Chapter 5

Managing Time and Time Intervals

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

- "Using Interval Timers" on page 55 discusses the use of timers, which interrupt a program when a certain amount of time has elapsed.
- "Using Timestamps" on page 63 discusses the ways of getting the current time in order to record the moment when something happens.

Using Interval Timers

IRIX supports a variety of programming interfaces to interval timer services. However, the timers are implemented using a single underlying mechanism.

Timed Pauses

In many instances a program, or a process within a multiprocess program, needs to suspend execution for a period of time. IRIX contains a variety of functions that provide this capability. The functions differ in their precision and in their portability. Table 5-1 contains a summary.

Table 5-1 Functions for Timed Suspensions

Reference Page	Precision	Portability	Operation
sleep(3C)	second	POSIX	Suspend for a number of seconds or until a signal arrives.

Table 5-1 Functions for Timed Suspensions

Reference Page	Precision	Portability	Operation
usleep(3C)	microsecond	SGI	Suspend for a number of microseconds or until a signal arrives.
nanosleep(2)	nanosecond	POSIX	Suspend for a number of seconds and nanoseconds or until a signal arrives.

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

POSIX Timer Support

POSIX standard 1003.1b-1993 defines several functions related to interval timers. Of these, IRIX 6.2 supports only the **alarm()** function. Using this function you can cause a signal of type SIGALRM to be sent to the calling process after a specified number of seconds has elapsed. The remaining POSIX timer functions allow intervals to be specified in units of nanoseconds, for repeating intervals, and for signals at absolute times. These functions will be implemented in a release of IRIX following 6.2.

BSD Timer Support

As described under “Timer Interrupts (Itimers)” on page 38, IRIX supports the BSD UNIX functions for interval timing, which defines three kinds of software interval timers. All three are used the same way:

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

The three kinds of timers differ in the time base they use, the precision with which you can specify the intervals, and in the signals they send, as summarized in Table 5-2.

Table 5-2 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

Of the three, only the ITIMER_REAL, which measures elapsed time, is useful for real-time processes. The signal it generates, SIGALRM, is always delivered as soon as possible. (Other signals are not delivered until a scheduling interval occurs; see "Signal Delivery and Latency" on page 122.)

Note: Interval timers are not normally used with the Frame Scheduler. See "Using Timers with the Frame Scheduler" on page 125

Using an Itimer

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

Example 5-1 Timer Initialization

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

```
    period.it_value.tv_usec = periodInMicrosec;
    if (setitimer(ITIMER_REAL, &period, NULL))

    {
        perror("setitimer");
        return -1;
    }
    return 0; /* indicate success */
}
```

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

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

Time Data Structures

The include files *time.h* and *sys/time.h* define several data types and data structures related to time with confusing relationships between them. Some features of these structures are summarized in Table 5-3.

Table 5-3 Time Data Structure Usage

Data Type	Declared In	Contains	Some Functions Using this Type
time_t	*time.h*	long int with time in seconds since 00:00:00 UTC, January 1, 1970	time(2), ctime(3C), cftime(3C), difftime(3C)
timeval	*sys/time.h*	*time_t* of seconds, long of microseonds	adjtime(2), getitimer(2), getrusage(3C), gettimeofday(3C), select(2), utimes(3B), utmpx(4)
itimerval	*sys/time.h*	two *timeval* fields for first interval and repeat interval	getitimer(2) and setitimer(2)

Table 5-3 (continued) Time Data Structure Usage

Data Type	Declared In	Contains	Some Functions Using this Type
timespec_t	*time.h*	*time_t* of seconds, long of nanoseconds	clock_gettime(2), nanosleep(2), aio_suspend(3), sigtimedwait(3)
itimerspec	*time.h*	two *timespec* fields for first interval and repeat interval	(none at this time)
tm	*time.h*	int fields for seconds, minutes, hours, day, month, year, etc.	localtime(2), gmtime(2), strftime(3C)

Time Signal Latency

It takes time for the kernel to deliver the SIGALRM that notifies your program at the end of the interval. (The issue of signal latency in general is discussed under "Signal Delivery and Latency" on page 122.) The signal latency is less for SIGALRM than for other signals, since the kernel initiates a scheduling cycle immediately after the timer interrupt, without waiting for the end of a fixed time slice. When the program is running or ready to run, in a CPU that has been restricted and isolated (as discussed in Chapter 6), the latency is fairly short and consistent from one signal to the next. (Even so, it is not advisable to use a repeating itimer as the time base for a real-time program). Under less favorable conditions, signal latency can be variable and sometimes lengthy (tens of milliseconds) relative to a fast timer frequency.

How Timers Are Managed

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

- Some systems have no hardware support for interval timers, so the kernel has to rely on frequent, periodic interrupts as a time base.

  In these systems, the precision of timer interrupts is controlled by a tuning paramater, the *fasthz* variable.

- In the Challenge/Onyx and POWER-Challenge architecture, each CPU has a clock comparator that the kernel can program to cause an interrupt after a specific interval has elapsed.

In these systems, timer interrupts have sub-microsecond precision.

Timer Management in Challenge, Onyx, and POWER-Challenge

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

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

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

Timer Management Without a Clock Comparator

In all uniprocessor systems and in the Crimson series (the only multiprocessor systems supported by IRIX 6.2 that lack a hardware clock comparator) the kernel manages interval timers using a periodic interrupt, as follows.

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

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

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

Using Short Timer Intervals

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

A process running at a nondegrading, real-time priority (see "Setting a Nondegrading Real-Time Priority" on page 71) can make use of an interval that is shorter than the *HZ* frequency (10 milliseconds). The actual interval that elapses is different in different versions of IRIX.

Fast Timers With a Clock Comparator

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

Fast Timers Without a Clock Comparator

In uniprocessors and Crimson systems, the minimum effective timer resolution for real-time processes is set by the *fasthz* system tuning parameter. Any timer request is rounded up to the nearest multiple of the *fasthz* interval.

You can inspect and change the current value of *fasthz* using the *systune* command

```
# systune -i
Updates will be made to running system and /unix.install
systune-> fasthz
        fasthz = 1000 (0x3e8)
systune-> quit
```

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

In systems that have no clock comparator, the kernel implements fast timers by shortening the periodic timer interval to the *fasthz* frequency whenever a process sets a timer at an interval that is not a multiple of HZ The frequent interrupts needed to support fast timers cause an overhead load of several percent of the power of one CPU. In these systems you should try to design your real-time application to use intervals that are multiples of 10 milliseconds, so that the fast interrupt rate is not needed. When that is not feasible, you can assign the interrupt to a particular CPU (see "Assigning the fasthz Processor" on page 79).

Selecting the fasthz Value

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

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

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

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

Which CPU Handles Timer Interrupts

Normally, interval timer interrupts are taken by the CPU in which the itimer was initialized. However, when you isolate a particular CPU (see "Isolating a CPU From TLB Interrupts" on page 84), all itimers pending for that CPU are retargeted to the assigned

Clock CPU (see "Assigning the Clock Processor" on page 79). Any new itimer requests made in an isolated CPU after it has been isolated are taken on that CPU.

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

When timer interrupts are handled in any CPU, timer handling can temporarily be out of synchronization with the time-of-day service. There is a kernel tuning parameter, *itimer_on_clkcpu*, which if set forces all timer interrupts to be taken on the CPU that is the clock CPU. This parameter should only be set when it is crucial that all processes see an exact, consistent relationship between itimer intervals and time of day stamp values. Because it forces timer handling to a single CPU, it can increase the latency of SIGALRM delivery.

Using Timestamps

There are two sources of timestamps, as described earlier in "Timestamps" on page 40. They differ in their accessibility, precision, and accuracy.

Using the Time of Day

There are two system functions that return the current system time of day, one based on BSD UNIX and one on POSIX.

Using BSD gettimeofday()

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

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

The nominal resolution of this timestamp is 1 microsecond. However, it is not practical for the kernel to update an internal timestamp with this frequency. The actual timestamp value is updated at intervals that are convenient to the kernel. The intervals are not

regular, and they differ with hardware and with the version of IRIX in use. However they are never greater than 10 milliseconds.

This does not mean that successive calls to **gettimeofday()** return the same value. On the contrary, experimentation reveals that successive calls always return different values. (See the sample program under "Getting the Time of Day Stamp" on page 184.)

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

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

Using POSIX clock_gettime()

The POSIX-compliant **clock_gettime()** function returns the time of day as two long integers in a structure

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

```
typedef struct timespec {
   long tv_sec; /* seconds */
   long tv_usec; /* and nanoseconds */
} timespec_t;
```

The nominal resolution of this timestamp is 1 nanosecond. However, it is not practical for the kernel to update an internal timestamp with this frequency. The actual timestamp value is updated at intervals that are convenient to the kernel. The intervals are not regular, and they differ with hardware and with the version of IRIX in use. However they are never greater than 10 milliseconds.

Normally an IRIX system uses one of the time-synchronization daemons, either *timed* or *timeslave*, to keep the local clock accurate. These daemons use **adjtime()** to adjust the time of day when necessary. (See the timed(1), timeslave(1), and adjtime(2) reference pages.)

Thus the timestamp returned by **clock_gettime()** should reflect the true local time, with an inaccuracy of at worst -10 milliseconds.

Either the *date* command or the time daemon can adjust the current time by a negative increment. As a result, **clock_gettime()** can in rare circumstances return duplicate values, or a value that is less than the value from the preceding call (see the date(1) reference page).

Using the Cycle Counter

All Silicon Graphics systems have a free-running counter that is updated by hardware at a high frequency. You can map the image of this counter into the process address space, then sample its value as an integer. (For a discussion on mapping segments of memory, see the book *Topics in IRIX Programming*.)

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

You obtain the size fo the cycle counter in bytes (4 or 8) using **syssgi**(SGI_CYCLECNTR_SIZE). You obtain the address of the cycle counter (in the kernel's virtual address space) using **syssgi**(SGI_QUERY_CYCLECNTR), a call that also returns the counter precision in picoseconds (1e-12 seconds, millionths of a microsecond). See the syssgi(2) reference page for details. As can be seen from an example program ("Mapping and Reading the Cycle Counter" on page 175), this method of reading the cycle counter is somewhat complex since it has to take into account two variables: the precision of the clock in the current system, and the programming model of the compiled program (32-bit or 64-bit).

You can also interrogate the cycle counter value, converted to a *timespec_t* form, using the **clock_gettime()** function (see clock_gettime(2) for use of the nonstandard CLOCK_SGI_CYCLE option). The first time this function is used, maps the cycle counter into the process address space. After that, each call merely fetches the hardware value and formats it into the fields of a *timespec_t* structure. By using **clock_gettime()** you eliminate the complexity of **syssgi()** and **mmap()** calls, and the function is portable to all current Silicon Graphics, Inc. systems.

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

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

Comparing the Timestamps

The two timestamp sources can be compared on several different attributes, listed in Table 5-4.

Table 5-4 Comparison of Timestamp Functions

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

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

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