

Script Control

IN THIS CHAPTER

Handling signals

Running scripts in the background

Forbidding hang-ups

Controlling a Job

Modifying script priority

Automating script execution

As you start building advanced scripts, you'll probably wonder how to run and control them on your Linux system. So far in this book, the only way we've run scripts is directly from the command line interface in real-time mode. This isn't the only way to run scripts in Linux. Quite a few options are available for running your shell scripts. There are also options for controlling your scripts. Various control methods include sending signals to your script, modifying a script's priority, and switching the run mode while a script is running. This chapter examines the different ways you can control your shell scripts.

Handling Signals

Linux uses signals to communicate with processes running on the system. Chapter 4 described the different Linux signals and how the Linux system uses these signals to stop, start, and kill processes. You can control the operation of your shell script by programming the script to perform certain commands when it receives specific signals.

Signaling the bash shell

There are more than 30 Linux signals that can be generated by the system and applications. Table 16-1 lists the most common Linux system signals that you'll run across in your shell script writing.

TABLE 16-1 **Linux Signals**

Signal	Value	Description
1	SIGHUP	Hangs up the process
2	SIGINT	Interrupts the process
3	SIGQUIT	Stops the process
9	SIGKILL	Unconditionally terminates the process
15	SIGTERM	Terminates the process if possible
17	SIGSTOP	Unconditionally stops, but doesn't terminate, the process
18	SIGTSTP	Stops or pauses the process, but doesn't terminate
19	SIGCONT	Continues a stopped process

By default, the bash shell ignores any `SIGQUIT` (3) and `SIGTERM` (15) signals it receives (so an interactive shell cannot be accidentally terminated). However, the bash shell does not ignore any `SIGHUP` (1) and `SIGINT` (2) signals it receives.

If the bash shell receives a `SIGHUP` signal, such as when you leave an interactive shell, it exits. Before it exits, however, it passes the `SIGHUP` signal to any processes started by the shell, including any running shell scripts.

With a `SIGINT` signal, the shell is just interrupted. The Linux kernel stops giving the shell processing time on the CPU. When this happens, the shell passes the `SIGINT` signal to any processes started by the shell to notify them of the situation.

As you probably have noticed, the shell passes these signals on to your shell script program for processing. However, a shell script's default behavior does not govern these signals, which may have an adverse effect on the script's operation. To avoid this situation, you can program your script to recognize signals and perform commands to prepare the script for the consequences of the signal.

Generating signals

The bash shell allows you to generate two basic Linux signals using key combinations on the keyboard. This feature comes in handy if you need to stop or pause a runaway script.

Interrupting a process

The `Ctrl+C` key combination generates a `SIGINT` signal and sends it to any processes currently running in the shell. You can test this by running a command that normally takes a long time to finish and pressing the `Ctrl+C` key combination:

```
$ sleep 100
^C
$
```

The Ctrl+C key combination sends a SIGINT signal, which simply stops the current process running in the shell. The `sleep` command pauses the shell's operation for the specified number of seconds and returns the shell prompt. By pressing the Ctrl+C key combination before the time passed, the `sleep` command terminated prematurely.

Pausing a process

Instead of terminating a process, you can pause it in the middle of whatever it's doing. Sometimes, this can be a dangerous thing (for example, if a script has a file lock open on a crucial system file), but often it allows you to peek inside what a script is doing without actually terminating the process.

The Ctrl+Z key combination generates a SIGTSTP signal, stopping any processes running in the shell. Stopping a process is different than terminating the process. Stopping the process leaves the program in memory and able to continue running from where it left off. In the "Controlling the Job" section later in this chapter, you learn how to restart a process that's been stopped.

When you use the Ctrl+Z key combination, the shell informs you that the process has been stopped:

```
$ sleep 100
^Z
[1]+  Stopped                  sleep 100
$
```

The number in the square brackets is the *job number* assigned by the shell. The shell refers to each process running in the shell as a *job* and assigns each job a unique job number within the current shell. It assigns the first started process job number 1, the second job number 2, and so on.

If you have a stopped job assigned to your shell session, bash warns you if you try to exit the shell:

```
$ sleep 100
^Z
[1]+  Stopped                  sleep 100
$ exit
exit
There are stopped jobs.
$
```

You can view the stopped jobs using the `ps` command:

```
$ sleep 100
^Z
[1]+  Stopped                  sleep 100
$
$ ps -l
```

```
F S UID    PID  PPID  C PRI NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 501    2431  2430  0  80  0  - 27118 wait   pts/0 00:00:00 bash
0 T 501    2456  2431  0  80  0  - 25227 signal pts/0 00:00:00 sleep
0 R 501    2458  2431  0  80  0  - 27034 -      pts/0 00:00:00 ps
$
```

In the `S` column (process state), the `ps` command shows the stopped job's state as `T`. This indicates the command is either being traced or is stopped.

If you really want to exit the shell with a stopped job still active, just type the `exit` command again. The shell exits, terminating the stopped job. Alternately, now that you know the PID of the stopped job, you can use the `kill` command to send a `SIGKILL` signal to terminate it:

```
$ kill -9 2456
$
[1]+  Killed                  sleep 100
$
```

When you kill the job, initially you don't get any response. However, the next time you do something that produces a shell prompt (such as pressing the Enter key), you'll see a message indicating that the job was killed. Each time the shell produces a prompt, it also displays the status of any jobs that have changed states in the shell. After you kill a job, the next time you force the shell to produce a prompt, it displays a message showing that the job was killed while running.

Trapping signals

Instead of allowing your script to leave signals ungoverned, you can trap them when they appear and perform other commands. The `trap` command allows you to specify which Linux signals your shell script can watch for and intercept from the shell. If the script receives a signal listed in the `trap` command, it prevents it from being processed by the shell and instead handles it locally.

The format of the `trap` command is:

```
trap commands signals
```

On the `trap` command line, you just list the commands you want the shell to execute, along with a space-separated list of signals you want to trap. You can specify the signals either by their numeric value or by their Linux signal name.

Here's a simple example of using the `trap` command to capture the `SIGINT` signal and govern the script's behavior when the signal is sent:

```
$ cat test1.sh
#!/bin/bash
# Testing signal trapping
#
```

```

trap "echo ' Sorry! I have trapped Ctrl-C'" SIGINT
#
echo This is a test script
#
count=1
while [ $count -le 10 ]
do
    echo "Loop #$count"
    sleep 1
    count=$(( $count + 1 )
done
#
echo "This is the end of the test script"
#

```

The `trap` command used in this example displays a simple text message each time it detects the `SIGINT` signal. Trapping this signal makes this script impervious to the user attempting to stop the program by using the bash shell keyboard `Ctrl+C` command:

```

$ ./test1.sh
This is a test script
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
^C Sorry! I have trapped Ctrl-C
Loop #6
Loop #7
Loop #8
^C Sorry! I have trapped Ctrl-C
Loop #9
Loop #10
This is the end of the test script
$

```

Each time the `Ctrl+C` key combination was used, the script executed the `echo` statement specified in the `trap` command instead of not managing the signal and allowing the shell to stop the script.

Trapping a script exit

Besides trapping signals in your shell script, you can trap them when the shell script exits. This is a convenient way to perform commands just as the shell finishes its job.

To trap the shell script exiting, just add the `EXIT` signal to the `trap` command:

```

$ cat test2.sh
#!/bin/bash

```

```
# Trapping the script exit
#
trap "echo Goodbye..." EXIT
#
count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 1
    count=$(( $count + 1 ))
done
#
$
$ ./test2.sh
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
Goodbye...
$
```

When the script gets to the normal exit point, the trap is triggered, and the shell executes the command you specify on the `trap` command line. The `EXIT` trap also works if you prematurely exit the script:

```
$ ./test2.sh
Loop #1
Loop #2
Loop #3
^CGoodbye...
$
```

Because the `SIGINT` signal isn't listed in the `trap` command list, when the `Ctrl+C` key combination is used to send that signal, the script exits. However, before the script exits, because the `EXIT` is trapped, the shell executes the `trap` command.

Modifying or removing a trap

To handle traps differently in various sections of your shell script, you simply reissue the `trap` command with new options:

```
$ cat test3.sh
#!/bin/bash
# Modifying a set trap
#
trap "echo ' Sorry... Ctrl-C is trapped.'" SIGINT
```

```
#
count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 1
    count=$(( $count + 1 ])
done
#
trap "echo ' I modified the trap!'" SIGINT
#
count=1
while [ $count -le 5 ]
do
    echo "Second Loop #$count"
    sleep 1
    count=$(( $count + 1 ])
done
#
$
```

After the signal trap is modified, the script manages the signal or signals differently. However, if a signal is received before the trap is modified, the script processes it per the original trap command:

```
$ ./test3.sh
Loop #1
Loop #2
Loop #3
^C Sorry... Ctrl-C is trapped.
Loop #4
Loop #5
Second Loop #1
Second Loop #2
^C I modified the trap!
Second Loop #3
Second Loop #4
Second Loop #5
$
```

You can also remove a set trap. Simply add two dashes after the `trap` command and a list of the signals you want to return to default behavior:

```
$ cat test3b.sh
#!/bin/bash
# Removing a set trap
#
trap "echo ' Sorry... Ctrl-C is trapped.'" SIGINT
#
```

```
count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 1
    count=$(( $count + 1 ))
done
#
# Remove the trap
trap -- SIGINT
echo "I just removed the trap"
#
count=1
while [ $count -le 5 ]
do
    echo "Second Loop #$count"
    sleep 1
    count=$(( $count + 1 ))
done
#
$ ./test3b.sh
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
I just removed the trap
Second Loop #1
Second Loop #2
Second Loop #3
^C
$
```

TIP

You can use a single dash instead of a double dash after the `trap` command to return signals to their default behavior. Both the single and double dash work properly.

After the signal trap is removed, the script handles the `SIGINT` signal in its default manner, terminating the script. However, if a signal is received before the trap is removed, the script processes it per the original trap command:

```
$ ./test3b.sh
Loop #1
Loop #2
Loop #3
^C Sorry... Ctrl-C is trapped.
Loop #4
```



```

Loop #5
I just removed the trap
Second Loop #1
Second Loop #2
^C
$

```

In this example, the first Ctrl+C key combination was used to attempt to terminate the script prematurely. Because the signal was received before the trap was removed, the script executed the command specified in the trap. After the script executed the trap removal, then Ctrl+C could prematurely terminate the script.

Running Scripts in Background Mode

Sometimes, running a shell script directly from the command line interface is inconvenient. Some scripts can take a long time to process, and you may not want to tie up the command line interface waiting. While the script is running, you can't do anything else in your terminal session. Fortunately, there's a simple solution to that problem.

When you use the `ps` command, you see a whole bunch of different processes running on the Linux system. Obviously, all these processes are not running on your terminal monitor. This is called running processes in the *background*. In background mode, a process runs without being associated with a `STDIN`, `STDOUT`, and `STDERR` on a terminal session (see Chapter 15).

You can exploit this feature with your shell scripts as well, allowing them to run behind the scenes and not lock up your terminal session. The following sections describe how to run your scripts in background mode on your Linux system.

Running in the background

Running a shell script in background mode is a fairly easy thing to do. To run a shell script in background mode from the command line interface, just place an ampersand symbol (&) after the command:

```

$ cat test4.sh
#!/bin/bash
# Test running in the background
#
count=1
while [ $count -le 10 ]
do
    sleep 1
    count=$(( $count + 1 ])
done

```

```
#
$
$ ./test4.sh &
[1] 3231
$
```

When you place the ampersand symbol after a command, it separates the command from the bash shell and runs it as a separate background process on the system. The first thing that displays is the line:

```
[1] 3231
```

The number in the square brackets is the job number assigned by the shell to the background process. The next number is the Process ID (PID) the Linux system assigns to the process. Every process running on the Linux system must have a unique PID.

As soon as the system displays these items, a new command line interface prompt appears. You are returned to the shell, and the command you executed runs safely in background mode. At this point, you can enter new commands at the prompt.

When the background process finishes, it displays a message on the terminal:

```
[1] Done ./test4.sh
```

This shows the job number and the status of the job (Done), along with the command used to start the job.

Be aware that while the background process is running, it still uses your terminal monitor for STDOUT and STDERR messages:

```
$ cat test5.sh
#!/bin/bash
# Test running in the background with output
#
echo "Start the test script"
count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 5
    count=$(( $count + 1 ))
done
#
echo "Test script is complete"
#
$
$ ./test5.sh &
[1] 3275
```

```

$ Start the test script
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
Test script is complete

[1] Done                      ./test5.sh
$

```

You'll notice from the example that the output from the `test5.sh` script displays. The output intermixes with the shell prompt, which is why `Start the test script` appears next to the `$` prompt.

You can still issue commands while this output is occurring:

```

$ ./test5.sh &
[1] 3319
$ Start the test script
Loop #1
Loop #2
Loop #3
ls myprog*
myprog myprog.c
$ Loop #4
Loop #5
Test script is complete

[1]+ Done                      ./test5.sh
$$

```

While the `test5.sh` script is running in the background, the command `ls myprog*` was entered. The script's output, the typed command, and the command's output all intermixed with each other's output display. This can be confusing! It is a good idea to redirect `STDOUT` and `STDERR` for scripts you will be running in the background (Chapter 15) to avoid this messy output.

Running multiple background jobs

You can start any number of background jobs at the same time from the command line prompt:

```

$ ./test6.sh &
[1] 3568
$ This is Test Script #1

$ ./test7.sh &

```

```
[2] 3570
$ This is Test Script #2

$ ./test8.sh &
[3] 3573
$ And...another Test script

$ ./test9.sh &
[4] 3576
$ Then...there was one more test script

$
```

Each time you start a new job, the Linux system assigns it a new job number and PID. You can see that all the scripts are running using the `ps` command:

```
$ ps
  PID TTY          TIME CMD
 2431 pts/0        00:00:00 bash
 3568 pts/0        00:00:00 test6.sh
 3570 pts/0        00:00:00 test7.sh
 3573 pts/0        00:00:00 test8.sh
 3574 pts/0        00:00:00 sleep
 3575 pts/0        00:00:00 sleep
 3576 pts/0        00:00:00 test9.sh
 3577 pts/0        00:00:00 sleep
 3578 pts/0        00:00:00 sleep
 3579 pts/0        00:00:00 ps
$
```

You must be careful when using background processes from a terminal session. Notice in the output from the `ps` command that each of the background processes is tied to the terminal session (`pts/0`) terminal. If the terminal session exits, the background process also exits.

NOTE

Earlier in this chapter we mentioned that when you attempt to exit a terminal session, a warning is issued if there are stopped processes. However, with background processes, only some terminal emulators remind you that a background job is running, before you attempt to exit the terminal session.

If you want your script to continue running in background mode after you have logged off the console, there's something else you need to do. The next section discusses that process.

Running Scripts without a Hang-Up

Sometimes, you may want to start a shell script from a terminal session and let the script run in background mode until it finishes, even if you exit the terminal session. You can do this by using the `nohup` command.

The `nohup` command runs another command blocking any `SIGHUP` signals that are sent to the process. This prevents the process from exiting when you exit your terminal session.

The format used for the `nohup` command is as follows:

```
$ nohup ./test1.sh &
[1] 3856
$ nohup: ignoring input and appending output to 'nohup.out'

$
```

As with a normal background process, the shell assigns the command a job number, and the Linux system assigns a PID number. The difference is that when you use the `nohup` command, the script ignores any `SIGHUP` signals sent by the terminal session if you close the session.

Because the `nohup` command disassociates the process from the terminal, the process loses the `STDOUT` and `STDERR` output links. To accommodate any output generated by the command, the `nohup` command automatically redirects `STDOUT` and `STDERR` messages to a file, called `nohup.out`.

NOTE

If you run another command using `nohup`, the output is appended to the existing `nohup.out` file. Be careful when running multiple commands from the same directory, because all the output is sent to the same `nohup.out` file, which can get confusing.

The `nohup.out` file contains all the output that would normally be sent to the terminal monitor. After the process finishes running, you can view the `nohup.out` file for the output results:

```
$ cat nohup.out
This is a test script
Loop 1
Loop 2
Loop 3
Loop 4
Loop 5
Loop 6
Loop 7
Loop 8
Loop 9
Loop 10
This is the end of the test script
$
```

The output appears in the `nohup.out` file just as if the process ran on the command line.

Controlling the Job

Earlier in this chapter, you saw how to use the Ctrl+C key combination to stop a job running in the shell. After you stop a job, the Linux system lets you either kill or restart it. You can kill the process by using the `kill` command. Restarting a stopped process requires that you send it a SIGCONT signal.

The function of starting, stopping, killing, and resuming jobs is called *job control*. With job control, you have full control over how processes run in your shell environment. This section describes the commands used to view and control jobs running in your shell.

Viewing jobs

The key command for job control is the `jobs` command. The `jobs` command allows you to view the current jobs being handled by the shell:

```
$ cat test10.sh
#!/bin/bash
# Test job control
#
echo "Script Process ID: $$"
#
count=1
while [ $count -le 10 ]
do
    echo "Loop #$count"
    sleep 10
    count=$(( $count + 1 ))
done
#
echo "End of script..."
#
$
```

The script uses the `$$` variable to display the PID that the Linux system assigns to the script; then it goes into a loop, sleeping for 10 seconds at a time for each iteration.

You can start the script from the command line interface and then stop it using the Ctrl+Z key combination:

```
$ ./test10.sh
Script Process ID: 1897
Loop #1
Loop #2
^Z
[1]+  Stopped                  ./test10.sh
$
```

Using the same script, another job is started as a background process, using the ampersand symbol. To make life a little easier, the output of that script is redirected to a file so it doesn't appear on the screen:

```
$ ./test10.sh > test10.out &
[2] 1917
$
```

The `jobs` command enables you to view the jobs assigned to the shell. The `jobs` command shows both the stopped and the running jobs, along with their job numbers and the commands used in the jobs:

```
$ jobs
[1]+  Stopped                  ./test10.sh
[2]-  Running                  ./test10.sh > test10.out &
$
```

You can view the various jobs' PIDs by adding the `-l` parameter (lowercase L) on the `jobs` command:

```
$ jobs -l
[1]+  1897 Stopped              ./test10.sh
[2]-  1917 Running              ./test10.sh > test10.out &
$
```

The `jobs` command uses a few different command line parameters, as shown in Table 16-2.

TABLE 16-2 The `jobs` Command Parameters

Parameter	Description
<code>-l</code>	Lists the PID of the process along with the job number
<code>-n</code>	Lists only jobs that have changed their status since the last notification from the shell
<code>-p</code>	Lists only the PIDs of the jobs
<code>-r</code>	Lists only the running jobs
<code>-s</code>	Lists only stopped jobs

You probably noticed the plus and minus signs in the `jobs` command output. The job with the plus sign is considered the default job. It would be the job referenced by any job control commands if a job number wasn't specified in the command line.

The job with the minus sign is the job that would become the default job when the current default job finishes processing. There will be only one job with the plus sign and one job with the minus sign at any time, no matter how many jobs are running in the shell.

The following is an example showing how the next job in line takes over the default status, when the default job is removed. Three separate processes are started in the background. The `jobs` command listing shows the three processes, their PID, and their status. Note that the default process (the one listed with the plus sign) is the last process started, job #3.

```
$ ./test10.sh > test10a.out &
[1] 1950
$ ./test10.sh > test10b.out &
[2] 1952
$ ./test10.sh > test10c.out &
[3] 1955
$
$ jobs -l
[1] 1950 Running          ./test10.sh > test10a.out &
[2]- 1952 Running          ./test10.sh > test10b.out &
[3]+ 1955 Running          ./test10.sh > test10c.out &
$
```

Using the `kill` command to send a `SIGHUP` signal to the default process causes the job to terminate. In the next `jobs` listing, the job that previously had the minus sign now has the plus sign and is the default job:

```
$ kill 1955
$
[3]+  Terminated          ./test10.sh > test10c.out
$
$ jobs -l
[1]- 1950 Running          ./test10.sh > test10a.out &
[2]+ 1952 Running          ./test10.sh > test10b.out &
$
$ kill 1952
$
[2]+  Terminated          ./test10.sh > test10b.out
$
$ jobs -l
[1]+ 1950 Running          ./test10.sh > test10a.out &
$
```

Although changing a background job to the default process is interesting, it doesn't seem very useful. In the next section, you learn how to use commands to interact with the default process using no PID or job number.

Restarting stopped jobs

Under bash job control, you can restart any stopped job as either a background process or a foreground process. A foreground process takes over control of the terminal you're working on, so be careful about using that feature.

To restart a job in background mode, use the `bg` command:

```
$ ./test11.sh
^Z
[1]+  Stopped                  ./test11.sh
$
$ bg
[1]+  ./test11.sh &
$
$ jobs
[1]+  Running                  ./test11.sh &
$
```

Because the job was the default job, indicated by the plus sign, only the `bg` command was needed to restart it in background mode. Notice that no PID is listed when the job is moved into background mode.

If you have additional jobs, you need to use the job number along with the `bg` command:

```
$ ./test11.sh
^Z
[1]+  Stopped                  ./test11.sh
$
$ ./test12.sh
^Z
[2]+  Stopped                  ./test12.sh
$
$ bg 2
[2]+  ./test12.sh &
$
$ jobs
[1]+  Stopped                  ./test11.sh
[2]-  Running                  ./test12.sh &
$
```

The command `bg 2` was used to send the second job into background mode. Notice that when the `jobs` command was used, it listed both jobs with their status, even though the default job is not currently in background mode.

To restart a job in foreground mode, use the `fg` command, along with the job number:

```
$ fg 2
./test12.sh
This is the script's end...
$
```

Because the job is running in foreground mode, the command line interface prompt does not appear until the job finishes.

Being Nice

In a multitasking operating system (which Linux is), the kernel is responsible for assigning CPU time for each process running on the system. The *scheduling priority* is the amount of CPU time the kernel assigns to the process relative to the other processes. By default, all processes started from the shell have the same scheduling priority on the Linux system.

The scheduling priority is an integer value, from -20 (the highest priority) to +19 (the lowest priority). By default, the bash shell starts all processes with a scheduling priority of 0.

TIP

It's confusing to remember that -20, the lowest value, is the highest priority and 19, the highest value, is the lowest priority. Just remember the phrase, "Nice guys finish last." The "nicer" or higher you are in value, the lower your chance of getting the CPU.

Sometimes, you want to change the priority of a shell script, either lowering its priority so it doesn't take as much processing power away from other processes or giving it a higher priority so it gets more processing time. You can do this by using the `nice` command.

Using the nice command

The `nice` command allows you to set the scheduling priority of a command as you start it. To make a command run with less priority, just use the `-n` command line option for `nice` to specify a new priority level:

```
$ nice -n 10 ./test4.sh > test4.out &
[1] 4973
$
$ ps -p 4973 -o pid,ppid,ni,cmd
  PID  PPID  NI  CMD
  4973   4721  10  /bin/bash ./test4.sh
$
```

Notice that you must use the `nice` command on the same line as the command you are starting. The output from the `ps` command confirms that the `nice` value (column `NI`) has been set to 10.

The `nice` command causes the script to run at a lower priority. However, if you try to increase the priority of one of your commands, you might be in for a surprise:

```
$ nice -n -10 ./test4.sh > test4.out &
[1] 4985
$ nice: cannot set niceness: Permission denied

[1]+  Done                  nice -n -10 ./test4.sh > test4.out
$
```

The `nice` command prevents normal system users from increasing the priority of their commands. Notice that the job does run, even though the attempt to raise its priority with the `nice` command failed.

You don't have to use the `-n` option with the `nice` command. You can simply type the priority preceded by a dash:

```
$ nice -10 ./test4.sh > test4.out &
[1] 4993
$
$ ps -p 4993 -o pid,ppid,ni,cmd
  PID  PPID  NI  CMD
 4993   4721  10 /bin/bash ./test4.sh
$
```

However, this can get confusing when the priority is a negative number, because you must have a double-dash. It's best just to use the `-n` option to avoid confusion.

Using the `renice` command

Sometimes, you'd like to change the priority of a command that's already running on the system. That's what the `renice` command is for. It allows you to specify the PID of a running process to change its priority:

```
$ ./test11.sh &
[1] 5055
$
$ ps -p 5055 -o pid,ppid,ni,cmd
  PID  PPID  NI  CMD
 5055   4721   0 /bin/bash ./test11.sh
$
$ renice -n 10 -p 5055
5055: old priority 0, new priority 10
$
$ ps -p 5055 -o pid,ppid,ni,cmd
  PID  PPID  NI  CMD
 5055   4721  10 /bin/bash ./test11.sh
$
```

The `renice` command automatically updates the scheduling priority of the running process. As with the `nice` command, the `renice` command has some limitations:

- You can only `renice` processes that you own.
- You can only `renice` your processes to a lower priority.
- The root user can `renice` any process to any priority.

If you want to fully control running processes, you must be logged in as the root account or use the `sudo` command.

Running Like Clockwork

When you start working with scripts, you may want to run a script at a preset time, usually at a time when you're not there. The Linux system provides a couple of ways to run a script at a preselected time: the `at` command and the `cron` table. Each method uses a different technique for scheduling when and how often to run scripts. The following sections describe each of these methods.

Scheduling a job using the `at` command

The `at` command allows you to specify a time when the Linux system will run a script. The `at` command submits a job to a queue with directions on when the shell should run the job. The `at` daemon, `atd`, runs in the background and checks the job queue for jobs to run. Most Linux distributions start this daemon automatically at boot time.

The `atd` daemon checks a special directory on the system (usually `/var/spool/at`) for jobs submitted using the `at` command. By default, the `atd` daemon checks this directory every 60 seconds. When a job is present, the `atd` daemon checks the time the job is set to be run. If the time matches the current time, the `atd` daemon runs the job.

The following sections describe how to use the `at` command to submit jobs to run and how to manage these jobs.

Understanding the `at` command format

The basic `at` command format is pretty simple:

```
at [-f filename] time
```

By default, the `at` command submits input from `STDIN` to the queue. You can specify a filename used to read commands (your script file) using the `-f` parameter.

The `time` parameter specifies when you want the Linux system to run the job. If you specify a time that has already passed, the `at` command runs the job at that time on the next day.

You can get pretty creative with how you specify the time. The `at` command recognizes lots of different time formats:

- A standard hour and minute, such as 10:15
- An AM/PM indicator, such as 10:15PM
- A specific named time, such as now, noon, midnight, or teatime (4PM)

In addition to specifying the time to run the job, you can also include a specific date, using a few different date formats:

- A standard date format, such as MMDDYY, MM/DD/YY, or DD.MM.YY
- A text date, such as Jul 4 or Dec 25, with or without the year
- A time increment:
 - Now + 25 minutes
 - 10:15PM tomorrow
 - 10:15 + 7 days

When you use the `at` command, the job is submitted into a *job queue*. The job queue holds the jobs submitted by the `at` command for processing. There are 26 different job queues available for different priority levels. Job queues are referenced using lowercase letters, *a* through *z*, and uppercase letters *A* through *Z*.

NOTE

A few years ago, the `batch` command was another method that allowed a script to be run at a later time. The `batch` command was unique because you could schedule a script to run when the system was at a lower usage level. However, nowadays, the `batch` command is just simply a script, `/usr/bin/batch`, that calls the `at` command and submits your job to the `b` queue.

The higher alphabetically the job queue, the lower the priority (higher `nice` value) the job will run under. By default, `at` jobs are submitted to the `a` queue. If you want to run a job at a lower priority, you can specify a different queue letter using the `-q` parameter.

Retrieving job output

When the job runs on the Linux system, there's no monitor associated with the job. Instead, the Linux system uses the e-mail address of the user who submitted the job as `STDOUT` and `STDERR`. Any output destined to `STDOUT` or `STDERR` is mailed to the user via the mail system.

Here's a simple example using the `at` command to schedule a job to run on a CentOS distribution:

```
$ cat test13.sh
#!/bin/bash
# Test using at command
#
echo "This script ran at $(date +%B%d,%T)"
echo
sleep 5
echo "This is the script's end..."
#
```

```
$ at -f test13.sh now
job 7 at 2015-07-14 12:38
$
```

The `at` command displays the job number assigned to the job along with the time the job is scheduled to run. The `-f` option tells what script file to use and the `now` time designation directs `at` to run the script immediately.

Using e-mail for the `at` command's output is inconvenient at best. The `at` command sends e-mail via the `sendmail` application. If your system does not use `sendmail`, you won't get any output! Therefore, it's best to redirect `STDOUT` and `STDERR` in your scripts (see Chapter 15) when using the `at` command, as the following example shows:

```
$ cat test13b.sh
#!/bin/bash
# Test using at command
#
echo "This script ran at $(date +%B%d,%T)" > test13b.out
echo >> test13b.out
sleep 5
echo "This is the script's end..." >> test13b.out
#
$
$ at -M -f test13b.sh now
job 8 at 2015-07-14 12:48
$
$ cat test13b.out
This script ran at July14,12:48:18

This is the script's end...
$
```

If you don't want to use e-mail or redirection with `at`, it is best to add the `-M` option to suppress any output generated by jobs using the `at` command.

Listing pending jobs

The `atq` command allows you to view what jobs are pending on the system:

```
$ at -M -f test13b.sh teatime
job 17 at 2015-07-14 16:00
$
$ at -M -f test13b.sh tomorrow
job 18 at 2015-07-15 13:03
$
$ at -M -f test13b.sh 13:30
job 19 at 2015-07-14 13:30
$
$ at -M -f test13b.sh now
```

```

job 20 at 2015-07-14 13:03
$
$ atq
20      2015-07-14 13:03 = Christine
18      2015-07-15 13:03 a Christine
17      2015-07-14 16:00 a Christine
19      2015-07-14 13:30 a Christine
$

```

The job listing shows the job number, the date and time the system will run the job, and the job queue the job is stored in.

Removing jobs

After you know the information about what jobs are pending in the job queues, you can use the `atrm` command to remove a pending job:

```

$ atq
18      2015-07-15 13:03 a Christine
17      2015-07-14 16:00 a Christine
19      2015-07-14 13:30 a Christine
$
$ atrm 18
$
$ atq
17      2015-07-14 16:00 a Christine
19      2015-07-14 13:30 a Christine
$

```

Just specify the job number you want to remove. You can only remove jobs that you submit for execution. You can't remove jobs submitted by others.

Scheduling regular scripts

Using the `at` command to schedule a script to run at a preset time is great, but what if you need that script to run at the same time every day or once a week or once a month? Instead of having to continually submit `at` jobs, you can use another feature of the Linux system.

The Linux system uses the `cron` program to allow you to schedule jobs that need to run on a regular basis. The `cron` program runs in the background and checks special tables, called *cron tables*, for jobs that are scheduled to run.

Looking at the cron table

The `cron` table uses a special format for allowing you to specify when a job should be run. The format for the `cron` table is:

```
min hour dayofmonth month dayofweek command
```

The `cron` table allows you to specify entries as specific values, ranges of values (such as 1–5), or as a wildcard character (the asterisk). For example, if you want to run a command at 10:15 on every day, you would use this `cron` table entry:

```
15 10 * * * command
```

The wildcard character used in the *dayofmonth*, *month*, and *dayofweek* fields indicates that `cron` will execute the command every day of every month at 10:15. To specify a command to run at 4:15 PM every Monday, you would use the following:

```
15 16 * * 1 command
```

You can specify the *dayofweek* entry as either a three-character text value (mon, tue, wed, thu, fri, sat, sun) or as a numeric value, with 0 being Sunday and 6 being Saturday.

Here's another example: to execute a command at 12 noon on the first day of every month, you would use the following format:

```
00 12 1 * * command
```

The *dayofmonth* entry specifies a date value (1–31) for the month.

NOTE

The astute reader might be wondering just how you would be able to set a command to execute on the last day of every month because you can't set the *dayofmonth* value to cover every month. This problem has plagued Linux and Unix programmers, and has spawned quite a few different solutions. A common method is to add an *if-then* statement that uses the `date` command to check if tomorrow's date is 01:

```
00 12 * * * if [ `date +%d -d tomorrow` = 01 ] ; then ; command
```

This checks every day at 12 noon to see if it's the last day of the month, and if so, `cron` runs the command.

The command list must specify the full command pathname or shell script to run. You can add any command line parameters or redirection symbols you like, as a regular command line:

```
15 10 * * * /home/rich/test4.sh > test4out
```

The `cron` program runs the script using the user account that submitted the job. Thus, you must have the proper permissions to access the command and output files specified in the command listing.

Building the cron table

Each system user can have their own `cron` table (including the root user) for running scheduled jobs. Linux provides the `crontab` command for handling the `cron` table. To list an existing `cron` table, use the `-l` parameter:


```
$ crontab -l
no crontab for rich
$
```

By default, each user's `cron` table file doesn't exist. To add entries to your `cron` table, use the `-e` parameter. When you do that, the `crontab` command starts a text editor (see Chapter 10) with the existing `cron` table (or an empty file if it doesn't yet exist).

Viewing cron directories

When you create a script that has less precise execution time needs, it is easier to use one of the pre-configured `cron` script directories. There are four basic directories: hourly, daily, monthly, and weekly.

```
$ ls /etc/cron.*ly
/etc/cron.daily:
cups          makewhatis.cron  prelink        tmpwatch
logrotate     mlocate.cron     readahead.cron

/etc/cron.hourly:
0anacron

/etc/cron.monthly:
readahead-monthly.cron

/etc/cron.weekly:
$
```

Thus, if you have a script that needs to be run one time per day, just copy the script to the daily directory and `cron` executes it each day.

Looking at the anacron program

The only problem with the `cron` program is that it assumes that your Linux system is operational 24 hours a day, 7 days a week. Unless you're running Linux in a server environment, this may not necessarily be true.

If the Linux system is turned off at the time a job is scheduled to run in the `cron` table, the job doesn't run. The `cron` program doesn't retroactively run missed jobs when the system is turned back on. To resolve this issue, many Linux distributions also include the `anacron` program.

If `anacron` determines that a job has missed a scheduled running, it runs the job as soon as possible. This means that if your Linux system is turned off for a few days, when it starts back up, any jobs scheduled to run during the time it was off are automatically run.

This feature is often used for scripts that perform routine log maintenance. If the system is always off when the script should run, the log files would never get trimmed and could

grow to undesirable sizes. With `anacron`, you're guaranteed that the log files are trimmed at least each time the system is started.

The `anacron` program deals only with programs located in the `cron` directories, such as `/etc/cron.monthly`. It uses timestamps to determine if the jobs have been run at the proper scheduled interval. A timestamp file exists for each `cron` directory and is located in `/var/spool/anacron`:

```
$ sudo cat /var/spool/anacron/cron.monthly
20150626
$
```

The `anacron` program has its own table (usually located at `/etc/anacrontab`) to check the job directories:

```
$ sudo cat /etc/anacrontab
# /etc/anacrontab: configuration file for anacron

# See anacron(8) and anacrontab(5) for details.

SHELL=/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
# the maximal random delay added to the base delay of the jobs
RANDOM_DELAY=45
# the jobs will be started during the following hours only
START_HOURS_RANGE=3-22

#period in days   delay in minutes   job-identifier   command
1         5         cron.daily       nice run-parts /etc/cron.daily
7        25        cron.weekly      nice run-parts /etc/cron.weekly
@monthly 45        cron.monthly     nice run-parts /etc/cron.monthly
$
```

The basic format of the `anacron` table is slightly different from that of the `cron` table:

```
period delay identifier command
```

The `period` entry defines how often the jobs should be run, specified in days. The `anacron` program uses this entry to check against the jobs' timestamp file. The `delay` entry specifies how many minutes after the system starts the `anacron` program should run missed scripts. The `command` entry contains the `run-parts` program and a `cron` script directory name. The `run-parts` program is responsible for running any script in the directory passed to it.

Notice that `anacron` does not run the scripts located in `/etc/cron.hourly`. This is because the `anacron` program does not deal with scripts that have execution time needs of less than daily.

The identifier entry is a unique non-blank character string — for example, `cron-weekly`. It is used to uniquely identify the job in log messages and error e-mails.

Starting scripts with a new shell

The ability to run a script every time a user starts a new bash shell (even just when a specific user starts a bash shell) can come in handy. Sometimes, you want to set shell features for a shell session or just ensure that a specific file has been set.

Recall the startup files run when a user logs into the bash shell (covered in detail in Chapter 6). Also, remember that not every distribution has all the startup files. Essentially, the first file found in the following ordered list is run and the rest are ignored:

- `$HOME/.bash_profile`
- `$HOME/.bash_login`
- `$HOME/.profile`

Therefore, you should place any scripts you want run at login time in the first file listed.

The bash shell runs the `.bashrc` file any time a new shell is started. You can test this by adding a simple echo statement to the `.bashrc` file in your home directory and starting a new shell:

```
$ cat .bashrc
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
echo "I'm in a new shell!"
$
$ bash
I'm in a new shell!
$
$ exit
exit
$
```

The `.bashrc` file is also typically run from one of the bash startup files. Because the `.bashrc` file runs both when you log into the bash shell and when you start a bash shell, if you need a script to run in both instances, place your shell script inside this file.

Summary

The Linux system allows you to control your shell scripts by using signals. The bash shell accepts signals and passes them on to any process running under the shell process. Linux signals allow you to easily kill a runaway process or temporarily pause a long-running process.

You can use the `trap` statement in your scripts to catch signals and perform commands. This feature provides a simple way to control whether a user can interrupt your script while it's running.

By default, when you run a script in a terminal session shell, the interactive shell is suspended until the script completes. You can cause a script or command to run in background mode by adding an ampersand sign (&) after the command name. When you run a script or command in background mode, the interactive shell returns, allowing you to continue entering more commands. Any background processes run using this method are still tied to the terminal session. If you exit the terminal session, the background processes also exit.

To prevent this from happening, use the `nohup` command. This command intercepts any signals intended for the command that would stop it — for example, when you exit the terminal session. This allows scripts to continue running in background mode even if you exit the terminal session.

When you move a process to background mode, you can still control what happens to it. The `jobs` command allows you to view processes started from the shell session. After you know the job ID of a background process, you can use the `kill` command to send Linux signals to the process or use the `fg` command to bring the process back to the foreground in the shell session. You can suspend a running foreground process by using the Ctrl+Z key combination and place it back in background mode, using the `bg` command.

The `nice` and `renice` commands allow you to change the priority level of a process. By giving a process a lower priority, you allow the CPU to allocate less time to it. This comes in handy when running long processes that can take lots of CPU time.

In addition to controlling processes while they're running, you can also determine when a process starts on the system. Instead of running a script directly from the command line interface prompt, you can schedule the process to run at an alternative time. You can accomplish this in several different ways. The `at` command enables you to run a script once at a preset time. The `cron` program provides an interface that can run scripts at a regularly scheduled interval.

Finally, the Linux system provides script files for you to use for scheduling your scripts to run whenever a user starts a new bash shell. Similarly, the startup files, such as `.bashrc`, are located in every user's home directory to provide a location to place scripts and commands that run with a new shell.

In the next chapter, we look at how to write script functions. Script functions allow you to write code blocks once and then use them in multiple locations throughout your script.