Chapter 3

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

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

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

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

Kernel Facilities for Real-Time Programs

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

Kernel Optimizations

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

- Instruction paths to system calls and traps are optimized, including some hand coding, to maximize cache utilization.
- In the real-time dispatch class (described further in "Using Priorities and Scheduling Queues" on page 67), the run queue is kept in priority-sorted order for fast dispatching.
- Floating point registers are saved only if the next process needs them, and restored only if saved.
- Paging I/O is prioritized with the process priority.
- The kernel tries to redispatch a process on the same CPU where it most recently ran, in hopes of finding some of its data remaining in cache (see "Understanding Affinity Scheduling" on page 72).

Special Scheduling Disciplines

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

Nondegrading Priorities

A real-time process needs an unchanging priority. The kernel allows you to apply a nondegrading priority to a specified process. When this priority is in the range of real-time priorities (smaller than normal priorities), the process is scheduled from a real-time scheduling queue, which is tested before the normal dispatch queue. For more information, see "Setting a Nondegrading Batch Priority" on page 70 and "Setting a Nondegrading Real-Time Priority" on page 71.)

Deadline Scheduling

The kernel also supports a deadline scheduling discipline. Under deadline scheduling, a process can request a certain amount of processing time in every interval of a specified length—for example, 30 milliseconds in every 100 milliseconds. For more information, see "Using Deadline Scheduling" on page 74.

Gang Scheduling

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

For more information, see "Using Gang Scheduling" on page 73.

Processor Sets

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

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

The use of kernel scheduling queues, priorities, and processor sets is covered in more detail in Chapter 6, "Controlling CPU Workload,". When a real-time application requires only a fraction of the system's power, these tools may be sufficient to ensure the needed performance. For more critical applications, you need to replace the kernel scheduler with the Frame Scheduler (see "REACT/Pro Frame Scheduler" on page 25).

Locking Virtual Memory

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

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

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

Mapping Processes and CPUs

Normally IRIX tries to keep all CPUs busy, dispatching the next ready process to the next available CPU. (This simple picture is complicated by the needs of affinity scheduling, deadline scheduling, and gang scheduling). Since the number of ready processes changes all the time, dispatching is a random process. A process cannot predict how often or when it will next be able to run. For normal programs this does not matter, as long as each process continues to run at a satisfactory average rate.

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

- Restrict one or more CPUs from normal scheduling, so that they can run only the processes that are specifically assigned to them.
- Assign one or more processes to run on the restricted CPUs.

A process on a dedicated CPU runs when it needs to run, delayed only by interrupt service and by kernel scheduling cycles (if scheduling is enabled on that CPU). For details, see "Assigning Work to a Restricted CPU" on page 83. The REACT/Pro Frame Scheduler takes care of both steps automatically; see "REACT/Pro Frame Scheduler" on page 25.

Controlling Interrupt Distribution

In normal operations, CPUs receive frequent interrupts:

- I/O interrupts are "sprayed" to different CPUs to equalize workload.
- A scheduling clock causes an interrupt to every CPU every time-slice interval of 10 milliseconds.
- Whenever interval timers are in use ("Timers and Clocks" on page 38), 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 79. 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 42)
- a device interrupt from a specially-modified device driver
- a software call (normally used for debugging)

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

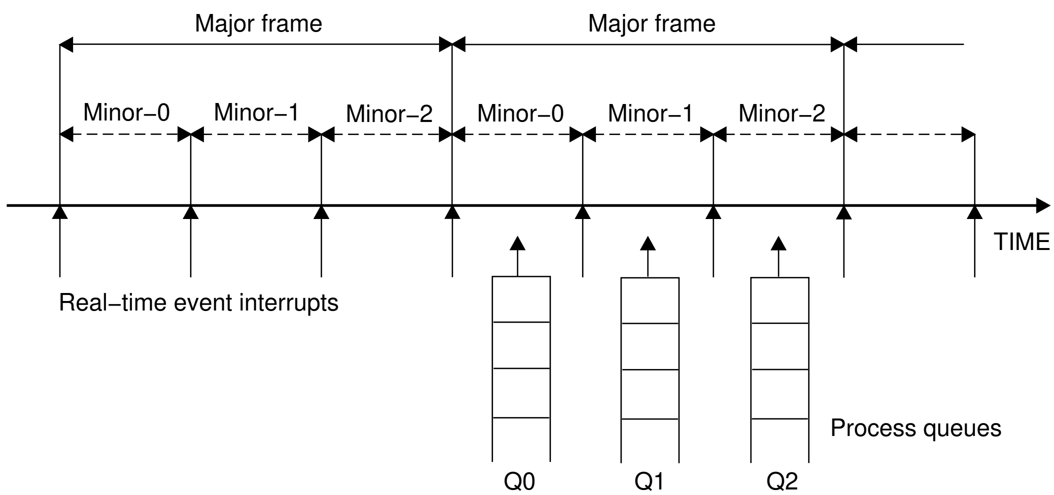


Figure 3-1 Major and Minor Frames

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

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

In other cases, you have 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 29).

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 (see "Barriers" on page 34).
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 7, “Using the Frame Scheduler”. Sample programs that illustrate the Frame Scheduler are described under “Frame Scheduler Examples” on page 225.

Interprocess Communication

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

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

Shared Memory Segments

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

IRIX Shared Memory Arenas

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

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

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

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

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

SVR4-Compatible Shared Memory

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

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

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

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

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

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

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

Semaphores

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

A process that needs to use the resource executes a “P” operation on the semaphore. This operation tests and decrements the count in the semaphore. If the count is 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*viv*es it.

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

IRIX Semaphores

IRIX supports a set of semaphore operations designed for low-overhead coordination between multiple concurrent processes. You create these semaphores within a shared arena created with **usinit()** (see "IRIX Shared Memory Arenas" on page 29). The **usnewsema()** call creates a semaphore. You specify the arena handle and the initial value for the semaphore (that is, the count of resources that it represents, typically 1).

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

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

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

For more information on semaphores, refer to *Topics in IRIX Programming*, and to the usnewsema(3), usnewpollsema(3), uspsems(3), usvsema(3), and poll(2) reference pages. The sample program shown in "Interprocess Communication" on page 186 uses IRIX semaphores, and demonstrates the use of metering information.

SVR4-Compatible Semaphores

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

Like a shared memory segment (see "SVR4-Compatible Shared Memory" on page 30), a set of semaphores is somewhat like a file in that it

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

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

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

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

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

Locks

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

A lock is functionally the same as a semaphore with a count of 1. The set-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.

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

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

For more information on locks, refer to *Topics in IRIX Programming*, and to the usnewlock(3), ussetlock(3) and usunsetlock(3) reference pages. See also the sample code of "Interprocess Communication" in Appendix A.

Barriers

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

A barrier is created by **newbarrier()** in an arena built by **usinit()**. The barrier is used by some fixed number (*N*) of processes. When each process is ready to rendezvous with the others, it issues **barrier**(*N*). As each process arrives at the barrier, it is suspended. When the *N*th process calls **barrier()**, all the processes resume execution.

A barrier is the computing equivalent of *N* coworkers who agree to go to lunch together. When each person realizes it is lunch time, he or she goes to the lobby. When the *N*th coworker reaches the lobby, all of them depart together for lunch. A barrier is very useful in initializing an application based on the Frame Scheduler (see "Preparing the System" on page 113).

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

The logic of the second, helper, process is as follows:

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

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

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

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

Mutual Exclusion Primitives

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

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

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

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

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

The Load Linked and Store Conditional instructions only operate on memory locations that can be cached. Uncached pages (for example, pages implemented as reflective shared memory, see “Reflective Shared Memory” on page 42) cannot be set by the **test_and_set()** functions.

Signals

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

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

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

Table 3-1 Signal Handling Interfaces

Function	SVR4-compatible Calls	BSD 4.2 Calls	POSIX Calls
set and 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 5.3 supports POSIX signal handling as specified in document 1003.1b-1993. This includes FIFO queueing new signals when a signal type is held, up to a system maximum of queued signals. (The maximum can be adjusted using *systune*; see the systune(1) reference page.)

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

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 122.) 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 122.

Timers and Clocks

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

Timer Interrupts (Itimers)

IRIX supports the BSD UNIX feature of interval timers or “itimers,” and part of the POSIX timer definition.

BSD Itimers

An itimer is a request to have a signal sent at the expiration of a specified interval. In order to use an itimer, you establish a signal handler, then issue the **setitimer()** call. The timer can be a one-shot or it can repeat at a regular interval (see the setitimer(2) reference page).

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

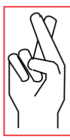
Table 3-2 Types of itimer

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

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

Timers and the resolution of the real-time timer are discussed further in Chapter 5, "Managing Time and Time Intervals." Sample code that sets up an itimer can be found under "Interprocess Communication" in Appendix A.

Note: Interval timers are usually not necessary, and should not be used, under the Frame Scheduler. See "Using Timers with the Frame Scheduler" on page 125.



In the release after 6.2 we have a better story for this topic.

POSIX Timers

The POSIX real-time standard 1003.1b-1993 specifies several timer-related functions which it is the intention of Silicon Graphics to support. However, in release 6.2 of IRIX, only the **nanosleep()** function is implemented (see the nanosleep(2) reference page).

IRIX also supports the POSIX-defined functions **alarm()** and **sleep()**. However, since these functions deal with intervals of seconds, they are of less interest to real-time programmers (see alarm(2) and sleep(2) reference pages).

The POSIX functions comparable to **setitimer()**, such as **timer_settime()**, will be implemented in a future release.

Timestamps

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

Time of Day Timestamp

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

The time of day timestamp is discussed further in Chapter 5, "Managing Time and Time Intervals." The sample program under "Getting the Time of Day Stamp" on page 184 tests the time-of-day clock to find out its true precision.

Hardware Cycle Counter

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

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

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

You sample the cycle counter by mapping it into the process's address space, then reading it as if it were a memory variable. The method is covered in Chapter 5, "Managing Time and Time Intervals." The sample program under "Mapping and Reading the Cycle Counter" on page 175 also demonstrates its use.

Interchassis Communication

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

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

Socket Programming

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 xxiii.

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 168.

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

External Interrupts

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

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

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

For details of the use and programming of external interrupts, see the *IRIX Device Driver Programmer's Guide*, and see 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 107.)