Chapter 7

Managing User Processes

Just as files can use up your available disk space, too many processes going at once can use up your available CPU time. When this happens, your system response time gets slower and slower until finally the system cannot execute any processes effectively. If you have not tuned your kernel to allow for more processes, the system will refuse new processes long before it reaches a saturation point. However, due to normal variations in system usage, you may experience fluctuations in your system performance without reaching the maximum number of processes allowed by your system.

Monitoring User Processes

Not all processes require the same amount of system resources. Some processes, such as database applications working with large files, tend to be *disk intensive*, requiring a great deal of reading from and writing to the disk as well as a large amount of space on the disk. These activities take up CPU time. Time is also spent waiting for the hardware to perform the requested operations. Other jobs, such as compiling programs or processing large amounts of data, are *CPU intensive*, since they require a great number of CPU instructions to be performed. Some jobs are *memory intensive*, such as a process that reads a great deal of data and manipulates it in memory. Since the disk, CPU, and memory resources are limited, if you have more than a few intensive processes running at once on your system, you may see a performance degradation.

As the administrator, you should be on the lookout for general trends in system usage, so you can respond to them and keep the systems running as efficiently as possible. If a system shows signs of being overloaded, and yet the total number of processes is low, your system may still be at or above reasonable capacity. The following sections show four ways to monitor your system processes.

Monitoring Processes With top

The *top* and *gr_top* commands are the most convenient utilities provided with IRIX to monitor the top CPU-using processes on your system. These utilities display the top such processes dynamically, that is, if a listed process exits, it is removed from the table and the next-highest CPU-using process takes its place. *gr_top* graphically displays the same information as *top*. If you are using a non-graphics server, you cannot use *gr_top* locally, but you can use it if you set the display to another system on the network that does have graphics capability. For complete information on configuring and using *top* and *gr_top*, consult the *top*(1) and *gr_top*(1) reference pages. For information on resetting the display, see "Displaying Windows on Alternate Workstations" on page 18.

Monitoring Processes With osview

The *osview* and *gr_osview* commands display kernel execution statistics dynamically. If you have a graphics workstation, you can use the *gr_osview*(1) tool, which provides a real-time graphical display of system memory and CPU usage. *osview* provides the same information in ASCII format. You can configure *gr_osview* to display several different types of information about your system's current status. In its default configuration, *gr_osview* provides information on the amount of CPU time spent on user process execution, system overhead tasks, interrupts, and idle time. For complete information on *osview* and *gr_osview*, see the *osview*(1) and *gr_osview*(1) reference pages.

Monitoring Processes With sar

The System Activity Reporter, *sar*, provides essentially the same information as *osview*, but it represents a "snapshot" of the system status, not a dynamic reflection. Because *sar* generates a single snapshot, it is easily saved and can be compared with a similar snapshot taken at another time. You can use *sar* automatically with *cron* to get a series of system snapshots over time to help you locate chronic system bottlenecks by establishing baselines of performance for your system at times of light and heavy loads, and under loads of various kinds (cpu load, network load, disk load, and so on). For complete information on *sar*, see the *sar*(1) reference pages. For more information on using sar to monitor system activity, see "Using timex(1), sar(1), and par(1)" on page 190.

Monitoring Processes With ps

The *ps -ef* command allows you to look at all the processes currently running on your system.The output of *ps -ef* follows the format shown in Table 7-1:

Table 7-1 Output format of the *ps -ef* Command

Name	PID	PPID	C	Time	TTY	CPU Time	Process
joe	23328	316	1	May 5	ttyq1	1:01	csh

In this table, the process shown is for the user "joe." In a real situation, each user with processes running on the system is represented. Each field in the output contains some useful information.

Name	The login name of the user who "owns" the process.
PID	The process identification number.
PPID	The process identification number of the parent process that spawned or forked the listed process.
C	Current execution priority. The higher this number, the lower the scheduling priority. This number is based on the recent scheduling of the process and is not a definitive indicator of its overall priority.
Time	The time when the process began executing. If it began more than 24 hours before the *ps* command was given, the date on which it began is displayed.
TTY	The TTY (Terminal or window) with which the process is associated.
CPU	The total amount of CPU time expended to date on this process. This field is useful in determining which processes are using the most CPU time. If a process uses a great deal in a brief period, it can cause a general system slowdown.

For even more information, including the general system priority of each process, use the *-l* flag to *ps*. For complete information on interpreting *ps* output, see the *ps*(1) reference page.

Prioritizing Processes With nice

IRIX provides methods for users to force their CPU-intensive processes to execute at a lower priority than general user processes. The */bin/nice*(1) and *npri*(1M) commands allow the user to control the priority of their processes on the system. The *nice* command functions as follows:

```
nice [ –increment ] command
```

When you form your command line using */bin/nice*, you fill in the **increment** field with a number between 1 and 19. If you do not fill in a number, a default of 10 is assumed. The higher the number you use for the **increment**, the lower your process' priority will be (19 is the lowest possible priority; all numbers greater than 19 are interpreted as 19). The *csh*(1) shell has its own internal *nice* functions, which operate differently from the *nice* command, and are documented in the *csh*(1) reference page.

After entering the *nice* command and the increment on your command line, give the **command** as you would ordinarily enter it. For example, if the user "joe" wants to make his costly compile command described in the *ps -ef* listing above happen at the lowest possible priority, he forms the command line as follows:

```
nice –19 cc –o prog prog.c
```

If a process is invoked using *nice*, the total amount of CPU time required to execute the program does not change, but the time is spread out, since the process executes less often.

The superuser (*root*) is the only user who can give *nice* a negative value and thereby *increase* the priority of a process. To give *nice* a negative value, use two minus signs before the increment. For example:

```
nice ––19 cc –o prog prog.c
```

The above command endows that process with the highest priority a user process may possess. The superuser should not use this feature frequently, as even a single process that has been upgraded in priority causes a significant system slowdown for all other users. Note that */bin/csh* has a built-in *nice* program that uses slightly different syntax than that described here. For complete information on *csh*, see the *csh*(1) reference page.

The *npri* command allows users to make their process' priority *nondegrading*. In the normal flow of operations, a process loses priority as it executes, so large jobs typically use fewer CPU cycles per minute as they grow older. (There is a minimum priority, too. This priority degradation simply serves to maintain performance for simple tasks.) By using *npri*, the user can set the *nice* value of a process, make that process non-degrading, and also set the default time slice that the CPU allocates to that process. *npri* also allows you to change the priority of a currently running process. The following example usage of *npri* sets all the possible variables for a command:

```
npri -h 10 -n 10 -t 3 cc -o prog prog.c
```

In this example, the *-h* flag sets the nondegrading priority of the process, while the *-n* flag sets the absolute *nice* priority. The *-t* flag sets the time slice allocated to the process. IRIX uses a 10-millisecond time slice as the default, so the example above sets the time slice to 30 milliseconds. For complete information about *npri* and its flags and options, see the *npri*(1) reference page.

Changing the Priority of a Running Process

The superuser can change the priority of a running process with the *renice*(1M) or *npri* commands. Only the superuser can use these commands. *renice* is used as follows:

```
renice increment pid [-u user] [-g pgrp]
```

In the most commonly used form, *renice* is invoked on a specific process that is using system time at an overwhelming rate. However, you can also invoke it with the *-u* flag to lower the priority of all processes associated with a certain user, or with the *-g* flag to lower the priorities of all processes associated with a process group. More options exist and are documented in the *renice*(1M) reference page.

The *npri* command can also be used to change the parameters of a running process. This example changes the parameters of a running process with *npri*:

```
npri -h 10 -n 10 -t 3 -p 11962
```

The superuser can use *renice* or *npri* to increase the priority of a process or user, but this can severely impact system performance.

Terminating Processes

From time to time a process may use so much memory, disk, or CPU time that your only alternative is to terminate it before it causes a system crash. Before you kill a process, make sure that the user who invoked the process will not try to invoke it again. You should, if at all possible, speak to the user before killing the process, and at a minimum you should notify the user that the process was prematurely terminated and give a reason for the termination. If you do this, the user can reinvoke the process at a lower priority or possibly use the system's job processing facilities (*at*, *batch*, and *cron*) to execute the process at another time.

To terminate a process, you use the *kill* command. Typically, for most terminations, you should use the *kill -15* variation. The *-15* flag indicates that the process is to be allowed time to exit gracefully, closing any open files and descriptors. The *-9* flag to *kill* terminates the process immediately, with no provision for cleanup. If the process you are going to kill has any child processes executing, using the *kill -9* command may cause those child processes to continue to exist on the process table, though they will not be responsive to input. The *wait*(1) command, given with the process number of the child process, removes them. For complete information about the syntax and usage of the *kill* command, see the *kill*(1) reference page. You must always know the PID of the process you intend to kill with the *kill* command.

Killing Processes by Name with the killall(1M) Command

The *killall*(1M) command allows you to kill processes by their command name. For example, if you wish to kill the program *a.out* that you invoked, use the syntax:

```
killall a.out
```

This command allows you to kill processes without the time-consuming task of looking up the process ID number with the *ps*(1M) command.

Note: This command kills all instances of the named program running under your shell and if invoked with no arguments, kills all processes on the system that are killable by the user who invoked the command. For ordinary users, these are simply the processes invoked and forked by that user, but if invoked by *root*, all processes on the system will be killed. For this reason, this command should be used carefully.