REACT™ Real-Time Programmer’s Guide

Document Number 007-2499-003

CONTRIBUTORS

Written by David Cortesi
Illustrated by Gloria Ackley
Production by Lorrie Williams
Engineering contributions by Rich Altmaier, Joe Caradonna, Jeffrey Heller, Ralph Humphries, and Luis Stevens
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

REACT™ Real-Time Programmer's Guide
Document Number 007-2499-003

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 the IRIX™ operating system provides 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 REACT for IRIX 6.4, and REACT/Pro 3.0.

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. In the online book, the name of a reference page ("man page") is red in color (for example: mpin(2), sproc(2)). You can click on these names to cause the reference page to open automatically in a separate terminal window.

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

Chapter 3, “Controlling CPU Workload,” describes how you can isolate a CPU and dedicate almost all of its cycles to your program’s use.

Chapter 4, “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 5, “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 6, “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, see *Programming on Silicon Graphics Systems: An Overview*; part number 007-2476-*nnn*.
- The *WindView™ for IRIX Programmer’s Guide*, part number 007-2824-*nnn*, tells how to use a graphical performance analysis tool that can be of great help in debugging and tuning a real-time application on a multiprocessor system.
- The *IRIX Device Driver Programmer’s Guide*, part number 007-0911-*nnn*, gives details on all types of device control, including programmed I/O (PIO) and direct memory

access (DMA) from the user process, as well as discussing the design and construction of device drivers and other kernel-level modules.

- Administration of a multiprocessor is covered in a family of six books, including
  - *IRIX Admin: System Configuration and Operation* (007-2859-*nnn*)
  - *IRIX Admin: Disks and Filesystems* (007-2825-*nnn*)
  - *IRIX Admin: Peripheral Devices* (007-2861-*nnn*)
- For details of the architecture of the CPU, processor cache, processor bus, and virtual memory, see the *MIPS R4000 Microprocessor User's Manual, 2nd Ed.* by Joseph Heinrich. This and other chip-specific documents are available for downloading from the MIPS home page, http://www.mips.com.
- For details of some IRIX system facilities not covered in this book, *Topics in IRIX Programming*, part number 007-2478-*nnn* and *MIPS Compiling and Performance Tuning Guide*, 007-2479-*nnn* (both available with the IRIX Developer's Option).
- For programming inter-computer connections using sockets, *IRIX Network Programming Guide*, part number 007-0810-*nnn*.
- For coding functions in assembly language, *MIPSpro Assembly Language Programmer's Guide*, part number 007-2418-*nnn*.

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 merely slow, not incorrect.

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, dedicated process-control computers are sometimes a more economical choice 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.
- External devices to display the state of the model; for example, one or more video displays, audio speakers, or a simulated instrument panel.
- 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 moving the controls in response to what is shown on the display.

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 it updates the display. 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 rates higher and lower than these are used in special situations.

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

The REACT/Pro Frame Scheduler helps you organize a multi-process application to achieve a specified frame rate. (See Chapter 4, “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 lags behind control inputs, 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 a nuclear or conventional power generation plant. 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 attach 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 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 inputs to the computer under test are the simulator’s display output. The output signals of the computer under test are the simulator’s control inputs.

Depending on the hardware being exercised, the simulator may have to maintain a very high frame rate, up to 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 the following major parts:

1. Sources of data, for example telemetry. Often the source or sources are interfaced to the VME bus, but serial ports, SCSI devices, and other device types are also used..
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. The display is typically not required to maintain a particular frame rate, however.

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.

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 Ada 95, a new implementation of the language. Ada 95 programs can call any function that is available to a C program, so all the facilities described in this book are available, although the calling syntax may vary slightly. Ada offers additional features that are useful in real-time programming; for example, it includes a partial implementation of POSIX threads which is used to implement Ada tasking.

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 46).

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, 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

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

This chapter provides an overview of the real-time support in IRIX and REACT / Pro.

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 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 23), 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.
- 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 25).

Special Scheduling Disciplines

The default IRIX scheduling algorithm is designed to ensure fairness among time-shared users. Called an “earnings-based” scheduler, the kernel credits each process group with a certain number of microseconds on each dispatch cycle. The process with the fattest “bank account” is dispatched first. If a process exhausts its “bank account” it is preempted.

Nonpreemptive Priority Range

While effective for its purpose, the earnings-based scheduler is not suitable for a real-time process, which cannot tolerate being preempted for any reason. The kernel supports a range of priorities that are higher than the time-sharing priorities, and which are not subject to “earnings” controls.

Gang Scheduling

When your program is structured as a process group, 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 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 the manual *Topics in IRIX Programming* (see “Other Useful Books” on page xviii).

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, and gang scheduling). Since the number of ready processes changes all the time, dispatching is a random process. A normal process cannot predict how often or when it will next be able to run. For normal programs this does not matter, as long as each process continues to run at a satisfactory average rate.

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 31. The REACT / Pro Frame Scheduler takes care of both steps automatically; see “REACT / Pro Frame Scheduler” on page 10.

Controlling Interrupt Distribution

In normal operations, CPUs receive frequent interrupts:

- I/O interrupts from devices attached to, or near, that CPU.
- 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 19), 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 28. 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 during every frame interval. When there is more to do in a frame than one CPU can do, 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, in sequence, on 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 21)
- 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 2-1.

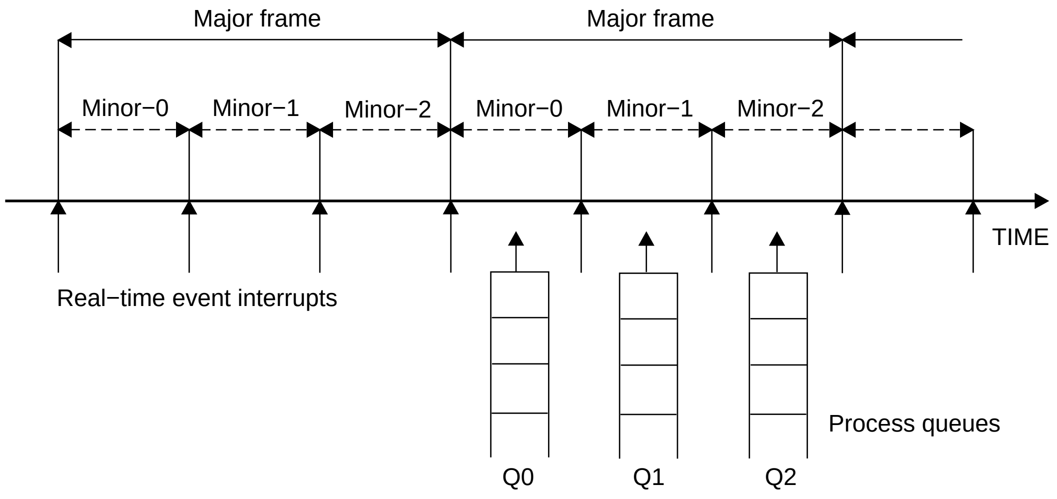

Figure 2-1 Major and Minor Frames

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

Sometimes what is here called a “major frame” 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 14).

When the real-time activities can be handled in a single CPU, the master process that initiates the program contains these steps:

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

   Each process initializes itself and then waits at a barrier.
4. Enqueue each activity process to the Frame Scheduler that will dispatch it (another library call).

   The master process specifies the process ID and the minor frame or frames in which the process should run, and a scheduling discipline.
5. Join the barrier where the activity processes are waiting.

   When all processes are ready to proceed, all are released.
6. Start the Frame Scheduler going (a library call).
7. Wait for a signal indicating it is time to shut down.
8. Terminate the Frame Schedulers.

A 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 to execute a library call to ”join” the scheduler. 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. Typically the error action is to send a signal to the master process. The master process can interrogate the Frame Scheduler, and stop it or restart it.

The Frame Scheduler is discussed in more detail in Chapter 4, “Using the Frame Scheduler”. Sample programs that illustrate the Frame Scheduler are described under “Frame Scheduler Examples” on page 143.

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. IRIX supports the POSIX and the SVR4 models of shared memory, as well as a system of shared arenas unique to IRIX. These facilities are covered in detail in the manual *Topics in IRIX Programming* (see “Other Useful Books” on page xviii).

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 greater than zero before the operation, at least one resource unit is available. The count is reduced by 1 and the process continues executing. When the count is not greater than zero, 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 three forms of semaphore: POSIX-compliant, SVR4-compatible, and Silicon Graphics. All three forms are discussed in the manual *Topics in IRIX Programming* (see “Other Useful Books” on page xviii)

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-lock 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.

For more information on locks, refer to the manual *Topics in IRIX Programming* (see "Other Useful Books" on page xviii), and to the usnewlock(3), ussetlock(3) and usunsetlock(3) reference pages.

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 xviii.

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 20) 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.

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 70.) 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 70.

Signal Families

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 2-1). Your real-time program should use the POSIX interface for signals.

Table 2-1 Signal Handling Interfaces

Function	SVR4-compatible Calls	BSD 4.2 Calls	POSIX Calls

set and 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)

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 supports POSIX signal handling as specified in IEEE 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.

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. IRIX supports the POSIX clock and timer facilities as specified in IEEE 1003.1b-1993, as well as the BSD itimer facility. The timer facilities are covered in *Topics in IRIX Programming* (see “Other Useful Books” on page xviii).

Hardware Cycle Counter

The hardware cycle counter is a high-precision hardware counter that is updated continuously. The precision of the cycle counter depends on the machine in use, but in most systems it is a 64-bit counter.

You sample the cycle counter by calling the POSIX function **clock_gettime()** specifying the CLOCK_SGI_CYCLE clock type.

The frequency with which the cycle counter is incremented also depends on the hardware system. You can learn the resolution of the clock by calling the POSIX function **clock_getres()**.

Note: The cycle counter is synchronyzed only to the CPU crystal and is not intended as a perfect time standard. If you use it to measure intervals between events, be aware that it can drift by as much as 100 microseconds per second, depending on the hardware system in 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

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

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

Message-Passing Interface (MPI)

The Message-Passing Interface (MPI) is a standard architecture and programming interface for designing distributed applications. Silicon Graphics, Inc. supports MPI in the POWERChallenge Array product. For details on MPI in Silicon Graphics systems, see the World-Wide Web page http://www.sgi.com/Products/PowerChallengeArray/TechInfo/MPI/. For the MPI standard, see http://www.mcs.anl.gov/mpi/index.html.

The performance of both sockets and MPI depends on the speed of the underlying network. The network that connects nodes (systems) in an Array product has a very high bandwidth.

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 115.

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 Origin200, Origin2000, and Challenge/Onyx systems support external interrupt lines for both incoming and outgoing external interrupts. Software support for these lines is described in the *IRIX Device Driver Programmer’s Guide* (see “Other Useful Books” on page xviii) and the ei(7) reference page. 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 55.)

Chapter 3

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 23 covers scheduling concepts, tells how to set nondegrading priorities, and explains affinity scheduling, gang scheduling, and deadline scheduling.
- "Minimizing Overhead Work" on page 28 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 35 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 IRIX supports 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 4.

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. (However, the CPUs in a multiprocessor are not necessarily synchronized. Different CPUs may take tick interrupts at a different times.)

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

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

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 34.

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. A typical process is usually blocked for I/O before it reaches 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. When a CPU needs a process to run, it normally takes the one with the highest priority.

When programming a priority you should never use a constant or absolute number. Instead, obtain the current, minimum, and maximum priority numbers using the POSIX function **sched_getparam()**, **sched_get_priority_min()**, and **sched_get_priority_max()**.

Understanding Affinity Scheduling

Affinity scheduling is a special scheduling discipline used in multiprocessor 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 52). 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 on an arbitrary scale.

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 14).

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, using the priority of the highest-priority process in the gang. 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 3-1

Example 3-1 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.

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 3-2.

Example 3-2 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 26). This will keep members of the gang executing concurrently longer.

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 3-3 (see also the sysmp(2) reference page).

Example 3-3 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;
}
```

Unavoidable Timer Interrupts

In machines based on the R4x00 CPU, even when the clock and fast timer duties are removed from a CPU, that CPU still gets an unwanted interrupt as a 5 microsecond “blip” every 80 seconds. Systems based on the R8000 and R10000 CPUs are not affected, and 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 from the VME bus 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*.

Note: Sprayed interrupts are not an issue with the Origin2000 family.

Assigning Interrupts to CPUs

To minimize the latency of real-time interrupts in the Challenge / Onyx, 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 with the specified level initiated on any 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 35).

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 29.

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 56). 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 cannot be directed to a different CPU.

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. You can exert three levels of increasing control: *restricted*, *isolated*, and *nonpreemptive*.

In general, the IRIX scheduling algorithms will run a process that is ready to run on any CPU. 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 group assignment made with *pset*. A restricted CPU remains part of a group, but does not perform any work you assign to the group 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 3-4.

Example 3-4 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 3-5 (see also the sysmp(2) reference page).

Example 3-5 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:

- The CPU assigned to handle the scheduling clock (“Assigning the Clock Processor” on page 28) must not be restricted.
- The REACT / Pro Frame Scheduler automatically restricts and isolates any CPU it uses. See Chapter 4.

Assigning Work to a Restricted CPU

After restricting 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 3-6 (see also the sysmp(2) reference page).

Example 3-6 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;
}
```

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 30, 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 24). 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, 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 3-7 (see the sysmp(2) reference page).

Example 3-7 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, interrupt response time is guaranteed not to exceed 200 microseconds in a properly configured system. The guarantee for Origin2000 and Onyx2 is the same (these systems generally achieve shorter response times in practice).

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 3-1 and discussed in the following topics.

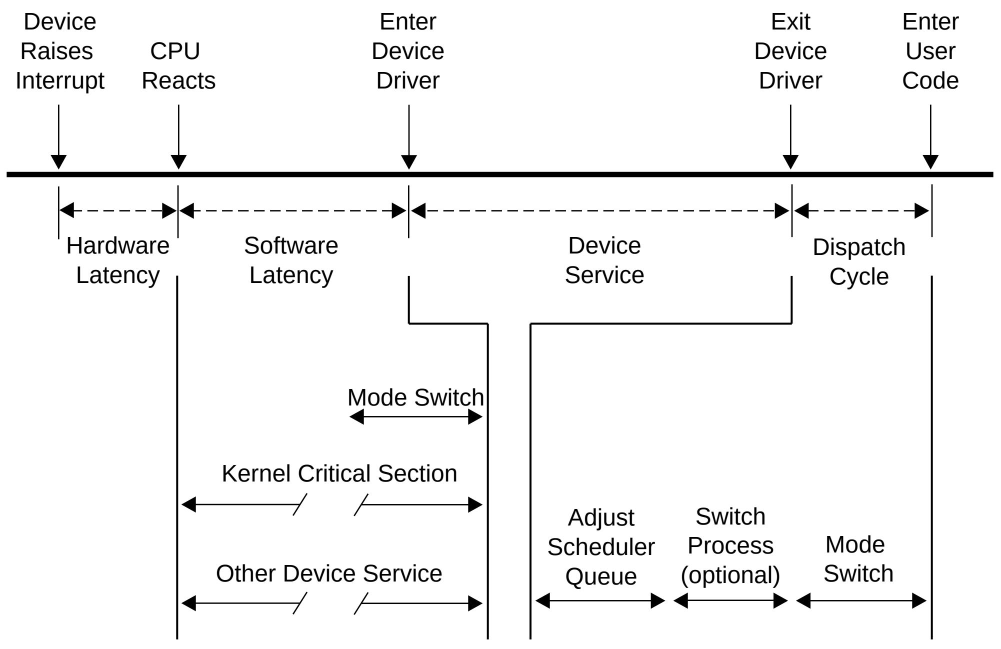

Figure 3-1 Components of Interrupt Response Time

Hardware Latency

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

The time taken for these events is the hardware latency, or interrupt propagation delay. In the Challenge/Onyx, the typical propagation delay is 2 microseconds. The worst-case delay can be much greater. The worst-case hardware latency can be significantly reduced by not placing high-bandwidth DMA devices such as graphics or HIPPI interfaces on the same hardware unit (POWERChannel-2 in the Challenge, module and hub chip in the Origin) used by the interrupting 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 30) and isolate it from TLB interrupts (see “Isolating a CPU From TLB Interrupts” on page 32)—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 “External Interrupts” on page 120).

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

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 29).

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 at a superior priority to the time-critical interrupts.

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 34), 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 50.)

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 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 115). A polling process can be dispatched one or more times per frame by the Frame Scheduler with low overhead.

Chapter 4

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 10.

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

- “Frame Scheduler Concepts” on page 44 details the operation and methods of the Frame Scheduler.
- “Selecting a Time Base” on page 55 covers the important choice of which source of interrupts should define a frame interval.
- “Using the Scheduling Disciplines” on page 57 explains the options for scheduling activities of different kinds.
- “Preparing the System” on page 61 reviews the system administration steps needed to prepare the CPUs that the Frame Scheduler will use.
- “Implementing a Single Frame Scheduler” on page 62 outlines the structure of an application that uses one CPU.
- “Implementing Synchronized Schedulers” on page 63 outlines the structure of an application that needs the power of multiple CPUs.
- “Handling Frame Scheduler Exceptions” on page 66 describes how overrun and underrun exceptions are dealt with.
- “Using Signals Under the Frame Scheduler” on page 70 discusses the issue of signal latency and the signals the Frame Scheduler generates.
- “Using Timers with the Frame Scheduler” on page 73 covers the use of itimers with the Frame Scheduler.
- “The Frame Scheduler Device Driver Interface” on page 74 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 32), 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 61). However, a Frame Scheduler CPU can be used to handle interrupts, although doing so runs a risk of causing overruns.

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 55.)

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*. The length of a major frame defines the application's true frame rate. The minor frames allow you to divide a major frame into sub-frames. Major and minor frames are depicted in Figure 4-1.

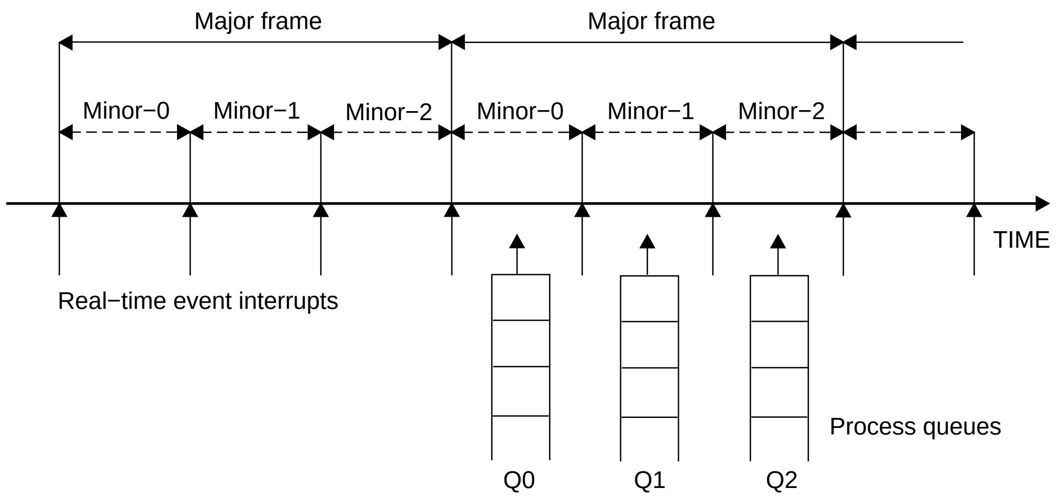

Figure 4-1 Major and Minor Frames

As pictured in Figure 4-1, the Frame Scheduler maintains a queue of processes for each minor frame. You enqueue each activity processes of your program to a specific minor frame. You determine the order of cyclic execution within a minor frame by the order in which you enqueue processes. You can:

- Enqueue multiple processes in one minor frame. They are run in queue sequence within the 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 4-1, 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 4-1, 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 57).

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 FRS Control Process

The process that creates a Frame Scheduler is called the *frs control* process. It is privileged in three respects:

- Its process ID (PID) is used to identify its Frame Scheduler in various functions.
- It can receive signals when errors are detected by the Frame Scheduler (see "Using Signals Under the Frame Scheduler" on page 70).
- 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. The API elements are declared in */usr/include/sys/frs.h*. The following are some important types are declared in */usr/include/sys/frs.h*:

typedef frs_fsched_info_t	A structure containing information about one scheduler, including its CPU number, time base, and number of minor frames. Used when creating a Frame Scheduler.
typedef frs_t	A structure containing an frs_fsched_info_t and the process ID of the frs control 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 Frame Scheduler and minor frame it uses and its scheduling discipline. Used when enqueuing a process.
typedef frs_recv_info_t	A structure containing error recovery options.

Library Interface for C Programs

The API library functions in */usr/lib/libfrs.so* are summarized in Table 4-1 on “Library Interface for C Programs” on page 47 for convenient reference.

Table 4-1 Frame Scheduler Operations

Operation	**Application Interface Options**

Create a Frame Scheduler	frs_t* **frs_create(**int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, pid_t *sync_master_pid*, int *num_slaves***);**
	frs_t* **frs_create_master(**int *cpu*, int *intr_source*, int *intr_qualifier*, int *n_minors*, int *num_slaves***);**
	frs_t* **frs_create_slave(**int *cpu*, frs_t* *sync_master_frs***);**

Enqueue an activity process to a Frame Scheduler	int **frs_enqueue(**frs_t* *frs*, pid_t *pid*, int *minor_index*, uint *discipline***);**
Join a Frame Scheduler (activity is ready to start)	int **frs_join(**frs_t* *frs***);**
Start scheduling (all activities enqueued)	int **frs_start(**frs_t* *frs***);**
Yield control after completing activity	int **frs_yield(**void**);**
Pause scheduling at end of minor frame	int **frs_stop(**frs_t* *frs***);**
Resume scheduling at next time-base interrupt	int **frs_resume(**frs_t* *frs***):**
Destroy a Frame Scheduler and send SIGKILL to its FRS control process	int **frs_destroy(**frs_t* *frs***);**

Interrogate a process queue	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 **frs_premove(**frs_t* *frs*, int *minor_index*, pid_t *remove_pid***);**
Reinsert a process in a queue, possibly changing discipline	int **frs_pinsert(**frs_t* *frs*, int *minor_index*, pid_t *insert_pid*, int *discipline*, pid_t *base_pid***);**
Retrieve error-recovery options	int **frs_getattr(** frs_t* *frs*, int *minor_index*, pid_t *pid*, frs_attr_t *att_index*, void* *options***);**
Set error-recovery options	int **frs_setattr(** frs_t* *frs*, int *minor_index*, pid_t *pid*, frs_attr_t *att_index*, void* *options***);**

System Call Interface for Fortran and Ada

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 because both languages have bindings for **schedctl()** (see the schedctl(2) reference page). The correspondence between the library functions and **schedctl()** calls is shown in Table 4-2.

Table 4-2 Frame Scheduler schedctl() Support

Library Function	Schedctl Syntax
frs_create()	int **schedctl(**MPTS_FRS_CREATE, frs_info_t* *frs_info***)**;
frs_enqueue()	int **schedctl(**MPTS_FRS_ENQUEUE, frs_queue_info_t* *frs_queue_info***)**;
frs_join()	int **schedctl(**MPTS_FRS_JOIN, pid_t *frs_master***)**;
frs_start()	int **schedctl(**MPTS_FRS_START, pid_t *frs_master***)**;
frs_yield()	int **schedctl(**MPTS_FRS_YIELD**)**;
f**rs_stop()**	int **schedctl(**MPTS_FRS_STOP, pid_t *frs_master***)**;
frs_resume()	int **schedctl(**MPTS_FRS_RESUME, pid_t *frs_master***)**;
frs_destroy()	int **schedctl(**MPTS_FRS_DESTROY, pid_t *frs_master***)**;
frs_getqueuelen()	int **schedctl(**MPTS_FRS_GETQUEUELEN, frs_queue_info_t* *frs_queue_info***)**;
frs_readqueue()	int **schedctl(**MPTS_FRS_READQUEUE, frs_queue_info_t* *frs_queue_info*, pid_t* *pidlist***)**;
frs_premove()	int **schedctl(**MPTS_FRS_PREMOVE, frs_queue_info_t* *frs_queue_info***)**;
frs_pinsert()	int **schedctl(**MPTS_FRS_PINSERT, frs_queue_info_t* *frs_queue_info*, pid_t **base_pid***)**;
frs_getattr()	int **schectl(**MPTS_FRS_GETATTR, frs_attr_info_t* *frs_attr_info***)**;
frs_setattr()	int **schectl(**MPTS_FRS_SETATTR, frs_attr_info_t* *frs_attr_info***)**;

Process Execution

An activity process that is enqueued to a Frame Scheduler has the basic structure shown in Example 4-1.

Example 4-1 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 begins (see “Starting Multiple Schedulers” on page 53). 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.

An activity process is never preempted. As long as it yields before the end of the frame, it can do its assigned work without interruption from other processes (it can be interrupted by hardware interrupts, if any hardware interrupts are allowed in that CPU). The Frame Scheduler preempts the process at the end of the minor frame.

Tip: Because an activity process cannot be preempted, it can often use global data without locks or semaphores. When the process that modifies a global variable is enqueued in a different minor frame from the processes that read the variable, 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 15) with high efficiency.

Tip: When a very short minor frame interval is used, it is possible for a process to have an overrun error in its first frame due to cache misses. A simple variation on the basic structure shown in Example 4-1 is to spend the first minor frame touching a set of important data structures in order to “warm up” the cache (see “Reducing Cache Misses” on page 52). This is sketched in Example 4-2.

Example 4-2 Alternate Skeleton of Activity Process

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

When an activity process is scheduled on more than one minor frame in a major frame, it can be designed to do nothing except warm the cache in the entire first major frame. To do this, the activity process function has to know how many minor frames it is scheduled on, and calls **frs_yield()** that many times in order to pass the first major frame.

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 are named in **frs_enqueue()** calls. (The queues can be reordered dynamically; see “Managing Activity Processes” on page 54.)

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 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 57).

At the end of a minor frame, the Frame Scheduler resets all *frs_run* 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.

Underrun and overrun exceptions are typically communicated via IRIX signals. The rules for sending these signals are covered under “Using Signals Under the Frame Scheduler” on page 70.

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 29 and “Assigning Interrupts to CPUs” on page 29), then the only competition for CPU cycles (other than a very few essential TLB interrupts) 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 35) and allow for it.

Using Multiple Synchronized Schedulers

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

You can create one master Frame Scheduler, 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 processes based on the same code, and enqueue each to a different CPU.) All synchronized Frame Schedulers use the same number of minor frames per major frame, which is taken from the definition of the master FRS.

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

Starting a Single Scheduler

A single Frame Scheduler comes into existence when the FRS control process calls **frs_create()**. Then the FRS controller calls **frs_enqueue()** one or more times to tell the new Frame Scheduler the PID values of the processes that it will schedule. The FRS controller calls **frs_start()** when it has enqueued all the processes. Each scheduled process must call **frs_join()** when it has initialized itself and is ready to be scheduled.

The Frame Scheduler requires the **frs_enqueue()** call for a given PID to precede the **frs_join()** call from the same PID. That is, an activity process cannot join the scheduler until the FRS controller has enqueued it—the **frs_join()** returns an error unless the calling process has been enqueued. After the Frame Scheduler receives the **frs_start()** call it waits until all enqueued processes have called **frs_join()**; then it begins the first minor frame.

Note: In version 1.0, 1.1, and 2.0 of REACT/Pro (the versions used with IRIX prior to version 6.2), the Frame Scheduler allowed a process to join prior to the enqueue. This flexibility was removed in version 3.0 (for IRIX 6.2) in order to simplify the implementation and to improve performance.

Starting Multiple Schedulers

A Frame Scheduler cannot start dispatching activities until

- the FRS controller has enqueued all the activity processes to their minor frames
- all the enqueued processes have done their own initial setup and have joined.

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

Each FRS controller 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 one or more **frs_enqueue()** calls, an appropriate number of **frs_join()** calls, and an **frs_start()** call. Each synchronized Frame Scheduler tells the master Frame Scheduler when it is ready. When all the schedulers are ready, the master Frame Scheduler gives the downbeat, and the first minor frame begins.

Pausing Frame Schedulers

Any Frame Scheduler can be made to pause and restart. Any process (typically but not necessarily the FRS controller) 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.

Note: If there is a process running Background discipline in the current minor frame, it will continue to execute until it yields or is blocked on a system service.

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. There is no way to restart all of a group of schedulers with the certainty that they start up on the same time-base interrupt.

Managing Activity Processes

The FRS control process creates the initial set of activity processes by calling **frs_enqueue()** prior to starting the Frame Scheduler. All the enqueued processes must call **frs_join()** before scheduling can begin. However, the FRS controller can change the set of activity processes dynamically while the Frame Scheduler is working, using the following functions:

frs_getqueuelen()	Get the number of processes currently in the queue for a specified minor frame.
f**rs_readqueue()**	Return the PID values of all queued processes for a specified minor frame as a vector of integers.
frs_premove()	Remove a process (specified by PID) from a minor frame queue.
frs_pinsert()	Insert a process (specified by PID and discipline) into a given position in a minor frame.

Using these functions, the FRS controller can change the queueing discipline of a process (by removing it and inserting it with a new discipline). The FRS controller can suspend a process by removing it from its queue; or can restart a process by putting it back in its queue.

Note: When an activity process is removed from the last or only queue it was in, it is returned to the normal IRIX scheduler and can begin to execute on some other CPU.When an activity process is removed from a queue, a signal may be sent to the removed process (see “Handling Signals in an Activity Process” on page 71). If a signal is sent to it, it will begin executing in its specified or default signal handler; otherwise, it will simply begin executing following **frs_yield()**. Once returned to the IRIX scheduler, a call to an FRS function such as **frs_yield()** returns an error (this also can be used to indicate the resumption of normal scheduling).

The FRS controller can also enqueue new processes that have not been scheduled before. The Frame Scheduler does not reject an **frs_pinsert()** call for a process that has not yet joined the scheduler. However, a process must call **frs_join()** before it can be scheduled.

If an enqueued process should be terminated for any reason, the Frame Scheduler removes the process from all queues in which it appears.

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 timer is not synchronized between processors, so it cannot be used to drive multiple synchronized schedulers. The on-chip 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 on-chip 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 19.

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 30). 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 120). 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 an external output jack to an external interrupt input jack on the same system, thus creating software-controlled external interrupts as an FRS time base. This would work in principle. However, if user process generating 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 74). 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: The use of software interrupts has a potential for causing a system deadlock if the interrupt-generating process contends for a resource that is also used by a frame-scheduled activity process. If any activity process calls IRIX system functions, the only way to be absolutely sure of avoiding deadlock is for the interrupt-generating process to avoid using any IRIX system functions. Note that C library functions such as **printf()** invoke system functions, and can lead to deadlocks in this case.

Using the Scheduling Disciplines

When an FRS control process 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 certain 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 60).

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 60).

When you specify Overrunnable+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 flags at the end of this minor frame (see “Scheduling Within a Minor Frame” on page 50).

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 CPUs 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 29).
3. Make sure that none of the real-time CPUs is managing the clock (see "Assigning the Clock Processor" on page 28). Normally the responsibility of handling 10ms scheduler interrupts is given to CPU 0.

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 146.)

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 shared by activity processes (see "Locking Pages in Memory" on page 49). (When **fork()** is used, each child process must lock its own address space.)
3. Create the Frame Scheduler using **frs_create_master()** (see Table 4-1 on "Library Interface for C Programs" on page 47).
4. Change the Frame Scheduler signals or exception policy, if desired (see "Setting Frame Scheduler Signals" on page 72 and "Setting Exception Policies" on page 67).
5. Create the activity processes using **sproc()** or **fork()** or, if they are independent processes, get them started and obtain their PID values.
6. Use **frs_enqueue()** to queue each activity process to the queue or queues on which it is to run.

   Each activity process independently uses **frs_join()** to let the Frame Scheduler know it is ready to start real-time execution. This call must follow the **frs_enqueue()** call for that process. The call blocks until scheduling begins, then returns to start the first frame dispatch of each activity process.
7. Set up signal handlers for signals from the Frame Scheduler (see "Using Signals Under the Frame Scheduler" on page 70). The handlers are set at this time, after creation of the activity processes, so that the activity processes do not inherit them.
8. Use **frs_start()** (Table 4-1) to enable scheduling.

   The Frame Scheduler begins scheduling processes as soon as all the activity processes have called **frs_join()**.

9. Wait for error and termination signals from the Frame Scheduler and for the termination of child processes.
10. Use **frs_destroy()** to terminate the Frame Scheduler.
11. 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.

Syncronized Scheduler Concepts

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

No single process may create more than one Frame Scheduler. This is because every Frame Scheduler must have a unique FRS control 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 control process for each sync-slave Frame Scheduler
- activity processes

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

Sync-slave schedulers can be stopped and restarted independently. However, when any scheduler, master or slave, is destroyed, all are immediately destroyed.

Synchronized Schedulers: the Sync-Master Process

A variety of program designs is possible but the simplest is possibly the set of processes described in the following paragraphs.

The master process executes first and performs these steps:

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 shared with lightweight processes.
3. Create the sync-master Frame Scheduler using the call **frs_create_master()**, and store its handle in a global location.
4. Create one FRS control 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 70).
7. Use **frs_start()** (Table 4-1) 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.

Synchronized Schedulers: Sync-Slave Processes

Each FRS control process for a synchronized scheduler (a sync-slave) will:

1. Create a synchronized Frame Scheduler using **frs_create_slave()**, specifying information about the master Frame Scheduler stored by the master process. The sync-master must exist. A sync-slave must specify the same time base and number of minor frames as the sync-master.
2. Change the Frame Scheduler signals or exception policy, if desired (see “Setting Frame Scheduler Signals” on page 72 and “Setting Exception Policies” on page 67).
3. Create the activity processes that will be scheduled by this synchronized Frame Scheduler, and use **frs_enqueue()** to enqueue them to their assigned minor frames.
4. Set up signal handlers for signals from the synchronized Frame Scheduler.
5. Use **frs_start()** to tell the synchronized Frame Scheduler that all activity processes have been enqueued.

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

For an example of this kind of program structure, refer to “Examples of Multiple Synchronized Schedulers” on page 147.

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.

Handling Frame Scheduler Exceptions

The FRS control 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 the Frame Scheduler should send. These policies have to be set before the scheduler is started. While the scheduler is running, the FRS controller 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 control 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 to the FRS controller
- Inject an additional minor frame
- Extend the frame by a specified number of microseconds
- Steal a specified number of microseconds from the following frame

The default action is to send a signal (the specific signals are listed under “Setting Frame Scheduler Signals” on page 72). The scheduler continues to run. The FRS control 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 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 55).

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 4-3 sets the policy of injecting a repeat frame. The caller specifies only the Frame Scheduler and the number of consecutive exceptions allowed.

Example 4-3 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 4-4 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 4-4 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 control 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 4-5 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 4-5 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 4-5 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.

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. In brief, an FRS sends signals to indicate that

- The FRS has been terminated
- Overrun or underrun have been detected
- A process has been dequeued

The rest of this topic details how to specify the signal numbers and how to handle the signals.

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 34), there are no clock tick interrupts, 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 Signals in the FRS Controller

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 control process.

Tip: 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 FRS control process for a synchronized Frame Scheduler should have handlers for Underrun and Overrun signals. The handler could report the error and issue **frs_destroy()** to shut down its scheduler. An FRS controller for a synchronized scheduler should use the default action for SIGHUP (Exit) so that completion of the **frs_destroy()** quietly terminates the FRS controller.

The FRS controller for the master (or only) Frame Scheduler should catch Underrun and Overrun exceptions, report them, and shut down its scheduler.

When an FRS is terminated with **frs_destroy()**, it sends SIGKILL to its FRS control process. This cannot be changed; and SIGKILL cannot be handled. Hence **frs_destroy()** is equivalent to termination for the FRS control process. (In the first release, the FRS sent SIGHUP, but this made deadlocks possible and had to be given up.)

Handling Signals in an Activity Process

A Frame Scheduler can send a signal to an activity process when the process is removed from any queue using **frs_premove()** (see “Managing Activity Processes” on page 54). The scheduler can also send a signal to an activity process when it is removed from the last or only minor frame to which it was enqueued (at which time a process is returned to normal IRIX scheduling).

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

Setting Frame Scheduler Signals

The frame scheduler sends signals to the FRS control process.

Note: In earlier versions of REACT / Pro, the Frame Scheduler sent these signals to *all* processes queued to that Frame Scheduler as well as the FRS controller. That is no longer the case. You can remove signal handlers for these signals from activity processes, if they exist.

The signal numbers used for most events can be modified. The signal numbers can be queried using **frs_getattr(**FRS_ATTR_SIGNALS**)** and changed using **frs_setattr(**FRS_ATTR_SIGNALS**)**, in each case passing an *frs_signal_info* structure. This structure contains room for four signal numbers, as shown in Table 4-3

Table 4-3 Signal Numbers Passed in frs_signal_info_t

Field Name	Signal Purpose	Default Signal Number
sig_underrun	Notify FRS controller of Underrun.	SIGUSR1
sig_overrun	Notify FRS controller of Overrun.	SIGUSR2
sig_dequeue	Notify an activity process that it has been dequeued with **frs_premove()**.	0 (do not send)
sig_unframesched	Notify an activity process that it has been removed from the last or only queue in which it was enqueued.	SIGRTMIN

Signal numbers must be changed before the Frame Scheduler is started. All the numbers must be specified to **frs_setattr()**, so the proper way to set any number is to first file the *frs_signal_info_t* using **frs_getattr()**. The function in Example 4-6 sets the signal numbers for Overrun and Underrun from its arguments.

Example 4-6 Function to Set Frame Scheduler Signals

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

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 70), so timer latency is unpredictable.

An FRS control process, because it is scheduled by IRIX, not the Frame Scheduler, can use interval timers.

Example 4-7 Minimal Activity Process as a Timer

```
frs_join(scheduler-handle)
do {
   usvsema(frs-controller-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 eight different device drivers can support the Frame Scheduler in any one system. The Frame Scheduler distinguishes device drivers by an ID number in the range 0 through 7 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 xviii). The generation of time-base signals can be added as a minor enhancement to a existing device driver.

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 in the *pfx***start()** or *pfx***init()** entry point, the driver calls a kernel function to specify its unique driver identifier between 0 and 7, and to register its *pfx***_frs_func_set()** and *pfx***_frs_func_clear()** functions. After this has been done, 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 real-time program creates a master Frame Scheduler and specifies this driver by its number as the source of interrupts (see "Device Driver Interrupt" on page 57). The Frame Scheduler calls the *pfx***_frs_func_set()** registered by this particular driver. This tells the driver that time signals are needed.
3. The device driver calls **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 *pfx***_frs_func_clear()** for the driver it is using. This tells the driver that time signals are no longer needed, and to cease calling **frs_handle_driverintr()** until it is once again initialized by a Frame Scheduler.

Device driver names, device driver structure, configuration files, and related topics are covered in the *IRIX Device Driver Programming Guide*.

Registering the Initialization and Termination Functions

A device driver must register two 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));
```

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 7. Different drivers in the same system must use different identifiers. A typical call resembles the code in Example 4-8.

Example 4-8 Exporting Device Driver Entry Points

```
/*
** Function called by the example driver to export
** its Frame Scheduler interface functions.
*/
frs_driver_export(3, example_frs_func_set, example_frs_func_clear);
```

Frame Scheduler Initialization Function

The device driver must provide a function with the following prototype:

`void` *pfx*`_frs_func_set ( intrgroup_t* intrgroup ) ;`

A skeleton of an initialization function for a Challenge / Onyx system running under IRIX 6.2 is shown in Example 4-9. The function is called by a new master Frame Scheduler—one 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 57). 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. 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 4-9 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 on the Challenge / Onyx need to call **vme_frs_install()** — do not call it on the Origin2000. As suggested by the code in Example 4-9, the arguments to **vme_frs_install()** can be taken from data supplied at boot time to the device driver's *pfx***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 *pfx***edtinit()** entry point is documented in the *IRIX Device Driver Programming Guide*.

Tip: 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 *prfx*_frs_func_clear (void) ;

A skeleton for this function is shown in Example 4-10. 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 4-10 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;
}
```

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 4-11. It delivers the interrupt to the Frame Scheduler on that CPU. The function to be invoked is

```
void frs_handle_driverintr(void);
```

Example 4-11 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 4-9 and cleared in Example 4-10.
   */
   if (FRS_is_active) frs_handle_driverintr();
   /*
   ** Step 3: any additional processing needed.
   */
   return;
}
```

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.

The call to **frs_handle_driverintr()** must be executed on a CPU controlled by the FRS that is using the driver. The only way to ensure this is to ensure that the hardware interrupt used by this driver is directed to that CPU. In IRIX 6.4 and later, you direct a hardware interrupt to a particular CPU by placing a DEVICE_ADMIN statement in the file */var/sysgen/system/irix.sm*. See comments in that file for the syntax.

Chapter 5

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 81 points out the uses of mapping a file into memory.
- "Asynchronous I/O" on page 82 describes the use of the asynchronous I/O feature of IRIX version 5.3 and later.
- "Synchronous Writing and Direct Writing" on page 95 documents the performance cost of knowing when disk output is complete.
- "Guaranteed-Rate I/O" on page 98 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 (for details on mapping files and other objects into memory, see the book *Topics in IRIX Programming*). A file that represents a data base of some kind—for example a file of scenery elements, or a file containing a precalculated table of operating parameters for simulated hardware—is best mapped into memory and accessed as a memory array. A mapped file of reasonable size can be locked into memory so that access to it is always fast.

You can also perform output on a memory-mapped file simply by storing into the memory image. When the mapped segment is also locked in memory, you control when the actual write takes place. Output happens only when the program calls **msync()** or changes the mapping of the file. At that time the modified pages are written. (See the msync(2) reference page.) 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, still operating under the identity of the calling process, 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 blocked process 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 still, 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 (see the fsync(2) and msync(2) reference pages). 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 95.)

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 (since version 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

Note: In IRIX 5.2 and IRIX 6.0, asynchronous I/O was implemented to conform to POSIX standard 1003.4 Draft 12, an earlier document. Support for the later POSIX standard1003.1b was implemented in IRIX 5.3. In releases following 5.3, support for POSIX 1003.1b-1993 is the only version of asynchronous I/O. It is no longer possible to compile programs that use the Draft-12 interface.

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(3)	Initiates asynchronous input from a file or device.
aio_write(3)	Initiates asynchronous output to a file or device.
lio_listio(3)	Initiates a list of operations to one or more files or devices.
aio_error(3)	Returns the status of an asynchronous operation.
aio_fsync(3)	Waits for all scheduled output for a file to complete.
aio_cancel(3)	Cancels pending, scheduled operations.

Each of these functions is described in detail in a reference page in 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. The important fields 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 94.
- 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 the *sigevent* is discussed under “Checking for Completion” on page 90.

Note: The IRIX 5.2 implementation 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 initialize asynchronous I/O simply by starting an operation with **aio_read()**, **lio_listio()**, or **aio_write()**. The first such call causes default initialization. This is the only form of initialization described by the POSIX standard. However, in a real-time program you often need to control at least 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 (*aio_threads*)

  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 (*aio_locks*)

  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 could execute concurrently. If in doubt, specify the number of subprocesses your program contains.

- the number of lightweight processes (sprocs) that will be sharing the use of asynchronous I/O (*aio_numusers*)

  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.

Other fields of the *aioinit_t* structure such as *aio_num* and *aio_usedba* 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 5-1 shows a subroutine to initialize asynchronous I/O, given counts of devices and calling processes.

Example 5-1 Initializing Asynchronous I/O

```
int initAIO(int numDevs, int numSprocs, int maxOps)
{
   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;
   if (maxOps) /* we do know max aiocbs at 1 time */
      A.aio_num = maxOps;
   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 30)—or create the Frame Schedulers (see “Starting Multiple Schedulers” on page 53).
- Assign real-time processes to their CPUs.

The asynchronous I/O processes created by **aioinit()** 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 queued.

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.

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 121.

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 (either 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.

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 some time. That depends on the file or device and 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 (see the aio_fsync(3) reference page). 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. The synchronize operation can take considerable time, depending on how much output data has been buffered. Its completion 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 121 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 121.

Polling for Status

You can check the progress of any asynchronous operation (including **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 function **inWait0()** under “Asynchronous I/O Example” on page 121. This function is used when the aiocb is initialized with SIGEV_NONE, meaning that no notification is to be returned at the completion of the operation. The function waits for an asynchronous operation to complete using a loop in the general form shown in Example 5-2.

Example 5-2 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 5-2)
- 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 121.

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 17).

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 15). Locks have very low overhead in cases such as this, where there 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 Example 5-3 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 5-3 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 5-3 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 (see the open(2) reference page).

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. If you combine O_SYNC access 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 I/O

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 transfer 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 **read()** or **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 5-1 was collected on a 4-processor Challenge system under IRIX 5.3, 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)

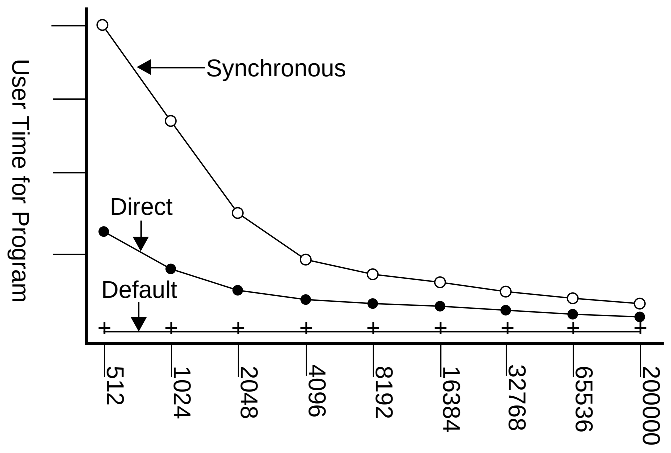

Figure 5-1 Effect of Blocksize on write() Performance

The values in Table 5-1reflect the total execution time for one run of the program, as reported by the *time* command (see the time(1) reference page).

Table 5-1 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 5-1, 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.
- A guaranteed-rate stream must be available. Unless extra-cost options are installed, a maximum of four streams can be in use at one time.

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 Admin: Disks and Filesystems* manual (see “Other Useful Books” on page xviii). For an example of how the **grio_request()** function is used, see the function starting in “Guaranteed-Rate Request” on page 140.

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 5-4.

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 I/O” on page 96).
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 102).

Example 5-4 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_XFLAG_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(3X) 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_remove_request(3X) 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 a 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_remove_request()** must be called from the same process ID that called **grio_request()**.
- 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 6

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

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 Programmer's Guide* (see "Other Useful Books" on page xviii).

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. Conventionally, every physical 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.

How Devices Are Defined

In IRIX 6.4, the */dev* filesystem still exists to support programs and shell scripts that depend on conventional names such as */dev/tty*. However, the true representation of all devices is built in a different file system rooted at */hw* (for hardware). You can explore the */hw* filesystem using standard commands such as *file*, *ls*, and *cd*. You will find that the conventional names in */dev* are implemented as links to device special files in */hw*. The creation and use of /hw, and the definition of devices in it, is described in detail in the *IRIX Device Driver Programmer's Guide*.

How Devices Are Used

To use a device, a process opens the device special file by passing the file 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 109) 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 sample, 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.

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 111.
- If it is a VME device with bus master (on-board DMA) capability, you should receive an IRIX device driver from the OEM. Consult *IRIX Admin: System Configuration and Operation* 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 109.

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 (see "Other Useful Books" on page xviii).

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 supported disk and tape devices. Other SCSI devices are controlled through a generic device driver that must be extended with programming for a specific device.

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. If you write a SCSI driver, it will control the device indirectly, by calling a host adapter driver.

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. The use of host adapter drivers is documented in the *IRIX Device Driver Programmer's Guide*.

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*, *dd*, and *mt* (see the tar(1), dd(1M) 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 83). You need to take pains to ensure that asynchronous operations 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. It is also described in detail in the *IRIX Device Driver Programmer's Guide*. 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.

For more on dlsib, see the *IRIX Device Driver Programmer's Guide*.

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(3dm) reference page (installed with the dmedia_dev package).

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(3dm) reference page(installed with the dmedia_dev package) .

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 6-1.

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.

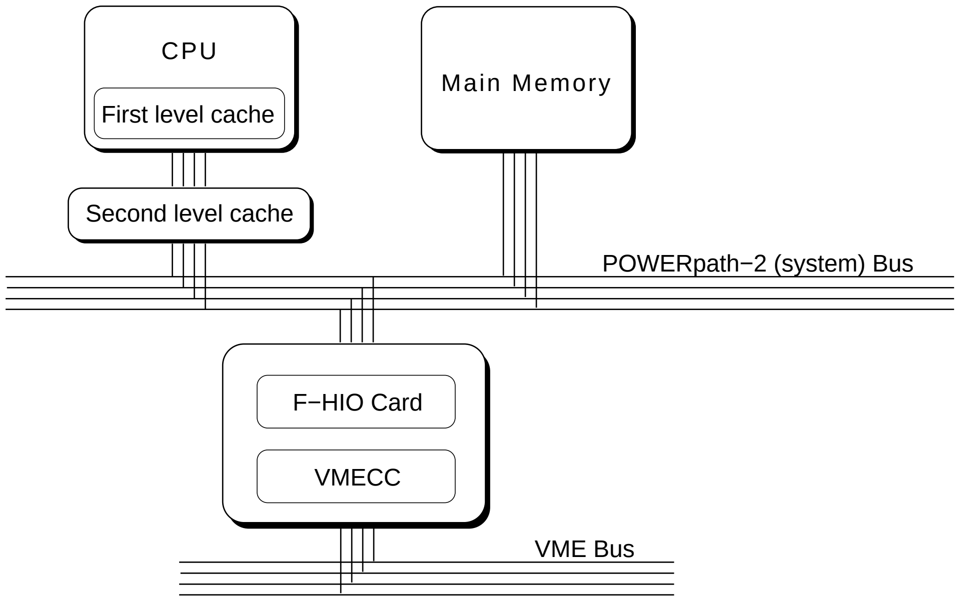

Figure 6-1 Multiprocessor CHALLENGE Data Path Components

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 6-1.

Table 6-1 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 6-2.

Table 6-2 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.

Note: VME addressing issues are discussed in greater detail from the standpoint of the device driver, in the *IRIX Device Driver Programmer's Guide*.

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 kernel and the VME device driver program registers in the VMECC to recognize fetches and stores to specific 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 12 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 12 unique mappings of at most 8 MB, or a single mapping of 96 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*.

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 (The use of PIO is covered in greater detail in the *IRIX Device Driver Programmer's Guide*. Memory mapping of I/O devices and other objects is covered in the book *Topics in IRIX Programming*.)

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 6-3.

Table 6-3 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 117.

User-Level Interrupt Handling

When a VME device that you control with PIO can generate interrupts, you can arrange to trap the interrupts in your own program. In this way you can program the device for some lengthy operation using PIO output to its registers, and then wait until the device returns an interrupt to say the operation is complete.

The programming details on user-level interrupts are covered in the *IRIX Device Driver Programmer's Guide*.

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 114.

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 6-4. Many factors can affect the performance of DMA, including the characteristics of the device.

Table 6-4 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 6-4.

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.

For more details of user DMA, see the *IRIX Device Driver Programmer's Guide*.

The typical performance of the DMA engine for D32 transfers is summarized in Table 6-5. 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 6-5 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

Table 6-5 VME Bus Bandwidth, DMA Engine, D32 Transfer

Transfer Size	Read	Write	Block Read	Block Write
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

Some of the factors that affect the performance of user DMA include

- The response time of the VME board to bus read and write requests
- The size of the data block transferred (as shown in Table 6-5)
- Overhead and delays in setting up each transfer

The numbers in Table 6-5 were achieved by a program that called **dma_start()** in a tight loop, in other words, with minimal overhead.

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. However, it waits in a test loop, polling the hardware until the operation is complete. As you can infer from Table 6-5, 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 24) 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 Origin200, Origin2000, and Challenge / Onyx hardware includes support for generating and receiving external interrupt signals. 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.)

For programming details of the external interrupt lines, see the *IRIX Device Driver Programmer's Guide*. You can also trap external interrupts with a user-level interrupt handler (see “User-Level Interrupt Handling” on page 116); this is also covered in the *IRIX Device Driver Programmer's Guide*.

Sample Programs

The programs in this appendix illustrate the use of some of the features discussed in the book. The following programs are included:

- "Asynchronous I/O Example" on page 121 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 140 demonstrates how to request a guaranteed rate of I/O transfer.
- "Frame Scheduler Examples" on page 143 describes the sample programs distributed with the REACT/Pro Frame Scheduler.

Asynchronous I/O Example

The program in Example A-1 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—work that does not normally require asynchronous I/O. However, this test program 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 copy the input files one at a time, using subroutine calls
- to 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 perform the following sequence of actions:

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. Then it waits on a semaphore that is posted from the signal handler function.
- **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.

Example A-1 Asynchronous I/O Example Program

```
/* ========================================================================
||  aiocat.c : This highly artificial example demonstrates 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. Each input file is
|| read in BLOCKSIZE units. The output file contains the data from
|| the input files in the order they were specified. Thus the
|| output should be the same as "cat infilename... >outfile".
||
|| When DO_SPROCS is compiled true, 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 _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  /* max sprocs: anything from 4 to 20 or so */
#define DO_SPROCS 1     /* 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, the 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 by 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 if ('-' == argv[argno][0])
    { /* is unknown -option */
        fprintf(stderr,"aiocat [-o outfile] [-a 0|1|2|3] infiles...\n");
        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 there is no need to reposition
                || because all aio_read calls have an implied lseek.
                || 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 = {0}; /* all fields initially zero */
        /*
        || Go with the default 5 for the number of aio-created procs,
        || as 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;
        /*
        || Warn aio of the number of user procs that will be
        || using its arena.
        */
        ainit.aio_numusers = nprocs;
        aio_sgi_init(&ainit);
    }
    /*
    || Process each input file, either in a child process or in
    || a subroutine call, as specified by the DO_SPROCS variable.
    */
    for (argno = 0; argno < nfiles; ++argno)
    {
        pc = &array[argno];
#if DO_SPROCS
#define CHILD_STACK 64*1024
        /*
        || For each input file, start a child process as an instance
        || of the selected method (-a argument).
        || If an error occurs, quit. That will send a SIGHUP to any
        || already-started child, which will kill it, too.
        */
        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
        /*
        || For each input file, call the selected (-a) method as a
        || subroutine to copy its file.
        */
        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.
    || Each child ensures that all its output is complete before it
    || invokes barrier().
    */
    barrier(convene,nprocs);
#endif
```

```
    /*
    || Close the output file and print some statistics.
    */
    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 polling with aio_error() with sginap(). 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 function to function.
*/
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;  /* unable to schedule a read */
        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.
        */
        if (!ret)
        {
            if (!(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, above.
    */

    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 declares 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 -- else cc gives a warning.  The cast
        || in the following statement is only to avoid this 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.
        */
        if (!ret)
        {
            if (!(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 number */
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->si_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 it 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.  In a more complicated
    || design, if a signal could arrive from more than one pending
    || operation, this function could not return status.
    */
    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;  /* could not start read, or it ended badly */
        /*
        || 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.
    */
    if (!ret)
    {
        if (!(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 polling the semaphore.  The callback
    || function might be entered before we reach this point, or after.
    */
    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.
    */
    if (!ret)
    {
        if (!(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*/
```

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 example programs distributed with the Frame Scheduler are found in the directory */usr/react/src/examples*. They are summarized in Table i and are discussed in more detail in the topics that follow.

Table i Summary of Frame Scheduler Example Programs

Directory	Features of Example
simple *r4k_intr*	Two processes scheduled on a single CPU at a frame rate slow enough to permit use of **printf()** for debugging. The examples differ in the time base used; and the *r4k_intr* code uses a barrier for synchronization.
mprogs	Like *simple,* but the scheduled processes are independent programs.
multi *ext_intr* *user_intr* *vsync_intr*	Three synchronous Frame Schedulers running lightweight processes on three processors. These examples are much alike, differing mainly in the source of the time base interrupt.
complete *stop_resume*	Like *multi* in starting three Frame Schedulers. Information about the activity processes is stored in arrays for convenient maintenance. The *stop_resume* code demonstrates **frs_stop()** and **frs_resume()** calls.
driver *dintr*	*driver* contains a pseudo-device driver that demonstrates the Frame Scheduler device driver interface. *dintr* contains a program based on *simple* that uses the example driver as a time base.
sixtyhz *memlock*	One process scheduled at a 60 Hz frame rate. The activity process in the *memlock* example locks its address space into memory before it joins the scheduler.
upreuse	Complex example that demonstrates the creation of a pool of reusable processes, and how they can be dispatched as activity processes on a Frame Scheduler.

Basic Example

The example in */usr/react/src/examples/simple* shows how to create a simple application using the Frame Scheduler API. The code in */usr/react/src/examples/r4kintr* is similar.

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 */usr/react/src/examples/mprogs* does the same work as example *simple* (see “Basic Example” on page 144). However, the activity processes A and B are physically loaded as separate commands. The main program establishes the single Frame Scheduler. The activity 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 activity processes. However, since they are started as separate programs, the master has no direct way of knowing their process IDs, which are needed for **frs_enqueue()**.
- The activity processes need to specify upon which minor frames they should be enqueued, and with what discipline.
- The master needs to enqueue the activities 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 activity processes 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 activity processes so they can terminate in an orderly way.

There are many ways in which these objectives could be met (for example, the three programs could share a shared-memory arena). In this example, the master and subordinates communicate using a simple protocol of messages exchanged using

msgget() and **msgput()** (see the msgget(2) and msgput(2) reference pages). 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.

Examples of Multiple Synchronized Schedulers

The example in */usr/react/src/examples/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 address of the Frame Scheduler it is to join, and the address of the “real-time” action it is to execute. In the sample code, all real-time actions are empty function bodies (feel free to load them down with code).

The examples in */usr/react/src/examples/ext_intr*, *user_intr*, and *vsync_intr* are all similar to multi, differing mainly in the time base used. The examples in *complete* and *stop_resume* are similar in operation, but more evolved and complex in the way they manage subprocesses.

Tip: It is helpful to use the *xdiff* program when comparing these similar programs—see the xdiff(1) reference page.

Example of Device Driver

The code in */usr/react/src/examples/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 79). This allows you to test the driver. Calling its **ioctl()** entry is equivalent to using **frs_usrintr()** (see “The Frame Scheduler API” on page 46).

The code in */usr/react/src/examples/dintr* contains a variant of the simple example that uses a device driver as the time base. The program *dintr/sendintr.c* opens the driver, calls **ioctl()** to send one time-base interrupt, and closes the driver. (It could easily be extended to send a specified number of interrupts, or to send an interrupt each time the return key is pressed.)

Examples of a 60 Hz Frame Rate

The example in directory */usr/react/src/examples/sixtyhz* demonstrates the ability to schedule a process at a frame rate of 60 Hz, a common 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.

This example also contains the code to query and to change the signal numbers used by the Frame Scheduler.

The example in */usr/react/src/examples/memlock* is similar to the sixtyhz example, but the activity process uses **plock()** to lock its address space. 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.)

Example of Managing Lightweight Processes

The code in */usr/react/src/examples/upreuse* implements a simulated real-time application based on a pool of reusable processes. A reusable process is created with **sproc()** and described by a *pdesc_t* structure. Code in *pqueue.c* builds and maintains a pool of processes. Code in *pdesc.c* provides functions to get and release a process, and to dispatch one to execute a specific function.

The code in *test_hello.c* creates a pool of processes and dispatches each one in turn to display a message. The code in *test_singlefrs.c* creates a pool of processes and causes them to join a Frame Scheduler.

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 of 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, and it must have enough memory to buffer 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. When all have arrived, 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 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 Programmer's Guide*, part number 007-0911-060.

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) and install(1) reference pages) 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 program involvement. 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 control process

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 control to one scheduler.

gang scheduling

A process scheduling discipline supported by IRIX. The processes of a *share 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 a shared resource. A process that wants to use the resource requests the lock that (by agreement) stands for that resource. The process releases the lock when it is finished using the resource. See *semaphore*.

major frame

The basic frame rate of a program running under the Frame Scheduler.

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 process ID, user and group IDs, working directory, open file handles, scheduler class, environment variables, and so on. See the fork(2) reference page for a list.

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* under the heading RSS. See *virtual size* and the ps(1) reference page.

scheduling discipine

The rules under which an activity process is dispatched by a Frame Scheduler, including whether or not the process is allowed to cause overrun or underrun exceptions.

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 in rotation. 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 specific 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. You can isolate a CPU from these unpredictable interrupts, under certain conditions.

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 and Crimson 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* under the heading SZ. The sum of all virtual sizes cannot exceed the size of the swap space. See *resident set size* and the ps(1) reference page.

virtual address space

The set of numbers that a process can validly use as memory addresses.

Index

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 007-2499-003.

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: 650-965-0964
- 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