Chapter 9

Managing Linux Processes

Chapter Objectives

1. Categorize the different types of processes on a Linux system.
2. View processes using standard Linux utilities.
3. Explain the difference between common kill signals.
4. Describe how binary programs and shell scripts are executed.
5. Create and manipulate background processes.
6. Use standard Linux utilities to modify the priority of a process.
7. Schedule commands to execute in the future using the at daemon.
8. Schedule commands to execute repetitively using the cron daemon.

A typical Linux system can run thousands of processes simultaneously, including those that you have explored in previous chapters. In this chapter, you focus on viewing and managing processes. In the first part of the chapter, you examine the different types of processes on a Linux system and how to view them and terminate them. You then discover how processes are executed on a system, run in the background, and prioritized. Finally, you examine the various methods used to schedule commands to execute in the future.

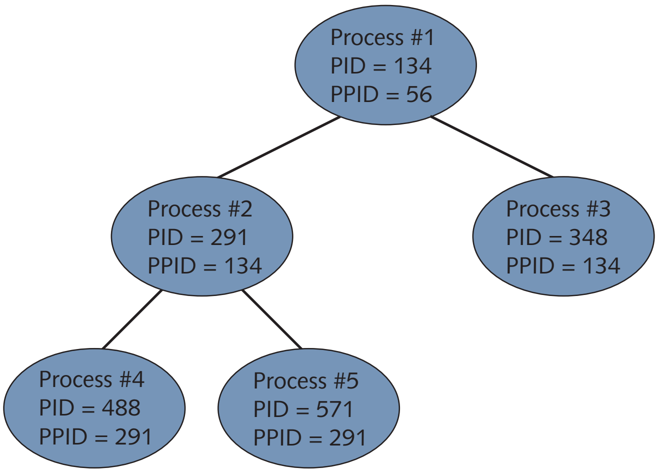
Linux Processes

Throughout this book, the terms "program" and "process" are used interchangeably. The same is true in the workplace. However, a fine distinction exists between these two terms. Technically, a **program** is an executable file on the filesystem that can be run when you execute it. A **process**, on the other hand, is a program that is running in memory and on the CPU. In other words, a process is a program in action.

If you start a process while logged in to a terminal, that process runs in that terminal and is labeled a **user process**. Examples of user processes include `ls`, `grep`, and `find`, not to mention most of the other commands that you have executed throughout this book. Recall that a system process that is not associated with a terminal is called a **daemon process**; these processes are typically started on system startup, but you can also start them manually. Most daemon processes provide system services, such as printing, scheduling, and system maintenance, as well as network server services, such as web servers, database servers, file servers, and print servers.

Every process has a unique **process ID (PID)** that allows the kernel to identify it uniquely. In addition, each process can start an unlimited number of other processes called **child processes**. Conversely, each process must have been started by an existing process called a **parent process**. As a result, each process has a **parent process ID (PPID)**, which identifies the process that started it. An example of the relationship between parent and child processes is depicted in Figure 9-1.

Figure 9-1 Parent and child processes



Note 1

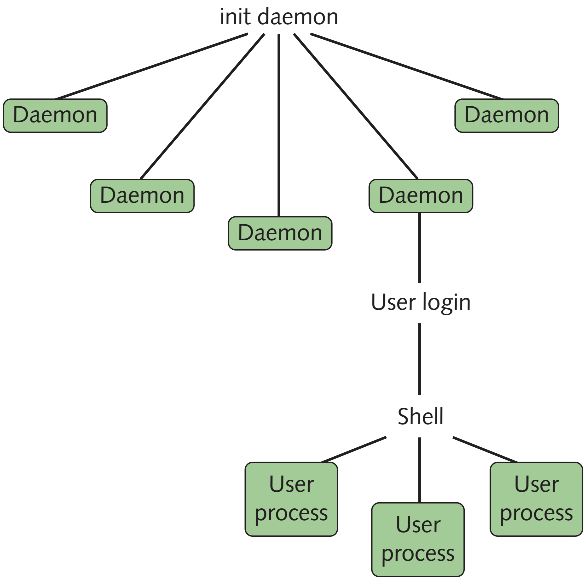
PIDs are not necessarily given to new processes in sequential order; each PID is generated from free entries in a process table used by the Linux kernel.

Note 2

Remember that although each process can have an unlimited number of child processes, it can only have one parent process.

The first process started by the Linux kernel is the initialize (or init) daemon, which has a PID of 1 and a PPID of 0, the latter of which refers to the kernel itself. The init daemon then starts most other daemons during the system initialization process, including those that allow for user logins. After you log in to the system, the login program starts a shell. The shell then interprets user commands and starts all user processes. Thus, each process on the Linux system can be traced back to the init daemon by examining the series of PPIDs, as shown in Figure 9-2.

Figure 9-2 Process genealogy



Note 3

The init daemon is often referred to as the "grandfather of all user processes."

Note 4

On Linux systems that use the UNIX SysV system initialization process, the init daemon will be listed as `init` within command output. On Linux systems that use the Systemd system initialization process, the init daemon will either be listed as `init` or `systemd` within command output.

Viewing Processes

Although several Linux utilities can view processes, the most versatile and common is the **`ps` command**. Without arguments, the `ps` command simply displays a list of processes that are running in the current shell. The following example shows the output of this command while the user is logged in to tty2:

```
[root@server1 ~]# ps
  PID TTY          TIME CMD
 2159 tty2     00:00:00 bash
 2233 tty2     00:00:00 ps
[root@server1 ~]#_
```

The preceding output shows that two processes were running in the terminal tty2 when the `ps` command executed. The command that started each process (CMD) is listed next to the time it has taken on the CPU (TIME), its PID, and terminal (TTY). In this case, the process took less than one second to run, and so the time elapsed reads nothing. To find out more about these processes, you could instead use the `–f`, or full, option to the `ps` command, as shown next:

```
[root@server1 ~]# ps -f
UID         PID  PPID  C STIME TTY          TIME CMD
root       2159  2156  0 16:18 tty2     00:00:00 -bash
root       2233  2159  3 16:28 tty2     00:00:00 ps -f
[root@server1 ~]#_
```

This listing provides more information about each process. It displays the user who started the process (UID), the PPID, the time it was started (STIME), as well as the CPU utilization (C), which starts at zero and is incremented with each processor cycle that the process runs on the CPU.

The most valuable information provided by the `ps –f` command is each process's PPID and lineage. The bash process (PID = 2159) displays a shell prompt and interprets user input; it started the `ps` process (PID = 2233) because the `ps` process had a PPID of 2159.

Because daemon processes are not associated with a terminal, they are not displayed by the `ps –f` command. To display an entire list of processes across all terminals and including daemons, you can add the `–e` option to any `ps` command, as shown in the following output:

```
[root@server1 ~]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 21:22 ?        00:00:00 /usr/lib/systemd/systemd
root         2     0  0 21:22 ?        00:00:00 [kthreadd]
root         3     2  0 21:22 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 21:22 ?        00:00:00 [kworker/0:0H]
root         6     2  0 21:22 ?        00:00:00 [kworker/u128:0]
root         7     2  0 21:22 ?        00:00:00 [migration/0]
root         8     2  0 21:22 ?        00:00:00 [rcu_bh]
```

```
root         9      2  0 21:22 ?       00:00:00 [rcu_sched]
root        10      2  0 21:22 ?       00:00:00 [watchdog/0]
root        11      2  0 21:22 ?       00:00:00 [khelper]
root        12      2  0 21:22 ?       00:00:00 [kdevtmpfs]
root        13      2  0 21:22 ?       00:00:00 [netns]
root        14      2  0 21:22 ?       00:00:00 [writeback]
root       394      1  0 21:22 ?       00:00:00 /sbin/auditd -n
avahi      422      1  0 21:22 ?       00:00:00 avahi-daemon: running
dbus       424      1  0 21:22 ?       00:00:00 /bin/dbus-daemon --system
chrony     430      1  0 21:22 ?       00:00:00 /usr/sbin/chronyd -u
root       431      1  0 21:22 ?       00:00:00 /usr/sbin/crond -n
root       432      1  0 21:22 ?       00:00:00 /usr/sbin/atd -f
root       435      1  0 21:22 ?       00:00:00 /usr/sbin/abrtd -d -s
root       437      1  0 21:22 ?       00:00:00 /usr/bin/abrt-watch-log
root       441      1  0 21:22 ?       00:00:00 /usr/sbin/gdm
root       446      1  0 21:22 ?       00:00:00 /usr/sbin/mcelog
root       481    441  0 21:22 ?       00:00:00 /usr/libexec/gdm-simple
polkitd    482      1  0 21:22 ?       00:00:00 /usr/lib/polkit-1/polkitd
root       488    481  0 21:22 tty1    00:00:00 /usr/bin/Xorg :0
root       551    481  0 21:22 ?       00:00:00 gdm-session-worker
root       552      1  0 21:22 ?       00:00:00 /usr/sbin/NetworkManager
gdm        852    551  0 21:23 ?       00:00:00 /usr/bin/gnome-session
gdm        856      1  0 21:23 ?       00:00:00 /usr/bin/dbus-launch
gdm       1018      1  0 21:23 ?       00:00:00 /bin/dbus-daemon --fork
root      1020    552  0 21:23 ?       00:00:00 /sbin/dhclient -d -sf
gdm       1045   1018  0 21:23 ?       00:00:00 /bin/dbus-daemon
root      1072      1  0 21:23 ?       00:00:00 /usr/libexec/upowerd
gdm       1077    852  0 21:23 ?       00:00:03 gnome-shell --mode=gdm
gdm       1087      1  0 21:23 ?       00:00:00 /usr/bin/pulseaudio
gdm       1148      1  0 21:23 ?       00:00:00 /usr/libexec/goa-daemon
root      1164      1  0 21:23 ?       00:00:00 login -- root
root      1175   1164  0 21:23 tty2    00:00:00 -bash
root      1742   1175  0 21:33 tty2    00:00:00 ps -ef
[root@server1 ~]#_
```

As shown in the preceding output, the kernel thread daemon (`kthreadd`) has a PID of 2 and starts most subprocesses within the actual Linux kernel because those subprocesses have a PPID of 2, whereas the init daemon (/usr/lib/systemd/systemd, PID=1) starts most other daemons because those daemons have a PPID of 1. In addition, there is a `?` in the TTY column for daemons and kernel subprocesses because they do not run on a terminal.

Because the output of the `ps –ef` command can be several hundred lines long on a Linux server, you usually pipe its output to the `less` command to send the output to the terminal screen page-by-page, or to the `grep` command, which can be used to display lines containing only certain information. For example, to display only the BASH shells on the system, you could use the following command:

```
[root@server1 ~]# ps –ef | grep bash
user1      2094   2008   0 14:29 pts/1     00:00:00 -bash
root       2159   2156   0 14:30 tty2      00:00:00 -bash
root       2294   2159   0 14:44 tty2      00:00:00 grep --color=auto bash
[root@server1 ~]#_
```

Notice that the `grep bash` command is also displayed alongside the BASH shells in the preceding output because it was running in memory at the time the `ps` command was executed. This might not always be the case because the Linux kernel schedules commands to run based on a variety of factors.

The -e and -f options are the most common options used with the ps command; however, many other options are available. The -l option to the ps command lists even more information about each process than the -f option. An example of using this option to view the processes in the terminal tty2 is shown in the following output:

```
[root@server1 ~]# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  2159  2156  0  80   0 -  1238 wait   tty2     00:00:00 bash
4 R     0  2295  2159  2  80   0 -   744 -      tty2     00:00:00 ps
[root@server1 ~]#_
```

The process flag (F) indicates particular features of the process; the flag of 4 in the preceding output indicates that the root user ran the process. The **process state** (S) column is the most valuable to systems administrators because it indicates what the process is currently doing. If a process is not being run on the processor at the current time, you see an S (interruptible sleep) in the process state column; processes are in this state most of the time and are awoken (interrupted) by other processes when they are needed, as seen with bash in the preceding output. You will see an R in this column if the process is currently running on the processor, a D (uninterruptible sleep) if it is waiting for disk access, or a T if it has stopped or is being traced by another process. In addition to these, you might also see a Z in this column, indicating a **zombie process**. When a process finishes executing, the parent process must check to see if it executed successfully and then release the child process's PID so that it can be used again. While a process is waiting for its parent process to release the PID, the process is said to be in a zombie state, because it has finished but still retains a PID. On a busy Linux server, zombie processes can accumulate and prevent new processes from being created; if this occurs, you can kill the parent process of the zombies, as discussed in the next section.

Note 5

Zombie processes are also known as defunct processes.

Note 6

To view a list of zombie processes on your entire system, you could use the ps -el | grep Z command.

Process priority (PRI) is the priority used by the kernel for the process; it is measured between 0 (high priority) and 127 (low priority). The **nice value** (NI) can be used to affect the process priority indirectly; it is measured between –20 (a greater chance of a high priority) and 19 (a greater chance of a lower priority). The ADDR in the preceding output indicates the memory address of the process, whereas the WCHAN indicates what the process is waiting for while sleeping. In addition, the size of the process in memory (SZ) is listed and measured in kilobytes; often, it is roughly equivalent to the size of the executable file on the filesystem.

Some options to the ps command are not prefixed by a dash character; these are referred to as Berkeley style options. The two most common of these are the a option, which lists all processes across terminals, and the x option, which lists processes that do not run on a terminal, as shown in the following output for the first 10 processes on the system:

```
[root@server1 ~]# ps ax | head -11
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:01 /usr/lib/systemd/systemd
    2 ?        S      0:00 [kthreadd]
    3 ?        S      0:00 [migration/0]
    4 ?        S<     0:00 [ksoftirqd/0]
```

```
    5 ?        S      0:00 [watchdog/0]
    6 ?        S      0:00 [migration/1]
    7 ?        S      0:00 [ksoftirqd/1]
    8 ?        S      0:00 [watchdog/1]
    9 ?        S      0:00 [events/0]
   10 ?        S      0:00 [events/1]
[root@server1 ~]#_
```

The columns just listed are equivalent to those discussed earlier; however, the process state column is identified with STAT and might contain additional characters to indicate the full nature of the process state. For example, a `W` indicates that the process has no contents in memory, a < symbol indicates a high-priority process, and an `N` indicates a low-priority process.

Note 7

For a full list of symbols that may be displayed in the STAT or S columns shown in prior output, consult the manual page for the `ps` command.

Several dozen options to the `ps` command can be used to display processes and their attributes; the options listed in this section are the most common and are summarized in Table 9-1.

Table 9-1 Common options to the `ps` command

Option	Description
`-e`	Displays all processes running on terminals as well as processes that do not run on a terminal (daemons)
`-f`	Displays a full list of information about each process, including the UID, PID, PPID, CPU utilization, start time, terminal, processor time, and command name
`-l`	Displays a long list of information about each process, including the flag, state, UID, PID, PPID, CPU utilization, priority, nice value, address, size, WCHAN, terminal, and command name
`-Z`	Displays SELinux context information about each process (discussed further in Chapter 14)
`a`	Displays all processes running on terminals
`x`	Displays all processes that do not run on terminals

The `ps` command is not the only command that can view process information. The kernel exports all process information subdirectories under the /proc directory. Each subdirectory is named for the PID of the process that it contains information for, as shown in the following output:

```
[root@server1 ~]# ls /proc
1     1174  1746  28   407  473  852        irq          slabinfo
10    1175  175   292  409  48   856        kallsyms     softirqs
1018  12    1754  3    411  481  9          kcore        stat
1020  1213  176   307  412  482  acpi       keys         swaps
1025  1216  1760  328  414  488  buddyinfo  key-users    sys
1045  1220  177   350  415  49   bus        kmsg         sysrq
1052  13    178   351  418  5    cgroups    kpagecount   sysvipc
1065  14    179   353  420  551  cmdline    kpageflags   timer_list
1072  15    18    354  421  552  consoles   loadavg      timer_stats
1077  16    180   357  422  58   cpuinfo    locks        tty
1080  164   181   358  424  59   crypto     mdstat       uptime
```

```
1087  165   1810  359  430  6    devices      meminfo       version
1097  167   188   370  431  60   diskstats    misc          vmallocinfo
11    168   189   371  432  62   dma          modules       vmstat
1111  169   19    372  435  64   driver       mounts        zoneinfo
1114  17    191   383  437  7    execdomains  mtrr
1117  170   2     384  440  72   fb           net
1122  171   20    385  441  748  filesystems  pagetypeinfo
1144  172   204   386  446  8    fs           partitions
1148  173   205   387  45   814  interrupts   sched_debug
1164  174   206   388  46   823  iomem        scsi
1171  1745  276   394  47   842  ioports      self
[root@server1 ~]#_
```

Thus, any program that can read from the /proc directory can display process information. For example, the **pstree command** displays the lineage of a process by tracing its PPIDs until the init daemon. The first 26 lines of this command are shown in the following output:

```
[root@server1 ~]# pstree | head -26
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager─┬─dhclient
        │                └─3*[{NetworkManager}]
        ├─2*[abrt-watch-log]
        ├─abrtd
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─alsactl
        ├─at-spi-bus-laun─┬─dbus-daemon───{dbus-daemon}
        │                 └─3*[{at-spi-bus-laun}]
        ├─at-spi2-registr───{at-spi2-registr}
        ├─atd
        ├─auditd─┬─audispd─┬─sedispatch
        │        │         └─{audispd}
        │        └─{auditd}
        ├─avahi-daemon───avahi-daemon
        ├─bluetoothd
        ├─chronyd
        ├─colord───2*[{colord}]
        ├─crond
        ├─2*[dbus-daemon───{dbus-daemon}]
        ├─dbus-launch
        ├─dconf-service───2*[{dconf-service}]
        ├─firewalld───{firewalld}
        ├─gdm─┬─gdm-simple-slav─┬─Xorg
        │     │                 ├─gdm-session─┬─gnome-session─┬─gnome-settings
        │     │                 │             │               ├─gnome-shell
[root@server1 ~]# _
```

The most common program used to display processes, aside from ps, is the **top command**. The top command displays an interactive screen listing processes organized by processor time. Processes that use the most processor time are listed at the top of the screen. An example of the screen that appears when you type the top command is shown next:

```
top - 21:55:15 up 32 min, 3 users, load average: 0.15, 0.06, 0.02
Tasks: 134 total, 1 running, 133 sleeping,  0 stopped,  0 zombie
%Cpu(s): 0.4 us, 0.4 sy, 0.0 ni, 95.9 id, 2.0 wa, 1.1 hi, 0.1 si, 0.0 st
MiB Mem:   7944.0 total, 7067.5 free, 467.3 used,  409.2 buff/cache
MiB Swap:  7944.0 total, 7944.0 free,   0.0 used, 7226.7 avail Mem

  PID USER    PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1077 gdm     20   0 1518192 150684  36660 R 19.4  7.4   0:33.93 gnome-shell
 2130 root    20   0  123636   1644   1180 R  0.7  0.1   0:00.05 top
    1 root    20   0   51544   7348   2476 S  0.0  0.4   0:00.98 systemd
```

```
  2 root    20   0       0       0       0 S  0.0  0.0   0:00.00 kthreadd
  3 root    20   0       0       0       0 S  0.0  0.0   0:00.01 ksoftirqd/0
  5 root     0 -20       0       0       0 S  0.0  0.0   0:00.00 kworker/0:0H
  6 root    20   0       0       0       0 S  0.0  0.0   0:00.02 kworker/u12+
  7 root    rt   0       0       0       0 S  0.0  0.0   0:00.00 migration/0
  8 root    20   0       0       0       0 S  0.0  0.0   0:00.00 rcu_bh
  9 root    20   0       0       0       0 S  0.0  0.0   0:00.34 rcu_sched
 10 root    rt   0       0       0       0 S  0.0  0.0   0:00.00 watchdog/0
 11 root     0 -20       0       0       0 S  0.0  0.0   0:00.00 khelper
 12 root    20   0       0       0       0 S  0.0  0.0   0:00.00 kdevtmpfs
 13 root     0 -20       0       0       0 S  0.0  0.0   0:00.00 netns
 14 root     0 -20       0       0       0 S  0.0  0.0   0:00.00 writeback
 15 root     0 -20       0       0       0 S  0.0  0.0   0:00.00 kintegrityd
 16 root     0 -20       0       0       0 S  0.0  0.0   0:00.00 bioset
```

Note that the `top` command displays many of the same columns that the `ps` command does, yet it contains a summary paragraph at the top of the screen and a cursor between the summary paragraph and the process list. From the preceding output, you can see that the `gnome-shell` uses the most processor time, followed by the top command itself (`top`) and the init daemon (`systemd`).

You might come across a process that has encountered an error during execution and continuously uses up system resources. These processes are referred to as **rogue processes** and appear at the top of the listing produced by the `top` command. The `top` command can also be used to change the priority of processes or kill them. Thus, you can stop rogue processes from the `top` command immediately after they are identified. Process priority and killing processes are discussed later in this chapter. To get a full listing of the different commands that you can use while in the top utility, press `h` to get a help screen.

Note 8

Rogue processes are also known as runaway processes.

Note 9

Many Linux administrators choose to install and use the **`htop` command** instead of `top`. The `htop` command provides the same functionality as `top` but displays results in color and includes a resource utilization graph at the top of the screen, as well as a usage legend at the bottom of the screen.

Killing Processes

As indicated earlier, a large number of rogue and zombie processes use up system resources. When system performance suffers due to these processes, you should send them a **kill signal**, which terminates a process. The most common command used to send kill signals is the **`kill` command**. All told, the `kill` command can send many different kill signals to a process. Each of these kill signals operates in a different manner. To view the kill signal names and associated numbers, you can use the `–l` option to the `kill` command, as shown in the following output:

```
[root@server1 ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
```

```
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[root@server1 ~]#_
```

Most of the kill signals listed in the preceding output are not useful for systems administrators. The five most common kill signals used for administration are listed in Table 9-2.

Table 9-2 Common administrative kill signals

Name	Number	Description
SIGHUP	1	Also known as the hang-up signal, it stops a process, then restarts it with the same PID. If you edit the configuration file used by a running daemon, that daemon might be sent a SIGHUP to restart the process; when the daemon starts again, it reads the new configuration file.
SIGINT	2	This signal sends an interrupt signal to a process. Although this signal is one of the weakest kill signals, it works most of the time. When you use the Ctrl+c key combination to kill a currently running process, a SIGINT is actually being sent to the process.
SIGQUIT	3	Also known as a core dump, the quit signal terminates a process by taking the process information in memory and saving it to a file called core on the filesystem in the current working directory. You can use the Ctrl+\ key combination to send a SIGQUIT to a process that is currently running.
SIGTERM	15	The software termination signal is the most common kill signal used by programs to kill other processes. It is the default kill signal used by the `kill` command.
SIGKILL	9	Also known as the absolute kill signal, it forces the Linux kernel to stop executing the process by sending the process's resources to a special device file called /dev/null.

To send a kill signal to a process, you specify the kill signal to send as an option to the `kill` command, followed by the appropriate PID of the process. For example, to send a SIGQUIT to a process called `sample`, you could use the following commands to locate and terminate the process:

```
[root@server1 ~]# ps –ef | grep sample
root       1199      1  0 Jun30 tty3     00:00:00 /sbin/sample
[root@server1 ~]# kill -3 1199
[root@server1 ~]#_
[root@server1 ~]# ps –ef | grep sample
[root@server1 ~]#_
```

Note 10

The `kill -SIGQUIT 1199` command does the same thing as the `kill -3 1199` command shown in the preceding output.

Note 11

If you do not specify the kill signal when using the `kill` command, the `kill` command uses the default kill signal, the SIGTERM signal.

Note 12

You can also use the **`pidof` command** to find the PID of a process to use as an argument to the `kill` command. For example, `pidof sample` will return the PID of the sample process.

When sending a kill signal to several processes, it is often easier to locate the process PIDs using the **`pgrep` command**. The `pgrep` command returns a list of PIDs for processes that match a regular expression, or other criteria. For example, to send a SIGQUIT to all processes started by the user bini whose process name starts with the letters `psql`, you could use the following commands to locate and terminate the process:

```
[root@server1 ~]# pgrep -u bini "^psql"
1344
1501
1522
[root@server1 ~]# kill -3 1344 1501 1522
[root@server1 ~]#_
[root@server1 ~]# pgrep -u bini "^psql"
[root@server1 ~]#_
```

Some processes have the ability to ignore, or **trap**, certain kill signals that are sent to them. The only kill signal that cannot be trapped by any process is the SIGKILL. Thus, if a SIGINT, SIGQUIT, and SIGTERM do not terminate a stubborn process, you can use a SIGKILL to terminate it. However, you should only use SIGKILL as a last resort because it prevents a process from closing open files and other resources properly.

Note 13

You can use the **`lsof` (list open files) command** to view the files that a process has open before sending a SIGKILL. For example, to see the files that are used by the process with PID 1399, you can use the `lsof -p 1399` command.

If you send a kill signal to a process that has children, the parent process terminates all of its child processes before terminating itself. Thus, to kill several related processes, you can simply send a kill signal to their parent process. In addition, to kill a zombie process, it is often necessary to send a kill signal to its parent process.

Note 14

To prevent a child process from being terminated when the parent process is terminated, you can start the child process with the **`nohup` command**. For example, executing the `nohup cathena` command within your shell would execute the cathena child process without any association to the parent shell process that started it.

Another command that can be used to send kill signals to processes is the **`killall` command**. The `killall` command works similarly to the `kill` command in that it takes the kill signal as an option; however, it uses the process name to kill instead of the PID. This allows multiple processes of the same name to be killed in one command. An example of using the `killall` command to send a SIGQUIT to multiple `sample` processes is shown in the following output:

```
[root@server1 ~]# ps -ef | grep sample
root       1729      1  0 Jun30 tty3     00:00:00 /sbin/sample
root      20198      1  0 Jun30 tty4     00:00:00 /sbin/sample
[root@server1 ~]# killall -3 sample
[root@server1 ~]#_
[root@server1 ~]# ps -ef | grep sample
[root@server1 ~]#_
```

Note 15

Alternatively, you could use the command `killall -SIGQUIT sample` to do the same as the `killall -3 sample` command used in the preceding output.

Note 16

As with the `kill` command, if you do not specify the kill signal when using the `killall` command, it sends a SIGTERM signal by default.

You can also use the **`pkill` command** to kill processes by process name. However, the `pkill` command allows you to identify process names using regular expressions as well as specify other criteria. For example, the `pkill -u bini -3 "^psql"` command will send a SIGQUIT signal to processes started by the bini user that begin with the letters `psql`.

In addition to the `kill`, `killall`, and `pkill` commands, the `top` command can be used to kill processes. While in the top utility, press the `k` key and supply the appropriate PID and kill signal when prompted.

Process Execution

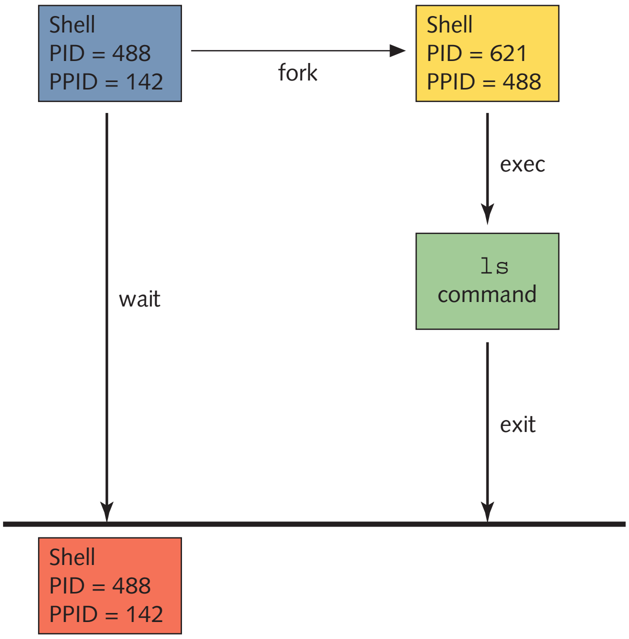
You can execute three main types of Linux commands:

- Binary programs
- Shell scripts
- Shell functions

Most commands, such as `ls`, `find`, and `grep`, are binary programs that exist on the filesystem until executed. They were written in a certain programming language and compiled into a binary format that only the computer can understand. Other commands, such as `cd` and `exit`, are built into the shell running in memory, and they are called shell functions. Shell scripts can also contain a list of binary programs, shell functions, and special constructs for the shell to execute in order.

When executing compiled programs or shell scripts, the shell that interprets the command you typed creates a new shell. This creation of a new subshell is known as **forking** and is carried out by the fork function in the shell. The new subshell then executes the binary program or shell script using its exec function. After the binary program or shell script has completed, the new shell uses its exit function to kill itself and return control to the original shell. The original shell uses its wait function to wait for the new shell to carry out the aforementioned tasks before returning a prompt to the user. Figure 9-3 depicts this process when a user types the `ls` command at the command line.

Figure 9-3 Process forking



Running Processes in the Background

As discussed in the previous section, the shell creates, or forks, a subshell to execute most commands on the Linux system. Unfortunately, the original shell must wait for the command in the subshell to finish before displaying a shell prompt to accept new commands. Commands run in this fashion are known as **foreground processes**.

Alternatively, you can omit the wait function shown in Figure 9-3 by appending an ampersand (`&`) character to the command. Commands run in this fashion are known as **background processes**. When a command is run in the background, the shell immediately returns the shell prompt for the user to enter another command. To run the `sample` command in the background, you can enter the following command:

```
[root@server1 ~]# sample &
[1] 2583
[root@server1 ~]#_
```

Note 17

Space characters between the command and the ampersand (`&`) are optional. In other words, the command `sample&` is equivalent to the command `sample &` used in the preceding output.

The shell returns the PID (2583 in the preceding example) and the background job ID (1 in the preceding example) so that you can manipulate the background job after it has been run. After the process has been started, you can use the `ps` command to view the PID or the **`jobs` command** to view the background job ID, as shown in the following output:

```
[root@server1 ~]# jobs
[1]+  Running                    sample &
[root@server1 ~]# ps | grep sample
2583 tty2     00:00:00 sample
[root@server1 ~]#_
```

To terminate the background process, you can send a kill signal to the PID (as shown earlier in this chapter), or you can send a kill signal to the background job ID. Background job IDs must be prefixed with a `%` character. To send the sample background process created earlier a SIGINT signal, you could use the following `kill` command:

```
[root@server1 ~]# jobs
[1]+  Running                  sample &
[root@server1 ~]# kill -2 %1
[1]+  Interrupt                sample
[root@server1 ~]# jobs
[root@server1 ~]#_
```

Note 18

You can also use the `killall -2 sample` command or the `top` utility to terminate the sample background process used in the preceding example.

After a background process has been started, you can move it to the foreground by using the **`fg` (foreground) command** followed by the background job ID. Similarly, you can pause a foreground process by using the Ctrl+z key combination. You can then send the process to the background with the **`bg` (background) command**. The Ctrl+z key combination assigns the foreground process a background job ID that is then used as an argument to the `bg` command. To start a sample process in the background and move it to the foreground, then pause it and move it to the background again, you can use the following commands:

```
[root@server1 ~]# sample &
[1] 7519
[root@server1 ~]# fg %1
sample

Ctrl+z

[1]+  Stopped                  sample
[root@server1 ~]# bg %1
[1]+ sample &
[root@server1 ~]# jobs
[1]+  Running                  sample &
[root@server1 ~]#_
```

When there are multiple background processes executing in the shell, the `jobs` command indicates the most recent one with a + symbol, and the second most recent one with a – symbol. If you place the % notation in a command without specifying the background job ID, the command operates on the most recent background process. An example of this is shown in the following output, in which four sample processes are started and sent SIGQUIT kill signals using the % notation:

```
[root@server1 ~]# sample &
[1] 7605
[root@server1 ~]# sample2 &
[2] 7613
[root@server1 ~]# sample3 &
[3] 7621
[root@server1 ~]# sample4 &
[4] 7629
[root@server1 ~]# jobs
```

```
[1]   Running                  sample &
[2]   Running                  sample2 &
[3]-  Running                  sample3 &
[4]+  Running                  sample4 &
[root@server1 ~]# kill -3 %
[root@server1 ~]# jobs
[1]   Running                  sample &
[2]-  Running                  sample2 &
[3]+  Running                  sample3 &
[root@server1 ~]# kill -3 %
[root@server1 ~]# jobs
[1]-  Running                  sample &
[2]+  Running                  sample2 &
[root@server1 ~]# kill -3 %
[root@server1 ~]# jobs
[1]+  Running                  sample &
[root@server1 ~]# kill -3 %
[root@server1 ~]# jobs
[root@server1 ~]#_
```

Process Priorities

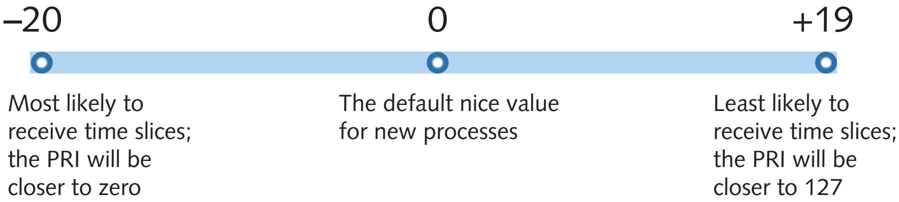
Recall that Linux is a multitasking operating system. That is, it can perform several tasks at the same time. Because most computers contain only a single CPU, Linux executes small amounts of each process on the processor in series. This makes it seem to the user as if processes are executing simultaneously. The amount of time a process has to use the CPU is called a **time slice**; the more time slices a process has, the more time it has to execute on the CPU and the faster it executes. Time slices are typically measured in milliseconds. Thus, several hundred processes can be executing on the processor in a single second.

The `ps -l` command lists the Linux kernel priority (PRI) of a process. This value is directly related to the amount of time slices a process has on the CPU. A PRI of 0 is the most likely to get time slices on the CPU, and a PRI of 127 is the least likely to receive time slices on the CPU. An example of this command is shown next:

```
[root@server1 ~]# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY      TIME CMD
4 S     0  3194  3192  0  75   0 -  1238 wait4  pts/1    00:00:00 bash
4 S     0  3896  3194  0  76   0 -   953 -      pts/1    00:00:00 sleep
4 S     0  3939  3194 13  75   0 -  7015 -      pts/1    00:00:01 gedit
4 R     0  3940  3194  0  77   0 -   632 -      pts/1    00:00:00 ps
[root@server1 ~]#_
```

The `bash`, `sleep`, `gedit`, and `ps` processes all have different PRI values because the kernel automatically assigns time slices based on several factors. You cannot change the PRI directly, but you can influence it indirectly by assigning a certain nice value to a process. A negative nice value increases the likelihood that the process will receive more time slices, whereas a positive nice value does the opposite. The range of nice values is depicted in Figure 9-4.

Figure 9-4 The nice value scale



All users can be "nice" to other users of the same computer by lowering the priority of their own processes by increasing their nice value. However, only the root user has the ability to increase the priority of a process by lowering its nice value.

Processes are started with a nice value of 0 by default, as shown in the NI column of the previous `ps -l` output. To start a process with a nice value of +19 (low priority), you can use the **`nice` command** and specify the nice value using the `-n` option and the command to start. If the `-n` option is omitted, a nice value of +10 is assumed. To start the `ps -l` command with a nice value of +19, you can issue the following command:

```
[root@server1 ~]# nice -n 19 ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY       TIME CMD
4 S     0  3194  3192  0  75   0 -  1238 wait4  pts/1   00:00:00 bash
4 S     0  3896  3194  0  76   0 -   953 -      pts/1   00:00:00 sleep
4 S     0  3939  3194  0  76   0 -  7015 -      pts/1   00:00:02 gedit
4 R     0  3946  3194  0  99  19 -   703 -      pts/1   00:00:00 ps
[root@server1 ~]#_
```

Notice from the preceding output that NI is 19 for the `ps` command, as compared to 0 for the `bash`, `sleep`, and `bash` commands. Furthermore, the PRI of 99 for the `ps` command results in fewer time slices than the PRI of 76 for the `sleep` and `gedit` commands and the PRI of 75 for the `bash` shell.

Conversely, to increase the priority of the `ps -l` command, you can use the following command:

```
[root@server1 ~]# nice -n -20 ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY       TIME CMD
4 S     0  3194  3192  0  75   0 -  1238 wait4  pts/1   00:00:00 bash
4 S     0  3896  3194  0  76   0 -   953 -      pts/1   00:00:00 sleep
4 S     0  3939  3194  0  76   0 -  7015 -      pts/1   00:00:02 gedit
4 R     0  3947  3194  0  60 -20 -   687 -      pts/1   00:00:00 ps
[root@server1 ~]#_
```

Note from the preceding output that the nice value of -20 for the `ps` command resulted in a PRI of 60, which is more likely to receive time slices than the PRI of 76 for the `sleep` and `gedit` commands and the PRI of 75 for the `bash` shell.

Note 19

On some Linux systems, background processes are given a nice value of 4 by default to lower the chance they will receive time slices.

After a process has been started, you can change its priority by using the **`renice` command** and specifying the change to the nice value, as well as the PID of the processes to change. Suppose, for example, three sample processes are currently executing on a terminal:

```
[root@server1 ~]# ps -l
F S  UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY        TIME CMD
4 S    0  1229  1228  0  71   0    -   617 wait4  pts/0  00:00:00 bash
```

```
4 S     0  1990  1229  0  69    0   -   483 nanosl pts/0   00:00:00 sample
4 S     0  2180  1229  0  70    0   -   483 nanosl pts/0   00:00:00 sample
4 S     0  2181  1229  0  71    0   -   483 nanosl pts/0   00:00:00 sample
4 R     0  2196  1229  0  75    0   -   768 -      pts/0   00:00:00 ps
[root@server1 ~]#_
```

To lower priority of the first two sample processes by changing the nice value from 0 to +15 and view the new values, you can execute the following commands:

```
[root@server1 ~]# renice +15 1990 2180
1990 (process ID) old priority 0, new priority 15
2180 (process ID) old priority 0, new priority 15
[root@server1 ~]# ps -l
F S   UID   PID  PPID  C PRI   NI ADDR    SZ WCHAN  TTY          TIME CMD
4 S     0  1229  1228  0  71    0   -   617 wait4  pts/0   00:00:00 bash
4 S     0  1990  1229  0  93   15   -   483 nanosl pts/0   00:00:00 sample
4 S     0  2180  1229  0  96   15   -   483 nanosl pts/0   00:00:00 sample
4 S     0  2181  1229  0  71    0   -   483 nanosl pts/0   00:00:00 sample
4 R     0  2196  1229  0  75    0   -   768 -      pts/0   00:00:00 ps
[root@server1 ~]#_
```

Note 20

You can also use the `top` utility to change the nice value of a running process. Press the `r` key, then supply the PID and the nice value when prompted.

Note 21

As with the `nice` command, only the root user can change the nice value to a negative value using the `renice` command.

The root user can use the `renice` command to change the priority of all processes that are owned by a certain user or group. To change the nice value to +15 for all processes owned by the users mary and bini, you could execute the command `renice +15 -u mary bini` at the command prompt. Similarly, to change the nice value to +15 for all processes started by members of the group sys, you could execute the command `renice +15 -g sys` at the command prompt.

Scheduling Commands

Although most processes are begun by users executing commands while logged in to a terminal, at times you might want to schedule a command to execute at some point in the future. For example, scheduling system maintenance commands to run during nonworking hours is good practice, as it does not disrupt normal business activities.

You can use two different daemons to schedule commands: the **at daemon (atd)** and the **cron daemon (crond)**. The at daemon can be used to schedule a command to execute once in the future, whereas the cron daemon is used to schedule a command to execute repeatedly in the future.

Scheduling Commands with atd

To schedule a command or set of commands for execution at a later time by the at daemon, you can specify the time as an argument to the **`at` command**; some common time formats used with the `at` command are listed in Table 9-3.

Table 9-3 Common `at` commands

Command	Description
`at 10:15pm`	Schedules commands to run at 10:15 PM on the current date
`at 10:15pm July 15`	Schedules commands to run at 10:15 PM on July 15
`at midnight`	Schedules commands to run at midnight on the current date
`at noon July 15`	Schedules commands to run at noon on July 15
`at teatime`	Schedules commands to run at 4:00 PM on the current date
`at tomorrow`	Schedules commands to run the next day
`at now + 5 minutes`	Schedules commands to run in five minutes
`at now + 10 hours`	Schedules commands to run in 10 hours
`at now + 4 days`	Schedules commands to run in four days
`at now + 2 weeks`	Schedules commands to run in two weeks
`at now` `at batch`	Schedules commands to run immediately

`at 9:00am 01/03/2023`
`at 9:00am 01032023`
`at 9:00am 03.01.2023`
	Schedules commands to run at 9:00 AM on January 3, 2023

After being invoked, the `at` command displays an `at>` prompt allowing you to type commands to be executed, one per line. After the commands have been entered, use the Ctrl+d key combination to schedule the commands using atd.

Note 22

The at daemon uses the current shell's environment when executing scheduled commands. The shell environment and scheduled commands are stored in the /var/spool/at directory on Fedora systems and the /var/spool/cron/atjobs directory on Ubuntu systems.

Note 23

If the standard output of any command scheduled using atd has not been redirected to a file, it is normally mailed to the user. You can check your local mail by typing `mail` at a command prompt. Because most modern Linux distributions do not install a mail daemon by default, it is important to ensure that the output of any commands scheduled using atd are redirected to a file.

To schedule the commands `date` and `who` to run at 10:15 PM on July 15, you can use the following commands:

```
[root@server1 ~]# at 10:15pm July 15
at> date > /root/atfile
at> who >> /root/atfile
at> Ctrl+d
job 1 at Wed Jul 15 22:15:00 2023
[root@server1 ~]#_
```

As shown in the preceding output, the at command returns an at job ID. You can use this ID to query or remove the scheduled command. To display a list of at job IDs, you can specify the -l option to the at command:

```
[root@server1 ~]# at -l
1       Wed Jul 15 22:15:00 2023 a root
[root@server1 ~]#_
```

Note 24

Alternatively, you can use the **atq command** to see scheduled at jobs. The atq command is simply a shortcut to the at -l command.

Note 25

When running the at -l command, a regular user only sees their own scheduled at jobs; however, the root user sees all scheduled at jobs.

To see the contents of the at job listed in the previous output alongside the shell environment at the time the at job was scheduled, you can use the -c option to the at command and specify the appropriate at job ID:

```
[root@server1 ~]# at -c 1
#!/bin/sh
# atrun uid=0 gid=0
# mail root 0
umask 22
XDG_VTNR=2; export XDG_VTNR
XDG_SESSION_ID=1; export XDG_SESSION_ID
HOSTNAME=server1; export HOSTNAME
SHELL=/bin/bash; export SHELL
HISTSIZE=1000; export HISTSIZE
QT_GRAPHICSSYSTEM_CHECKED=1; export QT_GRAPHICSSYSTEM_CHECKED
USER=root; export USER
MAIL=/var/spool/mail/root; export MAIL
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root
/bin; export PATH
PWD=/root; export PWD
LANG=en_US.UTF-8; export LANG
KDEDIRS=/usr; export KDEDIRS
HISTCONTROL=ignoredups; export HISTCONTROL
SHLVL=1; export SHLVL
XDG_SEAT=seat0; export XDG_SEAT
HOME=/root; export HOME
LOGNAME=root; export LOGNAME
LESSOPEN=\|\|/usr/bin/lesspipe.sh\ %s; export LESSOPEN
XDG_RUNTIME_DIR=/run/user/0; export XDG_RUNTIME_DIR
cd /root || {
         echo 'Execution directory inaccessible' >&2
         exit 1
}
${SHELL:-/bin/sh} << 'marcinDELIMITER2b2a920e'
date >/root/atfile
who >>/root/atfile
```

```
marcinDELIMITER2b2a920e
[root@server1 ~]#_
```

To remove the at job used in the preceding example, specify the -d option to the at command, followed by the appropriate at job ID, as shown in the following output:

```
[root@server1 ~]# at -d 1
[root@server1 ~]# at -l
[root@server1 ~]#_
```

Note 26

Alternatively, you can use the `atrm 1` command to remove the first at job. The **atrm command** is simply a shortcut to the `at -d` command.

If there are many commands to be scheduled using the at daemon, you can place these commands in a shell script and then schedule the shell script to execute at a later time using the -f option to the at command. An example of scheduling a shell script called myscript using the at command is shown next:

```
[root@server1 ~]# cat myscript
#this is a sample shell script
date > /root/atfile
who >> /root/atfile
[root@server1 ~]# at 10:15pm July 16 -f myscript
job 2 at Wed Jul 15 22:15:00 2023
[root@server1 ~]#_
```

If the /etc/at.allow and /etc/at.deny files do not exist, only the root user is allowed to schedule tasks using the at daemon. To give this ability to other users, create an /etc/at.allow file and add the names of users allowed to use the at daemon, one per line. Conversely, you can use the /etc/at.deny file to deny certain users access to the at daemon; any user not listed in this file is then allowed to use the at daemon. If both files exist, the system checks the /etc/at.allow file and does not process the entries in the /etc/at.deny file.

Note 27

On Fedora systems, only an /etc/at.deny file exists by default. Because this file is initially left blank, all users are allowed to use the at daemon. On Ubuntu systems, only an /etc/at.deny file exists by default, and lists daemon user accounts. As a result, the root user and other regular user accounts are allowed to use the at daemon.

Scheduling Commands with cron

The at daemon is useful for scheduling tasks that occur on a certain date in the future but is ill suited for scheduling repetitive tasks, because each task requires its own at job ID. The cron daemon is better suited for repetitive tasks because it uses configuration files called **cron tables** to specify when a command should be executed.

A cron table includes six fields separated by space or tab characters. The first five fields specify the times to run the command, and the sixth field is the absolute pathname to the command to be executed. As with the at command, you can place commands in a shell script and schedule the shell script to run repetitively; in this case, the sixth field is the absolute pathname to the shell script. Each of the fields in a cron table is depicted in Figure 9-5.

Figure 9-5 User cron table format

1 2 3 4 5 command

1 = Minute past the hour (0–59)
2 = Hour (0–23)
3 = Day of month (1–31)
4 = Month of year (1–12)
5 = Day of week
- 0 = Sun (or 7 = Sun)
- 1 = Mon
- 2 = Tues
- 3 = Wed
- 4 = Thurs
- 5 = Fri
- 6 = Sat

Thus, to execute the /root/myscript shell script at 5:20 PM and 5:40 PM Monday to Friday regardless of the day of the month or month of the year, you could use the cron table depicted in Figure 9-6.

Figure 9-6 Sample user cron table entry

1	2	3	4	5	command
20,40	17	*	*	1–5	/root/myscript

The first field in Figure 9-6 specifies the minute past the hour. Because the command must be run at 20 minutes and 40 minutes past the hour, this field has two values, separated by a comma. The second field specifies the time in 24-hour format, with 5 PM being the 17th hour. The third and fourth fields specify the day of month and month of year, respectively, to run the command. Because the command might run during any month regardless of the day of month, both fields use the * wildcard shell metacharacter to match all values. The final field indicates the day of the week to run the command; as with the first field, the command must be run on multiple days, but a range of days was specified (day 1 to day 5).

Two types of cron tables are used by the cron daemon: user cron tables and system cron tables. User cron tables represent tasks that individual users schedule and exist in the /var/spool/cron directory on Fedora systems and the /var/spool/cron/crontabs directory on Ubuntu systems. System cron tables contain system tasks and exist in the /etc/crontab file as well as the /etc/cron.d directory.

User Cron Tables

On a newly installed Fedora system, all users have the ability to schedule tasks using the cron daemon because the /etc/cron.deny file has no contents. However, if you create an /etc/cron.allow file and add a list of users to it, only those users will be able to schedule tasks using the cron daemon. All other users are denied. Conversely, you can modify the /etc/cron.deny file to list those users who are denied the ability to schedule tasks. Thus, any users not listed in this file are allowed to schedule tasks. If both files exist, only the /etc/cron.allow file is processed. If neither file exists, all users are allowed to schedule tasks, which is the case on a newly installed Ubuntu system.

To create or edit a user cron table, you can use the `–e` option to the **`crontab` command**, which opens the nano editor by default on Fedora systems. You can then enter the appropriate cron table entries. Suppose, for example, that the root user executed the `crontab –e` command on a Fedora system. To schedule /bin/command1 to run at 4:30 AM every Friday and /bin/command2 to run at 2:00 PM on the first day of every month, you can add the following lines while in the nano editor:

```
30 4 * * 5 /bin/command1
0 14 1 * * /bin/command2
```

Note 28

When you run the `crontab -e` command on an Ubuntu system, you are prompted for the editor to use.

When the user saves the changes and quits the nano editor, the information is stored in the file /var/spool/cron/*username*, where *username* is the name of the user who executed the `crontab -e` command. In the preceding example, the file would be named /var/spool/cron/root.

To list your user cron table, you can use the `-l` option to the `crontab` command. The following output lists the cron table created earlier:

```
[root@server1 ~]# crontab -l
30 4 * * 5 /bin/command1
0 14 1 * * /bin/command2
[root@server1 ~]#_
```

Furthermore, to remove a cron table and all scheduled jobs, you can use the `-r` option to the `crontab` command, as illustrated next:

```
[root@server1 ~]# crontab -r
[root@server1 ~]# crontab -l
no crontab for root
[root@server1 ~]#_
```

The root user can edit, list, or remove any other user's cron table by using the `-u` option to the `crontab` command followed by the user name. For example, to edit the cron table for the user mary, the root user could use the command `crontab -e -u mary` at the command prompt. Similarly, to list and remove mary's cron table, the root user could execute the commands `crontab -l -u mary` and `crontab -r -u mary`, respectively.

System Cron Tables

Linux systems are typically scheduled to run many commands during nonbusiness hours. These commands might perform system maintenance, back up data, or run CPU-intensive programs. While Systemd timer units can be configured to run these commands, they are often scheduled by the cron daemon from entries in the system cron table /etc/crontab, which can only be edited by the root user. The default /etc/crontab file on a Fedora system is shown in the following output:

```
[root@server1 ~]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root

# For details see man 4 crontabs

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR
# |  |  |  |  |        sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed

[root@server1 ~]#_
```

The initial section of the cron table specifies the environment used while executing commands. The remainder of the file contains comments that identify the format of a cron table entry. If you add your own cron table entries to the bottom of this file, they will be executed as the root user. Alternatively, you may prefix the command within a system cron table entry with the user account that it should be executed as. For example, the /bin/cleanup command shown in the following output will be executed as the apache user every Friday at 11:30 PM:

```
[root@server1 ~]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root

# For details see man 4 crontabs

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR
# |  |  |  |  |        sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed

 30 23  *  *  5   apache  /bin/cleanup

[root@server1 ~]#_
```

You can also place a cron table with the same information in the /etc/cron.d directory. Any cron tables found in this directory can have the same format as /etc/crontab and are run by the system. In the following example, the cron daemon is configured to run the sa1 command every 10 minutes as the root user, and the sa2 command at 11:53 PM as the root user each day:

```
[root@server1 ~]# cat /etc/cron.d/sysstat
# Run system activity accounting tool every 10 minutes
*/10 * * * * root /usr/lib/sa/sa1 -S DISK 1 1
# Generate a daily summary of process accounting at 23:53
53 23 * * * root /usr/lib/sa/sa2 -A
[root@server1 ~]#_
```

Note 29

The **watch command** can be used instead of the cron daemon for scheduling very frequent tasks. For example, to run the sa1 command shown in the previous output every 1 minute (60 seconds), you could run the `watch -n 60 /usr/lib/sa/sa1 -S DISK 1 1` command.

Many administrative tasks are performed on an hourly, daily, weekly, or monthly basis. If you have a task of this type, you don't need to create a system cron table. Instead, you can place a shell script that runs the appropriate commands in one of the following directories:

- Scripts that should be executed hourly in the /etc/cron.hourly directory
- Scripts that should be executed daily in the /etc/cron.daily directory
- Scripts that should be executed weekly in the /etc/cron.weekly directory
- Scripts that should be executed monthly in the /etc/cron.monthly directory

On Fedora systems, the cron daemon runs the /etc/cron.d/0hourly script, which executes the contents of the /etc/cron.hourly directory 1 minute past the hour, every hour on the hour. The /etc/cron.hourly/0anacron file starts the **anacron daemon**, which then executes the contents of the /etc/cron.daily, /etc/cron.weekly, and /etc/cron.monthly directories at the times specified in /etc/anacrontab.

On Ubuntu systems, cron table entries within the /etc/crontab file are used to execute the contents of the /etc/cron.hourly, as well as the contents of the /etc/cron.daily, /etc/cron.weekly, and /etc/cron.monthly directories using the anacron daemon.

Note 30

If the computer is powered off during the time of a scheduled task, the cron daemon will simply not execute the task. This is why cron tables within the /etc/cron.daily, /etc/cron.weekly, and /etc/cron.monthly directories are often executed by the anacron daemon, which will resume task execution at the next available time if the computer is powered off during the time of a scheduled task.

Summary

- Processes are programs that are executing on the system.
- User processes are run in the same terminal as the user who executed them, whereas daemon processes are system processes that do not run on a terminal.
- Every process has a parent process associated with it and, optionally, several child processes.
- Process information is stored in the /proc filesystem. You can use the `ps`, `pstree`, `pgrep`, and `top` commands to view this information.
- Zombie and rogue processes that exist for long periods of time use up system resources and should be killed to improve system performance.
- You can send kill signals to a process using the `kill`, `killall`, `pkill`, and `top` commands.
- The shell creates, or forks, a subshell to execute most commands.
- Processes can be run in the background by appending an `&` to the command name. The shell assigns each background process a background job ID such that it can be manipulated afterward.
- The priority of a process can be affected indirectly by altering its nice value; nice values range from –20 (high priority) to +19 (low priority). Only the root user can increase the priority of a process.
- You can use the at and cron daemons to schedule commands to run at a later time. The at daemon schedules tasks to occur once at a later time, whereas the cron daemon uses cron tables to schedule tasks to occur repetitively in the future.

Key Terms

anacron daemon
`at` command
at daemon (atd)
`atq` command
`atrm` command
background process
`bg` (background) command
child process
cron daemon (crond)
cron table
`crontab` command
daemon process
`fg` (foreground) command
foreground process
forking
`htop` command
`jobs` command
`kill` command
kill signal
`killall` command
`lsof` (list open files) command
`nice` command
nice value
`nohup` command
parent process
parent process ID (PPID)
`pgrep` command
`pidof` command
`pkill` command
process
process ID (PID)
process priority
process state
program
`ps` command
`pstree` command
`renice` command
rogue process
time slice
`top` command
trap
user process
`watch` command
zombie process