
{
   int ret = sysmp(MP_FASTCLOCK,cpu);
   if (-1 == ret) perror("sysmp(MP_FASTCLOCK)");
   return ret;
}
```

Note: On Challenge / Onyx and POWER-Challenge architectures, assigning the fasthz CPU is allowed but has no effect. Timer interrupts are taken only as required, not at the *fasthz* rate, and are targeted to the CPU where they were initiated.

Unavoidable Timer Interrupts

Prior to IRIX version 5.3, even when the clock and fast timer duties were removed from a CPU, that CPU still received a timer interrupt approximately every 42 seconds. This was the result of the maximum value, 0x7fffffff, counting down in a hardware timer. The resulting interrupt was processed in the normal timer-handling code, which used nearly 100 microseconds before recognizing the interrupt as unwanted.

Thus in IRIX 5.2 and IRIX 6.0, every CPU gets a 100 microsecond interrupt every 42 seconds. This can interfere with the timing of a real-time program with a high frame rate, or can extend the latency of an interrupt handler.

Starting in IRIX 5.3 and IRIX 6.0.1, the interrupt frequency is halved, to approximately every 80 seconds. More important, a fast path in the timer code recognizes the unwanted interrupt and exits in 5 microseconds. Thus in these later systems, the only unwanted interrupt in an isolated CPU is a 5 microsecond “blip” every 80 seconds. Processes running under the Frame Scheduler are not affected even by this small interrupt.

Isolating a CPU From Sprayed Interrupts

By default, the Challenge / Onyx hardware directs I / O interrupts to CPUs in rotation (called *spraying interrupts*). You do not want a real-time process interrupted at unpredictable times to handle I / O. The system administrator can isolate one or more CPUs from sprayed interrupts by placing the NOINTR statement in the configuration file */var/sysgen/system/irix.sm*. The syntax is

`NOINTR` *cpu#* [*cpu#*]...

After modifying *irix.sm*, rebuild the kernel using the command */etc/autoconfig -vf*.

Assigning Interrupts to CPUs

To minimize the latency of real-time interrupts, you can arrange for the VME-bus interrupts with real-time significance to be delivered to a specified CPU where no other interrupts are handled. This is done with the IPL (Interrupt Priority Level) statement in the */var/sysgen/system/irix.sm* file. The syntax is

`IPL` *level# cpu#*

Interrupts from the specified level of the VME bus will be delivered to the specified CPU. After modifying *irix.sm*, rebuild the kernel using the command */etc/autoconfig -vf*.

For more on how to handle time-critical interrupts see “Minimizing Interrupt Response Time” on page 107).

The best way to handle non-critical interrupts is to allow the hardware to “spray” them to all available CPUs. You can protect specific CPUs from interrupts as discussed under “Isolating a CPU From Sprayed Interrupts” on page 100.

Understanding the Vertical Sync Interrupt

In systems with dedicated graphics hardware, the graphics hardware generates a variety of hardware interrupts. The most frequent of these is the vertical sync interrupt, which marks the end of a video frame. The vertical sync interrupt can be used by the Frame Scheduler as a time base (see “Vertical Sync Interrupt” on page 126). Certain GL and Open GL functions are internally synchronized to the vertical sync interrupt (for an example, refer to the gsync(3g) reference page).

All the interrupts produced by dedicated graphics hardware are at an inferior priority compared to other hardware. All graphics interrupts including the vertical sync interrupt are directed to CPU 0. They are not “sprayed” in rotation, and they are not configurable.

Restricting a CPU From Scheduled Work

For best performance of a real-time process or for minimum interrupt response time, you need to use one or more CPUs without competition from other scheduled processes. There are three levels of increasing control you can exert: *restricted*, *isolated*, and *nonpreemptive*.

In general, the IRIX scheduling algorithms will use any CPU to run a process that is ready to run. This is modified by considerations of

- affinity—CPUs are made to execute the processes that have developed affinity to them
- processor group assignments—the *pset* command can force a specified group of CPUs to service only a given scheduling queue

You can *restrict* one or more CPUs from running any scheduled processes at all. The only processes that can use a restricted CPU are processes that you assign to those CPUs.

Note: Restricting a CPU overrides any assignment made with *pset*. The restricted CPU remains in its group, but does not perform any work you assign using *pset*.

You can find out the number of CPUs that exist, and the number that are still unrestricted, using the **sysmp()** function as in Example 6-8.

Example 6-8 Number of Processors Available and Total

```
#include <sys/sysmp.h>
int CPUsInSystem = sysmp(MP_NPROCS);
int CPUsNotRestricted = sysmp(MP_NAPROCS);
```

To restrict one or more CPUs, you can use *mpadmin*. For example, to restrict CPUs 4 and 5, you can use

```
mpadmin -r 4
mpadmin -r 5
```

The equivalent operation from within a program uses **sysmp()** as in Example 6-9 (see also the sysmp(2) reference page).

Example 6-9 Restricting a CPU

```
#include <sys/sysmp.h>
int restrictCpuN(int cpu)
{
   int ret = sysmp(MP_RESTRICT,cpu);
   if (-1 == ret) perror("sysmp(MP_RESTRICT)");
   return ret;
}
```

You remove the restriction, allowing the CPU to execute any scheduled process, with *mpadmin -u* or with **sysmp(**MP_EMPOWER**)**.

Note: The following points are important to remember:

- Prior to IRIX 5.3, you need to restrict CPUs very early in the boot process. See “Avoiding Timer Interrupts Before IRIX 5.3” on page 81.
- The CPU assigned to handle the scheduling clock (“Assigning the Clock Processor” on page 98) must not be restricted.
- The REACT/Pro Frame Scheduler automatically restricts and isolates any CPU it uses. See Chapter 7.

Assigning Work to a Restricted CPU

Having restricted a CPU, you can assign processes to it using the command *runon* (see the runon(1) reference page). For example, to run a program on CPU 3, you could use

```
runon 3 ~rt/bin/rtapp
```

The equivalent operation from within a program uses **sysmp()** as in Example 6-10 (see also the sysmp(2) reference page).

Example 6-10 Assigning the Calling Process to a CPU

```
#include <sys/sysmp.h>
int runMeOn(int cpu)
{
   int ret = sysmp(MP_MUSTRUN,cpu);
   if (-1 == ret) perror("sysmp(MP_MUSTRUN)");
   return ret;
}
```

Note: If you assign a process to a CPU that has been given other work with *pset*, the process can be blocked indefinitely. See “Processor Set Contradictions” on page 97.

You remove the assignment, allowing the process to execute on any available CPU, with **sysmp(**MP_RUNANYWHERE**)**. There is no command equivalent.

The assignment to a specified CPU is inherited by processes created by the assigned process. Thus if you assign a real-time program with *runon*, all the processes it creates run on that same CPU. More often you will want to run multiple processes concurrently on multiple CPUs. There are three approaches you can take:

1. Use the REACT/Pro Frame Scheduler, letting it restrict CPUs for you.
2. Let the parent process be scheduled normally using a nondegrading real-time priority. After creating child processes with **sproc()**, use **schedctl(**SCHEDMODE,SGS_GANG**)** to cause the share group to be gang-scheduled. Assign a processor group to service the gang-scheduled process queue.

   The CPUs that service the gang queue cannot be restricted. However, if yours is the only gang-scheduled program, those CPUs will effectively be dedicated to your program.
3. Let the parent process be scheduled normally. Let it restrict as many CPUs as it will have child processes. Have each child process invoke **sysmp(**MP_MUSTRUN,*cpu***)** when it starts, each specifying a different restricted CPU.

Isolating a CPU From TLB Interrupts

As described under “Translation Lookaside Buffer Updates” on page 13, when the kernel changes the address space in a way that could invalidate TLB entries held by other CPUs, it broadcasts an interrupt to all CPUs, telling them to update their translation lookaside buffers (TLBs).

You can *isolate* the CPU so that it does not receive broadcast TLB interrupts. When you isolate a CPU, you also restrict it from scheduling processes. Thus isolation is a superset of restriction, and the comments in the preceding topic, “Restricting a CPU From Scheduled Work” on page 101, also apply to isolation.

The command is *mpadmin -I*; the function is **sysmp(**MP_ISOLATE, *cpu#***)**. After isolation, the CPU will synchronize its TLB and instruction cache only when a system call is executed. This removes one source of unpredictable delays from a real-time program and helps minimize the latency of interrupt handling.

Note: The REACT / Pro Frame Scheduler automatically restricts and isolates any CPU it uses.

When an isolated CPU executes only processes whose address space mappings are fixed, it receives no broadcast interrupts from other CPUs. Actions by processes in other CPUs that change the address space of a process running in an isolated CPU can still cause interrupts at the isolated CPU. Among the actions that change the address space are:

- Causing a page fault. When the kernel needs to allocate a page frame in order to read a page from swap, and no page frames are free, it invalidates some unlocked page. This can render TLB and cache entries in other CPUs invalid. However, as long as an isolated CPU executes only processes whose address spaces are locked in memory, such events cannot affect it.
- Extending a shared address space with **brk()**. Allocate all heap space needed before isolating the CPU.

- Using **mmap()**, **munmap()**, **mprotect()**, **shmget()**, or **shmctl()** to add, change or remove memory segments from the address space; or extending the size of a mapped file segment when MAP_AUTOGROW was specified and MAP_LOCAL was not. All memory segments should be established before the CPU is isolated.
- Starting a new process with **sproc()**, thus creating a new stack segment in the shared address space. Create all processes before isolating the CPU; or use **sprocsp()** instead, supplying the stack from space allocated previously.
- Accessing a new DSO using **dlopen()** or by reference to a delayed-load external symbol (see the dlopen(3) and DSO(5) reference pages). This adds a new memory segment to the address space but the addition is not reflected in the TLB of an isolated CPU.
- Calling **cacheflush()** (see the cacheflush(2) reference page).
- Using DMA to read or write the contents of a large (many-page) buffer. For speed, the kernel temporarily maps the buffer pages into the kernel address space, and unmaps them when the I/O completes. However, these changes affect only kernel code. An isolated CPU processes a pending TLB flush when the user process enters the kernel for an interrupt or service function.

Isolating a CPU When Performer™ Is Used

The Performer™ graphics library supplies utility functions to isolate CPUs and to assign Performer processes to the CPUs. You can read the code of these functions in the file */usr/src/Performer/src/lib/libpfutil/lockcpu.c*. They use CPUs starting with CPU number 1 and counting upward. The functions can restrict as many as 1+2**pipes* CPUs, where *pipes* is the number of graphical pipes in use (see the pfuFreeCPUs(3pf) reference page for details). The functions assume these CPUs are available for use.

If your real-time application uses Performer for graphics—which is the recommended approach for high-performance simulators—you should use the libpfutil functions with care. Possibly you will need to replace them with functions of your own. Your functions can take into account the CPUs you reserve for other time-critical processes. If you already restrict one or more CPUs, you can use a Performer utility function to assign Performer processes to those CPUs.

Making a CPU Nonpreemptive

After a CPU has been isolated, you can turn off the dispatching "tick" for that CPU (see "Tick Interrupts" on page 86). This eliminates the last source of overhead interrupts for that CPU. It also ends preemptive process scheduling for that CPU. This means that the process now running will continue to run until

- it gives up control voluntarily by blocking on a semaphore or lock, requesting I/O, or calling **sginap()**
- it calls a system function and, when the kernel is ready to return from the system function, it finds a process of higher priority is ready to run

Some effects of this change within the specified CPU include the following:

- IRIX will no longer age degrading priorities. Priority ageing is done on clock tick interrupts.
- IRIX will no longer preempt a low-priority process when a high-priority process becomes runnable, except when the low-priority process calls a system function.
- Signals (other than SIGALARM) can only be delivered after I/O interrupts or on return from system calls. This can extend the latency of signal delivery.

Normally an isolated CPU runs only a few, related, time-critical processes that have equal priorities, and that coordinate their use of the CPU through semaphores or locks. When this is the case, the loss of preemptive scheduling is outweighed by the benefit of removing the overhead and unpredictability of interrupts.

To make a CPU nonpreemptive you can use *mpadmin*. For example, to isolate CPU 3 and make it nonpreemptive, you can use

```
mpadmin -I 3
mpadmin -D 3
```

The equivalent operation from within a program uses **sysmp()** as shown in Example 6-11 (see the sysmp(2) reference page).

Example 6-11 Making a CPU nonpreemptive

```
#include <sys/sysmp.h>
int stopTimeSlicingOn(int cpu)
{
   int ret = sysmp(MP_NONPREEMPTIVE,cpu);
   if (-1 == ret) perror("sysmp(MP_NONPREEMPTIVE)");
   return ret;
}
```

You reverse the operation with **sysmp(**MP_PREEMPTIVE**)** or with *mpadmin* -C.

Minimizing 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. IRIX guarantees a maximum interrupt response time on certain systems, but you have to configure the system properly to realize the guaranteed time.

Maximum Response Time Guarantee

In Challenge / Onyx and POWER-Challenge systems running IRIX 5.2, 5.3, and 6.0, interrupt response time is guaranteed not to exceed 200 microseconds in a properly configured system.

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 200 microseconds 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 these sequential parts:

Hardware latency	The time required to make a CPU respond to an interrupt signal.
Software latency	The time to set aside other work and enter the device driver code.
Device service time	The time the device driver spends processing the interrupt, which must be minimal.
Dispatch cycle time	The time to choose the next user process to run, and to return to its code.

The parts are diagrammed in Figure 6-1 and discussed in the following topics.

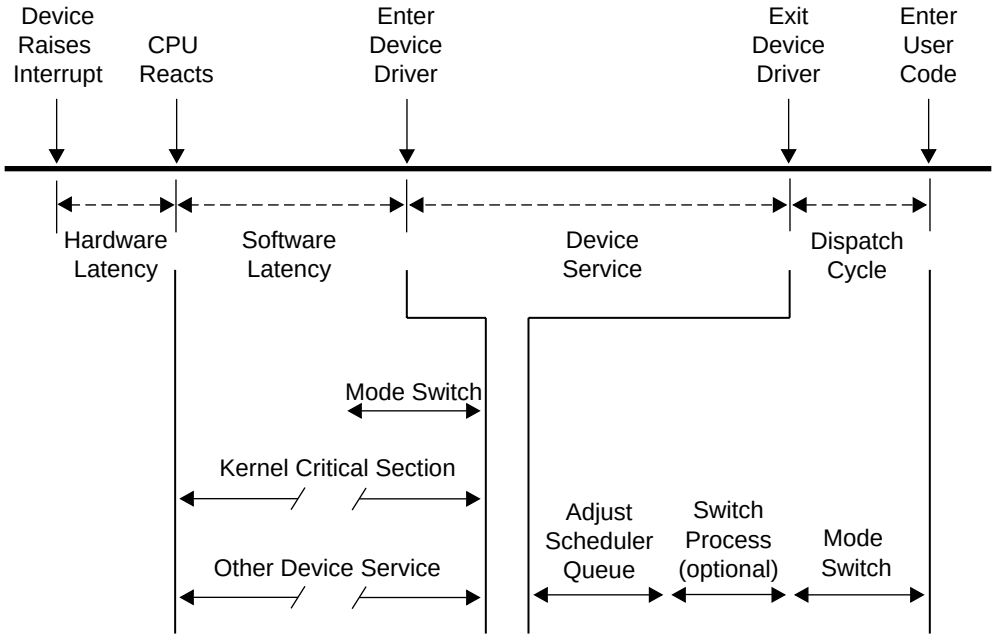

Figure 6-1 Components of Interrupt Response Time

Hardware Latency

When a VME device requests an interrupt, one of the 7 VME IRQ lines is set active. The Challenge / Onyx VCAM VME Controller contains interrupt destination registers that are programmed by the IRIX kernel to direct IRQ lines to specific CPUs. (The programming is in the IPL and NOINTR configuration statements. See “Isolating a CPU From Sprayed Interrupts” on page 100 and “Assigning Interrupts to CPUs” on page 100).

The VCAM VME Controller places an interrupt request to a specific CPU on the POWERpath-2 bus. The destination CPU records the interrupt in its interrupt register and, if interrupts at that level are not masked off, it responds by trapping to an interrupt vector.

The time taken for these events is the hardware latency, or interrupt propagation delay. The typical propagation delay is 2 microseconds. The theoretical worst-case delay is 8 microseconds, but this requires a very large system configuration. For typical configurations, 4 microseconds is an appropriate estimate of worst-case delay.

The worst-case hardware latency can be significantly reduced by not placing either graphics or HIPPI interfaces on the POWERchannel-2 interface used for VME devices.

Software Latency

Some instructions have to be executed before control reaches the device driver. When the interrupt arrives, the software will be in one of three states:

- executing user code or noncritical kernel code

  Entry to the device driver requires only a mode switch, a small number of instructions.

- executing a critical section in the kernel

  The kernel masks interrupts while in critical sections. The mode switch occurs when the critical section ends.

- executing another device driver at the same or higher interrupt level

  The mode switch occurs when the other device service ends.

Kernel Critical Sections

Most of the IRIX kernel code is noncritical and executed with interrupts enabled. However, 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, the interrupt level is raised in that CPU. New interrupts are not serviced until the critical section is complete.

Although most kernel critical sections are short, there is *no guarantee* on the length of a critical section. In order to achieve 200 microsecond 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 (see “Restricting a CPU From Scheduled Work” on page 101) and isolate it from TLB interrupts (see “Isolating a CPU From TLB Interrupts” on page 104)—or to use the Frame Scheduler.

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 isolated 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, for example **fork()**, **brk()**, or **mmap()**. The processes must also avoid generating external interrupts with long pulse widths (see “Generating External Signals” on page 195).

In general, the processes that use 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.

Service Time for Other Devices

While a device driver interrupt handler is executing, interrupts at the same or inferior priority are masked. During the interrupt handling, devices at a superior priority can interrupt and be handled. When the interrupt handler exits, interrupts are unmasked. Any pending interrupt at the same or inferior priority will then be taken before the kernel returns to the interrupted process. Thus the handling of an interrupt could be delayed by one or more device service times at either a superior or an inferior priority level.

Since device drivers are often provided by third parties, there is *no guarantee* on the service time of a device. In order to achieve 200 microsecond response time, you must ensure that the time-critical devices supply the only interrupts directed to that CPU. The system administrator assigns interrupt levels to devices using the VECTOR statement in the */var/sysgen/system* file. Then the assigned level is directed to a CPU using the IPL statement (see "Assigning Interrupts to CPUs" on page 100).

Device Service Time

The time spent servicing an interrupt should be negligible. The interrupt handler should do very little processing, 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.

Device drivers supplied by SGI indeed spend negligible time in interrupt service. Device drivers from third parties are an unknown quantity. Hence the 200 microsecond guarantee is not in force when third-party device drivers are used on the same CPU.

Dispatch Cycle

When the device driver interrupt handler exits, the kernel returns to a user process. This may be the same process that was interrupted, or a different one.

Adjust Scheduler Queue

Typically, the result of the interrupt is to make a sleeping process runnable. The runnable process is entered in one of the scheduler queues. (This work may be done while still within the interrupt handler, as part of a device driver library routine such as **wakeup()**.)

Switch Processes

If the CPU was idling when the interrupt arrived, and if the interrupt has made a process runnable, the kernel spends some time setting up the context of the process to be run.

If the CPU has not been made nonpreemptive (see “Making a CPU Nonpreemptive” on page 106), and if the interrupt has made a superior-priority process runnable, the interrupted process will be preempted. The kernel has to save the context of the inferior-priority process before setting up the context of the new process.

If the CPU has been made nonpreemptive, there is no process switch. The kernel always returns to the interrupted process, if there was one.

In short, the kernel may spend time saving the context of one process, and may spend time setting up the context of another process.

Note: In a CPU controlled by the Frame Scheduler, control always returns to the interrupted process in minimal time.

Mode Switch

A number of instructions are required to exit kernel mode and resume execution of the user process. Among other things, this is the time 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; see “Locking Program Text and Data” on page 66.)

Minimal Interrupt Response Time

To summarize, you can ensure interrupt response time of less than 200 microseconds for one specified device interrupt provided you configure the system as follows:

- The interrupt is directed to a specific CPU, not “sprayed”; and is the highest-priority interrupt received by that CPU.
- The interrupt is handled by an SGI-supplied device driver, or by a device driver from another source that promises negligible (10 microsecond or less) processing time.
- That CPU does not receive any other “sprayed” interrupts.
- That CPU is restricted from executing general UNIX processes, isolated from TLB interrupts, and made nonpreemptive—or is managed by the Frame Scheduler.
- Any process you assign to that CPU avoids system calls other than interprocess communication and allocation within an arena.

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

Tip: If interrupt service time is a critical factor in your design, consider the possibility of using VME programmed I/O to poll for data, instead of using interrupts. It takes at most 4 microseconds to poll a VME bus address (see “PIO Access” on page 190). A polling process can be dispatched one or more times per frame by the Frame Scheduler with low overhead.

Chapter 7

Using the Frame Scheduler

The REACT/Pro Frame Scheduler makes it easy to structure a real-time program as a family of independent, cooperating processes, running on multiple CPUs, scheduled in sequence at the frame rate of the application. For an overview of the Frame Scheduler, see “REACT/Pro Frame Scheduler” on page 26.

This chapter contains details on the operation and use of the Frame Scheduler, under these main headings:

- “Frame Scheduler Concepts” on page 116 details the operation and methods of the Frame Scheduler.
- “Selecting a Time Base” on page 125 covers the important choice of which source of interrupts should define a frame interval.
- “Using the Scheduling Disciplines” on page 128 explains the options for scheduling activities of different kinds.
- “Preparing the System” on page 132 reviews the system administration steps needed to prepare the CPUs that the Frame Scheduler will use.
- “Implementing a Single Frame Scheduler” on page 132 outlines the structure of an application that uses one CPU.
- “Implementing Synchronized Schedulers” on page 133 outlines the structure of an application that needs the power of multiple CPUs.
- “Handling Frame Scheduler Exceptions” on page 136 describes how overrun and underrun exceptions are dealt with.
- “Using Signals Under the Frame Scheduler” on page 141 discusses the issue of signal latency and the signals the Frame Scheduler generates.
- “Using Timers with the Frame Scheduler” on page 143 covers the use of itimers with the Frame Scheduler.
- “The Frame Scheduler Device Driver Interface” on page 144 documents the way that a kernel-level device driver can generate time-base interrupts for a Frame Scheduler.

Frame Scheduler Concepts

One Frame Scheduler dispatches selected processes at a real-time rate on one CPU. You can also create multiple, synchronized Frame Schedulers so as to dispatch concurrent processes on multiple CPUs.

Frame Scheduler Basics

A Frame Scheduler takes over the scheduling and dispatching of processes on one CPU. It isolates the CPU (see “Isolating a CPU From TLB Interrupts” on page 104), and completely supersedes the operation of the normal IRIX scheduler on that CPU. Only processes enqueued to the Frame Scheduler can use the CPU. IRIX dispatching priorities are not relevant on that CPU.

The execution of normal processes, daemons, and pending timeouts are all migrated to other CPUs—typically to CPU 0, which cannot be owned by a Frame Scheduler. All interrupt handling is usually directed away from a Frame Scheduler CPU as well (see “Preparing the System” on page 132).

Frame Scheduling

Instead of scheduling processes according to priorities with an attempt at fairness, the Frame Scheduler dispatches them according to a strict, cyclic rotation governed by a repetitive time base. The time base determines the fundamental frame rate. (See “Selecting a Time Base” on page 125.)

The interrupts from the time base define *minor frames*. You tell the Frame Scheduler a fixed number of minor frames that should be considered a *major frame*, which defines the application’s true frame rate.

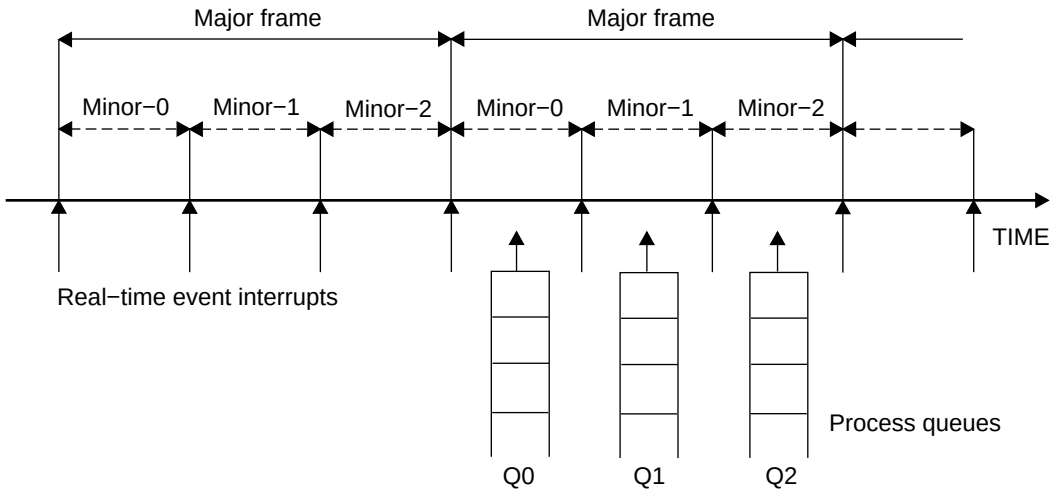

Figure 7-1 Major and Minor Frames

As shown in Figure 7-1, the Frame Scheduler maintains a queue of processes for each minor frame within a major frame. You enqueue the activity processes of your program in a specific order within specific minor frames. You can:

- Enqueue multiple processes in one minor frame. All must complete their work within the minor frame interval.
- Enqueue the same process to run in more than one minor frame. Say that process *double* is to run twice as often as process *solo*. You could enqueue *double* to Q0 and Q2 in Figure 7-1, and enqueue *solo* to Q1.
- Enqueue a process that takes more than a minor frame to complete its work. If process *sloth* could need more than one minor interval, you could enqueue it to Q0, Q1 and Q2 in Figure 7-1, such that it would be able to continue working in all three minor frames until it completed.
- Enqueue a background process that is allowed to run only when all others have completed, to use up any remaining time within a minor frame.

All these options are controlled by scheduling disciplines you specify for each process as you enqueue it (see “Using the Scheduling Disciplines” on page 128).

The processes that a Frame Scheduler dispatches are typically child processes of the process that creates the Frame Scheduler, but that is not a requirement. Any process can be enqueued, even one that starts execution as a separate command.

The process that creates a Frame Scheduler is called the *frs-master* process. It is privileged in two respects:

- It can receive signals when errors are detected by the Frame Scheduler (see "Using Signals Under the Frame Scheduler" on page 141).
- It cannot itself be enqueued to the Frame Scheduler. It continues to be dispatched by IRIX. It executes on some other CPU than the one the Frame Scheduler uses.

The Frame Scheduler API

The details of the Frame Scheduler API can be found in the frs(3) reference page. However, it is summarized in Table 7-1 on "The Frame Scheduler API" on page 119 for convenient reference.

The following special types are declared in *frs.h*:

typedef frs_fsched_info_t	A structure containing information about one scheduler, including the CPU number it owns. Used when creating a Frame Scheduler.
typedef frs_t	A structure containing an frs_fsched_info_t and the process ID of the frs-master of the master Frame Scheduler. Used to create or specify any Frame Scheduler.
typedef frs_queue_info_t	A structure containing information about one activity process, the CPU it uses, its queue number, and its scheduling discipline. Used when enqueuing a process.
typedef frs_recv_info_t	A structure containing error recovery options.

Each Frame Scheduler function is available in two ways: as a system call to **schedctl()**, or as one or more library calls to functions in the *frs* library, */usr/lib/libfrs.so*. The system call is accessible from FORTRAN and Ada programs. The library calls are convenient for C and C++ use.

Table 7-1 Frame Scheduler Operations

Operation	Application Interface Options
Create a Frame Scheduler	int **schedctl(**MPTS_FRS_CREATE, frs_fsched_info_t* *frs_fsched_info***)**; frs_t* **frs_create(**int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, pid_t *sync_master_pid*, int* *sync_slaves*, int *num_slaves***)**; frs_t* **frs_create_master(**int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, int* *sync_slaves*, int *num_slaves***)**; frs_t* **frs_create_slave(**int *cpu*, frs_t* *sync_master_frs***)**;
Enqueue an activity process to a Frame Scheduler	int **schedctl(**MPTS_FRS_ENQUEUE, frs_queue_info_t* *frs_queue_info***)**; int **frs_enqueue(**frs_t* frs, pid_t *pid*, int *minor_index*, uint *discipline***)**;
Join a Frame Scheduler (activity is ready to start)	int **schedctl(**MPTS_FRS_JOIN, int *cpu***)**; int **frs_join(**frs_t* *frs***)**;
Start scheduling (all activities enqueued)	int **schedctl(**MPTS_FRS_START, int *cpu***)**; int **frs_start(**frs_t* *frs***)**;
Yield control after completing activity	int **schedctl(**MPTS_FRS_YIELD**)** int **frs_yield(**void**)**;
Pause scheduling at end of minor frame	int **schedctl(**MPTS_FRS_STOP, int *cpu***)**; int **frs_stop(**frs_t* *frs***)**;
Resume scheduling at next time-base interrupt	int **schedctl(**MPTS_FRS_RESUME, int *cpu***)**; int **frs_resume(**frs_t* *frs***)**:
Destroy a Frame Scheduler	int **schedctl(**MPTS_FRS_DESTROY, int *cpu***)**; int **frs_destroy(**frs_t* *frs***)**;
Interrogate a process queue	int **schedctl(**MPTS_FRS_GETQUEUELEN, frs_queue_info_t* *frs_queue_info***)**; int **schedctl(**MPTS_FRS_READQUEUE, frs_queue_info_t* *frs_queue_info*, pid_t* *pidlist***)**; int **frs_getqueuelen(**frs_t* *frs*, int *minor_index***)**; int **frs_readqueue(**frs_t* *frs*, int *minor_index*, pid_t* *pidlist***)**;
Remove a process from a queue	int **schedctl(**MPTS_FRS_PREMOVE, frs_queue_info_t* *frs_queue_info***)**; int **frs_premove(**frs_t* *frs*, int *minor_index*, pid_t *remove_pid***)**;
Reinsert a process in a queue (possibly changing discipline**)**	int **schedctl(**MPTS_FRS_PINSERT, frs_queue_info_t* *frs_queue_info*, int *base_pid***)**; int **frs_pinsert(**frs_t* *frs*, int *minor_index*, pid_t *insert_pid*, int *discipline*, pid_t *base_pid***)**;
Set / retrieve error-recovery options	int **schectl(**MPTS_FRS_SETATTR, frs_attr_info_t***)**; int **frs_setattr(** frs_t* *frs*, int *minor_index*, pid_t *pid*, frs_attr_t *att_index*, frs_recv_info_t* *options***)**; int **schectl(**MPTS_FRS_GETATTR, frs_attr_info_t***)**; int **frs_getattr(** frs_t* *frs*, int *minor_index*, pid_t *pid*, frs_attr_t *att_index*, frs_recv_info_t* *options***)**;

Process Execution

An activity process that is enqueued to a Frame Scheduler has the basic structure shown in Example 7-1.

Example 7-1 Skeleton of an Activity Process

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

When the process is ready to start real-time execution, it calls **frs_join()**. This call blocks until all enqueued processes are ready and scheduling has begun (see “Starting the Schedulers” on page 124). When **frs_join()** returns, the process is running in its first minor-frame execution.

The process then performs whatever activity it is supposed to complete in each minor frame. When it completes that work, it calls **frs_yield()**. This gives up control of the CPU until the next minor frame in which the process is enqueued.

Each activity process executes without interruption. As long as it yields before the end of the frame, it can do its assigned work without interruption of any kind. The Frame Scheduler preempts it at the end of the minor frame.

Tip: This means that activity processes can often operate on global data without using locks or semaphores. If the process that modifies global data is enqueued in a different minor frame from the processes that read the global data, there can be no access conflicts between them.

Conflicts are still possible between two processes that are queued to the same minor frame in different, synchronized Frame Schedulers. However, such processes are guaranteed to be running concurrently. This means they can use spin-locks (see “Locks” on page 35) with high efficiency.

Tip: A possible variation on the pattern for processes with a very short minor frame interval is the following: Immediately after calling **frs_join()**, spend the first minor frame touching a set of important data structures in order to "warm up" the cache—as sketched in Example 7-2. The idea is to avoid an overrun exception in the first frame due to cache misses (see "Reducing Cache Misses" on page 68).

Example 7-2 Alternate Skeleton of Activity Process

```
/* Initialize data structures etc. */
frs_join(scheduler-handle); /* Much time could pass here. */
/* Here touch important data structures, doing no work */
do
{
   frs_yield();
   /* Perform the activity. */
} while(1);
_exit();
```

Scheduling Within a Minor Frame

Processes in a minor frame queue are dispatched in queue order. Initially, queue order is the order in which processes were named in **frs_enqueue()** calls. (The queues can be reordered dynamically; see **frs_premove()** and **frs_pinsert()** in Table 7-1.)

Scheduler Flags frs_run and frs_yield

The Frame Scheduler keeps two status flags per queued process, named *frs_run* and *frs_yield*. If a process is ready to run when its turn comes, it is dispatched and its *frs_run* flag is set to indicate that this process has run at least once within this minor frame.

When a process yields, its *frs_yield* flag is set to indicate that the process has already run and released the processor. It will not be activated again within this minor frame.

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

Detecting Overrun and Underrun

When a time base interrupt occurs to indicate the end of the minor frame, the Frame Scheduler checks the flags for each process. If the *frs_run* flag has not been set, that process never ran and therefore is a candidate for an *underrun* exception. If the *frs_run* flag is set but the *frs_yield* flag is not, the process is a candidate for an *overrun* exception. Whether these exceptions are declared depends on the scheduling discipline assigned to the process.

Scheduling disciplines are explained under “Using the Scheduling Disciplines” on page 128).

At the end of a minor frame, the Frame Scheduler resets all *frs_run* flags. It also resets all *frs_yield* flags except for those of processes that use the Continuable discipline in that minor frame. For those processes, the residual *frs_yield* flags keeps the processes that have yielded from being dispatched in the next minor frame. All *frs_yield* flags are reset at the end of a major frame.

Underrun and overrun exceptions are typically communicated via IRIX signals. These signals are sent to *all* processes in the FRS queues (see “Using Signals Under the Frame Scheduler” on page 141).

Estimating Available Time

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

Estimation is simplified by the fact that only the enqueued processes can execute in a CPU controlled by the Frame Scheduler. You need to estimate the maximum time each process can consume between one call to **frs_yield()** and the next.

Frame Scheduler processes do compete for CPU cycles with I/O interrupt service in the same CPU. If you direct I/O interrupts away from the CPU (see “Isolating a CPU From Sprayed Interrupts” on page 100 and “Assigning Interrupts to CPUs” on page 100), then the only competition for CPU cycles is the overhead of the Frame Scheduler itself, and it has been carefully optimized for least overhead.

Alternatively, you may assign specific I/O interrupts to a CPU used by the Frame Scheduler. In that case, you must estimate the time that interrupt service will consume (see “Maximum Response Time Guarantee” on page 107), and schedule fewer processes to compensate.

Using Multiple Synchronized Schedulers

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

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

Each Frame Scheduler has its own queues of processes. A given process can be enqueued to only one CPU. (However, you could create multiple instances of the same process and enqueue each to a different CPU.) Normally, all synchronized Frame Schedulers use the same number of minor frames per major frame, although this is not a requirement.

A process can have the frs-master relationship to only one Frame Scheduler. In order to create multiple, synchronized Frame Schedulers, you must create a process to be the frs-master of each one. Typically these will be lightweight processes created with **sproc()**.

Starting the Schedulers

A Frame Scheduler cannot start dispatching activities until

- the frs-master has enqueued all the activity processes to their minor frames
- all the enqueued processes have done their own initial setup and are ready to run their real-time loop

When multiple Frame Schedulers are used, none can start until all are ready.

The frs-master tells its Frame Scheduler that it has enqueued all activities by calling **frs_start()**. Each activity process tells its Frame Scheduler that it is ready to begin real-time processing by calling **frs_join()**.

A Frame Scheduler is ready when it has received an **frs_start()** call and an appropriate number of **frs_join()** calls. The Frame Scheduler is prepared to have these calls come in any order; that is, a process is allowed to join before or after it has been enqueued.

Each synchronized Frame Scheduler tells the master Frame Scheduler when it is ready. When all the synchronized schedulers are ready, the master Frame Scheduler gives the downbeat, and the first minor frame begins.

Pausing Frame Schedulers

Any Frame Scheduler can be made to pause and restart. Any process (typically but not necessarily the frs-master) can call **frs_stop()**, specifying a particular Frame Scheduler. That scheduler continues dispatching processes from the current minor frame until all have yielded. Then it goes into an idle loop until a call to **frs_resume()** tells it to start. It resumes on the next time-base interrupt, with the next minor frame in succession.

Since a Frame Scheduler does not stop until the end of a minor frame, you can stop and restart a group of synchronized schedulers by calling **frs_stop()** for each one before the end of a minor frame. You can cause them all to resume at the same time by calling **frs_resume()** for each one between one time-base interrupt and the next.

Selecting a Time Base

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

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

When you use multiple, synchronized Frame Schedulers, the master Frame Scheduler creates an *interrupt group*, a hardware mechanism that distributes the time-base interrupt to each synchronized CPU. This ensures that minor-frame boundaries are synchronized across all the Frame Schedulers. (For details of the interrupt group mechanism, you can read “Group Interrupts on Challenge and Onyx Systems,” a technical paper distributed with the REACT/Pro product.)

On-Chip Timer Interrupt

Each processor chip contains a free-running timer that is used by IRIX for normal process scheduling. This R4000 timer is not synchronized between processors, so it cannot be used to drive multiple synchronized schedulers. The R4000 timer can be used as a time base when only one CPU is used and there is a reason to not use the high-precision timer described in the next topic.

To use the R4000 timer, specify FRS_INTRSOURCE_R4KTIMER as the interrupt source, and the minor frame interval in microseconds, to **frs_create()**.

High-Resolution Timer

The high-resolution timer and clock is a timer that is synchronous across all processors, and is ideal to drive synchronous schedulers. On Challenge and Onyx systems this timer is based on the high resolution counter discussed under “Hardware Cycle Counter” on page 42.

To use this timer, specify FRS_INTRSOURCE_CCTIMER, and the minor frame interval in microseconds, to **frs_create()**.

The IRIX kernel uses this timer for managing timer events. When your program creates the master Frame Scheduler, the Frame Scheduler migrates all timeout events to CPU 0, leaving the timer on the scheduled CPU free.

An interrupt group is not required to coordinate multiple Frame Schedulers when this time base is used. The high-resolution timers in all CPUs are synchronized automatically.

Vertical Sync Interrupt

An interrupt is generated for every vertical retrace by the graphics subsystem (see “Understanding the Vertical Sync Interrupt” on page 101). The frame rate will be either 50 Hz or 60 Hz, depending on the installed hardware. This interrupt is especially appropriate for a visual simulator, since it defines a frame rate that matches the graphics subsystem frame rate.

To use the vertical sync interrupt, specify FRS_INTRSOURCE_VSYNC to **frs_create()**. An error is returned if this system lacks a graphics subsystem.

When multiple synchronized schedulers are used, the master Frame Scheduler allocates an interrupt group to distribute the vertical sync interrupt.

External Interrupts

An external interrupt is generated via a signal applied to the external interrupt sockets on a Challenge or Onyx system (see “External Interrupts” on page 195). To use external interrupts as a time base, specify FRS_INTRSOURCE_EXTINTR to **frs_create()**.

When multiple synchronized schedulers are used, the master Frame Scheduler receives the interrupt, and allocates an interrupt group that is used to make the interrupt simultaneously available to the synchronized schedulers.

Note: External output signals can be generated by software using **ioctl()** to the external interrupt driver. An imaginative designer might think of connecting the external output jacks to the external interrupt input jacks, thus creating software-controlled external interrupts as an FRS time base. This would work in principle. However, if the interrupts are generated by a user process that makes any other system calls, there is a possibility of system deadlock.

Device Driver Interrupt

A user-written, kernel-level device driver can supply the time-base interrupt (see “The Frame Scheduler Device Driver Interface” on page 144). The Frame Scheduler allocates an interrupt group. The device driver must direct interrupts to it.

To use a device driver as a time base, specify FRS_INTRSOURCE_DRIVER and the device driver’s identifying number, to **frs_create()**.

Software Interrupt

A programmed, software-generated interrupt can be used as the time base. Any user process can send this interrupt to the master Frame Scheduler by calling **frs_userintr()**.

Note: Software interrupts are primarily intended for application debugging. It is not feasible for a user process to generate interrupts with the kind of regularity that a real-time scheduler requires.

To use software interrupts as a time base, specify FRS_INTRSOURCE_USER to **frs_create()**.

Caution: User interrupts are normally triggered by a process running on CPU 0. This can lead to system deadlock if the user process generating the interrupts issues system calls other than frs_userintr().

Using the Scheduling Disciplines

When an frs-master enqueues a process to a minor frame (using **frs_enqueue()**), it must specify a scheduling discipline that tells the Frame Scheduler how the process is expected to use its time within that minor frame.

Realtime Discipline

In the simplest case, an activity process should start during the minor frame to which it is queued, and should complete its work and yield within the same minor frame.

If the process is not ready to run (for example, is blocked on I/O) during the entire minor frame, an *underrun* exception is said to occur. If the process fails to complete its work and yield within the minor frame interval, an *overrun* exception is said to occur.

The Frame Scheduler calls this strict discipline the Realtime scheduling discipline.

The simplest case of a Frame Scheduler would consist of

- one minor frame per major frame—the time base is also the frame rate
- one or more activities enqueued to the frame with Realtime discipline

This model could describe a simple kind of simulator in which three activities—poll the inputs; calculate the new status; update the display—must be repeated in that order during every frame.

In this scenario, each activity must start and must finish in every frame. If one fails to start, or fails to finish, the real-time program is broken in some way and must take some action.

However, realistic designs need the flexibility to have processes that

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

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

Background Discipline

The Background discipline is mutually exclusive with the other disciplines. The Frame Scheduler only dispatches a Background process when all other processes queued to that minor frame have run and have yielded. Since the Background process cannot be sure it will run and cannot predict how much time it will have, the concepts of underrun and overrun do not apply to it.

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

Underrunable Discipline

You specify Underrunable discipline with Realtime discipline to prevent detection of underrun exceptions. You specify Underrunable in two cases:

- When a process needs to run only when some event has occurred such as a lock being released or a semaphore being posted.
- When a process may need more than one minor frame (see “Using Multiple Consecutive Minor Frames” on page 130).

When you specify Realtime+Underrunable, the process is not required to start in that minor frame. However, if it starts, it is required to yield before the end of the frame or an overrun exception is raised.

Overrunnable Discipline

You specify Overrunnable discipline with Realtime discipline to prevent detection of overrun exceptions. You specify it in two cases:

- When it truly does not matter if the process fails to complete its work within the minor frame—for example, a calculation of a game strategy which, if it fails to finish, merely makes the computer a less dangerous opponent.
- When a process may need more than one minor frame (see “Using Multiple Consecutive Minor Frames” on page 130).

When you specify Overrunnable with Realtime, the process is not required to call **frs_yield()** before the end of the frame. Even so, the process is preempted at the end of the frame. It does not have a chance to run again until the next minor frame in which it is enqueued. At that time it resumes where it was preempted, with no indication that it was preempted.

Continuable Discipline

You specify Continuable discipline with Realtime discipline to prevent the Frame Scheduler from clearing the *frs_yield* flag at the end of this minor frame (see “Scheduling Within a Minor Frame” on page 121).

The result is that, if the process yields in this frame, it need not run or yield in the following frame. The residual *frs_yield* flag value, carried forward to the next frame, applies. You specify Continuable discipline with other disciplines in order to let a process execute just once in a block of consecutive minor frames.

Using Multiple Consecutive Minor Frames

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

You must decide the absolute maximum time the process could consume between starting up and calling **frs_yield()**. If this is unpredictable, or if it is predictably longer than the major frame, the process cannot be scheduled by the Frame Scheduler. It should probably run in another CPU under the IRIX scheduler.

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

The discipline settings for each frame should be:

First frame	Realtime + Overrunnable + Continuable—the process must start in this frame (not Underrunable) but is not required to yield (Overrunnable). If it yields, it is not restarted in the following minor frame (Continuable).
Intermediate	Realtime+Underrunable+Overrunnable+Continuable—the process need not start (it might already have yielded, or might be blocked) but is not required to yield. If it does yield (or if it had yielded in a preceding minor frame), it is not restarted in the following minor frame (Continuable).
Final frame	Realtime+Underrunable—the process need not start (it might already have yielded) but if it starts, it must yield in this frame (not Overrunnable). The process can start a new run in the next minor frame to which it is queued (not Continuable).

A process can be enqueued for one or more of these multi-frame sequences in one major frame. For example, suppose that the minor frame rate is 60 Hz, and a major frame contains 60 minor frames (1 Hz). You have a process that should run at a rate of 5 Hz and can use up to 3/60 second at each dispatch. You would enqueue the process to 5 sequences of 3 consecutive frames each. It would start in frames 0, 12, 24, 36, and 48. Frames 1, 13, 25, 37 and 49 would be intermediate frames, and 2, 14, 26, 38 and 50 would be final frames.

Preparing the System

Before a real-time program executes, you must set up the system in the following ways.

1. Choose the CPU or CPUs that the real-time program will use. CPU 0 (at least) must be reserved for IRIX system functions.
2. Decide which processors will handle I/O interrupts. By default, IRIX distributes I/O interrupts across all available processors as a means of balancing the load (referred to as *spraying interrupts*). CPUs that are used for real-time programs should be removed from the distribution set (see "Assigning Interrupts to CPUs" on page 100).
3. Make sure that none of the real-time CPUs is managing the clock (see "Assigning the Clock Processor" on page 98). Normally the responsibility of handling 10ms scheduler interrupts is given to CPU 0.
4. Make sure none of the real-time CPUs is handling the fast timer ("Assigning the fasthz Processor" on page 98). This responsibility is typically given to CPU 0 along with all other housekeeping.

Each Frame Scheduler takes care of restricting and isolating its CPU, so that the CPU is used only be processes scheduled by the Frame Scheduler.

Implementing a Single Frame Scheduler

When the activities of your real-time program can be handled within a major frame interval by a single CPU, your program needs to create only one Frame Scheduler.

Typically your program has a top-level process (called the master process here) to handle start-up and termination, and one or more activity processes that are dispatched by the Frame Scheduler. The activity processes are typically lightweight processes created using **sproc()**, but that is not a requirement—the activity processes can be created with **fork()**, and they need not be children of the master process. (See for instance "Example of Scheduling Separate Programs" on page 228.)

In general, these are the steps that the master process follows:

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, and other resources.
2. Lock the address space segments it will share with lightweight processes (see “Locking Pages in Memory” on page 65).
3. Create the Frame Scheduler using the call **frs_create_master()** (see Table 7-1 on “The Frame Scheduler API” on page 119).
4. Create the activity processes using **sproc()** or **fork()**. (When **fork()** is used, each child process must lock its own address space.)
5. Use **frs_enqueue()** to queue each child to the Frame Scheduler queue or queues on which it is to run (see Table 7-1 on “The Frame Scheduler API” on page 119).

   Each activity process independently uses **frs_join()** (Table 7-1) to let the Frame Scheduler know it is ready to start real-time execution. This call blocks until scheduling begins, then returns to start the first frame dispatch of each activity process.
6. Set up signal handlers for signals from the Frame Scheduler (see “Using Signals Under the Frame Scheduler” on page 141). The handlers are set now so that the activity processes do not inherit them.
7. Use **frs_start()** (Table 7-1) to enable scheduling.

   The Frame Scheduler begins scheduling processes as soon as all the activity processes have called **frs_join()**.
8. Wait for error and termination signals from the Frame Scheduler and for the termination of child processes.
9. Use **frs_destroy()** to terminate the Frame Scheduler.
10. Tidy up the global resources as required.

Implementing Synchronized Schedulers

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

No single process may create more than one Frame Scheduler. This is because every Frame Scheduler must have a unique frs-master process to which it can send signals. As a result, the program will have three types of processes:

- a master process that sets up global data and creates the master Frame Scheduler
- one frs-master for each synchronized Frame Scheduler
- activity processes

A variety of designs is possible but the simplest is possibly the set of processes described below.

The master process will:

1. Initialize global resources such as memory-mapped segments, memory arenas, files, asynchronous I/O, and other resources. One global resource is the process ID of the master process.
2. Lock the address space it will share with lightweight processes.
3. Create the master Frame Scheduler using the call **frs_create_master()**, and store its handle in a global location.
4. Create one frs-master process for each synchronized CPU to be used.
5. Create the activity processes that will be scheduled by the master Frame Scheduler and use **frs_enqueue()** to enqueue them to their assigned minor frames.
6. Set up signal handlers for signals from the Frame Scheduler (see “Using Signals Under the Frame Scheduler” on page 141).
7. Use **frs_start()** (Table 7-1) to tell the master Frame Scheduler that its activity processes are all enqueued.

   The master Frame Scheduler will start scheduling processes as soon as all processes have called **frs_join()** for their respective schedulers.
8. Wait for termination or error signals.
9. Use **frs_destroy()** to terminate the master Frame Scheduler.
10. Tidy up global resources as required.

Each frs-master process will:

1. Create a synchronized Frame Scheduler using **frs_create_slave()** (Table 7-1 on “The Frame Scheduler API” on page 119), specifying information about the master Frame Scheduler stored by the master process.
2. Create the activity processes that will be scheduled by this Frame Scheduler, and use **frs_enqueue()** to enqueue them to their assigned minor frames.
3. Set up signal handlers for signals from the synchronized Frame Scheduler.
4. Use **frs_start()** to tell the synchronized Frame Scheduler that all activity processes have been enqueued.

   The Frame Scheduler notifies the master Frame Scheduler when all processes have called **frs_join()**. When the master Frame Scheduler starts broadcasting interrupts, scheduling will begin.
5. Wait for termination or error signals.
6. Use **frs_destroy()** to terminate the synchronized Frame Scheduler.

For an example of this kind of program structure, refer to “Example of Multiple Synchronized Schedulers” on page 230.

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

Tip: A program of this type is place where you might use a barrier (see “Barriers” on page 36). For example, you want to be sure that all Frame Schedulers are successfully created before you start to create activity processes. The master and the frs-master processes could rendezvous at a barrier after creating their schedulers, but before starting to create activity processes or enqueue them.

Handling Frame Scheduler Exceptions

The frs-master process for a scheduler controls the handling of the Overrun and Underrun exceptions. It can specify how these exceptions should be handled, and what signals its Frame Scheduler should send. These policies have to be set before the scheduler is started. While the scheduler is running, the frs-master can query the number of exceptions that have occurred.

Exception Types

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

The Underrun exception indicates that a process that should have started in a minor frame did not start. Possibly the process has terminated. More likely it was blocked in some kind of wait because of an unexpected delay in I/O, or a deadlock on a lock or semaphore.

Exception Handling Policies

The frs-master process can establish one of four policies for handling overrun and underrun exceptions. When it detects an exception, the Frame Scheduler can:

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

The default action is to send a signal (the specific signals are listed under “Setting Frame Scheduler Signals” on page 140). The frs-master process can then take action, for example, terminating the Frame Scheduler.

Injecting a Repeat Frame

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

Extending the Current Frame

The policies of extending the frame, either with more time or by stealing time from the next frame, are allowed only when the time base is an on-chip or high-resolution timer (see "Selecting a Time Base" on page 125).

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

When stealing time from the following frame, the Frame Scheduler returns to the original time base at the end of the following minor frame—provided that the processes queued to that following frame can finish their work in a reduced amount of time. If they do not, the Frame Scheduler will steal time from the next frame still.

Dealing With Multiple Exceptions

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

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

Setting Exception Policies

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

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

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

The recovery modes and other constants are declared in */usr/include/sys/frs.h*. The function in Example 7-3 sets the policy of injecting a repeat frame. The caller specifies only the Frame Scheduler and the number of consecutive exceptions allowed.

Example 7-3 Function to Set INJECTFRAME Exception Policy

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

The function in Example 7-4 sets the policy of stretching the current frame (a function to set the policy of stealing time from the next frame would be nearly identical). The caller specifies the Frame Scheduler, the number of consecutive exceptions, and the stretch time in microseconds.

Example 7-4 Function to Set STRETCH Exception Policy

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

Querying Counts of Exceptions

When you set a policy that permits exceptions, the frs-master process can query for counts of exceptions. This is done with a call to **frs_getattr()**, passing the handle to the Frame Scheduler, the number of the minor frame, and the process ID of the process within that frame.

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

The function in Example 7-5 takes a Frame Scheduler handle and a minor frame number. It gets the list of process IDs queued to that that minor frame, and returns the sum of all exceptions for all of them.

Example 7-5 Function to Return a Sum of Exception Counts

```
#define THE_MOST_PIDS 250
int
totalExcepts(frs_t * theFRS, int theMinor)
{
    int numPids = frs_getqueuelen(theFRS, theMinor);
    int j, sum;
    pid_t allPids[THE_MOST_PIDS];
```

```
    if ( (numPids <= 0) || (numPids > THE_MOST_PIDS) )
        return 0; /* invalid minor #, or no procs queued? */

    if (!frs_readqueue(theFRS, theMinor, allPids))
        return 0; /* unexpected problem with reading IDs */

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

Tip: If a function such as the one in Example 7-5 is to be called frequently, it is a good idea to prepare the arrays of process IDs once and save them. The repeated calls to **frs_getqueuelen()** and **frs_readqueue()** can be avoided.

Setting Frame Scheduler Signals

The frame scheduler sends 3 signals, which are by default as follows:

SIGUSR2	Signals the Overrun exception
SIGUSR1	Signals the Underrun exception
SIGHUP	Signals the end of real-time dispatching due to a call to **frs_destroy()**

These signals are sent to *all* processes queued to that Frame Scheduler and to the frs-master process that created that scheduler.

All three signal numbers can be modified. The signal numbers are changed using **frs_setattr()**, in this case passing it an *frs_signal_info* structure, which is declared as follows:

```
typedef struct frs_signal_info {
    int sig_underrun;        /* SIGUSR1 */
    int sig_overrun;         /* SIGUSR2 */
    int sig_syncterm;        /* reserved, must be set to 0 */
    int sig_terminate;       /* SIGHUP */
} frs_signal_info_t;
```

This operation also must be done before the Frame Scheduler is started. A signal number specified as 0 is left unchanged. The function in Example 7-6 sets the signal numbers for Overrun and Underrun.

Example 7-6 Function to Set Frame Scheduler Signals

```
int
setUnderOverSignals(frs_t *frs, int underSig, int overSig)
{
  frs_signal_info_t work;
  bzero((void*)&work,sizeof(work));
  work.sig_underrun = underSig;
  work.sig_overrun = overSig;
  return frs_setattr(frs,0,0,FRS_ATTR_SIGNALS,(void*)&work);
}
```

Using Signals Under the Frame Scheduler

The Frame Scheduler itself sends signals to the processes using it. And processes can communicate by sending signals to each other.

Signal Delivery and Latency

When a process is scheduled by the IRIX kernel, it receives a pending signal the next time the process exits from the kernel domain. For most signals, this could occur

- when the process is dispatched after a wait or preemption
- upon return from some system call
- upon return from the kernel's usual 10-millisecond tick interrupt

(SIGALRM is delivered as soon as the kernel is ready to return to user processing after the timer interrupt, in order to preserve timer accuracy.) Thus, for a process that is ready to run, in a CPU that has not been made nonpreemptive, normal signal latency is at most 10 milliseconds, and SIGALARM latency is less. However, when the receiving process is not ready to run, or when there are competing processes with superious priorities, the delivery of a signal is delayed until the next time the receiving process is scheduled.

When the CPU is nonpreemptive (see “Making a CPU Nonpreemptive” on page 106), there are no clock ticks, so signals can only be delivered following a system call.

Signal latency can be greater when running under the Frame Scheduler. Like the normal IRIX scheduler, the Frame Scheduler delivers pending signals to a process when it next returns to the process from the kernel domain. This can occur

- when the process is dispatched at the start of a minor frame where it is enqueued
- upon return from some system call

The upper bound on signal latency in this case is the interval between the minor frames to which that process is queued. If the process is scheduled only once in a major frame, it might not receive a signal until a full major frame interval after the signal is sent.

Handling Frame Scheduler Signals

When a Frame Scheduler detects an Overrun or Underrun exception that it cannot recover from, and when it is ready to terminate, it sends a signal to the frs-master process and to all processes that have joined that scheduler. The different processes should handle these signals in different ways.

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

The activity processes can use the default system action of Exit for all three signals. This quietly terminates all activity processes on an error or when the Frame Scheduler terminates.

The frs-master process for a synchronized Frame Scheduler should have handlers for Underrun and Overrun signals. The handler should report the error and issue **frs_destroy()** to shut down its scheduler. An frs-master for a synchronized scheduler should use the default action for SIGHUP (Exit) so that completion of the **frs_destroy()** quietly terminates the frs-master.

The frs-master for the master (or only) Frame Scheduler should catch the exceptions, report them, and shut down its scheduler. It should also catch SIGHUP and use the occasion to close down shared resources in an orderly way. Alternatively, it can set the Ignore action for SIGHUP, and use **wait()** to wait for the termination of all its child processes (see the wait(2) reference page), before tidying up the shared resources.

Using Timers with the Frame Scheduler

In general, interval timers and the Frame Scheduler do not mix. The expiration of an interval is marked by a signal. However, signal delivery to an activity process can be delayed (see “Signal Delivery and Latency” on page 141), so timer latency is unpredictable.

An frs-master process, because it is scheduled by IRIX, not the Frame Scheduler, can use interval timers. However, it has a more reliable time base available in the activity processes it creates. The frs-master can create a global semaphore on which it waits with **uspsems()** (see “IRIX Semaphores” on page 33). The minimal activity process shown in Example 7-7 can be enqueued to one or more minor frames to provide a repeating interval at any multiple of the major-frame interval.

Example 7-7 Minimal Activity Process as a Timer

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

The Frame Scheduler Device Driver Interface

The Frame Scheduler provides a device driver interface to allow any device with a kernel-level device driver to generate the time-base interrupt. As many as 16 different device drivers can support the Frame Scheduler in any one Challenge / Onyx system. The Frame Scheduler distinguishes device drivers by an ID number in the range 0 through 15 that is coded into each driver.

Note: The structure of an IRIX kernel-level device driver is discussed in the *IRIX Device Driver Programming Guide* (see "Other Useful Books" on page xxiii).

In order to interact with the Frame Scheduler, a driver provides two routines, one for initialization and one for termination, which it exports during driver initialization. After a master Frame Scheduler has initialized a device driver, the driver calls a Frame Scheduler entry point to signal the occurrence of each interrupt.

Device Driver Overview

The following sequence of actions occurs when a device driver is used as a source of time-base interrupts for the Frame Scheduler.

1. During its initialization at system boot time, the driver exports the *prefix***_frs_func_set()** and *prefix***_frs_func_clear()** functions, specifying a unique driver identifier between 0 and 15. After these functions have been exported the Frame Scheduler is aware of the existence of this driver, and will allow programs to request it as the source of interrupts.
2. Later, a program creates a master Frame Scheduler specifying this driver as the source of interrupts. The Frame Scheduler verifies that this driver has exported the initialization and termination functions. Then it invokes *prefix***_frs_func_set(***intrgroup***)** for this particular driver. This tells the driver that time signals are needed.
3. As long as the master Frame Scheduler needs its time signals, the device driver invokes **frs_handle_driverintr()** each time its interrupt handling routine is entered. This informs the Frame Scheduler that an interrupt has been received.

4. When the Frame Scheduler is being terminated, it invokes *prefix*_**frs_func_clear()** for the associated driver. This tells the driver that time signals are no longer needed, and to cease calling **frs_handle_driverintr()** until it is again initialized by a Frame Scheduler.

The *prefix* in function names is the name of the loadable device driver as specified in its *master.d* file. Device driver names, device driver structure, configuration files, and related topics are covered in the *IRIX Device Driver Programming Guide*.

Frame Scheduler Initialization Function

The device driver must provide a function with the following prototype:

```
void prefix_frs_func_set ( intrgroup_t* intrgroup ) ;
```

A skeleton of an initialization function is shown in Example 7-8. The function is called by a new master Frame Scheduler that is created with an interrupt source parameter of FRS_INTRSOURCE_DRIVER and an interrupt qualifier specifying this device driver's number (see "Device Driver Interrupt" on page 127). A device driver is used by only one Frame Scheduler at a time.

The argument *intrgroup* is passed by the Frame Scheduler to identify the interrupt group it has allocated. A VME device driver must set the hardware devices it manages so that interrupts are directed to this interrupt group (see the paper "Group Interrupts on Challenge and Onyx Systems" distributed with REACT/Pro). The actual group identifier may be obtained using the macro:

```
intrgroup_get_groupid(intrgroup)
```

The effective destination may be obtained using the following macro:

```
EVINTR_GROUPDEST(intrgroup_get_groupid(intrgroup))
```

Example 7-8 Device Driver Initialization Function

```
/*
** Frame Scheduler initialization function
** for the External Interrupts Driver
*/
int FRS_is_active = 0;
int FRS_vme_install = 0;
void
example_frs_func_set(intrgroup_t* intrgroup)
{
   int s;
   ASSERT(intrgroup != 0);
   /*
   ** Step 1 (VME only):
   ** In a VME device driver, set up the hardware to send
   ** the interrupt to the appropriate destination.
   ** This is done with vme_frs_install() which takes:
   ** * (int) the VME adapter number
   ** * (int) the VME IPL level
   ** * the intrgroup as passed to this function.
   */
   FRS_vme_install = vme_frs_install(
      my_edt.e_adap, /* edt struct from example_edtinit */
      ((vme_intrs_t *)my_edt.e_bus_info)->v_brl,
      intrgroup);
   /*
   ** Step 2: any hardware initialization required.
   */
   /*
   ** Step 3: note that we are now in use.
   */
   FRS_is_active = 1;
}
```

Only VME device drivers need to call **vme_frs_install()**. As suggested by the code in Example 7-8, the arguments to **vme_frs_install()** can be taken from data supplied at boot time to the device driver's *prefix***edtinit()** function:

- the adapter number is in the *edt.e_adap* field
- the configured interrupt priority level is in the *vme_intrs.v_brl* addressed by the *edt.e_bus_info* field

The *prefix***edtinit()** function is documented in the *IRIX Device Driver Programming Guide*.

Note: The vme_frs_install() function is a dynamic version of the VECTOR configuration statement. You are not required to use the IPL value from the configuration file.

Frame Scheduler Termination Function

The device driver must provide a function with the following prototype:

```
void prefix_frs_func_clear ( void ) ;
```

A skeleton for this function is shown in Example 7-9. The Frame Scheduler that initialized a device driver calls this function when the Frame Scheduler is terminating. The Frame Scheduler deallocates the interrupt group to which interrupts were directed.

The device driver should clean up data structures and make sure that the device is in a safe state. A VME device driver must call **vme_frs_uninstall()**.

Example 7-9 Device Driver Termination Function

```
/*
** Frame Scheduler termination function
*/
void
example_frs_func_clear(void)
{
   /*
   ** Step 1: any hardware steps to quiesce the device.
   */

   /*
   ** Step 2 (VME only):
   ** Break the link between interrupts and the interrupt
   ** group by calling vme_frs_uninstall() passing:
   ** * (int) the VME adapter number
   ** * (int) the VME IPL level
   ** * the value returned by vme_frs_install()
   */
   vme_frs_uninstall(
      my_edt.e_adap, /* edt struct from example_edtinit */
```

```
      ((vme_intrs_t *)my_edt.e_bus_info)->v_brl,
      FRS_vme_install);
   /*
   ** Step 3: note we are no longer in use.
   */
   FRS_is_active = 0;
}
```

Exporting the Initialization and Termination Functions

A device driver must export the Frame Scheduler interface functions to make them known to the Frame Scheduler. This call, which occurs during the device driver's own initialization, also makes the driver known as a source of time-base interrupts:

```
frs_driver_export(int frs_driver_id,
   void (*frs_func_set)(intrgroup_t*),
   void (*frs_func_clear)(void));
```

A typical call resembles the code in Example 7-10.

The parameter *frs_driver_id* is the driver's identification number. A real-time program specifies the same number to **frs_create_master()** in order to select this driver as the source of interrupts. The identifier is an integer between 0 and 15. Different drivers in the same system must use different identifiers.

Example 7-10 Exporting Device Driver Entry Points

```
/*
** Function called by the example driver to export
** its Frame Scheduler interface functions.
*/
frs_driver_export(12,
   example_frs_func_set,
   example_frs_func_clear);
```

Generating Interrupts

A driver has to call the Frame Scheduler interrupt handler from within the driver's interrupt handler using code similar to that shown in Example 7-11. This handler is entered concurrently on each CPU where the master or a synchronized Frame Scheduler is running. It delivers the interrupt to the Frame Scheduler on that CPU. The function to be invoked is

```
void frs_handle_driverintr(void);
```

It is possible for an interrupt handler to be entered at a time when the Frame Scheduler for its processor is not active; that is, after **frs_destroy()** has been called and before the driver termination function has been entered. The **frs_handle_driverintr()** function checks for this and does nothing when nothing is required.

Example 7-11 Generating an Interrupt From a Device Driver

```
void example_intr()
{
   /*
   ** Step 1: anything required by the hardware
   */
   /*
   ** Step 2: if connected to the Frame Scheduler, send
   ** an interrupt to it. Flag FRS_is_active is set in
   ** Example 7-8 and cleared in Example 7-9.
   */
   if (FRS_is_active) frs_handle_driverintr();
   /*
   ** Step 3: any additional processing needed.
   */
   return;
}
```

Chapter 8

Optimizing Disk I/O for a Real-Time Program

A real-time program sometimes needs to 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 IRIX supports that can help you meet these performance goals, including these topics:

- "Memory-Mapped I/O" on page 151 points out the uses of mapping a file into memory.
- "Asynchronous I/O" on page 152 describes the use of the asynchronous I/O feature of IRIX version 5.3 and later.
- "Synchronous Writing and Direct Writing" on page 166 documents the performance cost of knowing when disk output is complete.
- "Guaranteed-Rate I/O" on page 169 describes the use of the guaranteed-rate feature of XFS.

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 (see "Mapping a File for I/O" on page 57). A file that represents a data base of some kind—for example a file of scenery elements, or a file containing a precalculated, multidimensional 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 (see "Locking Mapped Files Into Memory" on page 67).

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 that time the modified pages are written. The time-consuming call to **msync()** can be made from an asynchronous process.

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.

Conventional Synchronous I/O

Conventional I/O in UNIX 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.

Synchronous Input

The normal sequence of operations for IRIX input is as follows:

1. Normal code in a process invokes the system call **read()**, either directly or indirectly—for example, by accessing a new page of a memory-mapped file, or by calling a library function that calls **read()**.
2. The kernel locates the device holding the wanted data. Still operating under the identity of the calling process, the kernel enters the read entry point of the device driver.
3. The device driver initiates the input operation and blocks the calling process, for example by waiting on a semaphore in the kernel address space.
4. The kernel schedules another process to use the CPU.
5. Later, the device completes the input operation and causes a hardware interrupt.
6. The kernel interrupt handler enters the device driver interrupt entry point.
7. The device driver, finding that the data has been received, unblocks the sleeping process, for example by posting a semaphore.
8. The kernel recalculates the scheduling queues to account for the fact that a process that had been blocked can now run.
9. Then or perhaps later, depending on scheduling priorities, the kernel schedules the original process to run on some CPU.

10. The unblocked process exits the device driver read function and returns to user code, the read being complete.

During steps 4-8, the process that requested input is blocked. The duration of the delay is unpredictable. For example, the delay can be negligible if the data is already in a buffer in memory. It can be as long as one rotation time of a disk, if the disk is positioned on the correct cylinder. It can be longer if the disk has to seek. The probability of seeking depends on the way the file is arranged on the disk surface and also on the I/O operations of other processes in the system.

Synchronous 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. (An alternative for disk output is to use direct output, discussed under “Synchronous Writing and Direct Writing” on page 166.)

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. Device drivers for VME devices are often supplied by third parties.

Asynchronous I/O Basics

A real-time process needs to 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 design of processes and data structures. However, a standard solution is available.

Two Implementation Versions

IRIX 5.3 supports asynchronous I/O library calls conforming to POSIX document 1003.1b-1993. You use relatively simple calls to initiate input or output. The library package handles the details of

- initiating several lightweight processes to perform I/O
- allocating a shared memory arena and the locks, semaphores, and/or queues used to coordinate between the I/O processes
- queueing multiple input or output requests to each of multiple file descriptors
- reporting results back to your processes, either on request, through signals, or through callback functions

In IRIX 5.2 and IRIX 6.0, asynchronous I/O is implemented to conform to POSIX standard 1003.4 Draft 12, an earlier document. The later POSIX standard1003.1b, and the implementation of it version 5.3, are improved.

Programs compiled in IRIX 5.2 continue to work in IRIX 5.3. However, the two versions of asynchronous I/O cannot be mixed in the same program. *Conversion of programs to the 5.3 interface, where possible, is recommended.* The remainder of this topic describes the later implementation.

In IRIX 5.3, where both the older and newer interfaces are supported, you must take two steps to compile with the later (improved) version of asynchronous I/O:

- Define the compiler variable _ABI_SOURCE. This indicates conformance with the MIPS Application Binary Interface.
- Include the library parameter **-labi** in the link.

Note: In releases following 5.3, support for POSIX 1003.1b-1993 will be the default and these steps will no longer be required.

Asynchronous I/O Functions

Once you have opened the files and initialized asynchronous I/O, you perform asynchronous I/O by calling some of these functions:

aio_read()	Initiates asynchronous input from a file or device.
aio_write()	Initiates asynchronous output to a file or device.
lio_listio()	Initiates a list of operations to one or more files or devices.
aio_error()	Returns the status of an asynchronous operation.
aio_fsync()	Waits for all scheduled output for a file to complete.
aio_cancel()	Cancels pending, scheduled operations.

Each of these functions is described in detail in a reference page in reference volume 3.

Asynchronous I/O Control Block

Each asynchronous I/O request is represented by an instance of *struct aiocb*, a data structure that your program must allocate. Its principle contents are as follows.

- The file descriptor that is the target of the operation.

  File descriptors are returned by **open()** (see the open(2) reference page). A file descriptor used for asynchronous I/O can represent any file or device—not only a disk file.

- The address and size of a buffer to supply or receive the data.
- The file position for the operation as it would be passed to **lseek()** (see the lseek(2) reference page)

  The use of this value is discussed under “Multiple Operations to One File” on page 165.
- a *sigevent* structure, whose contents indicate what, if anything, should be done to notify your program of the completion of the I/O.

  The use of this value is discussed under “Checking for Completion” on page 160.

Note: The older (IRIX 5.2) version also accepted a request priority value. Request priorities are no longer supported. The field exists for compatibility and for possible future use, but must currently contain zero.

Initializing Asynchronous I/O

You can initialize asynchronous I/O in either of two ways. One way is simple; the other gives you control over the initialization.

Implicit Initialization

You can simply begin using asynchronous I/O by calling **aio_read()**, **lio_listio()**, or **aio_write()** to begin an operation. The first such call initializes asynchronous I/O. This is the only form of initialization described by the POSIX standard. However, in a real-time program you often need to control the timing of initialization.

Initializing with aio_sgi_init()

You can take greater control of asynchronous I/O by calling **aio_sgi_init()** (refer to the aio_sgi_init(3) reference page and to the declarations in */usr/include/aio.h*). The argument to this call can be a null pointer, indicating you want default values, or you can pass an *aioinit_t* structure. The principal fields of this structure specify

- the number of asynchronous processes to execute I/O

  The default is 5 processes; the minimum is 2. Specify 1 more than the number of I/O operations that could reasonably be executed in parallel on the available hardware. For example if you will be doing asynchronous I/O to one disk file and one tape drive, there could be at most two concurrent I/O operations, so there is no need to have more than 3 (1 more than 2) asynchronous processes.

- the number of locks that the asynchronous I/O processes should preallocate

  The default used by **aio_init()** is 3 locks; the minimum is 1. Specify the maximum number of simultaneous **lio_listio(**LIO_NOWAIT**)**, **aio_fsync()**, and **aio_suspend()** calls that your program can execute concurrently.

- the number of lightweight processes (sprocs) that will be sharing the use of asynchronous I/O.

  The default is 5; the minimum is 2. Specify 1 more than the number of different sproc'd processes that will be requesting asynchronous I/O.

The remaining fields of the *aioinit_t* structure are not used at this time and must be zero. Zero-valued fields are taken as a request for the default for that field. Example 8-1 shows a subroutine to initialize asynchronous I/O, given counts of devices and calling processes.

Example 8-1 Initializing Asynchronous I/O

```
int initAIO(int numDevs, int numSprocs)
{
   aioinit_t A = {0}; /* ensure zero'd fields */
   if (numDevs) /* we do know how many devices */
      A.aio_threads = 1+numDevs;
   if (numSprocs) /* we do know how many sprocs */
      A.aio_locks = A.aio_numusers = 1+numSprocs;
   return aioinit(&A);
}
```

When to Initialize

The time at which initialization occurs is important. If you initialize in a process that has been assigned to run on an isolated CPU, the asynchronous I/O processes will also run on that CPU. You probably want the I/O processes to run under normal dispatching on unrestricted CPUs. In that case, the proper sequence of initialization is:

- Open all file descriptors and verify that files and devices are ready.
- Initialize asynchronous I/O. The lightweight processes created by **aioinit()** inherit the attributes of the calling process, including its current priority and access to open file descriptors.
- Isolate any CPUs that are dedicated to real-time work (see "Restricting a CPU From Scheduled Work" on page 101)—or create the Frame Schedulers (see "Starting the Schedulers" on page 124).
- Assign real-time processes to their CPUs. The asynchronous I/O processes continue to be scheduled according to their priority in whatever CPUs remain available.

Scheduling Asynchronous I/O

You schedule an input or output operation by calling **aio_read()** or **aio_write()**, passing an *aiocb* structure to describe the operation (see the aio_read(3) and aio_write(3) reference pages). The operation is queued to the file descriptor, but it will not execute until one of the asynchronous I/O processes is available. The return code from the library call says nothing about the I/O operation itself; it merely indicates whether or not the *aiocb* could be recorded.

You can find examples of the use of **aio_read()**, **aio_write()**, and **aio_fsync()** in the program beginning on "Asynchronous I/O Example" on page 239.

You can schedule a list of operations using **lio_listio()** (see the lio_listio(3) reference page). The advantage of this function is that you can request a single notification (which can be a signal or a callback) when all of the operations in the list are complete. Alternatively, you can be notified of the completion of each one as it happens.

Note: It is important to use a given *aiocb* for only one operation at a time, and to not modify an *aiocb* until its operation is complete.

When an asynchronous I/O process is free, it takes a queued *aiocb* and performs the equivalent function to **lseek()** (if a file position is specified), then the equivalent of **read()** or **write()**. The asynchronous process may be blocked for more or less time, depending on the file or device, and depending on the options that were specified when it was opened. When the operation is complete, the asynchronous process notifies the initiating process using the method requested in the *aiocb*.

You can cancel a started operation, or all pending operations for a given file descriptor, using **aio_cancel()** (see the aio_cancel(3) reference page).

Assuring Data Integrity

With sequential I/O, you call **fsync()** to ensure that all buffered data has been written. However, you cannot use **fsync()** with asynchronous I/O, since you are not sure when the **write()** calls will execute. The **aio_fsync()** function queues the equivalent of an **fsync()** call for asynchronous execution. This function takes an *aiocb*. The file descriptor in it specifies which file is to be synchronized.

The **fsync()** operation is done following all other asynchronous operations that are pending when **aio_fsync()** is called. Completion of the synchronize operation (it can take considerable time, depending on how much output data has been buffered) is reported in the same ways as completion of a read or write (see the next topic). The example program starting in "Asynchronous I/O Example" on page 239 contains calls to **aio_fsync()**.

Checking the Progress of Asynchronous Requests

You can test the progress and completion of an asynchronous operation by polling. Your program can be informed of the completion of an operation in a variety of ways. All of the methods discussed here are demonstrated in the example program that starts in "Asynchronous I/O Example" on page 239.

Polling for Status

You can check the progress of any asynchronous operation (including an **aio_fsync()**) using **aio_error()**. As long as the operation is incomplete, this function returns EIINPROGRESS. When the operation is complete, you can check the final return code from **read()**, **write()**, or **fsync()** using **aio_return()** (see the aio_error(3) and aio_return(3) reference pages).

To see in an example of polling for status, see code shown in page 248. It waits for an asynchronous operation to complete using a function has the general form shown in Example 8-2.

Example 8-2 Polling for Asynchronous Completion

```
int waitForEndOfAsyncOp(aiocb *pab)
{
    while (EINPROGRESS == (ret = aio_error(pab)))
        sginap(0);
    return ret;
}
```

The function result is the final return code from the read, write, or sync operation that was started. Under the Frame Scheduler, the call to **sginap()** would be replaced with a call to **frs_yield()**.

Checking for Completion

In the *aiocb*, the program can specify one of three things to be done when the operation is complete:

- Nothing; take no action.
- Send a signal of a specified number.
- Invoke a callback function directly from the asynchronous process.

In addition, the **aio_suspend()** function blocks its caller until one of a list of pending operations is complete (see the aio_suspend(3) reference page).

These choices give you a wide variety of design options. Your program can

- periodically poll the *aiocb* using **aio_error()** until it completes (shown in Example 8-2)
- use **aio_suspend()** to wait until one of a list of operations completes
- set up an empty signal handler function and use **sigsuspend()** or **sigwait()** to wait until a signal arrives (see the sigsuspend(2) and sigwait(3) reference pages)
- use either a signal handler function or a callback function to report completion—for example, the function can post a semaphore.

Most of these methods are demonstrated in the program starting in "Asynchronous I/O Example" on page 239.

Tip: When operating under the Frame Scheduler, a handler or callback function can simply set a flag. An activity process can test the flag in each minor frame, calling **frs_yield()** immediately if the flag is not set.

Establishing a Completion Signal

You request a signal from an asynchronous operation by setting these values in the *aiocb* (refer to */usr/include/aio.h* and */usr/include/sys/signal.h*):

aio_sigevent.sigev_notify	Set to SIGEV_SIGNAL.
aio_sigevent.sigev_signo	The number of the signal. This should be one of the POSIX real-time signal numbers (see "Signals" on page 38).
aio_sigevent.sigev_value	A value to be passed to the signal handler. This can be used to inform the signal handler of which I/O operation has completed; for example, it could be the address of the *aiocb*.

When you set up a signal handler for asynchronous completion, do so using **sigaction()** and specify the SA_SIGINFO flag (see the sigaction(2) reference page). This has two benefits: any new completion signal that arrives while the first is being handled is queued; and the *aio_sigev.sigev_value* word is passed to the handler in a *siginfo* structure.

Establishing a Callback Function

You request a callback at the end of an asynchronous operation by setting the following values in the *aiocb*:

aio_sigevent.sigev_notify	Set to SIGEV_CALLBACK.
aio_sigevent.sigev_func	The address of the callback function. Its prototype must be `void` *functionName*`(union sigval);`
aio_sigevent.sigev_value	A word to be passed to the callback function. This can be used to inform the function of which I/O operation has completed; for example, it could be the address of the *aiocb*.

The callback function is invoked from the asynchronous process when the **read()**, **write()** or **fsync()** operation finishes. This notification method has the lowest overhead and shortest latency, but it requires careful design to avoid race conditions in the use of shared variables.

The asynchronous processes are created with **sproc()**, so they share the address space of the process that initialized asynchronous I/O. They typically execute in a different CPU from the real-time processes using that address space. Since the callback function could be entered at any time, it must coordinate its use of shared data structures. This is a good place to use a lock (see "Locks" on page 35). Locks have very low overhead in cases such as this, where is likely to be little contention for the use of the lock.

Tip: You can call **aio_read()** or **aio_write()** from within a callback function or within a signal handler. This lets you start another operation with the least delay.

The code in in Example 8-3 demonstrates a hypothetical set of subroutines to schedule asynchronous reads and writes using a single *aiocb*. The principle functions and global variables it uses are:

pendingIO	An array of records, each holding one request for an I/O operation.
dontTouchThatStuff	A lock used to gain exclusive use of *pendingIO*.

scheduleRead() A function that accepts a request to read some amount of data, from a specified file descriptor, at a specified file offset. It places the request in *pendingIO* and then, if no asynchronous operation is under way, initiates it.

yeahWeFinishedOne() The callback function that is entered when an asynchronous operation completes. If any more operations are pending, it initiates one.

initiatePending() A function that initiates one selected pending operation. It prepares the aiocb structure, including the specification of **yeahWeFinishedOne()** as the callback function. The lock *dontTouchThatStuff* must be held before this function is called.

Note: The code in Example 8-3 is not intended to be realistic and is not recommended as a model. In order to demonstrate the use of callback functions and the *aiocb*, it essentially duplicates work that could be done by the **lio_listio()** feature of asynchronous I/O.

Example 8-3 Set of Functions to Schedule Asynchronous I/O

```
#define _ABI_SOURCE
#include <signal.h>
#include <aio.h>
#include <ulocks.h>
#define MAX_PENDING 10
#define STATUS_EMPTY 0
#define STATUS_ACTIVE 1
#define STATUS_PENDING 2
static struct onePendingIO {
    int status;
    int theFile;
    void *theData;
    off_t theSize;
    off_t theSeek;
    int readNotWrite;
    } pendingIO[MAX_PENDING];
static unsigned numPending;
static struct aiocb theAiocb;
static ulock_t dontTouchThatStuff;
static unsigned scanner;
```

```
static void initiatePending(int P);
static void
yeahWeFinishedOne(union sigval S)
{
    ussetlock(dontTouchThatStuff);
    pendingIO[S.sival_int].status = STATUS_EMPTY;
    if (numPending)
    {
        while (pendingIO[scanner].status != STATUS_PENDING)
        {
            if (++scanner >= MAX_PENDING)
                scanner = 0;
        }
        initiatePending(scanner);
    }
    usunsetlock(dontTouchThatStuff);
}
static void
initiatePending(int P) /* lock must be held on entry */
{
    theAiocb.aio_fildes = pendingIO[P].theFile;
    theAiocb.aio_buf = pendingIO[P].theData;
    theAiocb.aio_nbytes = pendingIO[P].theSize;
    theAiocb.aio_offset = pendingIO[P].theSeek;
    theAiocb.aio_sigevent.sigev_notify = SIGEV_CALLBACK;
    theAiocb.aio_sigevent.sigev_func = yeahWeFinishedOne;
    theAiocb.aio_sigevent.sigev_value.sival_int = P;
    if (pendingIO[P].readNotWrite)
        aio_read(&theAiocb);
    else
        aio_write(&theAiocb);
    pendingIO[P].status = STATUS_ACTIVE;
    --numPending;
}
/*public*/ int
scheduleRead( int FD, void *pdata, off_t len, off_t pos )
{
    int j;
    if (numPending >= MAX_PENDING)
        likeTotallyFreakOut();
    ussetlock(dontTouchThatStuff);
    for(j=0; pendingIO[j].status != STATUS_EMPTY; ++j)
        ;
    pendingIO[j].theFile = FD;
    pendingIO[j].theData = pdata;
```

```
    pendingIO[j].theSize = len;
    pendingIO[j].theSeek = pos;
    pendingIO[j].readNotWrite = 1;
    pendingIO[j].status = STATUS_PENDING;
    if (1 == ++numPending)
        initiatePending(j);
    usunsetlock(dontTouchThatStuff);
}
```

Holding Callbacks Temporarily

You can temporarily prevent callback functions from being entered using the **aio_hold()** function. This function is not defined in the POSIX standard; it is added by the MIPS ABI standard. Use it as follows:

- Call **aio_hold(**AIO_HOLD_CALLBACK**)** to prevent any callback function from being invoked.
- Call **aio_hold(**AIO_RELEASE_CALLBACK**)** to allow callback functions to be invoked. Any that were held are now called.
- Call **aio_hold(**AIO_ISHELD_CALLBACK**)** returns 1 if callbacks are currently being held; otherwise it returns 0.

Multiple Operations to One File

When you queue multiple operations to a single file descriptor, the asynchronous I/O package does not always guarantee the order of their execution. There are three ways you can ensure the sequence of operations.

You can open any output file descriptor passing the flag O_APPEND (see the open(1) reference page). Asynchronous write requests to a file opened with O_APPEND are executed in the sequence of the calls to **aio_write()** or the sequence they are listed for **lio_listio()**. You can use this feature to ensure that a sequence of records is appended to a file in sequence.

For files that support **lseek()**, you can specify any order of operations by specifying the file offset in the *aiocb*. The asynchronous process executes an absolute seek to that offset as part of the operation. Even if the operations are not performed in the sequence they were requested, the data is transferred in sequence. You can use this feature to ensure that multiple requests for sequential disk input are stored in sequential locations.

For non-disk input operations, the only way you can be certain that operations are done in sequence is to schedule them one at a time, waiting for each one to complete.

Synchronous Writing and Direct Writing

Two options of **open()** give you more control over the timing of output.

Using Synchronous Writing

When you open a disk file and do not specify the O_SYNC flag, a call to **write()** for that file returns as soon as the data has been copied to a buffer managed by the device driver.

The actual disk write may not take place until considerable time has passed. A common pool of disk buffers is used for all disk files. Disk buffering is integrated with the virtual memory paging mechanism. A daemon executes periodically and initiates output of buffered blocks according to the age of the data and the needs of the system.

Tip: The number of disk blocks that are written in each output operation is set by the *dwcluster* tuning variable. The system administrator can adjust this value with *systune* (see the systune(1) reference page).

The default management of disk output improves performance in general but has two drawbacks:

- All output data must be copied from the buffer in process address space to a buffer in the kernel address space. For small or infrequent writes, the copy time is negligible, but for large quantities of data it adds up.
- You do not know when the written data is actually safe on disk. A system crash could prevent the output of a large amount of buffered data.

You can force the writing of all pending output for a file by calling **fsync()** (see the fsync(2) reference page). This gives you a way of creating a known checkpoint of a file. However, **fsync()** blocks until all buffered writes are complete, possibly a long time.

When you open a disk file specifying O_SYNC, each call to **write()** blocks until the data has been written to disk. This gives you a way of ensuring that all output is complete as it is created. By combining O_SYNC with asynchronous I/O, you can let the asynchronous process suffer the delay.

The O_SYNC option requires completed output even when the amount of data written is less than the physical blocksize of the disk, or when the output data does not align with the physical boundaries of disk blocks. This can lead to writing and rewriting the same disk blocks, wasting time. A file opened with O_SYNC also copies data to kernel memory before writing.

Using Direct Writing

You can avoid both sources of delay by using the option O_DIRECT. Under this option, writes to the file take place directly from your program's buffer—the data is not copied to a buffer in the kernel first. In order to use O_DIRECT you are required to write data in quantities that are multiples of the disk blocksize. This ensures that a block is written only once. (The requirements for O_DIRECT use are documented in the open(2) and fcntl(2) reference pages.)

Control does not return from an O_DIRECT **write()** until the disk write is complete. However, you can open a file O_DIRECT and use the use file descriptor for asynchronous I/O.

Performance Comparison

The data displayed in Figure 8-1 was collected on a 4-processor Challenge system using a test program that wrote approximately 250,000 bytes of binary data using a specified blocksize and one of three options:

- default: asynchronous buffered write
- synchronous writes (option O_SYNC)
- direct writes (option O_DIRECT)

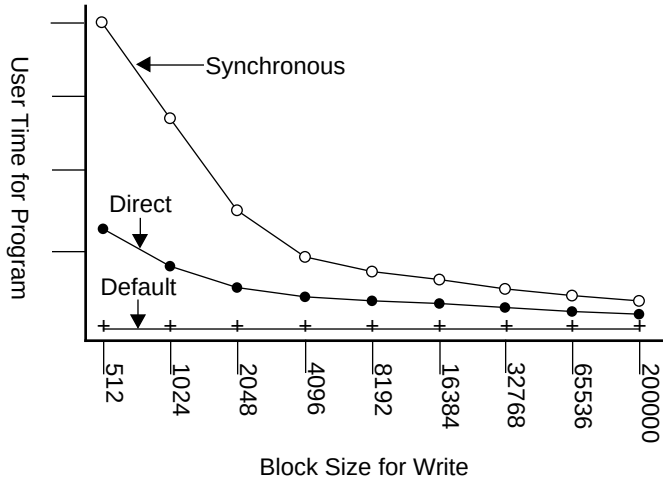

Figure 8-1 Effect of Blocksize on write() Performance

The values in Table 8-1reflect the total execution time for one run of the program, as reported by the *time* command (see the time(1) reference page).

Table 8-1 Data on Which Figure 8-1 is Based

Blocksize	O_SYNC	O_DIRECT	Asynchronous
512	40.4	13.9	2.7
1024	25.3	8.5	2.6
2048	12.9	5.8	2.6
4096	8.5	4.4	2.6
8192	6.2	3.7	2.6
16384	5.0	3.4	2.6
32768	4.4	3.1	2.5
65536	4.1	3.0	2.5
200000	3.9	2.9	2.5

Blocksize was almost irrelevant for asynchronous writes, because the only delay was the time to switch to kernel mode and block-copy the data from the program buffer to a kernel buffer. The actual disk operations occurred asynchronously, in another CPU, and so are not reflected in the *time* output. As shown in Figure 8-1, O_DIRECT is considerably faster than O_SYNC.

Using a Delayed System Buffer Flush

When your application has both clearly defined times when all unplanned disk activity should be prevented, and clearly defined times when disk activity can be tolerated, you can use the **syssgi()** function to control the kernel's automatic disk writes.

Prior to a critical section of length *s* seconds that must not be interrupted by unplanned disk writes, use **syssgi()** as follows:

```
syssgi(SGI_BDFLUSHCNT,s);
```

The kernel will not initiate any deferred disk writes for *s* seconds. At the start of a period when disk activity can be tolerated, initiate a flush of the kernel's buffered writes with **syssgi()** as follows:

```
syssgi(SGI_SSYNC);
```

Note: This technique is most useful in a uniprocessor—code executing in an isolated CPU of a multiprocessor is not affected by kernel disk writes.

Guaranteed-Rate I/O

Under specific conditions, your program can demand a guaranteed rate of data transfer. You would use this feature, for example, to ensure input of picture data for real-time video display, or to ensure disk output of high-speed telemetry data capture.

Guaranteed-Rate I/O Basics

Guaranteed-rate I/O (GRIO) is applied on a file basis. The file must have these characteristics for any guarantee to be granted:

- The file must be managed by XFS. EFS, the older IRIX file system, does not support GRIO.
- The file must be contained in the real-time subvolume of a logical volume created by XLV.

  The real-time subvolume of an XLV volume can span multiple disk partitions, and can be striped. The real-time subvolume differs from the more common data subvolume in that it contains only data, no file system management data such as directories or inodes.

  Note: Real-time subvolumes cannot include RAID partitions.
- The predictive failure analysis feature and the thermal recalibration feature of the drive firmware must be disabled, as these can make device access times unpredictable.

You can request either of two types of guarantee. A *hard guarantee* asks XFS and IRIX to subordinate all other considerations, including data integrity, to meet the guaranteed rate. A *soft guarantee* asks IRIX to make its best effort at the rate, accepting that error correction might cause glitches.

You can qualify either type of guarantee as being for Video On Demand (VOD), indicating a particular, special use of a striped volume. These three types of guarantee are discussed further in the following topics.

For information about using XFS, XLV, and how to prepare a real-time subvolume for GRIO, see the *IRIX Advanced Site and Server Administrator Guide*. For an example of how the **grio_request()** function is used, see the function starting in “Guaranteed-Rate Request” on page 258.

Creating a Real-time File

You can only request a guaranteed rate from a real-time disk file. A real-time disk file is identified by the fact that it is stored within the real-time subvolume of an XFS logical volume.

The file management information for all files in a volume (the directories as well as XFS management records) are stored in the data subvolume. A real-time subvolume contains only the data of real-time files. A real-time subvolume comprises an entire disk device or partition and uses a separate SCSI controller from the data subvolume. Because of these constraints, the GRIO facility can predict the data rate at which it can transfer the data of a real-time file.

You create a real-time file in the following steps, which are illustrated in Example 8-4.

1. Open the file with the options O_CREAT, O_EXCL, and O_DIRECT. That is, the file must not exist at this point, and must be opened for direct I/O (see “Using Direct Writing” on page 167).
2. Modify the file descriptor to set its extent size, which is the minimum amount by which the file will be extended when new space is allocated to it, and also to establish that the new file is a real-time file. This is done using **fcntl()** with the FS_FSSETXATTR command. Check the value returned by **fcntl()** as several errors can be detected at this point.

   The extent size must be chosen to match the characteristics of the disk; for example it might be the “stripe width” of a striped disk.
3. Write any amount of data to the new file. Space will be allocated in the real-time subvolume instead of the data subvolume because of step (2). Check the result of the first **write()** call carefully, since this is another point at which errors could be detected.

Once created, you can read and write a real-time file the same as any other file, except that it must always be opened with O_DIRECT. You can use a real-time file with asynchronous I/O, provided it is not under a guarantee (see “Sharing Access to Guaranteed Files” on page 174).

Example 8-4 Function to create a real-time file

```
#include <sys/fcntl.h>
#include <sys/fs/xfs_itable.h>
int createRealTimeFile(char *path, __uint32_t esize)
{
   struct fsxattr attr;
   bzero((void*)&attr,sizeof(attr));
   attr.fsx_xflags = XFS_FLAG_REALTIME;
   attr.fsx_extsize = esize;
   int rtfd = open(path, O_CREAT + O_EXCL + O_DIRECT );
   if (-1 == rtfd)
      {perror("open new file"); return -1; }
   if (-1 == fcntl(rtfd, F_FSSETXATTR, &attr) )
      {perror("fcntl set rt & extent"); return -1; }
   return rtfd; /* first write to it creates file*/
}
```

Requesting a Guarantee

To obtain a guaranteed rate, a program places a reservation for a specified part of the I/O capacity of a file. In the request, the program specifies

- the file descriptor to be used
- the start time and duration of the reservation
- the time unit of interest, typically 1 second
- the amount of data required in any one unit of time

For example, a reservation might specify: now, for 90 minutes, 1 megabyte per second. A process places a reservation by calling **grio_request()** (refer to the grio_request(2) reference page).

XFS (in a GRIO daemon) keeps information on the transfer capacity of all real-time subvolumes, as well as the capacity of the controllers and busses to which they are attached. When you request a reservation, XFS tests whether it is possible to transfer data at that rate, from that file, during that time period.

This test considers the capacity of the hardware as well as any other reservations that apply during the same time period to the same subvolume, drives, or controllers. Each reservation consumes some part of the total capacity.

When XFS predicts that the guaranteed rate can be met, it accepts the reservation. Over the reservation period, the available capacity of the subvolume is reduced by the promised rate. Other processes can place reservations against any capacity that remains.

If XFS predicts that the guaranteed rate cannot be met at some time in the reservation period, XFS returns the maximum data rate it could supply. The program can reissue the request for that available rate. However, this is a new request that is evaluated afresh.

During the reservation period, the process can use **read()** and **write()** to transfer up to the guaranteed number of bytes in each time unit. XFS raises the priority of requests as needed in order to ensure that the transfers take place. However, a request that would transfer more than the promised number of bytes within a 1-second unit is blocked until the start of the next time unit.

Releasing a Guarantee

A guarantee ends under three circumstances,

- when the process calls **grio_remove_request()** (see the grio_request(2) reference page)
- when the requested duration expires
- when all file descriptors held by the requesting process that refer to the guaranteed file are closed (an exception is discussed in the next topic)

When a guarantee ends, the guaranteed transfer capacity becomes available for other processes to reserve. When a guarantee expires but the file is not closed, the file remains usable for ordinary I/O, with no guarantee of rate.

Sharing Access to Guaranteed Files

Other processes can use a file or the hardware it resides on, even though guarantees are active. XFS never grants guarantees for the whole capacity of the I/O path; it always reserves some capacity. Non-guaranteed I/O requests are delayed within any 1-second interval until guarantees have been met, and may be executed bit by bit in smaller units, but they will finally be completed.

Once a guarantee is granted, the guarantee is uniquely identified with the file, through the I-node number, and with the process, through the process ID. However, it is possible to have the same file (I-node) open under different file descriptors. This has important implications:

- All requests from that process to that file are handled under the guarantee—even if they are issued to different file descriptors. (It is not possible for single process to request both guaranteed and nonguaranteed I/O to the same file.)
- It is not possible for one process to have two guarantees on the same file. The second guarantee request is rejected, even if it uses a different file descriptor.
- Only the process that received a guarantee can remove the guarantee—that is, **grio_request()** and **grio_remove_request()** must be called from the same process ID.
- A rate guarantee is not shared by other processes created with **fork()** or **sproc()**—even though they may have shared access to the file descriptor used with **grio_request()**. Each process that wants guaranteed access must obtain its own guarantee.

The last point has the important implication that you cannot use a rate guarantee with asynchronous I/O. An input requested using **aio_read()** is executed by a different process than the one that requested the guaranteed rate. That read is treated as non-guaranteed, and executed on a time-available basis.

A complication can arise when a guaranteed rate is obtained by one process of a process group created with **sproc()**. When the PR_SDIR flag (synchronize file descriptors; see the sproc(2) reference page) is used, a rate guarantee obtained by one process of the group cannot be terminated simply by closing all file descriptors. It can be terminated explicitly, or by the time expiring, or by the whole process group terminating.

Hard Guarantees

When a program requests a hard guarantee, it asserts that nothing, not even data integrity, should interfere with data transfer. A hard guarantee can be given only when

- the SCSI controller or controllers that attach the real-time subvolume have only disks attached to them—no tapes or other nondisk devices

  I/O to a non-disk device can delay disk data transfer.

- sector remapping in the drive firmware, as well as any device driver retry and correction mechanisms, is disabled

  Error retry can introduce unpredictable delays in data transfer.

When your program requests I/O under a hard guarantee, any device error is returned directly to the program. No effort is made to retry the failure. If the drive contains a bad sector, the bad sector is read and returned with no indication of error.

Soft Guarantees

A soft guarantee can be granted for a subvolume that has error retry and sector remapping enabled. Your program accepts a possible, occasional failure to meet the specified rate in exchange for having errors retried and possibly corrected.

In addition, a soft guarantee can be granted when the disk controller also controls non-disk devices such as scanners and printers. Use of these devices during the guarantee period can prevent the guaranteed rate from being met.

Video On Demand (VOD) Guarantees

You specify the VOD disk layout as a modifier on either a hard or soft guarantee (see the grio_request(2) reference page and */usr/include/sys/grio.h*). A VOD guarantee can be requested only for a *striped volume*. In a striped volume, fixed-size segments of the volume space that are logically sequential ("stripes") are physically located on successive drives. The potential data rate of a striped volume is higher because the multiple drives can be used in parallel.

However, in order to achieve the higher rate, the striped volume must be used concurrently by multiple processes, each reading in a different stripe. The maximum rate is reached when as many processes are reading sequentially in stripe-sized units as the subvolume has drives.

When a program requests a VOD guarantee, it must specify a data rate that equals one stripe-width per second. VOD guarantees can be given concurrently to several processes for the same subvolume. As long as all the processes read different stripes, the guaranteed rate can be sustained for each.

When the first VOD guarantee is granted against a striped volume, the XFS system begins VOD-style I/O scheduling for that volume. This establishes a strict cyclic rotation of time intervals during which any disk in the striped volume can be read. In general, a process must be ready for access when its turn in the rotation comes up. If it is not ready, it can be delayed by as many seconds as there are disks in the volume.

- The first access by a process to a striped volume under VOD scheduling can be delayed.
- If the process fails to request its next access before the beginning of the next second of time, it can miss its assigned slot and be delayed.
- When a process uses **lseek()** to move to a stripe other than the next stripe in sequence, its next I/O request can be delayed.

Chapter 9

Managing Device Interactions

A real-time program is defined by its close relationship to external hardware. This chapter reviews the ways that IRIX gives you to access and control external devices:

- "Device Drivers" on page 177 summarizes the role that device drivers play in the IRIX system, and points to sources for information on how you can write a device driver.
- "SCSI Devices" on page 182 describes the facilities of the dslib package, with which you can write code to control SCSI devices directly.
- "The VME Bus" on page 185 describes two methods for accessing the most popular interface for real-time devices.
- "Serial Ports" on page 193 summarizes the use of serial ports for real-time input and output.
- "External Interrupts" on page 195 on page 186 summarizes the facilities offered by the External Interrupt device driver.

Device Drivers

It is a basic concept in UNIX that all I/O is done by reading or writing files. All I/O devices—disks, tapes, printers, terminals, and VME cards—are represented as files in the file system. Each device is represented by an entry in the */dev* file system hierarchy. The purpose of each *device special file* is to associate a device name with a a *device driver*, a module of code that is loaded into the kernel either at boot time or dynamically, and is responsible for operating that device at the kernel's request.

Note: This section contains an overview for readers who are not familiar with the details of the UNIX I/O system. All these points are covered in much greater detail in the *IRIX Device Driver Programming Guide* (see "Other Useful Books" on page xxiii).

How Devices Are Defined

When a device special file is created in the */dev* file system, it is associated with the device driver that will manage it. The connection between the device name and the device driver is made through major and minor *device numbers*, which are recorded with the device name in the file system. To see these numbers, try a command such as

```
ls -l /dev/dsk
```

Device creation is documented in the system(4) reference page, the makedev(1) and mknod(1) reference pages, and in the */dev/MAKEDEV* script. Device administration is covered in detail in the *IRIX Advanced Site and Server Administration Guide*.

The major device number selects the device driver. The minor number is passed to the device driver each time it is entered; it encodes such useful parameters as logical unit number or density. In some cases, a device is represented by more than one name in the */dev* hierarchy. This associates it with more than one device driver, or else causes the device driver to treat the device differently depending on the name through which it is accessed.

For example, a disk device can appear in both the */dev/dsk* and the */dev/rdsk* directories, and the same disk can appear under several names in each directory, with each name standing for a different partition of the disk. (The naming of disk devices is documented in the dksc(7) reference page.) Again, a tape drive usually appears multiple times in */dev/mt*, with each name receiving different treatment from the tape device driver—names containing "ns," for example, are written with integers in non-byte-swapped order for compatibility with other systems.

How Devices Are Used

To use a device, a process opens the special device file by passing its pathname to **open()** (see the open(2) reference page). For example, a generic SCSI device might be opened by a statement such as this.

```
int scsi_fd = open("/dev/scsi/sc0d11l0",O_RDWR);
```

The returned integer is the *file descriptor*, a number that indexes an array of control blocks maintained by IRIX in the address space of each process. With a file descriptor, the process can call other system functions that give access to the device. Each of these system calls is implemented in the kernel by transferring control to an entry point in the device driver.

Device Driver Entry Points

Each device driver supports one or more of the following operations:

open	Notifies the driver that a process wants to use the device.
close	Notifies the driver that a process is finished with the device.
interrupt	Entered by the kernel upon a hardware interrupt, notes an event reported by a device, such as the completion of a device action, and possibly initiates another action.
read	Entered from the function **read()**, transfers data from the device to a buffer in the address space of the calling process.
write	Entered from the function **write()**, transfers data from the calling process's address space to the device.
control	Entered from the function **ioctl()**, performs some kind of control function specific to the type of device in use.

Not every driver supports every entry point. For example, the generic SCSI driver (see "Generic SCSI Device Driver" on page 184) supports only the open, close, and control entries.

Device drivers in general are documented with the device special files they support, in volume 7 of the reference pages. For a sampling, review:

- dsk(7m), documenting the standard IRIX SCSI disk device driver
- smfd(7m), documenting the diskette and optical diskette driver
- tps(7m), documenting the SCSI tape drive device driver
- plp(7), documenting the parallel line printer device driver
- klog(7), documenting a “device” driver that is not a device at all, but a special interface to the kernel

If you review a sample of entries in volume 7, as well as other reference pages that are called out in the topics in this chapter, you will understand the wide variety of functions performed by device drivers.

Device Driver Use

Here is a typical device operation scenario:

1. A process, running on any CPU, calls **read()**, **write()**, or **ioctl()** requesting a device operation.
2. After switching to kernel mode and the kernel’s address space, the system call enters the device driver’s read, write, or control entry point. The system is still executing on behalf of the user process, on the CPU where the call was made.
3. The device driver initiates an I/O or control action at the device.
4. Some operations can be completed immediately (in which case go to step 10). Others require a delay while the device operates. The device driver suspends itself with a device driver service routine such as **sleep()** or **psema()**. (See the sleep(d3) or psema(d3x) reference pages in *IRIX Device Driver Reference Pages*.)
5. Since the current process is suspended, the kernel selects another process to run on the calling CPU.
6. The I/O action ends with a hardware interrupt, causing the interrupt-handling CPU—which is not necessarily the same CPU that initiated the operation—to enter the device driver’s interrupt handler.

7. The interrupt handler awakens the waiting process with a device driver service routine such as **wakeup()** or **vsema()**. (See the *IRIX Device Driver Reference Pages*.)
8. The interrupt handler possibly initiates another I/O action.
9. The process that requested the operation is now dispatchable, and the kernel schedules it to run, based on its priority, affinity, and other factors (“Scheduling Concepts” on page 86).
10. The process executes the final instructions in the device driver code and switches mode back to the user address space, the function complete.

Taking Control of Devices

When your program needs direct control of a device, you have the following choices:

- If it is a device for which IRIX or the device manufacturer distributes a device driver, find the device driver reference page in volume 7 to learn the device driver’s support for **read()**, **write()**, **mmap()**, and **ioctl()**. Use these functions to control the device.
- If it is a VME device without bus master capability, you can control it directly from your program using programmed I/O or user-initiated DMA. Both options are discussed under “The VME Bus” on page 185.
- If it is a VME device with bus master (on-board DMA) capability, you should receive an IRIX device driver from the OEM. Consult the *IRIX Advanced Site and Server Administrator Guide* to install the device and its driver. Read the OEM reference page to learn the device driver’s support for **read()**, **write()**, and **ioctl()**.
- If it is a SCSI device that does not have built-in IRIX support, you can control it from your own program using the generic SCSI device driver. See “Generic SCSI Device Driver” on page 184 on page 175.

In the remaining case, you have a device with no driver. In this case you must create a device driver. This process is documented in the *IRIX Device Driver Programmer’s Guide*, which contains extensive information and sample code.

A device driver operates in the kernel domain and uses a large number of kernel functions that are not available to normal processes—for example, the device driver has to be able to map a buffer in the user process address space into the kernel address space and then into real memory so that it can be used for I/O. The *IRIX Device Driver Reference Pages* document these kefunctions.

SCSI Devices

The SCSI interface is the principal way of attaching disk, cartridge tape, CD-ROM, and digital audio tape (DAT) devices to the system. It can be used for other kinds of devices, such as scanners and printers.

IRIX contains device drivers for disk and tape. Other SCSI devices are controlled through a generic device driver that must be extended with programming for a specific device.

SCSI Hardware on CHALLENGE and Onyx Systems

Two SCSI-2 controllers are incorporated in each POWER Channel-2 board. Additional controllers can be added in two groups of 3 on HIO modules, for a maximum of 8 controllers per channel. The SCSI controllers are DMA devices that make near-optimum use of the POWER Channel-2 bus bandwidth.

Using 8-bit data transfers, a data rate of approximately 7 megabytes per second can be reached, per controller. Using 16-bit SCSI, data rates near 14 MB per second can be achieved, per controller.

Note: These data rates can be achieved by configuring the system to perform DMA directly into the requesting process's address space without buffering. This requires the use of Direct file access for disk files (see "Synchronous Writing and Direct Writing" on page 166). The generic SCSI device driver also performs DMA directly in or out of the process's buffers (see "Generic SCSI Device Driver" on page 184).

All controllers are capable of differential output and so can be cabled longer distances from the system cabinet.

SCSI Adapter Support

The detailed, board-level programming of the host SCSI adapters is done by an IRIX-supplied host adapter driver. The services of this driver are available to the SCSI device drivers that manage the logical devices. The host adapter drivers handle the low-level communication over the SCSI interface, such as programming the SCSI interface chip or board, negotiating synchronous or wide mode, and handling disconnect/reconnect. SCSI device drivers call on host adapter drivers using indirect calls through a table of adapter functions.

System Disk Device Driver

The naming conventions for disk and tape device files are documented in the intro(7) reference page. In general, devices in */dev/*[*r*]*dsk* are disk drives, and devices in */dev/*[*r*]*mt* are tape drives.

Disk devices in */dev/*[*r*]*dsk* are operated by the SCSI disk controller, which is documented in the dks(7) reference page. It is possible for a program to open a disk device and read, write, or memory-map it, but this is almost never done. Instead, programs open, read, write, or map files; and the EFS or XFS file system interacts with the device driver.

System Tape Device Driver

Tape devices in */dev/*[*r*]*mt* are operated by the magnetic tape device driver, which is documented in the tps(7) reference page. Users normally control tapes using such commands as *tar* and *mt* (see the tar(1) and mt(1) reference pages), but it is also common for programs to open a tape devices and then to use **read()**, **write()**, and **ioctl()** to interact with the device driver.

Since the tape device driver supports the read/write interface, you can schedule tape I/O through the asynchronous I/O interface (see "Asynchronous I/O Basics" on page 154). You need to take pains to ensure that asynchronous writes to a tape are executed in the proper sequence; see "Multiple Operations to One File" on page 156.

Generic SCSI Device Driver

Generally, non-disk, non-tape SCSI devices are installed in the */dev/scsi* directory. Devices so named are controlled by the generic SCSI device driver, which is documented in the ds(7m) reference page.

Unlike most kernel-level device drivers, the generic SCSI driver does not support interrupts, and does not support the **read()** and **write()** functions. Instead, it supports a wide variety of **ioctl()** functions that you can use to issue SCSI commands to a device. In order to invoke these operations you prepare a *dsreq* structure describing the operation and pass it to the device driver. Operations can include input and output as well as control and diagnostic commands.

The programming interface supported by the generic SCSI driver is quite primitive. A library of higher-level functions makes it easier to use. This library is documented in the dslib(3x) reference page. The most important functions in it are listed below:

- **dsopen()**, which takes a device pathname, opens it for exclusive access, and returns a *dsreq* structure to be used with other functions.
- **fillg0cmd()**, **fillg1cmd()**, and **filldsreq()**, which simplify the task of preparing the many fields of a *dsreq* structure for a particular command.
- **doscsireq()**, which calls the device driver and checks status afterward.

The *dsreq* structure for some operations specifies a buffer in memory for data transfer. The generic SCSI driver handles the task of locking the buffer into memory (if necessary) and managing a DMA transfer of data.

When the **ioctl()** function is called (through **doscsireq()** or directly), it does not return until the SCSI command is complete. You should only request a SCSI operation from a process that can tolerate being blocked.

Upon the basic dslib functions are built several functions that execute specific SCSI commands, for example, **read08()** performs a read. However, there are few SCSI commands that are recognized by all devices. Even the read operation has many variations, and the **read08()** function as supplied is unlikely to work without modification. The dslib library functions are not complete. Instead, you must alter them and extend them with functions tailored to a specific device.

The source for dslib, and some example programs that use dslib, can be found in the 4DGifts distribution, in */usr/people/4Dgifts/examples/devices/devscsi*.

CD-ROM and DAT Audio Libraries

A library of functions that enable you to read audio data from an audio CD in the CD-ROM drive is distributed with IRIX. This library was built upon the generic SCSI functions supplied in dslib. The CD audio library is documented in the cdintro(3) reference page.

A library of functions that enable you to read and write audio data from a digital audio tape is distributed with IRIX. This library was built upon the functions of the magnetic tape device driver. The DAT audio library is documented in the DTintro(3) reference page.

The VME Bus

Each CHALLENGE XL, POWER CHALLENGE, or Onyx system includes full support for the VME interface, including all features of Revision C.2 of the VME specification, and the A64 and D64 modes as defined in Revision D. VME devices can access system memory addresses, and devices on the system bus can access addresses in the VME address space.

The naming of VME devices in */dev/vme*, and other administrative issues, are covered in the usrvme(7) reference page.

CHALLENGE Hardware Nomenclature

A number of special terms are used to describe the multiprocessor CHALLENGE support for VME. The terms are described in the following list. Their relationship is shown graphically in Figure 9-1.

POWERpath-2 Bus	The primary system bus, connecting all CPUs and I/O channels to main memory.
POWER Channel-2	The circuit card that interfaces one or more I/O devices to the POWERpath-2 bus.
F-HIO card	Adapter card used for cabling a VME card cage to the POWER Channel
VMECC	VME control chip, the circuit that interfaces the VME bus to the POWER Channel.

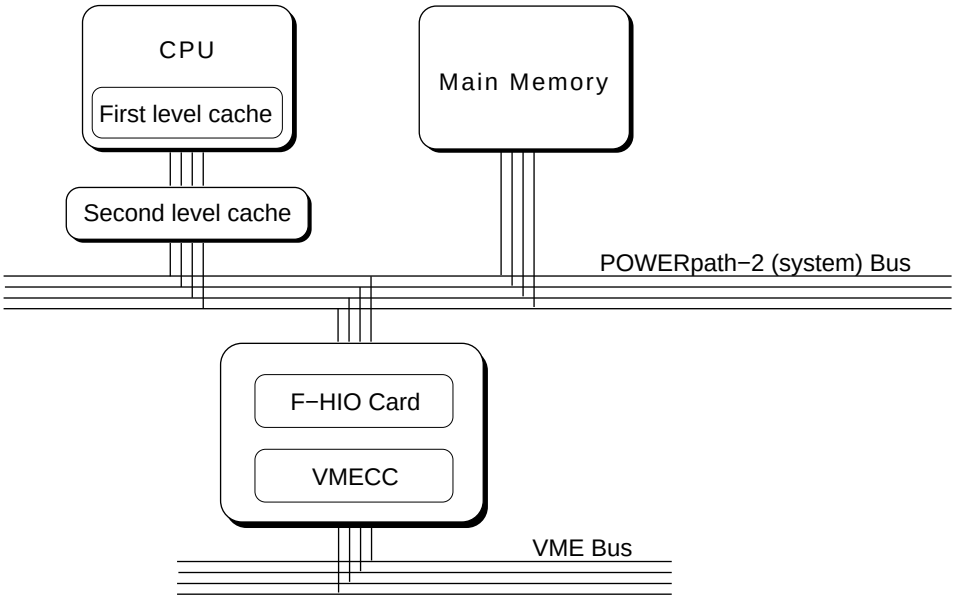

Figure 9-1 Multiprocessor CHALLENGE Data Path Components

VME Bus Attachments

All multiprocessor CHALLENGE systems contain a 9U VME bus in the main card cage. Systems configured for rack-mount can optionally include an auxiliary 9U VME card cage, which can be configured as 1, 2, or 4 VME busses. The possible configurations of VME cards are shown in Table 9-1.

Table 9-1 Multiprocessor CHALLENGE VME Cages and Slots

Model	Main Cage Slots	Aux Cage Slots (1 bus)	Aux Cage Slots (2 busses)	Aux Cage Slots (4 busses)
Challenge L	5	n.a.	n.a.	n.a.
Onyx Deskside	3	n.a.	n.a.	n.a.
Challenge XL	5	20	10 and 9	5, 4, 4, and 4
Onyx Rack	4	20	10 and 9	5, 4, 4, and 4

Each VME bus after the first requires an F cable connection from an F-HIO card on a POWER Channel-2 board, as well as a Remote VCAM board in the auxiliary VME cage. Up to three VME busses (two in the auxiliary cage) can be supported by the first POWER Channel-2 board in a system. A second POWER Channel-2 board must be added to support four or more VME busses. The relationship among VME busses, F-HIO cards, and POWER Channel-2 boards is detailed in Table 9-2.

Table 9-2 POWER Channel-2 and VME bus Configurations

Number of VME Busses	PC-2 #1 FHIO slot #1	PC-2 #1 FHIO slot #2	PC-2 #2 FHIO slot #1	PPC-2 #2 FHIO slot #2
1	unused	unused	n.a.	n.a.
2	F-HIO short	unused	n.a.	n.a.
3 (1 PC-2)	F-HIO short	F-HIO short	n.a.	n.a.
3 (2 PC-2)	unused	unused	F-HIO	unused
4	unused	unused	F-HIO	F-HIO
5	unused	unused	F-HIO	F-HIO

F-HIO short cards, which are used only on the first POWER Channel-2 board, supply only one cable output. Regular F-HIO cards, used on the second POWER Channel-2 board, supply two. This explains why, although two POWER Channel-2 boards are needed with four or more VME busses, the F-HIO slots on the first POWER Channel-2 board remain unused.

VME Address Space Mapping

A device on the VME bus has access to an address space in which it can read or write. Depending on the device, it uses 16, 32, or 64 bits to define a bus address. The resulting numbers are called the A16, A32, and A64 address spaces.

There is no direct relationship between an address in the VME address space and the set of real addresses in the Challenge/Onyx main memory. An address in the VME address space must be translated twice:

- The VMECC and POWER Channel devices establish a translation from VME addresses into addresses in real memory.
- The IRIX kernel assigns real memory space for this use, and establishes the translation from real memory to virtual memory in the address space of a process or the address space of the kernel.

Address space mapping is done differently for programmed I/O, in which slave VME devices respond to memory accesses by the program, and for DMA, in which master VME devices read and write directly to main memory.

PIO Address Space Mapping

In order to allow programmed I/O, the **mmap()** system function establishes a correspondence between a segment of a process's address space and a segment of the VME address space. The **mmap()** function, or the equivalent kernel functions that are available to device drivers, program registers in the VMECC to recognize fetches and stores to main memory real addresses and to translate them into reads and writes on the VME bus. The devices on the VME bus must react to these reads and writes as slaves; DMA is not supported by this mechanism.

One VMECC can map as many as 16 different segments of memory. Each segment can be as long as 8 MB. The segments can be used singly or in any combination. Thus one VMECC can support 16 unique mappings of at most 8 MB, or a single mapping of 128 MB, or combinations between.

DMA Mapping

DMA mapping is based on the use of page tables stored in system main memory. This allows DMA devices to access the virtual addresses in the address spaces of user processes. The real pages of a DMA buffer can be scattered in main memory, but this is not visible to the DMA device. DMA transfers that span multiple, scattered pages can be performed in a single operation.

The kernel functions that establish the DMA address mapping are available only to device drivers. For information on these, refer to the *IRIX Device Driver Programmer's Guide*.

Each DMA stream from the VME bus is assigned a virtual base address in VME address space which corresponds to a pointer to the beginning of a page table in main memory. Bits 20 through 12 of the VME virtual address are used as the virtual page number (VPN), an offset into the page table (I/O operations always use 4 KB pages, regardless of the page size of process address spaces). The page table entry is used as the page address in main memory.

The hardware of the POWER Channel-2 supports up to 8 DMA streams simultaneously active on a single VME bus without incurring a loss of performance.

Program Access to the VME Bus

Your program accesses the devices on the VME bus in one of two ways, through programmed I/O (PIO) or through DMA. Normally, VME cards with Bus Master capabilities always use DMA, while VME cards with slave capabilities are accessed using PIO.

The Challenge/Onyx architecture also contains a unique hardware feature, the DMA Engine, which can be used to move data directly between memory and a slave VME device.

PIO Access

You perform PIO to VME devices by mapping the devices into memory using the **mmap()** function (see “Mapping a VME Device” on page 62).

PIO is suitable for applications that transfer small amounts of data. It is especially suitable for polling a large number of low-bandwidth devices. The bandwidth available for PIO is limited because PIO access cannot be pipelined. The entire path from the CPU through the POWER Channel-2 and VMECC to the VME bus remains occupied from the time a PIO input starts until it completes. (In contrast, DMA transfers are expedited by data prefetching, which is done in the hardware.)

Each PIO read requires two transfers over the POWERpath-2 bus: one to send the address to be read, and one to retrieve the data. The latency of a single PIO input is approximately 4 microseconds. PIO write is somewhat faster, since the address and data are sent in one operation. Typical PIO performance is summarized in Table 9-3.

Table 9-3 VME Bus PIO Bandwidth

Data Unit Size	Read	Write
D8	0.2 MB/second	0.75 MB/second
D16	0.5 MB/second	1.5 MB/second
D32	1 MB/second	3 MB/second

When a system has multiple VME busses, you can program concurrent PIO operations from different CPUs to different busses, effectively multiplying the bandwidth by the number of busses. It does not improve performance to program concurrent PIO to a single VME bus.

Tip: When transferring more than 32 bytes of data, you can obtain higher rates using the DMA Engine. See “DMA Engine Access to Slave Devices” on page 191.

DMA Access to Master Devices

VME bus cards with Bus Master capabilities transfer data using DMA. These transfers are controlled and executed by the circuitry on the VME card. The DMA transfers are directed by the address mapping described under “DMA Mapping” on page 189.

DMA transfers from a Bus Master are always initiated by a kernel-level device driver. In order to exchange data with a VME Bus Master, you open the device and use **read()** and **write()** calls. The device driver sets up the address mapping and initiates the DMA transfers. The calling process is typically blocked until the transfer is complete and the device driver returns.

The typical performance of a single DMA transfer is summarized in Table 9-4.

Table 9-4 VME Bus Bandwidth, VME Master Controlling DMA

Data Transfer Size	Reading	Writing
D8	0.4 MB / sec	0.6 MB / sec
D16	0.8 MB / sec	1.3 MB / sec
D32	1.6 MB / sec	2.6 MB / sec
D32 BLOCK	22 MB / sec (256 byte block)	24 MB / sec (256 byte block)
D64 BLOCK	55 MB / sec (2048 byte block)	58 MB / sec (2048 byte block)

Up to 8 DMA streams can run concurrently on each VME bus. However, the aggregate data rate for any one VME bus will not exceed the values in Table 9-4.

DMA Engine Access to Slave Devices

A DMA engine is included as part of each POWER Channel-2. The DMA engine is unique to the Challenge / Onyx architecture. It performs efficient, block-mode, DMA transfers between system memory and VME bus slave cards—cards that would normally be capable of only PIO transfers.

The DMA engine greatly increases the rate of data transfer compared to PIO, provided that you transfer at least 32 contiguous bytes at a time. The DMA engine can perform D8, D16, D32, D32 Block, and D64 Block data transfers in the A16, A24, and A32 bus address spaces.

All DMA engine transfers are initiated by a special device driver. However, you do not access this driver through open / read / write system functions. Instead, you program it through a library of functions. The functions are documented in the udmalib(3x) reference page. They are used in the following sequence:

1. Call **dma_open()** to initialize action to a particular VME card.
2. Call **dma_allocbuf()** to allocate storage to use for DMA buffers.
3. Call **dma_mkparms()** to create a descriptor for an operation, including the buffer, the length, and the direction of transfer.
4. Call **dma_start()** to execute a transfer. This function does not return until the transfer is complete.

The performance of the DMA engine for D32 transfers is summarized in Table 9-5. Performance with D64 Block transfers is somewhat less than twice the rate shown in Table 9-5. Transfers for larger sizes are faster because the setup time is amortized over a greater number of bytes.

Table 9-5 VME Bus Bandwidth, DMA Engine, D32 Transfer

Transfer Size	Read	Write	Block Read	Block Write
32	2.8 MB / sec	2.6 MB / sec	2.7 MB / sec	2.7 MB / sec
64	3.8 MB / sec	3.8 MB / sec	4.0 MB / sec	3.9 MB / sec
128	5.0 MB / sec	5.3 MB / sec	5.6 MB / sec	5.8 MB / sec
256	6.0 MB / sec	6.7 MB / sec	6.4 MB / sec	7.3 MB / sec
512	6.4 MB / sec	7.7 MB / sec	7.0 MB / sec	8.0 MB / sec
1024	6.8 MB / sec	8.0 MB / sec	7.5 MB / sec	8.8 MB / sec
2048	7.0 MB / sec	8.4 MB / sec	7.8 MB / sec	9.2 MB / sec
4096	7.1 MB / sec	8.7 MB / sec	7.9 MB / sec	9.4 MB / sec

The **dma_start()** function operates in user space; it is not a kernel-level device driver. This has two important effects. First, overhead is reduced, since there are no mode switches between user and kernel, as there are for **read()** and **write()**. This is important since the DMA engine is often used for frequent, small inputs and outputs.

Second, **dma_start()** does not block the calling process, in the sense of suspending it and possibly allowing another process to use the CPU. It waits in a test loop until the operation is complete. As you can infer from Table 9-5, typical transfer times range from 50 to 250 microseconds. You can calculate the approximate duration of a call to **dma_start()** based on the amount of data and the operational mode.

You can use the udmalib functions to access a VME Bus Master device, if the device can respond in slave mode. However, this would normally be less efficient than using the Master device's own DMA circuitry.

While you can initiate only one DMA engine transfer per bus, it is possible to program a DMA engine transfer from each bus in the system, concurrently.

Serial Ports

Occasionally a real-time program has to use an input device that interfaces through a serial port. This is not a recommended practice for several reasons: the serial device drivers and the STREAMS modules that process serial input are not optimized for deterministic, real-time performance; and at high data rates, serial devices generate many interrupts.

When there is no alternative, a real-time program will typically open one of the files named */dev/tty**. The names, and some hardware details, for these devices are documented in the serial(7) reference page. Information specific to two serial adapter boards is in the duart(7) reference page and the cdsio(7) reference page.

When a process opens a serial device, a line discipline STREAMS module is pushed on the stream by default. If the real-time device is not a terminal and doesn't support the usual line controls, this module can be removed. Use the I_POP ioctl (see the streamio(7) reference page) until no modules are left on the stream. This minimizes the overhead of serial input, at the cost of receiving completely raw, unprocessed input.

An important feature of current device drivers for serial ports is that they try to minimize the overhead of handling the many interrupts that result from high character data rates. The serial I/O boards interrupt at least every 4 bytes received, and in some cases on every character (at least 480 interrupts a second, and possibly 1920, at 19,200 bps). Rather than sending each input byte up the stream as it arrives, the drivers buffer a few characters and send multiple characters up the stream.

When the line discipline module is present on the stream, this behavior is controlled by the *termio* settings, as described in the termio(7) reference page for non-canonical input. However, a real-time program will probably not use the line-discipline module. The hardware device drivers support the SIOC_ITIMER ioctl that is mentioned in the serial(7) reference page, for the same purpose.

The SIOC_ITIMER function specifies the number of clock ticks (see "Tick Interrupts" on page 86) over which it should accumulate input characters before sending a batch of characters up the input stream. A value of 0 requests that each character be sent as it arrives (do this only for devices with very low data rates, or when it is absolutely necessary to know the arrival time of each input byte). A value of 5 tells the driver to collect input for 5 ticks (50 milliseconds, or as many as 24 bytes at 19,200 bps) before passing the data along.

External Interrupts

The Challenge / Onyx hardware includes support for generating and receiving external interrupt signals. Four jacks for outgoing signals are available on the master IO4 board. Your program can change the level of these lines individually. Two jacks for incoming interrupt signals are also provided. The input lines are logically OR'd together and presented as a single interrupt; your program cannot distinguish one input line from another.

The electrical interface to the external interrupt lines is documented at the end of the ei(7) reference page.

Your program controls and receives external interrupts by interacting with the external interrupt device driver. This driver is associated with the special device file */dev/ei*, and is documented in the ei(7) reference page. (External interrupt support and the ei(7) page are first available in IRIX 5.3.)

Generating External Signals

Your program can generate an outgoing signal on any one of the four external interrupt lines. To do so, it first opens */dev/ei*. Then it can use **ioctl()** on the file descriptor to switch the outgoing lines. The principal functions are summarized in Table 9-6.

Table 9-6 Generating Outgoing External Signals

Operation	Typical ioctl() Call
Set pulse width to *N* microseconds	ioctl(*eifd*, EIIOCSETOPW, *N*);
Send a pulse on line *L* (*L* is 0, 1, 2, or 3)	ioctl(*eifd*, EIIOCSTROBE, 1<<*L*)
Set line *L* high (active, asserted)	ioctl(*eifd*, EIIOCSETHI, 1<<*L*)
Set line *L* low (inactive, deasserted)	ioctl(*eifd*, EIIOCSETLO, 1<<*L*)

The external interrupt lines are controlled directly from a CPU; they are not managed by a chip or I/O controller. In order to ensure precise, repeatable pulse widths, the device driver generates a pulse by asserting the line, then spinning in a disabled loop until the specified pulse time has elapsed, and deasserting the line. Clearly, if the pulse width is set to much more than the default of 5 microseconds, pulse generation could interfere with the handling of other interrupts (see “Kernel Critical Sections” on page 110).

The calls to assert and deassert the outgoing lines do not tie up the kernel. However, it is usually not practical for even a real-time program to hope to generate repeatable pulse durations by this method—at least, not with duty cycles measured in microseconds. For one thing, the minimum guaranteed interrupt service time is 200 microseconds (see “Minimizing Interrupt Response Time” on page 107). If an interrupt occurs between the call to assert the signal and the call to deassert it, the pulse width can be stretched by 200 microseconds or more. Direct assertion of the outgoing signal should be used only when the desired signal frequency and pulse duration are measured in milliseconds.

Receiving Incoming External Interrupts

An important feature of the external interrupt input line is that interrupts are triggered by the level of the signal, not by the transition from deasserted to asserted. This means that, whenever external interrupts are enabled and any of the input lines are in the asserted state, an external interrupt occurs. The interface between your program and the external interrupt device driver is affected by this hardware design.

Detecting An External Interrupt

The external interrupt handler maintains two important numbers:

- the expected input pulse duration in microseconds
- the minimum pulse-to-pulse interval, called the “stuck” pulse width because it is used to detect when an input line is “stuck” in the asserted state

When the external interrupt device driver is entered to handle an interrupt, it spins in a disabled loop until the expected input pulse duration has expired. At the end of this time the interrupt handler counts one external interrupt and returns to the kernel, which enables interrupts and returns to the interrupted process.

Normally the input line will be deasserted during the period the interrupt handler was spinning. However, if the input line is still asserted when the time expires, another external interrupt will occur immediately. The external interrupt handler notes that it has been reentered within the “stuck” pulse time since the last interrupt. It assumes that this is still the same input pulse as before. In order to prevent the stuck pulse from saturating the CPU with interrupts, the interrupt handler disables the external interrupt signal.

External interrupts remain disabled for one timer tick (“Tick Interrupts” on page 86). Then the external interrupt device driver reenables external interrupts. If an interrupt occurs immediately, the input line is still asserted. The handler disables external interrupts for another, longer delay. It continues to delay and to test the input signal in this manner until it finds the signal deasserted.

Setting the Expected Pulse Widths

You can set the expected input pulse width and the minimum pulse-to-pulse time using **ioctl()**. For example, you could set the expected pulse width using a function like Example 9-1.

Example 9-1 Function to Test and Set External Interrupt Pulse Width

```
int setEIPulseWidth(int eifd, int newWidth)
{
   int oldWidth;
   if ( (0==ioctl(eifd, EIIOCGETIPW, &oldWidth))
   &&   (0==ioctl(eifd, EIIOCSETIPW, newWidth)) )
      return oldWidth;
   perror("setEIPulseWidth");
   return 0;
}
```

The function retrieves the original pulse width and returns it. If either **ioctl()** call fails, it returns 0.

The default pulse width is 5 microseconds.Pulse widths shorter than 2.5 microseconds are not recommended.

Since the interrupt handler keeps interrupts disabled for this duration, you want to specify as short an expected width as possible. However, it is very important that all legitimate input pulses terminate within the expected time. When a pulse persists past the expected time, the interrupt handler is likely to detect a “stuck” pulse, and disable external interrupts for several milliseconds. Set the expected pulse width to a few microseconds longer than the longest valid pulse.

You can set the minimum pulse-to-pulse width using code like that in Example 9-1, using constants EIIOCGETSPW and EIIOCSETSPW.

The device driver notes the time of any legitimate signal that it detects. When another interrupt occurs within the “stuck” time following, the interrupt handler assumes it represents the same pulse as before. It discards the interrupt and disables interrupts for a time.

The default stuck-pulse time is 500 microseconds. You want to set this time to the nominal pulse-to-pulse interval, minus the largest amount of “jitter” that you anticipate in the signal. In the event that external signals are not produced by a regular oscillator, set this value to the expected pulse width plus the duration of the shortest expected “off” time, with a minimum of twice the expected pulse width.

For example, suppose you expect the input signal to be a 10 microsecond pulse at 1000 Hz, both numbers plus or minus 10%. Set the expected pulse width to 11 or 12 microseconds to ensure that all pulses are seen to complete. Set the stuck pulse width to 900 microseconds, so as to permit a legitimate pulse to arrive 10% early.

Receiving Interrupts

The external interrupt device driver offers you three different methods of receiving notification of an interrupt. You can

- have a signal of your choice delivered to your process
- execute an **ioctl()** call that polls for an interrupt, or that blocks until one is received
- call a library function that polls for an interrupt, or that spin-loops until one is received

You would use a signal when interrupts are infrequent and irregular, and when it is not important to know the precise arrival time. Signal latency can be milliseconds long in some cases (see “Signal Delivery and Latency” on page 141). For this reason, it would not be wise to use signals to handle a high rate of interrupts, nor to expect to time interrupts closely. Use a signal when, for example, the external interrupt represents a human-operated switch or some kind of out-of-range alarm condition.

The **ioctl(**EIIOCRECV**)** call tests for an interrupt, or suspends the caller until an interrupt arrives or a timeout expires (see the ei(7) reference page for details). Use this method when interrupts arrive frequently enough that it is worthwhile devoting a process to handling them.

The **ioctl()** call is a fairly costly method of polling, since it entails entry to and exit from the kernel. This is not significant if the polling is infrequent—for example, if one poll call is made every 60th of a second.

When the **ioctl()** call is used to wait for an interrupt, an unknown amount of time can pass between the moment when the interrupt handler unblocks the process, and the moment when the kernel dispatches the process. This makes it impossible to timestamp the interrupt at the microsecond level.

For minimum latency, a library function **eicbusywait()** is provided (see the ei(7) reference page). This function does not switch into kernel mode, so it is a very fast method of polling for an interrupt. However, if you ask it to wait until an interrupt occurs, it waits by spinning on a repeated test for an interrupt. This monopolizes the CPU, so this form of waiting can only be used reliably by a process running in an isolated, nonpreemptive CPU. (If there are other processes to run, or interrupts to handle, the polling loop in **eicbusywait()** shares the CPU and can be preempted for long periods.)

The benefit of **eicbusywait()** is that, in an isolated, nonpreemptive CPU, control returns to the calling process in negligible time after the interrupt handler has exited, so the interrupt can be handled quickly and timed precisely.

Appendix A

Sample Programs

The programs in this appendix illustrate the use of some of the features discussed in the book. The following programs are included:

- "Mapping and Reading the Cycle Counter" on page 202 illustrates the use of the cycle-counter.
- "Getting the Time of Day Stamp" on page 211 illustrates the use of **gettimeofday()** and shows how to test its precision.
- "Interprocess Communication" on page 212 illustrates some uses of arenas, semaphores, and interval timers.
- "Probing the Address Space" on page 222 displays the addresses assigned to a process address space, and illustrates some uses of **mmap()**.
- "Deadline Scheduling Subroutines" on page 224 illustrates the use of schedctl(2) to set a deadline scheduling policy.
- "Frame Scheduler Examples" on page 226 describes the sample programs distributed with the REACT/Pro Frame Scheduler.
- "Asynchronous I/O Example" on page 239 illustrates the use of asynchronous I/O including four different methods of testing for I/O completion, and also shows process creation with sproc() and the use of semaphores and barriers.
- "Guaranteed-Rate Request" on page 258 demonstrates how to request a guaranteed rate of I/O transfer.

Mapping and Reading the Cycle Counter

This program shows how to map the high-precision cycle counter into memory and sample it. By default, the file compiles to a library of the following functions:

getIPNumber() Returns the model number of the CPU board in this system. For example, an Indy contains an IP22 board, so in an Indy, this function returns the number 22.

mapTheTimer() Uses **mmap()** to map the cycle counter into the address space. Returns the unit-value of the timer in picoseconds; for example returns 21000 in a Challenge where the timer unit value is 21 nanoseconds.

timerBitCount() Returns the number of bits of precision in the timer, which varies with the CPU board type, either 24, 32 or 64 bits.

readTimer32() Returns the least-significant (or only) word of the timer value.

readTimer64() Returns the timer value as a 64-bit unsigned integer (extended with 0-bits when necessary).

When you define the compiler variable UNIT_TEST, it also compiles a **main()** that exercises the functions.

```
/*****************************************************************************
||
|| The functions in this module provide access to the free-running timer
|| on the CPU board of certain SGI systems.
||
|| getIPNumber()
||
|| This function returns the IPnn number -- the CPU board model number --
|| of the current system.  For example in an Indy it returns 22 (IP22),
|| while in a Challenge-XL it returns 19 (IP19).  Used in mapTheTimer to
|| determine number of bits in the available timer.
||
|| mapTheTimer()
||
|| This function tests the hardware environment. If the current system has
|| a timer, the function tries to map it into memory.  Errors can include:
||   * this hardware does not have a mappable timer
||   * error returned by syssgi(2)
||   * error returned by mmap(2)
|| When there is no error, the function returns a positive integer which is
|| the number of picoseconds represented by one unit increment of the timer.
|| In the event of an error, the function returns 0, and errno is set to
|| some error code.
||   mapTheTimer() can be called multiple times without harm.  To convert
|| its returned value to a fraction of a second, convert to double and
|| multiply by 1e-12.
||
|| readTimer32()
||
|| This function calls mapTheTimer(), if it has not been called already.
|| Thus the first attempt to read the clock will map it if necessary.
|| If the timer has been mapped, its least-significant bits are returned
|| as an unsigned 32-bit integer.
||   * if mapTheTimer() failed, the returned value is always 0
||   * if the timer has 24- or 32-bit precision, the returned value is
||     the whole timer value
||   * if the timer has 64-bit precision (e.g. Challenge), the returned
||     value is the low-order word.
||
|| readTimer64()
||
|| This function is like readTimerUL(), except that it returns an unsigned
|| 64-bit integer.
||   * if mapTheTimer() failed, the returned value is always 0
||   * if the timer has 24- or 32-bit precision, the returned value is
```

```
||     the whole timer value, extended with high-order 0-bits
||   * if the timer has 64-bit precision, the returned value is the whole
||     timer value.  The 64-bit timer is sampled in such as way as to
||     compensate for rollover while minimizing bus traffic.
||
|| timerBitCount()
||
|| Returns the number of bits of data in the timer value:
||    0 if mapTheTimer() has failed
||   24 in an Indigo and some other R3000 systems
||   32 in an Indy and some other R4000 uniprocessors
||   64 in a Challenge, Onyx, and other big machines.
||
||
|| main()
||
|| Compiled only when UNIT_TEST is defined, provides a functional test
|| platform for the above functions.
||
|| NOTE: in two of these routines we assume that this machine is operating
|| in big-endian mode, such that the least-significant 32 bits of a
|| long-long are at the higher word address.
||
|| This code, compiled with options -mips1 -g, has been tested
|| on machines with IP6, IP7, IP12, IP19, IP21, and IP22 cpu boards.
*************************************************************************/
#include <stddef.h>         /* for NULL */
#include <fcntl.h>          /* for O_RDONLY and open() */
#include <unistd.h>         /* for getpagesize() */
#include <sys/mman.h>       /* for constants used with mmap() */
#include <sgidefs.h>        /* for __psint_t, __uint32_t, and __int64_t */
#include <sys/syssgi.h>     /* for syssgi(), SGI_QUERY_CYCLECNTR */
#include <errno.h>          /* for errno global */
#include <invent.h>         /* getinvent() and related constants */

unsigned short getIPNumber()
{
inventory_t *pitem;
unsigned short ipno = 0; /* default meaning ?? */

    setinvent();
    while( (pitem = getinvent()) )
    {
        if ((pitem->inv_class == INV_PROCESSOR)
        &&  (pitem->inv_type  == INV_CPUBOARD) )
```

```
        {
            switch (pitem->inv_state)
            {
            case INV_IP4BOARD: ipno = 4; break;
            case INV_IP5BOARD: ipno = 5; break;
            case INV_IP6BOARD: ipno = 6; break;
            case INV_IP7BOARD: ipno = 7; break;
            case INV_IP9BOARD: ipno = 9; break;
            case INV_IP12BOARD: ipno = 12; break;
            case INV_IP17BOARD: ipno = 17; break;
            case INV_IP20BOARD: ipno = 20; break;
            case INV_IP19BOARD: ipno = 19; break;
            case INV_IP22BOARD: ipno = 22; break;
            case INV_IP21BOARD: ipno = 21; break;
            case INV_IP26BOARD: ipno = 26; break;
            default: break;
            }
            break;
        }
    }
    endinvent();
    return ipno;
}

/*****************************************************************************
|| The following globals are set up by mapTheTimer() the first time called.
||   timerMapAddress == NULL means mapTheTimer() has never been called
||                   == -1 means mapTheTimer() called and failed
||                   else it points to the timer in memory
||
|| The "volatile" declaration keeps the compiler from optimizing away
|| successive references to it.
||
||   timerPicoSecs   == 0 means the timer has not been mapped successfully
||                   else is the value returned by syssgi(QUERY_CYCLECOUNTER)
||
||   timerBits       == 0 means the timer has not been mapped successfully
||                   else is 24, 32 or 64 depending on the IP number.
||
|| If this code was redone in C++ (not a bad idea, feel free) these would
|| be class variables.
*****************************************************************************/
```

```
static volatile void * timerMapAddress = NULL;
static unsigned int timerPicoSecs = 0;
static unsigned short timerBits = 0;

unsigned int
timerBitCount()
{
    return timerBits;
}

unsigned int
mapTheTimer()
{
__uint32_t timerUnits = 0;  /* receives timer picosecond unit value */
__psint_t  timerPhysAddr;   /* receives timer absolute address */
__psint_t  timerPhysVPN;    /* page boundary of timerPhysAddr */
__psint_t   addrMask;       /* page offset bit mask */
int fdMem;                  /* file descriptor for /dev/mmem */

    if ( ! timerMapAddress)     /* first time through this code */
    {
        /*
        || Get the physical address of the clock in full. If there
        || is no cycle counter on this machine, syssgi returns -1.
        */
        timerPhysAddr = syssgi(SGI_QUERY_CYCLECNTR, &timerUnits);
        if ((__psint_t)-1 != timerPhysAddr) /* we have a timer */
        {
            /*
            || Trim out the offset from the address leaving the
            || page number part of the address. (VPN == virtual page number)
            */
            addrMask = getpagesize() - 1;
            timerPhysVPN = timerPhysAddr & ~addrMask;
            /*
            || Map the page containing the clock's address into the virtual
            || address space of this process.
            */
            fdMem = open("/dev/mmem", O_RDONLY);
            timerMapAddress = (void *) mmap(
                NULL,               /* addr = don't care */
                addrMask,           /* len = pagesize - 1 */
                PROT_READ,          /* prot = read-only */
                MAP_PRIVATE,        /* changes are unshared (n.a.) */
                fdMem,              /* map base is physical memory */
```

```
                (off_t)timerPhysVPN /* source address to map */
                );
            if ((__psint_t)-1 != (__psint_t)timerMapAddress)
            {/* mmap() succeeded. */
                /*
                || Restore possibly-nonzero offset bits to mapped address.
                */
                timerMapAddress = (void*) (
                    ((__psint_t)timerMapAddress) /* addr as int */
                    | (timerPhysAddr & addrMask)  /* plus offset bits */
                    );
                /*
                || Set globals for timer units and precision
                */
                timerPicoSecs = timerUnits;
                switch (getIPNumber())
                {
                case 12: /* IP12 (Indigo R3K) */
                    timerBits = 24; break;
                case 19: /* IP19 (Challenge, Onyx) */
                case 21:
                    timerBits = 64; break;
                default: /* everything else currently is... */
                    timerBits = 32; /* ...32 bits, Dave Olson says */
                }
            }
            else
                ; /* mmap() failed, timerMapAddress == -1, errno set */
        } /* end syssgi() successful */
        else
        {
            timerMapAddress = (void *)-1; /* no timer to map */
            errno = ENODEV;
        }
    } /* end attempting to initialize */
    return timerPicoSecs;
}
/*****************************************************************************
||
|| In both of the following routines, one goal is to minimize the number of
|| references to the mapped timer.  Reason: each such reference is an
|| uncached memory reference plus a bus access, taking at least 0.5 usec and
|| possibly more depending on the machine.  Unnecessary references to the
|| timer should be avoided when possible.
||
```

```
|| If the timer has 64 bits, return its least-significant word. Which word
|| is that?  This code assumes the big-endian model.  An alternative
|| would be to load the long-long value and force C to convert it.  That is
|| be portable but would hit the bus twice instead of once, nullifying the
|| speed advantage that this routine has over the one following.
||
|| Only the IP12 (Indigo 3K) has a 24-bit timer.
||
******************************************************************************/
__uint32_t
readTimer32()
{
__uint32_t ret = 0;

    if ( ! timerMapAddress ) mapTheTimer();
    if ( -1 != (__psint_t)timerMapAddress ) /* timer mapped ok */
    {
        if (64 == timerBits)
            ret = ((__uint32_t *)timerMapAddress)[1]; /* low word */
        else
        {
            ret = *((__uint32_t *)timerMapAddress);
        }
    }
    return ret;
}

/******************************************************************************
||
|| When the timer has 32 or 24 bits, just fake up a long-long and return it.
|| For long timers we must ask: was this code compiled under a 64-bit IRIX?
|| If so, the timer is accessed in a single load, which is an atomic access.
|| No special code is needed in this case.
||
|| When compiled under a 32-bit system, the generated code loads the timer
|| value in two "lw" instructions.  The low word of the timer overflows into
|| the high word about every 90 seconds, and if that happens between the
|| lw's, the result will be wrong.  Worse, we cannot be certain which of the
|| two words will be loaded first, the low or the high.
||
|| In order to minimize the number of bus accesses, we only test for overflow
|| when it is about to happen or just happened; hence this code does, on the
|| average, 2+1/256 accesses to the timer per call.
||
******************************************************************************/
```

```
__uint64_t
readTimer64()
{
union {
    struct { __uint32_t msw,lsw; }w;
    __uint64_t ll;
    } ret;

    ret.ll = 0;
    if ( ! timerMapAddress ) mapTheTimer();
    if ( -1 != (__psint_t)timerMapAddress ) /* clock mapped ok */
    {
        if (timerBits < 64)
        {
            ret.w.lsw = *((__uint32_t *)timerMapAddress);
        }
        else
        {
#if (64 == _MIPS_SZPTR) /* 64-bit program, 64-bit timer */
            ret.ll = *(__uint64_t *)timerMapAddress;
#else /* 32-bit program, 64-bit timer */
            ret.w.msw = ((__uint32_t *)timerMapAddress)[0];
            ret.w.lsw = ((__uint32_t *)timerMapAddress)[1];
            if ( ! ret.w.lsw & 0xff000000 )
            { /* recent overflow, resample the high word */
                __uint32_t sample2 = *((__uint32_t *)timerMapAddress);
                if (ret.w.msw != sample2)
                { /* overflow between loads */
                    ret.w.msw = sample2;
                    ret.w.lsw = 0;
                }
            }
#endif
        }
    }
    return ret.ll;
}

#ifdef UNIT_TEST
#include <stdio.h>

int main(int argc, char**argv)

{
    int     j;
```

```
int     numTix = 10;
unsigned int picosecs;
unsigned short tbits;
double  dmicsecs;

if (argc>1) numTix = atoi(argv[1]);
printf("This machine uses an IP%2d cpu board\n",getIPNumber());
if ( picosecs = mapTheTimer() )
{
    tbits = timerBitCount();
    dmicsecs = ((double)picosecs)/1e6;
    printf("The timer has %d bits of precision\n",tbits);
    printf("One timer unit == %d picoseconds or %g us\n",
                                picosecs, dmicsecs);
}
else
{
    perror("mapTheTimer");
    return errno;
}

if (64 > tbits)
{
    __uint32_t st1, st2, stx;
    st1 = readTimer32();
    for(j=0; j<numTix; ++j)
    {
        st2 = readTimer32();
        stx = st2 - st1;
        printf("0x%0lx - 0x%0lx = 0x%0lx (%g usecs)\n",
                st2,       st1,     stx, (stx*dmicsecs) );
        st1 = st2;
    }
}
else
{
    __uint64_t lt1, lt2, ltx;
    lt1 = readTimer64();
    for(j=0; j<numTix; ++j)
```

```
        {
            lt2 = readTimer64();
            ltx = lt2 - lt1;
            printf("0x%0llx - 0x%0llx = 0x%0llx (%g usecs)\n",
                    lt2,        lt1,      ltx, (ltx * dmicsecs));
            lt1 = lt2;
        }
    }
}
#endif
```

Getting the Time of Day Stamp

The following program test the precision of the time of day stamp returned by **gettimeofday()**. The function **getTODdiff()** contains an example call to **gettimeofday()**.

```
#include <sys/time.h>
#include <stdio.h>
#define LOOPS 1000

/*
 * Ths function loops on gettimeofday() until its returned time
 * changes by more than 1 microsec. This tells us the real
 * resolution of the time of day stamp -- it is basically the
 * dispatching timer frequency of the kernel.
 *
 * We look for a change of >1 usec because apparently gettimeofday()
 * never returns the same value twice. The values returned by two
 * calls in quick succession differ by 1 usec, on most calls.
 * Eventually, after a number of calls that varies with the platform,
 * the return differs by a larger amount, indicating that the
 * timestamp has been updated by the kernel timer routine.  This
 * is the event we wait for.
 *
 * The function also updates a maximum loop-count value.
 */

long getTODdiff(int * pMaxLoops)
{
    long first, second;
    int nloops = 0;
    struct timeval tod;
```

```
    struct timezone tz;
    gettimeofday(&tod, &tz);
    first = tod.tv_usec;
    do
    {
        gettimeofday(&tod, &tz);
        second = tod.tv_usec;
        ++nloops;
    } while (first == (second-nloops));
    if (first > second) second += 1000000;
    if (pMaxLoops)
        if (nloops > *pMaxLoops) *pMaxLoops = nloops;
    return second - first;
}
int main()
{
    int j;
    int maxLoops;
    long sum;
    double mean;
    /* get past the first call, which is likely to be short */
    sum = getTODdiff(NULL);
    for (j=0, sum=0 ; j< LOOPS; j++)
    {
        sum += getTODdiff(&maxLoops);
    }
    mean = sum/LOOPS;
    printf("Mean step size in gettimeofday() = %g\n", mean);
    printf("Max number of loops %d\n", maxLoops);
}
```

Interprocess Communication

The following sample program illustrates the use of some of the interprocess communication features of IRIX.

The program models a real-time data-collection program. The main process establishes an arena. Within the arena it creates a data structure that defines and manages a ring buffer. Then the main process uses **sproc()** to create three processes:

- **inputProcess()** generates random-integer “input data” and stores it in the ring buffer. To simulate an unpredictable and varying input rate, the process “receives” bursts of from 1 to 16 data items. The average input rate is calculable (see the commentary in the code).

  After generating a certain number of items, **inputProcess()** terminates. The number of items can be specified on the command line.

- **outputProcess()**—of which two instances are created—takes data from the ring buffer. To simulate a steady average output rate, each process sets a repeating itimer and takes one data item each time the timer expires. The itimer interval represents the simulated “processing time” of a data item. It can be specified on the command line.

After starting the three processes, the main process waits for one to terminate (if there are no errors, **inputProcess()** finished first). The main process dumps the metering information from the lock and semaphores, and terminates the program.

The three simulated real-time processes communicate through two semaphores and a lock.

- Semaphore *semRBdata* represents the number of data items now in the ring buffer. **inputProcess()** does the V operation, increasing the semaphore count with each input datum; **outputProcess()** does the P operation, decreasing the count with each output.
- Semaphore *semRBspace* represents the number of empty slots in the ring buffer. **inputProcess()** does the P operation and **outputProcess()** does the V operation.
- Lock *lockRBupdate* represents the right to alter the ring buffer index values. All processes set this lock before modifying the ring buffer, and clear it afterward.

The displayed metering data at the end of the program shows whether the output processes could keep up with the input process. The following example shows a case in which they did not keep up:

```
# /usr/sbin/npri -h 30 rbtest -t 3000

Lock information
 : LOCKDUMP at 0x4413a0 (last owner = 0)
 lock free
```

```
 lock meter: tries 7781 spins 0 hits 7781
 lock debug: owner pid -1

semRBdata information
 : SEMADUMP of 0x4411c0 count 159
 semaphore meter: value 159 nwait 0 maxnwait 1
 psemas 3811 phits 3808 vsemas 3970 vnowait 3967

semRBspace information
 : SEMADUMP of 0x4412b0 count 0
 semaphore meter: value 0 nwait 0 maxnwait 2
 psemas 3972 phits 2275 vsemas 3812 vnowait 2115

detaching arena file
```

When **inputProcess()** executed a P operation on *semRBspace*, it "hit" (passed through without waiting) only 2275 of 3972 times. In other words, more than a third of the time, the input process was blocked, waiting for an empty slot in the ring buffer. Similarly, **outputProcess()** performed 3812 V operations on semRBspace to release ring buffer slots. On 2115 of those times there was no process waiting. That means that 1697 times, there was a process (which must have been the input process) waiting on the semaphore. So on this particular machine, the simulated processing time of 3000 microseconds (specified on the command line as -t 3000) was too long to keep up with the input data rate.

```
#include <stdlib.h> /* for getopt() */
#include <signal.h>
#include <sys/time.h>
#include <ulocks.h>
#include <math.h> /* for random() and srandom() */
/*
 * The following declarations define a structure that controls a ring buffer.
 *
 *      rbElem_t    the type of thing that is stored in the buffer
 *      RB_MAXELS   the size of the ring buffer
 *      rbStruct    control and serialization items for the buffer
 *
 *  The buffer and structure are built together in an arena, and the
 *  address of the structure is the arena info (usgetinfo()).
 */
typedef long rbElem_t;  /* can be any scalar, but assumed long below */
#define RB_MAXELS 160   /* specify enough to buffer peak data rate */
typedef struct {
    rbElem_t * theBuffer;       /* -> [RB_MAXELS] of rbElem_t */
```

```
    usema_t * semRBdata;        /* -> semaphore for buffered data */
    usema_t * semRBspace;       /* -> semaphore for open buffer slots */
    ulock_t * lockRBupdate;     /* -> lock on the following words */
    int rbGet;      /* theBuffer[rbGet] is next data */
    int rbPut;      /* theBuffer[rbPut] is next slot */
} rbStruct;

/*
 * The following global variables are input parameters to the
 * child processes. They are set by main() from command line arguments.
 * For discussion of the default constants, see the prologs of the
 * inputProcess and outputProcess functions.
 */
#define MAX_BURST 16 /* data rate average 25*16/2 == 200/sec */
#define FINAL_COUNT ((MAX_BURST/2)*25)*10 /* run for 10 seconds about */
#define OUTPUT_TIME 10000 /* 10 ms delay or 100 Hz */

static int outputTimer = OUTPUT_TIME;   /* -t */
static int inputCount = FINAL_COUNT;    /* -c */

/*
 * Allocate the arena and populate it with ring buffer stuff.
 * If any error occurs, report it and return NULL.  The following filename
 * is used. It must address a writeable directory. If the file already
 * exists, it must be writeable to this process.
 */
#define ARENA_FILE "/usr/tmp/ring.buffer.arena"

usptr_t * allocRBStuff()
{
    usptr_t * arena;
    rbStruct * rbs;
    int okSoFar = 1;
    /*
     * Announce that we want metering info stored with our locks.
     */
    if (-1 == usconfig(CONF_LOCKTYPE, US_DEBUGPLUS) )
    {
        perror("usconfig(CONF_LOCKTYPE)");
        return NULL;
    }
    /*
     * Create the arena.
     */
    if ( NULL == (arena = usinit(ARENA_FILE) ) )
```

```
{
    perror("usinit");
    return NULL;
}
/*
 * From here on, a failure means we should usdetach().
 */
okSoFar = 0 != (rbs = (rbStruct *)uscalloc(1, sizeof(rbStruct), arena) );
if (okSoFar)
{
    okSoFar = 0 !=
        (rbs->theBuffer =
            (rbElem_t *)uscalloc(RB_MAXELS, sizeof(rbElem_t), arena));
    if (!okSoFar)
        fprintf(stderr, "Unable to allocate ring buffer in arena\n");
}
if (okSoFar)
{ /* value of semRBdata is 0 because no data in buffer yet */
    okSoFar = 0 !=
            (rbs->semRBdata = usnewsema(arena, 0) );
    if (!okSoFar)
        perror("usnewsema #1");
}
if (okSoFar)
{ /* value of semRBspace is number of empty slots in ring buffer */
    okSoFar = 0 !=
            (rbs->semRBspace = usnewsema(arena, RB_MAXELS) );
    if (!okSoFar)
        perror("usnewsema #2");
}
if (okSoFar)
{
    okSoFar = 0 !=
            (rbs->lockRBupdate = usnewlock(arena));
    if (!okSoFar)
        perror("usnewlock");
}
/*
* Set the semaphores to collect metering information.
*/
if (okSoFar)
{
    okSoFar =   (0==( usctlsema(rbs->semRBdata, CS_METERON) ) )
            &&  (0==( usctlsema(rbs->semRBspace, CS_METERON) ) );
    if (!okSoFar)
```

```
            perror("usctlsema(METERON)");
    }
    if (okSoFar)
    { /* stow the ring buffer structure as the arena info word */
        usputinfo(arena, (void *)rbs);
    }
    else
    { /* something went wrong, return null */
        usdetach(arena);
        arena = NULL;
    }
    return arena;
}
/*
 * Put an item into the ring buffer.  This dePletes the count of open
 * slots, and reViVes the count of waiting data.  It can block if the
 * ring buffer is full until getElement() has been called.  It can also
 * block briefly on the lock if another process is updating the ring buffer.
 */
void putElement(rbElem_t value, rbStruct * rbs)
{
    uspsema(rbs->semRBspace); /* dePlete the open slots */
    ussetlock(rbs->lockRBupdate); /* get exclusive use of rbPut */
    rbs->theBuffer[rbs->rbPut++] = value;
    if (rbs->rbPut >= RB_MAXELS)
        rbs->rbPut = 0;
    usunsetlock(rbs->lockRBupdate); /* release use of lock */
    usvsema(rbs->semRBdata);    /* reViVe the count of active data */
}
/*
 * Get an item from the ring buffer.  This procedure can block
 * if the ring buffer is empty.
 */
rbElem_t getElement(rbStruct * rbs)
{
    rbElem_t ret;
    uspsema(rbs->semRBdata); /* dePlete the available data */
    ussetlock(rbs->lockRBupdate); /* get exclusive use of rbGet */
    ret = rbs->theBuffer[rbs->rbGet++];
    if (rbs->rbGet >= RB_MAXELS)
        rbs->rbPut = 0;
    usunsetlock(rbs->lockRBupdate); /* release use of lock */
    usvsema(rbs->semRBspace);   /* reViVe the count of open slots */
    return ret;
}
```

```
/*
 * This is the body of the simulated data collection process.
 * The process actually runs at a constant rate of 25 Hz,  using
 * sginap(4) to pace itself (100 ticks per second / 4 ticks = 25Hz).
 * However, to simulate "data" received in bursts, it "receives" from
 * 1 to MAX_BURST items per iteration, an average of MAX_BURST/2,
 * for an average data rate of (25*MAX_BURST/2) items/second.
 *
 * With MAX_BURST at 16, that's 200 items/second.
 * This is the average rate the data writers must achieve, and the ring
 * buffer has to take up the slack during long bursts.
 *
 * At a rough approximation, the probability of a burst of length
 * n*MAX_BURST should be (1/MAX_BURST)^n.  (Which means that there is
 * a nonzero probability of a burst of any length you name, and you
 * cannot make a buffer big enough to completely preclude blockages.)
 *
 * However, with MAX_BURST==16 and RB_MAXEL==160, this buffer should
 * overflow once in ~1e-12 times, provided the data writers keep to the rate.
 *
 * The process executes until it has buffered FINAL_COUNT elements,
 * then terminates.  main() waits for this, and shuts down the program.
 */
void inputProcess(void *arena) /* prototype required by sproc() */
{
    rbStruct * rbs = usgetinfo((usptr_t *)arena);
    int myPid = getpid();
    rbElem_t datum;     /* this code assumes rbElem_t is "long" */
    int counter = inputCount;
    int burst;

    srandom(myPid);     /* seed random() */
    do
    {
        sginap(4);
        datum = (rbElem_t) random();
        burst = 1+(datum % MAX_BURST);
        for ( ; burst; --burst)
        {
            putElement(datum, rbs);
            --counter;
        }
    } while (counter > 0);
    /* exit, ending the process and satisfying wait() in main() */
}
```

```
/*
 * This is the body of the simulated data-output processes, 2 of which
 * are started.  (The purpose of starting 2 is merely to complicate the
 * use of the semaphores in rbStruct -- it isn't realistic.)
 *
 * Each process sets a repeating itimer with an interval of (outputTimer)
 * microseconds.  That global determines the "output data rate" that
 * can be achieved.  However, due to integer truncation effects in the
 * precision timer routines, you should not expect fine-grained
 * adjustments of this value to be effective. (Not to mention the
 * interference of other processes in the system, even when this
 * program runs with a real-time priority level.)
 *
 * The signal handler is empty. The POSIX sigsuspend() call is used
 * to block until the SIGALRM comes. When it comes, the empty handler
 * is called and then control returns from the sigsuspend().
 * Then one data item is fetched from the ring buffer.
 *
 * When the input rate averages 200/sec, each output process needs to
 * get signals at a rate of 100/sec, or an interval of 10000 usec.
 * (In fact, 3000 usec is not fast enough...)
 */

void uponSigalrm()
{
    return;
}

void outputProcess(void *arena) /* prototype for sproc() use */
{
    rbStruct * rbs = usgetinfo((usptr_t *)arena);
    sigset_t alarmSet, emptySet;
    struct sigaction alarmAct = {SA_RESTART, uponSigalrm, 0};
    struct itimerval timer = {{0, 0}, {0, 0}};
    rbElem_t datum;
    /*
     *  Prepare an empty set of signals to use with sigsuspend()
     */
    sigemptyset(&emptySet);
    /*
     * Prepare a mask to block SIGALRM, and apply it.
     */
    alarmSet = emptySet;
    sigaddset(&alarmSet, SIGALRM);
    sigprocmask(SIG_BLOCK, &alarmSet, NULL);
```

```
    /*
     * Set the action for SIGALRM to the empty handler.
     */
    if (sigaction(SIGALRM, &alarmAct, NULL))
    {
        perror("sigaction");
        return;
    }
    /*
     * Set a repeating itimer to deliver SIGALRMs regularly
     */
    timer.it_interval.tv_usec = outputTimer;
    timer.it_value.tv_usec = outputTimer;
    if (setitimer(ITIMER_REAL, &timer, NULL))
    {
        perror("setitimer");
        return;
    }
    /*
     * Loop getting a datum each timer pop
     */
    for (;;)
    {
        datum = getElement(rbs);
        sigsuspend(&emptySet);
    }
}
/*
 * The main() function:
 *      gets the arguments, if any
 *      sets up the arena
 *      starts the 3 processes
 *      waits for the outputProcess to terminate
 *      dumps the lock and semaphore info
 *      detaches the arena and unlinks its file
 */
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait() */

main(int argc, char**argv)
{
    pid_t kids[3];
    usptr_t * arena = allocRBStuff();
    rbStruct *rbs;
    int c;
```

```
/*
 * check that the arena and structures allocated OK
 */
if (arena)
    rbs = usgetinfo(arena);
else
    return -1; /* allocation failed, message issued */
/*
 * get command line arguments for input count and output delay
 */
while ((c = getopt(argc, argv, "c:t:")) != EOF)
{
    switch (c)
    {
    case 'c':
        inputCount = atoi(optarg);
        break;
    case 't':
        outputTimer = atoi(optarg);
        break;
    case '?':
        printf("usage: [-c input data count] [-t output time usec]\n");
        return -2;
    }
}
/*
 * create the outputProcess
 */
kids[0] = sproc(outputProcess, PR_SALL, (void *)arena);
if (-1 == kids[0])
{
    perror("sproc(outputProcess)");
    return -1;
}
/*
 * create the 2 inputProcess es
 */
kids[1] = sproc(inputProcess,  PR_SALL,  (void *)arena);
if (-1 == kids[1])
{
    perror("sproc(inputProcess 1)");
    return -1;
}
kids[2] = sproc(inputProcess,  PR_SALL,  (void *)arena);
if (-1 == kids[2])
```

```
    {
        perror("sproc(inputProcess 2)");
        return -1;
    }
    /*
     * wait until a child process (don't care which) ends.
     */
    wait(0);
    /*
     * dump the metering information from the lock.
     */
    if (usdumplock(rbs->lockRBupdate, stdout, "\nLock information\n\t"))
    {
        perror("usdumplock");
    }
    /*
     * dump the metering information from the two semaphores
     */
    if (usdumpsema(rbs->semRBdata, stdout, "\nsemRBdata information\n\t"))
    {
        perror("usdumpsema");
    }
    if (usdumpsema(rbs->semRBspace, stdout, "\nsemRBspace information\n\t"))
    {
        perror("usdumpsema");
    }
    printf("\ndetaching arena file\n");
    usdetach(arena);
    unlink(ARENA_FILE);
    return 0;
}
```

Probing the Address Space

The following sample program uses some generally unsafe coding tricks to get the addresses of segments for text, stack, library DSO and mapped data. It demonstrates the use of **mmap()** with */dev/zero*, with default and with absolute segment addresses.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
```

```
#include <sys/mman.h>
/*
 * Function to display an address with a caption
 */
void pr(char *addr, char *caption)
{
        printf("%s:\t%08lx\n",caption,addr);
}
/*
 * poke and prod the address space a bit
 */
int main()
{
/*
 * get a file descriptor for the nothing device
 */
int zero = open("/dev/zero",O_RDWR);
/*
 * map two segments at default addresses
 */
char * map1 = (char *)mmap(0,16384,PROT_WRITE,MAP_SHARED,zero,0);
char * map2 = (char *)mmap(0,16384,PROT_WRITE,MAP_SHARED,zero,0);
/*
 * map one segment at a designated address
 */
char * map3 = (char *)mmap((void *)0x30040000L,16384,
                        PROT_WRITE,MAP_SHARED+MAP_FIXED,zero, 0);
/*
 * get the address of this program, rounded down to a page
 */
char * poke = (char *)(((unsigned long)main) & 0xfffff000);
/*
 * get the address of some code in the libc DSO
 */
char * libr = (char *)fprintf;
/*
 * get the address of an item in our stack space
 */
char * stack = (char *)&zero;
/*
 * print all the above
 */
    pr( poke, "program text");
    pr( map1, "default map1");
    pr( map2, "default map2");
```

```
    pr( map3, "absolute map");
    pr( libr, "library code");
    pr( stack, "stack space");
/*
 * see if we can get away with patching our own text
 */
    if (!mprotect((void *)poke,4096,PROT_WRITE))
    {
            poke[0] = poke[0];
            printf("I wrote into program text\n");
    }
    else
    {
            perror("mprotect");
    }
}
```

Deadline Scheduling Subroutines

The following example contains two subroutines that simplify the interface to the **schedctl()** function for deadline scheduling. If the code is compiled with variable UNIT_TEST defined, it compiles a **main()** procedure that runs a test. Otherwise it compiles only the functions.

```
/*
 * Issue the schedctl(2) calls to set up deadline scheduling, using
 * the simpler interface of npri(1). That is, where schedctl() requires
 * you to set up a structure containing two intervals in nanoseconds,
 * setDeadlinePct() lets you specify an interval in milliseconds and a
 * percentage duty cycle.
 *
 * As a bonus, setDeadlinePolicy() is a short way to call for any of
 * the four policies, DL_ONLY, DL_ANY, DL_RELEASE (the rest of the period)
 * and DL_BLOCK (for the rest of the period).
 *
 */
#include <errno.h>
#include <sys/schedctl.h>
#include <stdio.h> /* for stderr, perror */

static void putMSinTimestruc(timestruc_t *ts, const int milliseconds)
{
        int ms = milliseconds;
```

```
        if (1000 > ms)
                ts->tv_sec = 0;
        else
        { /* set the seconds as well as the nanoseconds */
                ts->tv_sec = ms/1000;
                ms %= 1000;
        }
        /* set the nanoseconds: 1e3*1e6 == 1e9 */
        ts->tv_nsec = ms*1000000;
}

int setDeadlinePct(int pid, int period, int pct)
{
        struct sched_deadline dd = {{0,0},{0,0}};
        int retval;

        putMSinTimestruc(&dd.dl_period, period);
        putMSinTimestruc(&dd.dl_alloc, (period * pct)/100);
        if (-1 == (retval = schedctl(DEADLINE, pid, &dd)) )
        {
                if (ENOSPC == errno)
                { /* system cannot guarantee that duty cycle */
                        fprintf(stderr,"schedctl: cannot promise %d%% of %dms\n",
                                                pct, period);
                }
                else perror("schedctl");
        }
        return retval;
}

int setDeadlinePolicy(int pid, struct sched_deadline *policy)
{
        int retval = schedctl(DEADLINE,pid,policy);
        if (-1 == retval)
        {
                char msg[64];
                sprintf(msg,"schedctl(DEADLINE,%d, %ld)",pid,(long)policy);
                perror(msg);
        }
        return retval;
}
#ifdef UNIT_TEST
int main(int argc, char **argv)
{
        int pct = 25;
```

```
        int per = 100;
        int pid = 0; /* which means "self" to schedctl() */
        if (1 < argc)
        {
                pct = atoi(argv[1]);
        }
        if (2 < argc)
        {
                per = atoi(argv[2]);
        }
        if (3 < argc)
        {
                pid = atoi(argv[3]);
        }
        if ( (4 < argc) || (0==pct) || (0==per) )
        {
                fprintf(stderr,
"usage: setDeadline [ pct_duty_cycle [ period_ms [ pid ] ] ]\n");
                exit();
        }
        printf("schedule pid %d for %d%% of %dms --> %d\n",
                                pid, pct, per, setDeadlinePct( pid, per, pct));
        printf("policy DL_ONLY-->%d\n", setDeadlinePolicy(pid,DL_ONLY));
        printf("policy DL_ANY-->%d\n", setDeadlinePolicy(pid,DL_ANY));
        printf("policy DL_RELEASE-->%d\n", setDeadlinePolicy(pid,DL_RELEASE));
}
#endif /* UNIT_TEST */
```

Frame Scheduler Examples

A number of example programs are distributed with the REACT / Pro Frame Scheduler. This section describes them. Only one is reproduced here; the others are found on disk.

The following example programs are distributed with the Frame Scheduler:

simple	Several processes scheduled on a single CPU.
mprogs	Processes loaded as independent programs dispatched by a Frame Scheduler.
multi	Three synchronous Frame Schedulers running lightweight processes on three processors.

driver	A pseudo-driver that demonstrates the Frame Scheduler device driver interface.
sixtyhz	One process scheduled at a 60 Hz frame rate.
memlock	The sixtyhz example with added code to lock memory and to skip the first minor frame to account for an initial cold cache.

The code for each example is found in a subdirectory named for the example.

Basic Example

This example shows how to map a real-time application specification into a Frame Scheduler specification and also how to write the corresponding program using the Frame Scheduler API. The code is reproduced starting on “Code of Basic Example” on page 232.

Real-Time Application Specification

The application consists of two processes that have to periodically execute a specific sequence of code. The period for the first process, process A, is 600 milliseconds. The period for the other process, process B, is 2400 ms.

Note: Such long periods are unrealistic for real-time applications. However, they allow the use of **printf()** calls within the “real-time” loops in this sample program.

Frame Scheduler Design

The two periods and their ratio determine the selection of the minor frame period—600 ms—and the number of minor frames per major frame—4, for a total of 2400 ms.

The discipline for process A is strict real-time (FRS_DISC_RT). Underrun and overrrun errors should cause signals.

Process B should run only once in 2400 ms, so it operates as Continuable over as many as 4 minor frames. For the first 3 frames, its discipline is Overrunnable and Continuable. For the last frame it is strict real-time. The Overrunnable discipline allows process B to run without yielding past the end of each minor frame. The Continuable discipline ensures that once process B does yield, it is not resumed until the fourth minor frame has passed. The combination allows process B to extend its execution to the allowable period of 2400 ms, and the strict real-time discipline at the end makes certain that it yields by the end of the major frame.

There is a single Frame Scheduler so a single processor is used by both processes. Process A runs within a minor frame until yielding or until the expiration of the minor frame period. In the latter case the frame scheduler generates an overrun error signaling that process A is misbehaving.

When process A yields, the frame scheduler immediately activates process B. It runs until yielding, or until the end of the minor frame at which point it is preempted. This is not an error since process B is Overrunable.

Starting the next minor frame, the Frame Scheduler allows process A to execute again. After it yields, process B is allowed to resume running, if it has not yet yielded. Again in the third and fourth minor frame, A is started, followed by B if it has not yet yielded. At the interrupt that signals the end of the fourth frame (and the end of the major frame), process B must have yielded, or an overrun error is signalled.

Example of Scheduling Separate Programs

The code in directory *mprogs* is based on the code for the basic example (see “Basic Example” on page 227). However, the scheduled processes A and B are physically loaded as separate commands. The main program establishes the single Frame Scheduler. The real-time processes are started as separate programs. They communicate with the main program using SVR4-compatible interprocess communication messages (see the intro(2) and msgget(2) reference pages).

There are three separate executables in the mprogs example. The master program, in *master.c*, is a command that has the following syntax:

`master [-p` *cpu-number*`] [-s` *slave-count*`]`

The *cpu-number* specifies which processor to use for the one Frame Scheduler this program creates. The default is processor 1. The *slave-count* tells the master how many subordinate programs will be enqueued to the Frame Scheduler. The default is two programs.

The problems that need to be solved in this example are as follows:

- The frs-master program must enqueue the subordinate processes. However, since they are started as separate programs, the master has no direct way of knowing their process IDs.
- The subordinates need to specify upon which minor frames they should be enqueued, and with what discipline.
- The master needs to enqueue the subordinates in the proper order on their minor frames, so they will be dispatched in the proper sequence. Therefore the master has to distinguish the subordinates in some way; it cannot treat them as interchangeable.
- The subordinate programs must join the Frame Scheduler, so they need the handle of the Frame Scheduler to use as an argument to **frs_join()**. However, this information is in the master's address space.
- If an error occurs when enqueueing, the master needs to tell the subordinate so they can terminate in an orderly way.

There are many ways in which these objectives could be met. In this example, the master and subordinates communicate using a simple protocol of messages exchanged using **msgget()** and **msgput()**. The sequence of operations is as follows:

1. The master program creates a Frame Scheduler.
2. The master sends a message inviting the most important subordinate to reply. (All the message queue handling is in module *ipc.c*, which is linked by all three programs.)
3. The subordinate compiled from the file *processA.c* replies to this message, sending its process ID and requesting the FRS handle.
4. The subordinate process A sends a series of messages, one for each minor queue on which it should enqueue. The master enqueues it as requested.
5. The subordinate process A sends a "ready" message.

6. The master sends a message inviting the next most important process to reply.
7. The program compiled from *processB.c* will reply to this request, and steps 3-6 are repeated for as many slaves as the *slave-count* parameter to the master program. (Only two slaves are provided. However, you can easily create more using *processB.c* as a pattern.)
8. The master issues **frs_start()**, and waits for the termination signal.
9. The subordinates independently issue **frs_join()** and the real-time dispatching begins.

Example of Multiple Synchronized Schedulers

The example *multi* demonstrates the creation of three synchronized Frame Schedulers. The three use the cycle counter to establish a minor frame interval of 50 ms. All three Frame Schedulers use 20 minor frames per major frame, for a major frame rate of 1 Hz.

The following processes are scheduled in this example:

- Processes A and D require a frequency of 20 Hz
- Process B requires a frequency of 10 Hz and can consume up to 100 ms of execution time each time
- Process C requires a frequence of 5 Hz and can consume up to 200 ms of execution time each time
- Process E requires a frequency of 4 Hz and can consume up to 250 ms of execution time each time
- Process F requires a frequency of 2 Hz and can consume up to 500 ms of execution time each time
- Processes K1, K2 and K3 are background processes that should run as often as possible, when time is available.

The processes are assigned to processors as follows:

- Scheduler 1 runs processes A (20 Hz) and K1 (background).
- Scheduler 2 runs processes B (10 Hz), C (5 Hz), and K2 (background).
- Scheduler 3 runs processes D (20Hz), E (4 Hz), F (2 Hz), and K3.

In order to simplify the coding of the example, all real-time processes use the same function body, **process_skeleton()**, which is parameterized with the process name, the number of the Frame Scheduler it is to join, and the "real-time" function it is to execute. In the sample code, all real-time functions are empty function bodies.

Example of Device Driver

The code in directory *driver* contains a skeletal test-bed for a kernel-level device driver that interacts with the Frame Scheduler. Most of the driver functions consist of minimal or empty stubs. However, the **ioctl()** entry point to the driver (see the ioctl(2) reference page) simulates a hardware interrupt and calls the Frame Scheduler entry point, **frs_handle_driverintr()** (see "Generating Interrupts" on page 149). This allows you to test the driver. Calling its **ioctl()** entry is equivalent to using **frs_usrintr()** (see "The Frame Scheduler API" on page 118).

Example of a 60 Hz Frame Rate

The example in directory *sixtyhz* demonstrates the ability to schedule a process at a frame rate of 60 Hz, a common target rate in visual simulators. A single Frame Scheduler is created. It uses the cycle counter with an interval of 16,666 microseconds (16.66 ms, approximately 60 Hz). There is one minor frame per major frame.

One real-time process is enqueued to the Frame Scheduler. By changing the compiler constant LOGLOOPS you can change the amount of work it attempts to do in each frame.

Example of Memory and Cache Management

The example in directory *memlock* is almost identical to that in directory *sixtyhz*. (Use the *diff* command to display the altered code.)

The functional difference between the two examples is that the real-time process in *memlock.c* locks its own virtual address space so as to avoid page faults. Also, it executes one major frame's worth of **frs_yield()** calls immediately after return from **frs_join()**. The purpose of this is to "warm up" the processor cache with copies of the process code and data. (An actual application process could access its major data structures prior to this yield in order to speed up the caching process.)

Code of Basic Example

```
/*
 * This example creates one simple frame scheduler.
 *
 * PARAMETERS:
 * cpu: 1
 * intr_source: FRS_INTRSOURCE_CCTIMER
 * n_minors: 4
 * period: 600 [ms] (a long period to allow for the printf statements)
 *
 * PROCESSES:
 * Process A: Period of 600 [ms] (determined base minor frame period)
 * Process B: Period of 2400 [ms] (determined # of minor frames per major frame)
 */
#include <math.h>
#include <signal.h>
#include <sys/schedctl.h>
#include <sys/sysmp.h>
#include <stdio.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/frs.h>
/*
 * Some fixed real-time loop parameters
 */
#define NLOOPS 40
#define LOGLOOPS_A 150
#define LOGLOOPS_B 30000
#define JOINM_A
#define JOINM_B
#define YIELDM_A
#define YIELDM_B
/*
 * Error Signal handlers
 */
```

```
void
underrun_error()
{
   if ((int)signal(SIGUSR1, underrun_error) == -1) {
      perror("[underrun_error]: Error while resetting signal");
      exit(1);
   }
   fprintf(stderr, "[underrun_error], PID=%d\n", getpid());
   exit(2);
}

void
overrun_error()
{
   if ((int)signal(SIGUSR2, overrun_error) == -1) {
      perror("[overrun_error]: Error while resetting signal");
      exit(1);
   }
   fprintf(stderr, "[overrun_error], PID=%d\n", getpid());
   exit(2);
}

void
termination_signal()
{
   if ((int)signal(SIGHUP, termination_signal) == -1) {
      perror("[termination_signal]: Error while resetting signal");
      exit(1);
   }
   fprintf(stderr, "[termination_signal], PID=%d\n", getpid());
   exit(8);
}
/*
 * This is process A, scheduled once on every minor frame
 */
void
processA(frs_t* frs)
{
   int counter;
   double res;
   int i;
   int previous_minor;
   int pid = getpid();
   /*
    * First, we join to the frame scheduler
```

```
    */
   if (frs_join(frs) < 0) {
      perror("[processA]: frs_join failed");
      exit(1);
   }

#ifdef JOINM_A
   fprintf(stderr, "[processA (%d)]: Joined Frame Scheduler on cpu %d\n",
      pid, frs->frs_info.cpu);
#endif

   counter = NLOOPS;
   res = 2;
   /*
    * This is the real-time loop. The first iteration
    * is done right after returning from the join
    */
   do {
      for (i = 0; i < LOGLOOPS_A; i++) {
         res = res * log(res) - res * sqrt(res);
      }
      /*
       * After we are done with our computations, we
       * yield the cpu. THe yield call will not return until
       * it is our turn to execute again.
       */
      if ((previous_minor = frs_yield()) < 0) {
         perror("[processA]: frs_yield failed");
         exit(1);
      }

#ifdef YIELDM_A
      fprintf(stderr, "[processA (%d)]: Return from Yield; previous_minor: %d\n",
         pid, previous_minor);
#endif
   } while (counter--);

   fprintf(stderr, "[ProcessA (%d)]: Destroying Frame Scheduler Managing cpu %d\n",
      pid, frs->frs_info.cpu);

   if (frs_destroy(frs) < 0) {
      perror("[ProcessA]: frs_destroy failed\n");
      exit(1);
   }
   fprintf(stderr, "[ProcessA (%d)]: Exiting\n", pid);
```

```
   exit(0);
}


/*
 * This is process B, scheduled Continuable on every minor frame
 */
void
processB(frs_t* frs)
{
   int counter;
   double res;
   int i;
   int previous_minor;
   int pid = getpid();
   /*
    * First, we join to the frame scheduler
    */
   if (frs_join(frs) < 0) {
      perror("[processB]: frs_join failed");
      exit(1);
   }

#ifdef JOINM_B
   fprintf(stderr, "[processB (%d)]: Joined Frame Scheduler on cpu %d\n",
      pid, frs->frs_info.cpu);
#endif

   counter = NLOOPS;
   res = 2;
   /*
    * This is the real-time loop. The first iteration
    * is done right after returning from the join
    */
   do {
      for (i = 0; i < LOGLOOPS_B; i++) {
         res = res * log(res) - res * sqrt(res);
      }
      /*
       * After we are done with our computations, we
       * yield the cpu. THe yield call will not return until
       * it's our turn to execute again.
       */
      if ((previous_minor = frs_yield()) < 0) {
         perror("[processB]: frs_yield failed");
```

```
         exit(1);
      }

#ifdef YIELDM_B
      fprintf(stderr, "[processB (%d)]: Return from Yield; previous_minor: %d\n",
         pid, previous_minor);
#endif
   } while (counter--);

   fprintf(stderr, "[processB (%d)]: Exiting\n", pid);
   exit(0);
}
/*
 * Common subroutine to fork off a child process, passing
 * the handle of the frame scheduler as a parameter.
 */
int
create_process(void (*f)(frs_t*), frs_t* frs)
{
   int pid;
   switch ((pid = fork())) {
         case 0: /* child process, invoke processing function */
             (*f)(frs); /* no need for break, exit() called */
         case -1: /* error return from fork() */
             perror("[create_process]: fork() failed");
             exit(1);
         default: /* parent process, return to caller */
             return (pid);
   }
}

void
setup_signals()
{
   if ((int)signal(SIGUSR1, underrun_error) == -1) {
      perror("[setup_signals]: Error while setting underrun_error signal");
      exit(1);
   }

   if ((int)signal(SIGUSR2, overrun_error) == -1) {
      perror("[setup_signals]: Error while setting overrun_error signal");
      exit(1);
   }
   if ((int)signal(SIGHUP, termination_signal) == -1) {
      perror("[setup_signals]: Error while setting termination signal");
```

```
      exit(1);
   }
}

main(int argc, char** argv)
{
   frs_t* frs;
   int minor;
   int pid;
   int wstatus;
   int cpu_number;
   /*
    * Usage: simple [cpu_number]
    */
   if (argc == 1) {
      cpu_number = 1;
   } else {
      cpu_number = atoi(argv[1]);
   }

   printf("Running Frame Scheduler on Processor [%d]\n", cpu_number);

   /*
    * Initialize signals to catch
    * termination signals and underrun, overrun errors.
    */
   setup_signals();

   /*
    * Create the frame scheduler object
    * cpu = cpu_number,
    * interrupt source = CCTIMER
    * number of minors = 4
    * slave mask = 0, no slaves
    * period = 600 [ms] == 600000 [microseconds]
    */
   if ( (frs = frs_create_master(cpu_number,
                                 FRS_INTRSOURCE_CCTIMER,
                                 600000,
                                 4,
                                 NULL,
                                 0))
        == NULL) {
      perror("[main]: frs_create_master failed");
      exit(1);
```

```
}
/*
 * Create the processes and enqueue
 */
/*
 * ProcessA will be enqueued on all minor frame queues
 * with a strict RT discipline
 */
pid = create_process(processA, frs);
for (minor = 0; minor < 4; minor++) {
   if (frs_enqueue(frs, pid, minor, FRS_DISC_RT) < 0) {
      perror("[main]: frs_enqueue of ProcessA failed");
      exit(1);
   }
}
/*
 * ProcessB will be enqueued on all minor frames, but the
 * disciplines will differ. We need continuability for the first
 * 3 frames, and absolute real-time for the last frame.
 */
pid = create_process(processB, frs);
for (minor = 0; minor < 3; minor++) {
   int disc = FRS_DISC_RT |
              FRS_DISC_UNDERRUNNABLE |
              FRS_DISC_OVERRUNNABLE |
              FRS_DISC_CONT;
   if (frs_enqueue(frs, pid, minor, disc) < 0) {
      perror("[main]: frs_enqueue of ProcessB failed");
      exit(1);
   }
}
if (frs_enqueue(frs, pid, 3, FRS_DISC_RT | FRS_DISC_UNDERRUNNABLE) < 0) {
   perror("[main]: frs_enqueue of ProcessB failed");
   exit(1);
}
/*
 * Now we are ready to start the frame scheduler
 */
if (frs_start(frs) < 0) {
   perror("[main]: frs_start failed");
   exit(1);
}
/*
 * Wait for the two processes to finish
 * The actual exit will happen from the
```

```
    * termination signal handler.
    */
   wait(&wstatus);
   wait(&wstatus);
   exit(0);
}
```

Asynchronous I/O Example

The following program demonstrates the use some asynchronous I/O functions. The basic purpose of the program is to read a list of input files and write their concatenated contents as its output. However, it reads the input files using **aio_read()**, and writes the output files using **aio_write()** and **aio_fsync()**. In addition, it can be compiled in either of two ways,

- to read and copy the input files one at a time, by calling a subroutine
- to read and copy the input files concurrently, using a separate process for each input file

There is no functional advantage to using multiple processes. Doing so merely makes the example more interesting. It also demonstrates that, even though multiple processes ask for output at different points in the same file at the same time, the output is written to the requested offsets.

The reading and writing is done in one of four functions. The functions all have the following structure:

1. Initialize the *aiocb* for the type of notification desired. The type of notification is the principal difference between the functions: some use signals, some callback functions, some no notification.
2. Until the input file is exhausted,
    - Call **aio_read()** for up to one BLOCKSIZE amount from the next offset in the input file
    - Wait for the read to complete
    - Call **aio_write()** to write the data read to the next offset in the output file
    - Wait for the write to complete

3. Use **aio_fsync()** to ensure that output is complete and wait for it to complete.

The four functions, **inProc0()** through **inProc3()**, differ only in the method they use to wait for completion.

- **inProc0()** alternates calling **aio_error()** with **sginap()** until the status is other than EINPROGRESS.
- **inProc1()** calls **aio_suspend()** to wait for the current operation.
- **inProc2()** sets the *aiocb* to request a signal on completion. When the signal handler is entered, it posts a semaphore.
- **inProc3()** waits on a semaphore which is posted from a callback function.

You select which of the four function to use with the *-a* argument to the program. If you compile the program with the variable DO_SPROCS defined as 0, the chosen function is called as a subroutine once for each input file. If you compile with DO_SPROCS defined as 1, the chosen function is launched by **sprocsp()** once for each input file.

```
/* ============================================================================

    aiocat.c : This example demonstrates the use of asynchronous I/O.

The command syntax is:

    aiocat [ -o outfile ] [-a {0|1|2|3} ] infilename...

The output file is given by -o, with $TMPDIR/aiocat.out by default.
The aio method of waiting for completion is given by -a as follows:

    -a 0 poll for completion with aio_error() (default)
    -a 1 wait for completion with aio_suspend()
    -a 2 wait on a semaphore posted from a signal handler
    -a 3 wait on a semaphore posted from a callback routine

Up to MAX_INFILES input files may be specified.  The output file contains
the input files' contents in the order they were specified. Thus the
output should be the same as "cat infilename... >outfile".

All input files are read in BLOCKSIZE units. All I/O is done asynchronously
and concurrently using one sproc'd process per file.  Thus in a
```

```
multiprocessor concurrent input can be done.

============================================================================== */

#define _ABI_SOURCE     /* ask for POSIX 1003.1b-1993 version of aio & sigs */
#define _SGI_MP_SOURCE  /* see the "Caveats" section of sproc(2) */

#include <sys/time.h>   /* for clock() */
#include <errno.h>      /* for perror() */
#include <stdio.h>      /* for printf() */
#include <stdlib.h>     /* for getenv(), malloc(3c) */
#include <ulocks.h>     /* usinit() & friends */
#include <bstring.h>    /* for bzero() */
#include <sys/resource.h> /* for prctl, get/setrlimit() */
#include <sys/prctl.h>  /* for prctl() */
#include <sys/types.h>  /* required by lseek(), prctl */
#include <unistd.h>     /* ditto */
#include <sys/types.h>  /* wanted by sproc() */
#include <sys/prctl.h>  /* ditto */
#include <signal.h>     /* for signals - gets sys/signal and sys/siginfo */
#include <aio.h>        /* async I/O */

#define BLOCKSIZE 2048  /* input units -- play with this number */
#define MAX_INFILES 10  /* most sprocs: anything from 4 to 20 or so */
#define DO_SPROCS 0     /* set 0 to do all I/O in a single process */

#define QUITIFNULL(PTR,MSG) if (NULL==PTR) {perror(MSG);return(errno);}
#define QUITIFMONE(INT,MSG) if (-1==INT) {perror(MSG);return(errno);}

/*****************************************************************************
|| The following structure contains the info needed by one child proc.
|| The main program builds an array of MAX_INFILES of these.
|| The reason for storing the actual filename here (not a pointer) is
|| to force the struct to >128 bytes.  Then, when the procs run in
|| different CPUs on a CHALLENGE, their info structs will be in different
|| cache lines, and a store by one proc will not invalidate a cache line
|| for its neighbor proc.
*/
typedef struct child
{
        /* read-only to child */
    char fname[100];        /* input filename from argv[n] */
    int         fd;         /* FD for this file */
    void*       buffer;     /* buffer for this file */
    int         procid;     /* process ID of child process */
```

```
    off_t       fsize;      /* size of this input file */
        /* read-write to child */
    usema_t*    sema;       /* semaphore used methods 2 & 3 */
    off_t       outbase;    /* starting offset in output file */
    off_t       inbase;     /* current offset in input file */
    clock_t     etime;      /* sum of utime/stime to read file */
    aiocb_t     acb;        /* aiocb used for reading and writing */
} child_t;

/*************************************************************************
|| Globals, accessible to all processes
*/
char*       ofName = NULL;  /* output file name string */
int         outFD;          /* output file descriptor */
usptr_t*    arena;          /* arena where everything is built */
barrier_t*  convene;        /* barrier used to sync up */
int         nprocs = 1;     /* 1 + number of child procs */
child_t*    array;          /* array of child_t structs in arena */
int         errors = 0;     /* always incremented on an error */

/*************************************************************************
|| forward declaration of the child process functions
*/
void inProc0(void *arg, size_t stk);    /* polls with aio_error() */
void inProc1(void *arg, size_t stk);    /* uses aio_suspend() */
void inProc2(void *arg, size_t stk);    /* uses a signal and semaphore */
void inProc3(void *arg, size_t stk);    /* uses a callback and semaphore */

/*************************************************************************
// The main()
*/
int main(int argc, char **argv)
{
    char*       tmpdir;         /* ->name string of temp dir */
    int         nfiles;         /* how many input files on cmd line */
    int         argno;          /* loop counter */
    child_t*    pc;             /* ->child_t of current file */
    void (*method)(void *,size_t) = inProc0; /* ->chosen input method */
    char        arenaPath[128]; /* build area for arena pathname */
    char        outPath[128];   /* build area for output pathname */

    /*
    || Ensure the name of a temporary directory.
    */
    tmpdir = getenv("TMPDIR");
```

```
if (!tmpdir) tmpdir = "/var/tmp";

/*
|| Build a name for the arena file.
*/
strcpy(arenaPath,tmpdir);
strcat(arenaPath,"/aiocat.wrk");

/*
|| Create the arena. First, call usconfig() to establish the
|| minimum size (twice the buffer size per file, to allow for misc usage)
|| and the (maximum) number of processes that may later use
|| this arena.  For this program that is MAX_INFILES+10, allowing
|| for our sprocs plus those done by aio_sgi_init().
|| These values apply to any arenas made subsequently, until changed.
*/
{
    ptrdiff_t ret;
    ret = usconfig(CONF_INITSIZE,2*BLOCKSIZE*MAX_INFILES);
    QUITIFMONE(ret,"usconfig size")
    ret = usconfig(CONF_INITUSERS,MAX_INFILES+10);
    QUITIFMONE(ret,"usconfig users")
    arena = usinit(arenaPath);
    QUITIFNULL(arena,"usinit")
}

/*
|| Allocate the barrier.
*/
convene = new_barrier(arena);
QUITIFNULL(convene,"new_barrier")

/*
|| Allocate the array of child info structs and zero it.
*/
array = (child_t*)usmalloc(MAX_INFILES*sizeof(child_t),arena);
QUITIFNULL(array,"usmalloc")
bzero((void *)array,MAX_INFILES*sizeof(child_t));

/*
|| Loop over the arguments, setting up child structs and
|| counting input files.  Quit if a file won't open or seek,
|| or if we can't get a buffer or semaphore.
*/
for (nfiles=0, argno=1; argno < argc; ++argno )
```

```
{
    if (0 == strcmp(argv[argno],"-o"))
    { /* is the -o argument */
        ++argno;
        if (argno < argc)
            ofName = argv[argno];
        else
        {
            fprintf(stderr,"-o must have a filename after\n");
            return -1;
        }
    }
    else if (0 == strcmp(argv[argno],"-a"))
    { /* is the -a argument */
        char c = argv[++argno][0];
        switch(c)
        {
        case '0' : method = inProc0; break;
        case '1' : method = inProc1; break;
        case '2' : method = inProc2; break;
        case '3' : method = inProc3; break;
        default:
            {
                fprintf(stderr,"unknown method -a %c\n",c);
                return -1;
            }
        }
    }
    else
    { /* neither -o nor -a, assume input file */
        if (nfiles < MAX_INFILES)
        {
            /*
            || save the filename
            */
            pc = &array[nfiles];
            strcpy(pc->fname,argv[argno]);
            /*
            || allocate a buffer and a semaphore.  Not all
            || child procs use the semaphore but so what?
            */
            pc->buffer = usmalloc(BLOCKSIZE,arena);
            QUITIFNULL(pc->buffer,"usmalloc(buffer)")
            pc->sema = usnewsema(arena,0);
            QUITIFNULL(pc->sema,"usnewsema")
```

```
            /*
            || open the file
            */
            pc->fd = open(pc->fname,O_RDONLY);
            QUITIFMONE(pc->fd,"open")
            /*
            || get the size of the file. This leaves the file
            || positioned at-end, but all aio_read calls have an
            || implied lseek, so there is no need to reposition.
            || NOTE: there is no check for zero-length file; that
            || is a valid (and interesting) test case.
            */
            pc->fsize = lseek(pc->fd,0,SEEK_END);
            QUITIFMONE(pc->fsize,"lseek")
            /*
            || set the starting base address of this input file
            || in the output file.  The first file starts at 0.
            || Each one after starts at prior base + prior size.
            */
            if (nfiles) /* not first */
                pc->outbase =
                    array[nfiles-1].fsize + array[nfiles-1].outbase;
            ++nfiles;
        }
        else
        {
            printf("Too many files, %s ignored\n",argv[argno]);
        }
    }
} /* end for(argc) */

/*
|| If there was no -o argument, construct an output file name.
*/
if (!ofName)
{
    strcpy(outPath,tmpdir);
    strcat(outPath,"/aiocat.out");
    ofName = outPath;
}

/*
|| Open, creating or truncating, the output file.
|| Do not use O_APPEND, which would constrain aio to doing
|| operations in sequence.
```

```
    */
    outFD = open(ofName, O_WRONLY+O_CREAT+O_TRUNC,0666);
    QUITIFMONE(outFD,"open(output)")

    /*
    || If there were no input files, just quit, leaving empty output
    */
    if (!nfiles)
    {
        return 0;
    }

    /*
    || Note the number of processes-to-be, for use in initializing
    || aio and for use by each child in a barrier() call.
    */
    nprocs = 1+nfiles;

    /*
    || Initialize async I/O using aio_sgi_init(), in order to specify
    || a number of locks at least equal to the number of child procs
    || and in order to specify extra sproc users.
    */
    {
        aioinit_t ainit;
        bzero((void*)&ainit,sizeof(ainit)); /* all fields zero */
        /*
        || for the number of aio-created procs go with the default 5,
        || since we have no way of knowing the number of unique devices.
        */
#define AIO_PROCS 5
        ainit.aio_threads = AIO_PROCS;
        /*
        || Set the number of locks aio needs to the number of procs
        || we will start, minimum 3.
        */
        ainit.aio_locks = (nprocs > 2)?nprocs:3;
        /*
        || New field: warn aio of the number of user procs that will be
        || using its arena.
        */
        ainit.aio_numusers = nprocs;
        aio_sgi_init(&ainit);
    }
```

```
    /*
    || For each input file, start a child process.
    || If an error occurs, quit. That will send a SIGHUP to each
    || already-started child, which will kill it, too.
    */
    for (argno = 0; argno < nfiles; ++argno)
    {
        pc = &array[argno];

#if DO_SPROCS
#define CHILD_STACK 64*1024

        pc->procid = sprocsp(method     /* function to start */
                            ,PR_SALL    /* share all, keep FDs sync'd */
                            ,(void *)pc /* argument to child func */
                            ,NULL       /* absolute stack seg */
                            ,CHILD_STACK);  /* max stack seg growth */
        QUITIFMONE(pc->procid,"sproc")
#else
        fprintf(stderr,"file %s...",pc->fname);
        method((void*)pc,0);
        if (errors) break;
        fprintf(stderr,"done\n");
#endif
    }
#if DO_SPROCS
    /*
    || Wait for all the kiddies to get themselves initialized.
    || When all have started and reached barrier(), all continue.
    || If any errors occurred in initialization, quit.
    */
    barrier(convene,nprocs);
    /*
    || Child processes are executing now. Reunite the family round the
    || old hearth one last time, when their processing is complete.
    || If any errors seen at that point, quit.
    */
    barrier(convene,nprocs);
#endif
    /*
    || Close the output file. Each child process has ensured that
    || async writes it started have finished.
    */
    close(outFD);
    {
```

```
        clock_t timesum;
        long    bytesum;
        double  bperus;
        printf("    procid   time     fsize     filename\n");
        for(argno = 0, timesum = bytesum = 0 ; argno < nfiles ; ++argno)
        {
            pc = &array[argno];
            timesum += pc->etime;
            bytesum += pc->fsize;
            printf("%2d: %-8d %-8d %-8d  %s\n"
                    ,argno,pc->procid,pc->etime,pc->fsize,pc->fname);
        }
        bperus = ((double)bytesum)/((double)timesum);
        printf("total time %d usec, total bytes %d, %g bytes/usec\n"
                     ,timesum            , bytesum , bperus);
    }
    /*
    || Unlink the arena file, so it won't exist when this progam runs
    || again. If it did exist, it would be used as the initial state of
    || the arena, which might or might not have any effect.
    */
    unlink(arenaPath);
    return 0;
}

/*****************************************************************************
|| inProc0() alternates sginap() with polling with aio_error(). Under
|| the Frame Scheduler, it would use frs_yield() instead of sginap().
|| The general pattern of this function is repeated in the other three;
|| only the wait method varies from method to method.
*/

int inWait0(child_t *pch)
{
    int ret;
    aiocb_t* pab = &pch->acb;
    while (EINPROGRESS == (ret = aio_error(pab)))
    {
        sginap(0);
    }
    return ret;
}

void inProc0(void *arg, size_t stk)
{
```

```
    child_t *pch = arg;         /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;   /* base address of the aiocb_t in child_t */
    int ret;                    /* as long as this is 0, all is ok */
    int bytes;                  /* #bytes read on each input */

    /*
    || Initialize -- no signals or callbacks needed.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_NONE;
    pab->aio_buf = pch->buffer; /* always the same */
#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (ret = aio_read(pab))
            break;
        ret = inWait0(pch);
        if (ret)
            break;  /* nonzero read completion status */
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
            break;  /* no need to write a last block of 0 */
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
```

```
        if (ret = aio_write(pab))
            break;
        ret = inWait0(pch);
        if (ret)
            break;
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));

    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    ret = aio_fsync(O_SYNC,pab);
    ret = inWait0(pch);

    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
    return;
} /* end inProc1 */

/*****************************************************************************
|| inProc1 uses aio_suspend() to await the completion of each operation.
|| Otherwise it is the same as inProc0.
*/

int inWait1(child_t *pch)
{
    int ret;
    aiocb_t* susplist[1]; /* list of 1 aiocb for aio_suspend() */
    susplist[0] = &pch->acb;
    /*
    || Note: aio.h declare the 1st argument of aio_suspend() as "const."
    || The C compiler requires the actual-parameter to match in type,
    || so the list we pass must either be declared "const aiocb_t*" or
    || must be cast to that -- or cc gives a warning.  Thus the cast
```

```
    || in the following statement is only to avoid a warning.
    */
    ret = aio_suspend( (const aiocb_t **) susplist,1,NULL);
    return ret;
}

void inProc1(void *arg, size_t stk)
{
    child_t *pch = arg;         /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;   /* base address of the aiocb_t in child_t */
    int ret;                    /* as long as this is 0, all is ok */
    int bytes;                  /* #bytes read on each input */

    /*
    || Initialize -- no signals or callbacks needed.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_NONE;
    pab->aio_buf = pch->buffer; /* always the same */
#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (ret = aio_read(pab))
            break;
        ret = inWait1(pch);
        /*
        || If the aio_suspend() return is nonzero, it means that the wait
        || did not end for i/o completion but because of a signal. Since we
        || expect no signals here, we take that as an error.
        */
        if (!ret) /* op is complete */
            ret = aio_error(pab);  /* read() status, should be 0 */
        if (ret)
            break;  /* signal, or nonzero read completion */
```

```
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
            break;  /* no need to write a last block of 0 */
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
        if (ret = aio_write(pab))
            break;
        ret = inWait1(pch);
        if (!ret) /* op is complete */
            ret = aio_error(pab);  /* should be 0 */
        if (ret)
            break;
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));

    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    ret = aio_fsync(O_SYNC,pab);
    ret = inWait1(pch);

    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
} /* end inProc0 */
```

```
/*****************************************************************************
|| inProc2 requests a signal upon completion of an I/O. After starting
|| an operation, it P's a semaphore which is V'd from the signal handler.
*/

#define AIO_SIGNUM SIGRTMIN+1 /* arbitrary choice of signal */

void sigHandler2(const int signo, const struct siginfo *sif )
{
    /*
    || In this minimal signal handler we pick up the address of the
    || child_t info structure -- which was put in aio_sigevent.sigev_value
    || field during initialization -- and use it to find the semaphore.
    */
    child_t *pch = sif->_data._value.sival_ptr ;
    usvsema(pch->sema);
    return; /* stop here with dbx to print the above address */
}

int inWait2(child_t *pch)
{
    /*
    || Wait for any signal handler to post the semaphore.  The signal
    || handler could have been entered before this function is called,
    || or could be entered afterward.
    */
    uspsema(pch->sema);
    /*
    || Since this process executes only one aio operation at a time,
    || we can return the status of that operation.  This is only a
    || coding convenience. It could not be done if a signal could
    || arrive from more than one pending operation.
    */
    return aio_error(&pch->acb);
}

void inProc2(void *arg, size_t stk)
{
    child_t *pch = arg;         /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;   /* base address of the aiocb_t in child_t */
    int ret;                    /* as long as this is 0, all is ok */
    int bytes;                  /* #bytes read on each input */

    /*
```

```
    || Initialize -- request a signal in aio_sigevent. The address of
    || the child_t struct is passed as the siginfo value, for use
    || in the signal handler.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    pab->aio_sigevent.sigev_signo = AIO_SIGNUM;
    pab->aio_sigevent.sigev_value.sival_ptr = (void *)pch;
    pab->aio_buf = pch->buffer; /* always the same */
    /*
    || Initialize -- set up a signal handler for AIO_SIGNUM.
    */
    {
        struct sigaction sa = {SA_SIGINFO,sigHandler2};
        ret = sigaction(AIO_SIGNUM,&sa,NULL);
        if (ret) ++errors; /* parent will shut down ASAP */
    }

#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#else
    if (ret) return;
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (!(ret = aio_read(pab)))
            ret = inWait2(pch);
        if (ret)
            break;  /* read error */
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
```

```
            break;  /* no need to write a last block of 0 */
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
        if (!(ret = aio_write(pab)))
             ret = inWait2(pch);
        if (ret)
            break;
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));

    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    ret = aio_fsync(O_SYNC,pab);
    ret = inWait2(pch);

    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
} /* end inProc2 */

/*****************************************************************************
|| inProc3 uses a callback and a semaphore. It waits with a P operation.
|| The callback function executes a V operation.  This may come before or
|| after the P operation.
*/

void callBack3(union sigval usv)
{
    /*
```

```
    || The callback function receives the pointer to the child_t struct,
    || as prepared in aio_sigevent.sigev_value.sival_ptr.  Use this to
    || post the semaphore in the child_t struct.
    */
    child_t *pch = usv.sival_ptr;
    usvsema(pch->sema);
    return;
}

int inWait3(child_t *pch)
{
    /*
    || Suspend, if necessary, by pulling the semaphore.
    */
    uspsema(pch->sema);
    /*
    || Return the status of the aio operation associated with the sema.
    */
    return aio_error(&pch->acb);
}

void inProc3(void *arg, size_t stk)
{
    child_t *pch = arg;         /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;   /* base address of the aiocb_t in child_t */
    int ret;                    /* as long as this is 0, all is ok */
    int bytes;                  /* #bytes read on each input */

    /*
    || Initialize -- request a callback in aio_sigevent. The address of
    || the child_t struct is passed as the siginfo value to be passed
    || into the callback.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_CALLBACK;
    pab->aio_sigevent.sigev_func = callBack3;
    pab->aio_sigevent.sigev_value.sival_ptr = (void *)pch;
    pab->aio_buf = pch->buffer; /* always the same */

#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#endif
    pch->etime = clock();
```

```
do /* read and write, read and write... */
{
    /*
    || Set up the aiocb for a read, queue it, and wait for it.
    */
    pab->aio_fildes = pch->fd;
    pab->aio_offset = pch->inbase;
    pab->aio_nbytes = BLOCKSIZE;
    if (!(ret = aio_read(pab)))
        ret = inWait3(pch);
    if (ret)
        break;  /* read error */
    /*
    || get the result of the read() call, the count of bytes read.
    || Since aio_error returned 0, the count is nonnegative.
    || It could be 0, or less than BLOCKSIZE, indicating EOF.
    */
    bytes = aio_return(pab); /* actual read result */
    if (!bytes)
        break;  /* no need to write a last block of 0 */
    pch->inbase += bytes;   /* where to read next time */
    /*
    || Set up the aiocb for a write, queue it, and wait for it.
    */
    pab->aio_fildes = outFD;
    pab->aio_nbytes = bytes;
    pab->aio_offset = pch->outbase;
    if (!(ret = aio_write(pab)))
         ret = inWait3(pch);
    if (ret)
        break;
    pch->outbase += bytes;  /* where to write next time */
} while ((!ret) && (bytes == BLOCKSIZE));

/*
|| The loop is complete.  If no errors so far, use aio_fsync()
|| to ensure that output is complete.  This requires waiting
|| yet again.
*/
ret = aio_fsync(O_SYNC,pab);
ret = inWait3(pch);

/*
|| Flag any errors for the parent proc. If none, count elapsed time.
*/
```

```
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
} /* end inProc3 */
```

Guaranteed-Rate Request

The following subroutine simplifies the task of requesting a guaranteed rate of I/O transfer. The file descriptor passed to function **requestRate()** must describe a file located in the real-time subvolume of a volume managed by XLV and XFS.

```
/*
 * Simple function to request a guaranteed rate reservation.
 * Input:
 *      fd      file descriptor to be guaranteed
 *      dur     duration of guarantee in seconds
 *      bps     bytes per second required
 *      flag    one of SOFT_ or HARD_GUARANTEE [+VOD_LAYOUT]
 *              (extra entry points included for those who do not
 *              want to include sys/grio.h)
 *
 * Assumed:
 *      reservation start time of "1 second from now"
 *      guarantee unit time of 1 second
 *
 * Returns:
 *       0    success,  guarantee granted
 *      -1    error returned and displayed with perror()
 *      +n    n is the best bytes/second that XFS can offer
 *
 * Usage:
 *      #define BEST_RATE zillions
 *      #define MINIMAL_RATE somewhat_less
 *      if ( (ret = requestRate(fd, dur, BEST_RATE, SOFT_GUARANTEE)) )
 *      { // not a success
 *        if (ret >= MINIMAL_RATE) // acceptable lower rate offered
```

```
 *        ret = requestRate(fd, dur, ret, SOFT_GUARANTEE);
 *      }
 *      if (ret) // failed for some reason
 *      {
 *        if (0<ret) // not an error as such
 *           fprintf(stderr, "Cannot get rate\n");
 *        exit();
 *      }
 *      // guaranteed rate obtained, continue
 */
#include <sys/types.h>  /* for time_t */
#include <time.h>       /* for time() */
#include <errno.h>      /* for error codes */
#include <stdio.h>      /* [fs]printf() */
#include "grio.h"       /* for grio_* */

/*
 * This subroutine displays a diagnostic message to stderr when
 * grio_request() returns an error. perror() cannot be used in
 * this case because the generic messages are not descriptive.
 *
 */
void printGRIOerror( grio_resv_t *g )
{
    char work[80];
    char *msg = work;

    switch (g->gr_error)
    {
    case EINVAL:
    {
        msg = "unable to contact grio daemon";
        break;
    }
    case EBADF:
    {
        msg = "cannot stat file, or file already guaranteed";
        break;
    }
    case ESRCH:
    {
        msg = "invalid procid";
        break;
    }
    case ENOENT:
```

```
    {
        msg = "file has no (real-time?) extents";
        break;
    }
    case EIO:
    {
        msg = "incorrect start time";
        break;
    }
    case EACCES:
    {
        msg = (g->gr_flags & VOD_LAYOUT)
              ? "unable to provide VOD guarantee"
              : (
                (g->gr_flags & HARD_GUARANTEE)
                ? "unable to provide HARD guarantee"
                : "unable to provide SOFT guarantee"
            );
        break;
    }
    case ENOSPC:
    {
        sprintf(work, "out of bandwidth on device %s",
                    g->gr_errordev);
        break;
    }
    default: /* in case they think of something else */
    {
        sprintf(work, "error %d", g->gr_error);
    }
    }
    fprintf(stderr, "grio_request: %s.\n", msg);
}

/*
 * This function actually places the request.
 */
int requestRate( int fd, int dur, int bps, int flag)
{
    int ret;
    grio_resv_t grio;

    grio.gr_duration= dur;
    grio.gr_start   = 1+time(NULL);
    grio.gr_optime  = 1; /* unit time is 1 second */
```

```
    grio.gr_opsize  = bps;
    grio.gr_flags   = flag;
    ret = grio_request(fd, &grio);
    if (ret) /* not a success */
    {
        if ( (ENOSPC == grio.gr_error) /* insufficient bandwidth.. */
        &&   (grio.gr_opsize) ) /* ..but nonzero rate remaining */
            ret = grio.gr_opsize; /* return available rate */
        else /* some other problem or 0 bandwidth available */
            printGRIOerror(&grio);
    }
    return ret;
}
/*
 * When you don't want to #include sys/grio.h to define one constant...
 */
int requestHardRate( int fd, int dur, int bps )
{ return requestRate(fd, dur, bps, HARD_GUARANTEE); }

int requestSoftRate( int fd, int dur, int bps )
{ return requestRate(fd, dur, bps, SOFT_GUARANTEE); }

#ifdef UNIT_TEST
#include <sys/stat.h>
#include <fcntl.h>
/* requestRate pathname [rate [duration [flags ] ] ] */
int main(int argc, char **argv)
{
    int fd = open(argv[1], O_RDONLY);
    int bps = 1000000; /* 1MB */
    int dur = 60; /* a new york minute */
    int flg = SOFT_GUARANTEE;
    int rc;
    if (argc > 2) bps = atoi(argv[2]);
    if (argc > 3) dur = atoi(argv[3]);
    if (argc > 4) flg = atoi(argv[4]);
    printf("Requesting guarantee on fd=%d of %d bps for %d sec\n",
                                        fd,   bps,       dur);
    rc = requestRate(fd, dur, bps, flg);
    printf("requestRate() returns %d\n", rc);
}
#endif /*UNIT_TEST*/
```

Glossary

activity

When using the Frame Scheduler, the basic design unit: a piece of work that can be done by one process without interruption. You partition the real-time program into activities, and use the Frame Scheduler to invoke them in sequence within each frame interval.

address space

The set memory addresses that a process may legally access. The potential address space in IRIX is either 2^{32} (IRIX 5.3) or 2^{64} (IRIX 6.0); however only addresses that have been mapped by the kernel are legally accessible.

affinity scheduling

The IRIX kernel attempts to run a process on the same CPU where it most recently ran, in the hope that some of the process's data will still remain in the cache of that CPU. The process is said to have "cache affinity" for that CPU. ("Affinity" means "a natural relationship or attraction.")

arena

A segment of memory used as a pool for allocation of objects of a particular type. Usually the shared memory segment allocated by **usinit()**.

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, provided that it has enough memory buffers to deal with bursts at the *peak data rate*.

backing store

The disk location that contains the contents of a memory page. The contents of the page are retrieved from the backing store when the page is needed in memory. The backing store for executable code is the program or library file. The backing store for modifiable pages is the swap disk. The backing store for a memory-mapped file is the file itself.

barrier

A memory object that represents a point of rendezvous or synchronization between multiple processes. The processes come to the barrier asynchronously, and block there until all have arrived. Then all resume execution together.

context switch time

The time required for IRIX to set aside the context, or execution state, of one process and to enter the context of another; for example, the time to leave a process and enter a device driver, or to leave a device driver and resume execution of an interrupted user process.

deadline scheduling

A process scheduling discipline supported by IRIX version 5.3. A process may require that it receive a specified amount of execution time over a specified interval, for instance 70ms in every 100ms. IRIX adjusts the process's priority up and down as required to ensure that it gets the required execution time.

deadlock

A situation in which two (or more) processes are blocked because each is waiting for a resource held by the other.

device driver

Code that operates a specific hardware device and handles interrupts from that device. Refer to the *IRIX Device Driver Programming Guide*, part number 007-0911-040.

device numbers

Each I/O device is represented by a name in the */dev* file system hierarchy. When these "special device files" are created (see the makedev(1) reference page) they are given major and minor device numbers. The major number is

the index of a *device driver* in the kernel. The minor number is specific to the device, and encodes such information as its unit number, density, VME bus address space, or similar hardware-dependent information.

device service time

The amount of time spent executing the code of a *device driver* in servicing one interrupt. One of the three main components of *interrupt response time*.

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*.

direct memory access (DMA)

Independent hardware that transfers data between memory and an I/O device without direct program control. The Challenge/Onyx systems have a DMA engine for the VME bus.

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 time of 16.67 milliseconds.

frs-master

The process that creates a Frame Scheduler. Its process ID is used to identify the Frame Scheduler internally, so a process can only be frs-master to one scheduler.

gang scheduling

A process scheduling discipline supported by IRIX version 5.3. The processes of a process group can request to be scheduled as a gang; that is, IRIX attempts to schedule all of them concurrently when it schedules any of them—provided there are enough CPUs. When processes coordinate using locks, gang scheduling helps to ensure that one does not spend its whole time slice spinning on a lock held by another that is not running.

guaranteed rate

A rate of data transfer, in bytes per second, that definitely is available through a particular file descriptor.

hard guarantee

A type of *guaranteed rate* that is met even if data integrity has to be sacrificed to meat it.

heap

The *segment* of the *address space* devoted to static data and dynamically-allocated objects. Created by calls to the system function **brk()**.

interrupt

A hardware signal from an I/O device that causes the computer to divert execution to a device driver.

interrupt group

In the Challenge/Onyx hardware, each CPU has a register containing an interrupt group mask. Each interrupt source can be directed to a specific CPU or to an interrupt group number. When the interrupt destination is a group, all CPUs that have enabled that group receive the interrupt. The Frame Scheduler creates an interrupt group in order to synchronize minor frames among multiple synchronized CPUs.

interrupt latency

The amount of time that elapses between the arrival of an interrupt signal and the entry to the device driver that handles the interrupt.

interrupt response time

The total time from the arrival of an interrupt until the user process is executing again. Its three main components are *interrupt latency*, *device service time*, and *context switch time*.

locality of reference

The degree to which a program keeps memory references confined to a small number of locations over any short span of time. The better the locality of reference, the more likely a program will execute entirely from fast cache memory. The more scattered are a program's memory references, the higher is the chance that it will access main memory or, worse, load a page from swap.

locks

Memory objects that represent the exclusive right to use an atomic shared resource. A process that wants to use the resource requests the lock that (by agreement) stands for that resource. The process releases the lock when it is finished using the resource. See *semaphore*.

major frame

The basic frame rate of a program running under the Frame Scheduler.

minor frame

The scheduling unit of the Frame Scheduler, the period of time in which any scheduled process must do its work.

overrun

When incoming data arrives faster than a data collection system can accept it, so that data is lost, an overrun has occurred.

overrun exception

When a process scheduled by the Frame Scheduler should have yielded before the end of the minor frame and did not, an overrun exception is signalled.

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 *average data rate*.

process

The entity that executes instructions in a UNIX system. A process has access to an *address space* containing its instructions and data. The state of a process includes its set of machine register values, as well as many *process attributes*.

process attributes

Variable information about the state of a process. Every process has a number of attributes, including such things as its machine register contents, process ID, user and group IDs, working directory, open file handles, scheduler class, environment variables, and so on.

process group

See *share group*.

processor sets

Groups of one or more CPUs designated using the *pset* command.

programmed I/O (PIO)

Transfer of data between memory and an I/O device in byte or word units, using program instructions for each unit. Under IRIX, I/O to memory-mapped VME devices is done with PIO. See DMA.

race condition

Any situation in which two or more processes update a shared resource in an uncoordinated way. For example, if one process sets a word of shared memory to 1, and the other sets it to 2, the final result depends on the “race” between the two to see which can update memory last. Race conditions are prevented by use of *semaphores* or *locks*.

resident set size

The aggregate size of the valid (that is, memory-resident) pages in the address space of a process. Reported by *ps*(1) under the heading RSS. See *virtual size*.

segment

Any contiguous range of memory addresses. Segments as allocated by IRIX 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 *locks*.

share group

A group of two or more processes created with **sproc()**, including the original parent process. Processes in a share group share a common *address space* and can be scheduled as a gang (see *gang scheduling*). Also called a *process group*.

signal latency

The time that elapses from the moment when a signal is generated until the signal-handling function begins to execute. Signal latency is longer, and much less predictable, than *interrupt latency*.

soft guarantee

A type of *guaranteed rate* that XFS may fail to meet in order to retry device errors.

spraying interrupts

In order to equalize workload across all CPUs, the Challenge / Onyx systems direct each I / O interrupt to a different CPU chosen at random. In order to protect a real-time program from unpredictable interrupts, you can isolate specified CPUs from sprayed interrupts, or you can assign interrupts to specified CPUs.

striped volume

A logical disk volume comprising multiple disk drives, in which segments of data that are logically in sequence (“stripes”) are physically located on each drive in turn. As many processes as there are drives in the volume can read concurrently at the maximum rate.

translation lookaside buffer (TLB)

An on-chip cache of recently-used virtual-memory page addresses, with their physical-memory translations. The CPU uses the TLB to translate virtual addresses to physical ones at high speed. When the IRIX kernel alters the in-memory page translation tables, it broadcasts an interrupt to all CPUs, telling them to purge their TLBs.

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.

underrun exception

When a process scheduled by the Frame Scheduler should have started in a given minor frame but did not (owing to being blocked), an underrun exception is signalled. See *overrun exception*.

VERSA-Model Eurocard (VME) bus

A hardware interface and device protocol for attaching I/O devices to a computer. The VME bus is an ANSI standard. Many third-party manufacturers make VME-compatible devices. The Silicon Graphics Challenge/Onyx, Crimson, and IRIS-4D computer lines support the VME bus.

video on demand (VOD)

In general, producing video data at video frame rates. Specific to *guaranteed rate*, a disk organization that places data across the drives of a *striped volume* so that multiple processes can achieve the same guaranteed rate while reading sequentially.

virtual size

The aggregate size of all the pages that are defined in the address space of a process. The virtual size of a process is reported by *ps*(1) under the heading SZ. The sum of all virtual sizes cannot exceed the size of the swap space. See *resident set size.*

virtual address space

The set of numbers that a process can validly use as memory addresses.

Index

D

E

F

G

H

I

M

T

U

V

W

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2499-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389