Chapter 6

Controlling CPU Workload

This chapter describes how to use IRIX kernel features to make the execution of a real-time program predictable. Each of these features works in some way to dedicate hardware to your program's use, or to reduce the influence of unplanned interrupts on it. The main topics covered are:

- "Using Priorities and Scheduling Queues" on page 67 covers scheduling concepts, tells how to set nondegrading priorities, and explains affinity scheduling, gang scheduling, and deadline scheduling.
- "Using Processor Sets" on page 76 describes how to define sets of CPUs and how to assign them to specific kinds of work.
- "Minimizing Overhead Work" on page 79 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 87 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 in a multiprocessor system 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 7.

Scheduling Concepts

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

Tick Interrupts

In normal operation, the kernel pauses to make scheduling decisions every 10 milliseconds in every CPU. The duration of this interval, which is called the “tick” because it is the metronomic beat of the scheduler, is defined in *sys/param.h*. Every CPU is normally interrupted by a timer every tick interval. (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 86.

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. (There are actually multiple queues, as described in a later topic.) Every process has a priority and a “nice” value. When a CPU needs a process to run, it normally takes the one with the lowest sum of priority and nice value. Thus a *lower-numbered* priority value gives a process a *superior priority* to run.

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

Table 6-1 Priority Ranges

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

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

Aging Priorities

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

Scheduler Queues

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

Table 6-2 Scheduler Queues

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

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

```
pset -q
```

Setting a Nondegrading Batch Priority

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

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

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

Example 6-1 Setting a Nondegrading Batch Priority

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

The smallest numerical value a regular user can set in these ways is established by the system tuning parameter *ndpri_hilim.* To see its value use *systune*, as shown in Example 6-2.

Example 6-2 Displaying Tuning Variable ndpri_hilim

```
# systune -i
systune-> ndpri_hilim
          ndpri_hilim = 128 (0x80)
```

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

Setting a Nondegrading Real-Time Priority

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

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

The same change can be effected from within a process using **schedctl()**, as shown in Example 6-3.

Example 6-3 Setting a Real-Time Priority

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

The real-time priorities are those numerically less than or equal to the system tuning parameter *ndpri_lolim*, which is normally 39. You can view or change *ndpri_lolim* using *systune*, as shown in Example 6-2.

The kernel guarantees that a runnable process with one of the real-time band of priorities will never sit idle waiting for a process with a lower priority.

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

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

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

Understanding Affinity Scheduling

Affinity scheduling is a special scheduling discipline used in 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 as the amount of time the process used the CPU, with 300 microseconds or less having zero affinity, and 10 milliseconds or more having 100% affinity.

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

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

- The CPU runs a different process.

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

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

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

Using Gang Scheduling

You have been advised to design a real-time program as a family of cooperating, lightweight processes sharing an address space (see, for example, "Lightweight Process Creation With sproc()" on page 17). These processes typically coordinate their actions using locks or semaphores ("Interprocess Communication" on page 29).

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 6-4

Example 6-4 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.

Tip: Gang-scheduled processes are queued in one of the two gang queues (see “Scheduler Queues” on page 70). You can use pset to assign a set of CPUs to work only on the gang queues (see “Assigning a Processor Set to a Queue” on page 77).

Using Deadline Scheduling

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

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

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

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

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

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

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

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

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

For an example of using **schedctl()** to set deadline scheduling, see "Deadline Scheduling Subroutines" on page 200.

Note: The kernel uses a high-precision interval timer to measure usage under deadline scheduling. When deadline scheduling is in use, more frequent timer interrupts are generated. In some architectures this causes frequent kernel interrupts (see "Timer Management Without a Clock Comparator" on page 60 and "Assigning the fasthz Processor" on page 79).

Changing the Time Slice Duration

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

You can change the length of the time slice for one particular process using the **schectl()** function (see the schedctl(2) reference page). The code would resemble Example 6-5.

Example 6-5 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 73). This will keep members of the gang executing concurrently longer.

Using Processor Sets

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

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

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

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

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

Assigning a Process to a Processor Set

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

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

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

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

Example 6-6 Command to Assign Process to Processor Set

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

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

Tip: Keep in mind that affinity scheduling tends to keep a CPU-bound process on one CPU in any case (see "Understanding Affinity Scheduling" on page 72). In general, the dynamic operation of the IRIX scheduler, guided by a nondegrading priority or deadline scheduling, can do a better job of allocating CPUs to processes than you can do with a static assignment through *pset*.

Assigning a Processor Set to a Queue

Using *pset* you can assign a set of CPUs to service a particular scheduling queue (see "Scheduler Queues" on page 70). Only those CPUs will take processes from that queue, and they will take processes from no other queue. For example, to assign CPU 9 alone to service the batch queue, use:

```
pset -s 1009 9
```

```
pset -q bt 1009
```

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

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

Assigning a Discipline to a Processor Set

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

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

Processor Set Contradictions

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

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

Minimizing Overhead Work

A certain amount of CPU time must be spent on general housekeeping. Since this work is done by the kernel and triggered by interrupts, it can interfere with the operation of a real-time process. However, you can remove almost all such work from designated CPUs, leaving them free for real-time work.

First decide how many CPUs are required to run your real-time application (regardless of whether it will be scheduled normally, or as a gang, or by the Frame Scheduler). Then apply the following steps to isolate and restrict those CPUs. The steps are independent of each other. Each needs to be done to completely free a CPU.

Assigning the Clock Processor

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

```
mpadmin -c 0
```

The equivalent operation from within a program uses **sysmp()** as shown in Example 6-7 (see also the sysmp(2) reference page).

Example 6-7 Setting the Clock CPU

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

Assigning the fasthz Processor

When high precision timers are used, timer interrupts occur more frequently. In machines that lack a clock comparator, fast timer interrupts cause overhead processing (see "Fast Timers Without a Clock Comparator" on page 61). A particular CPU can be designated to handle this work. You can use the *-f* parameter of *mpadmin* to find out which CPU has responsibility:

```
% mpadmin -f
0
```

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

```
mpadmin -f 1
```

The equivalent operation from software uses **sysmp()**, as shown in Example 6-8.

Example 6-8 Setting the fasthz CPU

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

Note: On Challenge/Onyx and POWER-Challenge systems, assigning the fasthz CPU is allowed, but has no effect. Timer interrupts are taken only as required, not at the *fasthz* rate, and are targeted to the CPU where they were initiated. (See “Timer Management in Challenge, Onyx, and POWER-Challenge” on page 60.)

Unavoidable Timer Interrupts

Prior to IRIX version 5.3, even when the clock and fast timer duties were removed from a CPU, that CPU still received a timer interrupt approximately every 42 seconds. This was the result of the maximum value, 0x7fff ffff, counting down in a hardware timer. The resulting interrupt was processed in the normal timer-handling code, which used nearly 100 microseconds before recognizing the interrupt as unwanted.

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

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

Isolating a CPU From Sprayed Interrupts

By default, the Challenge/Onyx hardware directs I/O interrupts 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*.

Assigning Interrupts to CPUs

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

`IPL` *level# cpu#*

Interrupts 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 87).

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 81.

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 108). 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 6-9.

Example 6-9 Number of Processors Available and Total

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

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

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

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

Example 6-10 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 79) must not be restricted.
- The REACT/Pro Frame Scheduler automatically restricts and isolates any CPU it uses. See Chapter 7.

Assigning Work to a Restricted CPU

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 6-11 (see also the sysmp(2) reference page).

Example 6-11 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 82, 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 68). 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 6-12 (see the sysmp(2) reference page).

Example 6-12 Making a CPU nonpreemptive

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

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

Minimizing Interrupt Response Time

Interrupt response time is the time that passes between the instant when a hardware device raises an interrupt signal, and the instant when—interrupt service completed—the system returns control to a user process. IRIX guarantees a maximum interrupt response time on certain systems, but you have to configure the system properly to realize the guaranteed time.

Maximum Response Time Guarantee

In Challenge/Onyx and POWER-Challenge systems running IRIX 5.3 and 6.2, interrupt response time is guaranteed not to exceed 200 microseconds in a properly configured system.

This guarantee is important to a real-time program because it puts an upper bound on the overhead of servicing interrupts from real-time devices. You should have some idea of the number of interrupts that will arrive per second. Multiplying this by 200 microseconds yields a conservative estimate of the amount of time in any one second devoted to interrupt handling in the CPU that receives the interrupts. The remaining time is available to your real-time application in that CPU.

Components of Interrupt Response Time

The total interrupt response time includes these sequential parts:

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

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

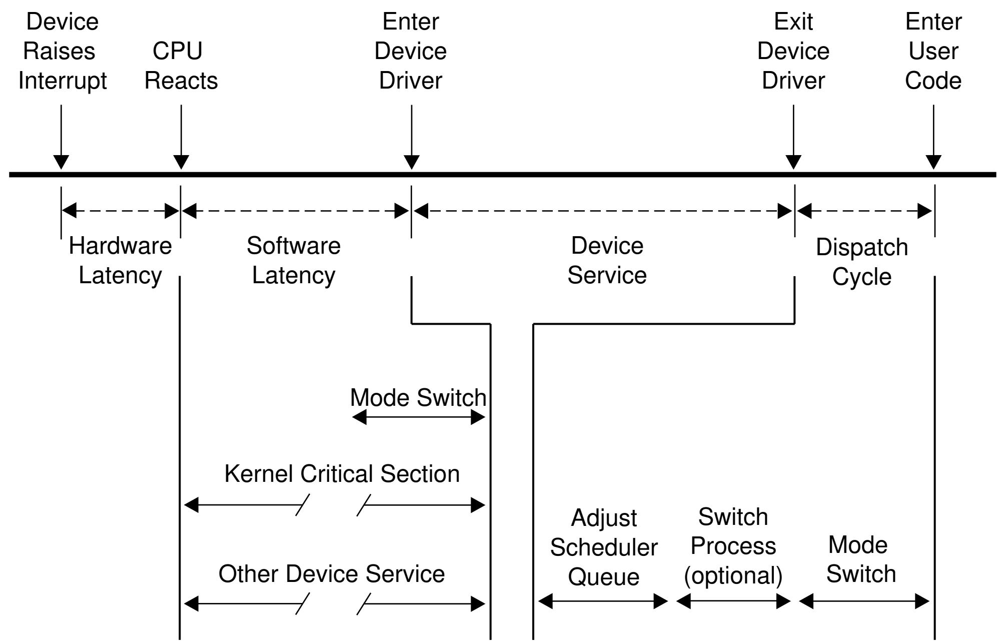

Figure 6-1 Components of Interrupt Response Time

Hardware Latency

When a VME device requests an interrupt, one of the 7 VME IRQ lines is set active. The Challenge/Onyx VCAM VME Controller contains interrupt destination registers that are programmed by the IRIX kernel to direct IRQ lines to specific CPUs. (The programming is in the IPL and NOINTR configuration statements. See “Isolating a CPU From Sprayed Interrupts” on page 81 and “Assigning Interrupts to CPUs” on page 81).

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

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

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

Software Latency

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

- executing user code or noncritical kernel code

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

- executing a critical section in the kernel

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

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

  The mode switch occurs when the other device service ends.

Kernel Critical Sections

Most of the IRIX kernel code is noncritical and executed with interrupts enabled. However, certain sections of kernel code depend on exclusive access to shared resources. Spin locks are used to control access to these critical sections. Once in a critical section, the interrupt level is raised in that CPU. New interrupts are not serviced until the critical section is complete.

Although most kernel critical sections are short, there is *no guarantee* on the length of a critical section. In order to achieve 200 microsecond response time, your real-time program must avoid executing system calls on the CPU where interrupts are handled. The way to ensure this is to restrict that CPU from running normal processes (see "Restricting a CPU From Scheduled Work" on page 82) and isolate it from TLB interrupts (see "Isolating a CPU From TLB Interrupts" on page 84)—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 173).

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 81).

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