

CHAPTER 12

Command-Line Master Class, Part 2

In Chapter 11, “Command-Line Master Class, Part 1,” you learned a number of useful commands. This chapter follows up with information about how to link commands together to create new command groups. It also looks at the three most popular Linux text editors: `vim`, `emacs`, and `nano`, as well as the `sed` and `awk` tools. This chapter also covers more commands and command-line interface (CLI) tools to help you become successful and efficient. Let’s jump right in.

Redirecting Output and Input

Sometimes, the output of a command is too long to view on one screen. At other times, there might be an advantage to preserving the output as a file, perhaps to be edited later. You can’t do that using `cat` or `less`, at least not using them as described so far. The good news is that it is possible using redirection.

Commands take their input and give their output in a standard place. This is called *standard input* and *standard output*. By default, this is configured as the output of a keyboard for standard input because it comes in to the computer from the keyboard and the screen for standard output because the computer displays the results of a command to the user using that screen. Standard input and standard output can be redirected.

To redirect output, you use `>` on the command line. Sometimes people read this as “in to.” Here is an example:

```
matthew@seymour:~$ cat /proc/cpuinfo > file.txt
```

IN THIS CHAPTER

- ▶ Redirecting Output and Input
- ▶ `stdin`, `stdout`, `stderr`, and Redirection
- ▶ Comparing Files
- ▶ Limiting Resource Use and Job Control
- ▶ Combining Commands
- ▶ Executing Jobs in Parallel
- ▶ Using Environment Variables
- ▶ Using Common Text Editors
- ▶ Working with Compressed Files
- ▶ Using Multiple Terminals with `byobu`
- ▶ Doing a Polite System Reset using `REISUB`
- ▶ Fixing an Ubuntu System that Will Not Boot
- ▶ Tips and Tricks
- ▶ References

You know the first part of this line reads the information about the CPU from the file `/proc/cpuinfo`. Usually this would print it to the screen, but here the output is redirected into a file called `file.txt`, created in your `/home` directory, because that is the directory in which you issued the command. You can now read, edit, or use this file in any way you like.

CAUTION

Be aware that you can overwrite the contents of a file by using a redirect in this manner, so be certain that either the destination file you name does not exist or that its current contents do not need to be preserved.

What if you want to take the contents of an existing file and use that data as an input to a command? It is as simple as reversing the symbol from `>` to `<`:

```
matthew@seymour:~$ cat < file.txt
```

This displays the contents of `file.txt` on the screen. At first glance, that does not seem useful because the command is doing the same thing it usually does: It is printing the contents of a file to the screen. Perhaps a different example would be helpful.

Ubuntu uses a software packaging system called `apt`, which is discussed in Chapter 9, “Managing Software.” By using a command from the `apt` stable, `dpkg`, you can quickly list all software that has been installed using `apt` on a system and record that info into a file by using a redirect:

```
matthew@seymour:~$ sudo dpkg --get-selections > pkg.list
```

This creates a text file named `pkg.list` that contains the list you want. You can open it with a text editor to confirm this. Then you can use this file as input to `dpkg`, perhaps on another system on which you want exactly the same software to be installed:

```
matthew@seymour:~$ sudo dpkg --set-selections < pkg.list
```

This tells `dpkg` to mark for installation any of the items in the list that are not already installed on the second system. One more quick command (included here for completeness of the example, even though it has nothing to do with redirection), and these will be installed:

```
matthew@seymour:~$ sudo apt-get -u dselect-upgrade
```

Earlier in the chapter you saw an example of using `cat` to display several files simultaneously. This example can be modified slightly to redirect the output into a file, thereby making a new file that includes the contents of the previous two, using the order in which they are listed:

```
matthew@seymour:~$ cat myfile.txt myotherfile.txt > combinedfile.txt
```

If you want to append information to the end of a text file, rather than replace its contents, use two greater-than signs, like this:

```
matthew@seymour:~$ echo "This is a new line being added." >> file.txt
```

If you want to suppress the output that you do not want to keep from a process, so that it does not get sent to standard output or saved, send it instead to a special location called the Null Device, like this, where `verboseprocess` is an example of a process that produces lots of output:

```
matthew@seymour:~$ verboseprocess > /dev/null
```

Add the power of redirection to the information in the next section, and you will begin to understand the potential and the power that a command-line-savvy user has and why so many who learn the command line absolutely love it.

stdin, stdout, stderr, and Redirection

When a program runs, it automatically has three input/output streams opened for it: one for input, one for output, and one for error messages. Typically, these are attached to the user's terminal (so they take input from or give output to the command line), but they can be directed elsewhere, such as to or from a file. These three streams are referred to and abbreviated as shown here, and each of these is assigned a number, shown in the third column:

Stream	Abbreviation	Number
Standard input	<code>stdin</code>	0
Standard output	<code>stdout</code>	1
Standard error, or error stream	<code>stderr</code>	2

In the earlier section "Redirecting Output and Input," you learned how to redirect input and output without needing to know about `stdin` or `stdout`. In addition, you can redirect where the `stderr` messages are sent and also do some more powerful things.

If you are running a program and want any error messages that appear to be directed into a text file instead of having them printed on a screen that might not be monitored, you can use the following command when starting the program or running a command (substitute *program* with the name of the program or command you are running):

```
matthew@seymour:~$ program 2> error.log
```

Here, any error messages from *program* are added to the end of a file named *error.log* in the current working directory.

If you want to redirect both `stderr` and `stdout` to a file, use the following:

```
matthew@seymour:~$ program &> filename
```

You can do the same thing by using the following:

```
matthew@seymour:~$ program >> filename 2>&1
```

In this case, any output from *program* is added to the end of a file named *filename*.

To redirect *stderr* to *stdout*, so that error messages are printed to the screen instead of another location (such as when a program or command is written in a way that already redirects those error messages to a file), use the following:

```
matthew@seymour:~$ program 2>&1
```

Comparing Files

The two things users most commonly want to know when comparing two files are what in the files is the same and what is different. This is especially useful when comparing current versions of configuration files with backup versions of the same files. There are commands to make these tasks easy. Because looking for differences is more common, we start there.

Finding Differences in Files with **diff**

The **diff** command compares files line by line and outputs any lines that are not identical. For example, this command outputs every line that is different between two files:

```
matthew@seymour:~$ diff file1 file2
```

If *file1* and *file2* are different versions of a configuration file—say the current file and a backup—the output quickly tells you what, if anything, has changed. This can help when a config file is automatically updated during an operating system upgrade or when you make a change that doesn't work as well as you had planned and then go back a couple weeks later to change the configuration back.

There are several options you may use when running **diff**. (The original UNIX-style versions, like **-i**, and the newer-style versions, like **--ignore-case**, are identical in what they do; it might simply be easier for you to remember one than the other.) Here are a few of the most useful ones to get you started:

- ▶ **-i** **or** **--ignore-case**—Ignores case differences in file contents
- ▶ **-b** **or** **--ignore-space-change**—Ignores changes in the amount of white space
- ▶ **-w** **or** **--ignore-all-space**—Ignores all white space
- ▶ **-q** **or** **--brief**—Outputs only whether the files differ
- ▶ **-l** **or** **--paginate**—Passes the output through **pr** to paginate it

Finding Similarities in Files with **comm**

The **comm** command compares files line by line and outputs any lines that are identical. For example, this command displays output in three columns, where column 1 shows

lines only in `file1`, column 2 shows lines only in `file2`, and column 3 shows every line that is the same between the two files:

```
matthew@seymour:~$ comm file1 file2
```

This is a much more detailed comparison than with `diff`, and the output can be overwhelming when all you want is to find or check for one or two simple changes. However, it can be incredibly useful when you aren't terribly familiar with either file, and you want to see how they compare.

Fewer options are available when running `comm`. These three are the ones you are most likely to be interested in:

- ▶ `-1`—Suppresses the output of column 1
- ▶ `-2`—Suppresses the output of column 2
- ▶ `-3`—Suppresses the output of column 3

Limiting Resource Use and Job Control

Computer systems run many processes at the same time. This is a good thing and allows users to multitask. Some processes require more system resources than others. Occasionally, a resource-intensive process may take up or require so many resources that it slows down the system for other processes. There are ways to deal with this. This section describes a few of the basics. You must have admin privileges to perform any of the actions in this section.

Listing Processes with `ps`

The `ps` command lists processes and gives you an extraordinary amount of control over its operation. A process is any running program or instance of a running program. There can be many copies of the same program running at the same time, and when that happens, each has its own process. Every process has its own address space, or designated part of the computer's memory that is reserved just for this process and its needs. A process group is created when any process begins, and it includes that process and any processes started by it.

In the UNIX/Linux world, a process (parent) has the ability to create another process (child) that executes some given code independently. This can be really useful for programs that need a lot of time to finish. For example, if you have a program that needs to calculate some complex equation, search large databases, or delete and clean up a lot of files, you can write it so that it will “spawn” a child process that performs the task, while the parent returns control to the user. In such a case, the user does not have to wait for the task to finish because the child process is running in the background.

It is important to know is that `ps` is typically used with what are known as BSD-style parameters. In the section “Finding Files by Searching with `find`” in Chapter 11, “Command-Line Master Class, Part 2,” we discussed UNIX-style, GNU-style, and X-style

parameters (`-c`, `--dosomething`, and `-dosomething`, respectively); BSD-style parameters are different because they use single letters without dashes.

So, the default use of `ps` lists all processes that you are running that are attached to the terminal. However, you can ask it to list all your processes attached to any terminal (or, indeed, no terminal) by adding the `x` parameter: `ps x`. You can ask it to list all processes for all users with the `a` parameter or combine that with `x` to list all processes for all users, attached to a terminal or otherwise: `ps ax`.

However, both of these are timid compared with the almighty `u` option, which enables user-oriented output. In practice, this makes a huge difference because you get important fields such as the username of the owner, how much CPU time and RAM are being used, when the process was started, and more. This outputs a lot of information, so you might want to try adding the `f` parameter, which creates a process forest by using ASCII art to connect parent commands with their children. You can combine all the options so far with this command: `ps faux`. (Yes, with a little imagination, you can spell words with the parameters.)

You can control the order in which the data is returned by using the `--sort` parameter. This takes either `a +` or `a -` (although the `+` is the default) followed by the field you want to sort by: `command`, `%cpu`, `pid`, and `user` are all popular options. If you use the minus sign, the results are reversed. The following command lists all processes, ordered by CPU usage in descending order (with output abridged to conserve space):

```
matthew@seymour:~$ ps aux --sort=-%cpu
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
matthew	17669	10.2	3.8	5256448	1603116	?	Sl	08:30	5:23	/usr/lib/virtua
matthew	6761	4.4	0.0	1226128	17544	?	SNl	Jan26	951:13	conky -d -c /ho
matthew	30849	3.8	0.7	1425084	310356	?	Sl	09:20	0:04	/opt/google/chr
matthew	18668	3.2	5.5	3692116	2297988	?	SLl	Feb05	227:08	/opt/google/chr
matthew	31211	3.0	0.0	698532	39348	?	Sl	09:22	0:00	/usr/bin/xfce4-
matthew	14553	2.8	1.0	2881828	422216	?	Sl	08:17	1:50	/usr/lib/libreo
root	1379	2.3	0.7	780372	304860	tty7	Ssl+	Jan26	490:58	/usr/lib/xorg/X
matthew	31215	0.4	0.0	21952	5524	pts/0	Ss	09:22	0:00	bash
systemd+	1316	0.2	0.0	66008	6284	?	Ss	Jan26	54:00	/lib/systemd/sy
matthew	5369	0.1	0.0	457476	39304	?	SLl	Jan26	31:23	/usr/bin/gnome-
matthew	5589	0.1	0.0	210212	30692	?	S	Jan26	22:40	xfwm4 --replace
matthew	5661	0.1	0.0	1840276	13292	?	S<l	Jan26	26:51	/usr/bin/pulsea
root	1	0.0	0.0	220548	8952	?	Ss	Jan26	0:15	/sbin/init spla
root	2	0.0	0.0	0	0	?	S	Jan26	0:00	[kthreadd]
root	4	0.0	0.0	0	0	?	S<	Jan26	0:00	[kworker/0:0H]
root	7	0.0	0.0	0	0	?	S<	Jan26	0:00	[mm _ percpu _ wq]
root	8	0.0	0.0	0	0	?	S	Jan26	0:08	[ksoftirqd/0]
root	9	0.0	0.0	0	0	?	S	Jan26	5:22	[rcu _ sched]
root	10	0.0	0.0	0	0	?	S	Jan26	0:00	[rcu _ bh]
root	11	0.0	0.0	0	0	?	S	Jan26	0:01	[migration/0]
root	12	0.0	0.0	0	0	?	S	Jan26	0:01	[watchdog/0]
root	13	0.0	0.0	0	0	?	S	Jan26	0:00	[cpuhp/0]

There are many other parameters for `ps`, including a large number of options for compatibility with other UNIXes, all of which can be found in the man page.

Listing Jobs with `jobs`

A *job* is any program you interactively start that doesn't then detach from the user and run on its own (like a daemon does). If you're running an interactive program, you can press Ctrl+Z to suspend it. Then you can start it back in the foreground (using `fg`, covered next) or in the background (using `bg`, covered with `fg`).

While the program is suspended or running in the background, you can start another program. You then have two jobs running. You can also start a program running in the background by appending an `&` like this:

```
matthew@seymour:~$ programname &
```

When started this way, a program runs as a background job. To list all the jobs you are running, you can use `jobs`:

```
matthew@seymour:~$ jobs
```

You can use several useful parameters with `jobs`, including the following:

- ▶ `-l`—Lists the process IDs along with the normal output
- ▶ `-n`—Displays information only about jobs that have changed status since the user was last notified of their status
- ▶ `-r`—Restricts output to running jobs
- ▶ `-s`—Restricts output to stopped jobs

Running One or More Tasks in the Background

Put the `&` (ampersand) symbol at the end of any command to make it run in the background:

```
matthew@seymour:~$ command &
[1] 11423
```

A background process runs without any user input. The shell is not forced to wait until the process is complete before it is freed up to allow more commands to be input and run. When you tell a command to run in the background, you are given its job number in brackets followed by its PID, or process ID number. You can use this to manage the process later, if necessary.

You can input a list of several commands to run in the background. Each will run separately and have its own PID. In this sample, `a`, `b`, and `c` represent commands:

```
matthew@seymour:~$ a & b & c &
[1] 11427
```

```
[2] 11428
[3] 11429
```

You can even use pipes within background processes, and you can combine multiples of each. The letters here represent individual commands:

```
matthew@seymour:~$ d & | e & f & g & | h &
[1] 11432
[2] 11433
[3] 11434
```

Notice that the line above becomes three separate background processes, even though five commands were issued. That is because commands that are piped together are treated as one process.

Moving Jobs to the Background or Foreground with `bg` and `fg`

The shell has the concept of foreground jobs and background jobs. *Foreground jobs* are process groups with control of the terminal, and *background jobs* are process groups without control of the terminal.

Let's say you start a job by running a program at the command line—maybe something like this, which could take a while to run:

```
matthew@seymour:~$ find . -type f -printf "%s\t%p\n" | sort -n
```

When you run this, it starts in the foreground, meaning the terminal is interactive with you only for this job, until the job is complete. This particular job will find all files in the current directory and its subdirectories and list them according to their size. You wouldn't likely want to tie up your terminal session the whole time the job is running. Say that you mean to run it with an `&` at the end so that it will run in the background, but you forget. No worries. You can press `Ctrl+Z` to suspend the job, and then you type this:

```
matthew@seymour:~$ bg
```

That's it. This causes the process to resume, but this time running in the background.

Both `bg` and `fg`, if entered with no further arguments, operate on the job you have most recently interacted with.

Remember that the `jobs` command lists all current jobs and their status (running, stopped, and so on). If you want to move a job running in the background to the foreground, first list the running jobs. Each one has a number next to it in the listing. Use the job number to move a job to the foreground, like this:

```
matthew@seymour:~$ fg %2
```

If you want to move a specific job to the background, just press `Ctrl+Z` and add the job number the same way:

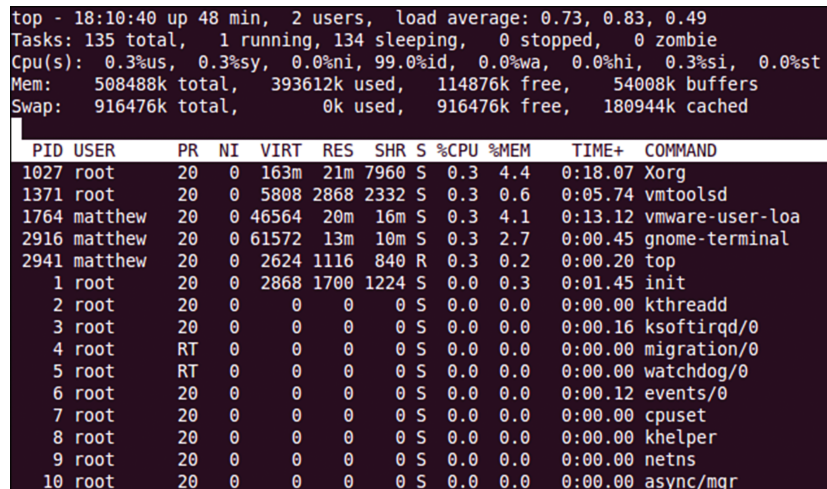
```
matthew@seymour:~$ bg %2
```


Remember that jobs running in this manner terminate when the shell is closed. If you want a job to continue after you exit, you should consider using a tool such as `byobu`, covered later in this chapter, or learn to run the process as a daemon, which is beyond the scope of this chapter and requires you to do some further research.

Printing Resource Usage with `top`

The `top` command is unusual in this list because the few parameters it takes are rarely, if ever, used. Instead, it has a number of commands you can use while it is running to customize the information it shows you. To get the most from these instructions, open two terminal windows. In the first one, run the program `yes` and leave it running; in the second one, run `top`.

When you run `top`, you see a display like the one shown in Figure 12.1.



```
top - 18:10:40 up 48 min, 2 users, load average: 0.73, 0.83, 0.49
Tasks: 135 total, 1 running, 134 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.3%us, 0.3%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 508488k total, 393612k used, 114876k free, 54008k buffers
Swap: 916476k total, 0k used, 916476k free, 180944k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1027	root	20	0	163m	21m	7960	S	0.3	4.4	0:18.07	Xorg
1371	root	20	0	5808	2868	2332	S	0.3	0.6	0:05.74	vmtoolsd
1764	matthew	20	0	46564	20m	16m	S	0.3	4.1	0:13.12	vmware-user-loa
2916	matthew	20	0	61572	13m	10m	S	0.3	2.7	0:00.45	gnome-terminal
2941	matthew	20	0	2624	1116	840	R	0.3	0.2	0:00.20	top
1	root	20	0	2868	1700	1224	S	0.0	0.3	0:01.45	init
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	0:00.16	ksoftirqd/0
4	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
6	root	20	0	0	0	0	S	0.0	0.0	0:00.12	events/0
7	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuset
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	khelper
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	netns
10	root	20	0	0	0	0	S	0.0	0.0	0:00.00	async/mgr

FIGURE 12.1 Use the `top` command to monitor and control processes.

The default sort order in `top` shows the most CPU-intensive tasks first. The first command there should be the `yes` process you just launched from the other terminal, but there should be many others also. Say that you want to filter out all the other users and focus on the user running `yes`. To do this, press `u` and enter the username you used when you ran `yes`. When you press Enter, `top` filters out processes not being run by that user.

The next step is to kill the PID of the `yes` command, so you need to understand what each of the important fields means:

- **PID**—The process ID
- **User**—The owner of the process
- **PR**—Priority
- **NI**—Niceness

- ▶ **virt**—Virtual image size, in kilobytes
- ▶ **res**—Resident size, in kilobytes
- ▶ **shr**—Shared memory size, in kilobytes
- ▶ **s**—Status
- ▶ **%CPU**—CPU usage
- ▶ **%Mem**—Memory usage
- ▶ **Time+**—CPU time
- ▶ **Command**—The command being run

Several of these fields are unimportant unless you have a specific problem. The important ones in this case are **PID**, **User**, **Niceness**, **%CPU**, **%MEM**, **Time+**, and **Command**. The **Niceness** of a process is how much time the CPU allocates to it compared to everything else on the system; 19 is the lowest, and -20 is the highest.

With the columns explained, you should be able to find the PID of the errant **yes** command launched earlier; it is usually the first number below **PID**. Now type **k**, enter that PID, and press Enter. You are prompted for a signal number (the manner in which you want the process killed), with 15 provided as the default. Signal 15 (also known as **SIGTERM**, for *terminate*) is a polite way of asking a process to shut down, and all processes that are not wildly out of control should respond to it. Give **top** a few seconds to update itself, and hopefully the **yes** command should be gone. If not, you need to be more forceful: Type **k** again, enter the PID, and press Enter. When prompted for a signal to send, enter 9 and press Enter to send **SIGKILL**, or “terminate whether you like it or not.”

You can choose the fields to display by pressing **f**. A new screen appears, listing all possible fields, along with the letter you need to press to toggle their visibility. Selected fields are marked with an asterisk and have their letter, as follows:

```
* A: PID          = Process Id
```

If you now press the **a** key, the screen changes to this:

```
a: PID          = Process Id
```

When you have made your selections, press Enter to return to the normal **top** view with your normal column selection.

You can also press **F** to select the field you want to use for sorting. This works the same way as the field selection screen, except that you can select only one field at a time. Again, press Enter to get back to **top** after you have made your selection, and it is updated with the new sorting.

If you press **B**, text bolding is enabled. By default, this bolds some of the header bar as well as any programs that are currently running (as opposed to sleeping), but if you press **x**, you can also enable bolding of the sorted column. You can use **y** to toggle bolding of running processes.

The last command to try is `r`, which enables you to renice—or adjust the niceness value of—a process. You need to enter the PID of the process, press Enter, and enter a new niceness value. Remember that 19 is the lowest and -20 is the highest; anything less than 0 is considered “high” and should be used sparingly.

You can combine the information you learn here with the information in Chapter 16, “System-Monitoring Tools,” for even more power over your system.

Setting Process Priority with `nice`

You can set the priority for individual processes to tell the kernel to either limit or give extra priority to a specific process. This is most useful when multiple concurrent processes are demanding more resources than are actually available, because this is the condition that generally causes slowdowns and bottlenecks. Processes set with a higher priority get a larger portion of the CPU time than lower-priority processes.

You can set the process priority when you first run a program by putting the command `nice` before whatever you are going to run and assigning the process a value that designates its priority. By default, all processes start with a priority of 0 (zero). `nice` can be set that to a maximum of -20, which is the highest priority, to a minimum of 19, the lowest priority. Only root can increase the priority of a process (set negative nice values) but any user can lower the priority of a process (set positive values).

Here is an example that takes the `tar` command used earlier in this chapter and sets its priority very low, because `tar` can demand significant system resources but is often not something whose output you require immediately. You could run the same command as earlier but with `nice` set to 19 to allow you to do something else with the system at the same time:

```
matthew@seymour:~$ sudo nice -n 19 tar czf compressedfilename.tgz directoryname
```

If a process is already running, you can reset the priority (some say “renice it”) by using `renice`. To adjust a specific process, first use `top` to learn the PID for the process and then use `-p PID`, as shown in this example that renices PID 20136 to priority 19:

```
matthew@seymour:~$ sudo renice 19 -p 20136
```

This command is a little more flexible, as it also allows priority adjustments to be made on all processes owned by a specific user or group in the system. Notice that `renice` is most commonly used to lower the priority of a system-slowing task, but it can also be used to bump up the priority for an urgent task, as shown in these examples. Here you first give all tasks by the user `mysql` (using `-u username`) a priority of -20 (top priority, remember?) and then give all tasks belonging to system users in the `website` group (using `-g group-name`) a priority of -20:

```
matthew@seymour:~$ sudo renice -20 -u mysql
matthew@seymour:~$ sudo renice -20 -g website
```

With the `ionice` command, you can adjust priority for disk access, similar to how `nice` and `renice` set priority for CPU access. The difference here is that there are only three class settings for priority. The lowest priority is Idle (3), which permits the process to access the disk only when other processes are not using the disk. In the middle is Best Effort (2), which is the default and allows the kernel to schedule access as its algorithms deem appropriate. The highest priority is Real Time (1), which gives this process the first access to the disk whenever it demands it, regardless of what other processes are running. The Real Time setting can be dangerous as it can cause other processes to lose their data; this isn't guaranteed to happen, but you should consider yourself warned, and you probably want to spend some time studying the man page for `ionice` before you set anything to Real Time.

To use `ionice`, find the PID for a process and use it to set the priority. Notice that there are no spaces between the flag and the value using this command. Here is an example of setting set the priority (using the `-c` flag, for *class*) to 3 for PID 24351 (using the `-p` flag):

```
matthew@seymour:~$ sudo ionice -c3 -p24351
```

You can find other useful tips for managing your system in Chapter 16, “System-Monitoring Tools.”

Combining Commands

So far in this chapter, you have been using commands only individually—and for the most part, that is what you do in practice. However, some of the real power of these commands lies in the capability to join them together to get exactly what you want. There are some extra little commands that we have not looked at that are often used as glue because they do one very simple thing that enables a more complex process to work.

Pipes

All the commands we have looked at have printed their information to the screen, but this is often flexible.

A pipe is a connector between one command's output and another command's input. Instead of sending its output to your terminal, you can use a pipe to send that output directly to another command as input.

Two of the commands we have looked at so far are `ps` and `grep`: the process lister and the string matcher. You can combine the two to find out which users are playing NetHack right now:

```
matthew@seymour:~$ ps aux | grep nethack
```

This creates a list of all the processes running right now and sends that list to the `grep` command, which filters out all lines that do not contain the word `nethack`. Ubuntu allows you to pipe as many commands as you can sanely string together. For example, you could add the `wc` command, which counts the numbers of lines, words, and characters in its input, to count precisely how many times *NetHack* is being run:

```
matthew@seymour:~$ ps aux | grep nethack | wc -l
```

The `-l` (lowercase *L*) parameter to `wc` prints only the line count.

Using pipes in this way is often preferable to using the `-exec` parameter with `find`, simply because many people consider `find` to be a black art, and using it less frequently is better. This is where the `xargs` command comes in: It converts output from one command into arguments for another.

For a good example, consider this mammoth `find` command from earlier:

```
matthew@seymour:~$ find / -name "*.txt" -size +10k -user matthew -not -perm +o=r
[ccc]-exec chmod o+r {} \;
```

This searches every directory from `/` onward for files matching `*.txt` that are greater than 10KB, are owned by user `matthew`, and do not have read permission for others. Then it executes `chmod` on each of the files. It is a complex command, and people who are not familiar with the workings of `find` might have problems understanding it. You can break it into two—a call to `find` and a call to `xargs`. The simplest conversion would look like this:

```
matthew@seymour:~$ find / -name "*.txt" -size +10k -user matthew -not -perm +o=r |
[ccc]xargs chmod o+r
```

This has eliminated the confusing `{ } \;` from the end of the command, but it does the same thing—and faster, too. The speed difference between the two is because using `-exec` with `find` causes it to execute `chmod` once for each file. However, `chmod` accepts many files at a time and, because you are using the same parameter each time, you should take advantage of that. The second command, using `xargs`, is called once with all the output from `find`, and it therefore saves many command calls. The `xargs` command automatically places the input at the end of the line, so the previous command might look something like this:

```
matthew@seymour:~$ xargs chmod o+r file1.txt file2.txt file3.txt
```

Not every command accepts multiple files, though, and if you specify the `-l` parameter, `xargs` executes its command once for each line in its input. If you want to check what it is doing, use the `-p` parameter to have `xargs` prompt you before executing each command.

For even more control, the `-i` parameter allows you to specify exactly where the matching lines should be placed in your command. This is important if you need the lines to appear before the end of the command or need it to appear more than once. Either way, using the `-i` parameter also enables the `-l` parameter so each line is sent to the command individually. The following command finds all files in `/home/matthew` that are larger than 10,000KB (10MB) and copies them to `/home/matthew/archive`:

```
matthew@seymour:~$ find /home/matthew -size +10000k | xargs -i cp {} ./home/
matthew/
[ccc]archive
```

Using `find` with `xargs` is a unique case. All too often, people use pipes when parameters would do the job just as well. For example, the following two commands are identical:

```
matthew@seymour:~$ ps aux --sort=-%cpu | grep -v `whoami`
matthew@seymour:~$ ps -N ux --sort=-%cpu
```

The former prints all users and processes and then pipes that to `grep`, which in turn filters out all lines that contain the output from the program `whoami` (the username in this case). So, the first line prints all processes being run by other users, sorted by CPU use. The second line does not specify the `a` parameter to `ps`, which makes it list only your parameters. It then uses the `-N` parameter to flip that, which means it is everyone but you, without the need for `grep`.

The reason people use the former is often just simplicity: Many people know only a handful of parameters to each command, so they can string together two commands simply rather than write one command properly. Unless the command is to be run regularly, this is not a problem. Indeed, the first line would be better because it does not drive people to the manual pages to find out what `ps -N` does.

You can string together any commands that use standard input and output formats. Another useful example is the following series, which verifies the installation of a named software package (in this example, a search for FTP-related software):

```
matthew@seymour:~$ dpkg --get-selections | grep ftp | sort
ftp                install
lftp               install
```

Here, `dpkg` is being told to list all installed packages, `grep` is searching that list for any line containing `ftp`, and `sort` is sorting alphabetically (which is not vital in this two-line example, but it is really useful if you have a large number of results).

Combining Commands with Boolean Operators

If you want to run a second command only if the first command is successfully completed, you can. Every command you issue to the system outputs an exit status: `0` for true (successful) and `1` for false (failed). The system receives these even if they are not displayed to the user. The `&&` operator, when added to the end of a command, reads that exit status and confirms its value as `0` for true before allowing the next command to be run. Again, the letters represent commands:

```
matthew@seymour:~$ i && k
```

You can do exactly the opposite with `||`, which runs the following command only if the first one returns an exit status of `1` for false:

```
matthew@seymour:~$ m || n
```

Running Separate Commands in Sequence

If you want to have a set of commands run in order but not use the output from one as the input to the next one, you can. Separating commands with a `;` (semicolon) causes the

system to treat each item as if it were entered on a new line after the previous item finished running. Let's say you have three commands: `doctor`, `rose`, and `tardis`. You could run each in order by using this command:

```
matthew@seymour:~$ doctor ; rose ; tardis
```

Note that the spaces before and after the semicolons are optional, but they do make the line easier to read.

12

Process Substitution

Sometimes the output of one or more commands is precisely what you want to use as the input to another command. You can use output redirection for this purpose, using what we call *process substitution*. In process substitution, you surround one or more commands with `()` and precede each list with a `<`. When you do this, do *not* insert a space between the `<` and the opening `(`. The resulting command looks like this:

```
matthew@seymour:~$ cat <(ls -al)
```

This first example is really the same as `ls -al | cat`. With only the output of one process being involved, it doesn't seem worth learning an additional command.

In the following example, you take the output of two `ls` commands as input to a `diff` command to compare the contents of two directories:

```
matthew@seymour:~$ diff <(ls firstdirectory) <(ls seconddirectory)
```

This is faster because you don't have to wait for temporary files to be written and then read; it saves both disk space and the time needed to clean up temporary files. One especially neat advantage of doing this is that `bash` automatically runs the multiple tasks being used as input in parallel, which is faster than doing them sequentially with redirects, like this:

```
matthew@seymour:~$ ls firstdirectory > file1.txt
matthew@seymour:~$ ls seconddirectory > file2.txt
matthew@seymour:~$ diff file1.txt file2.txt
```

Executing Jobs in Parallel

When you issue a command in the terminal, it is executed by the CPU (the central processing unit, or processor). Common processors include Intel's i3, i5, i7, and Xeon series as well as AMD's Ryzen and Epyc. Most computers today have more than one processor (more accurately stated as "processor core," each of which essentially functions as a separate processor). This allows multiple jobs to run at the same time. Programs that are written to take advantage of multiple processors or cores concurrently can run much faster by splitting large tasks into chunks and sending the chunks to multiple processors or cores in parallel.

Most terminal commands are not (yet?) written to take advantage of multiple processors or cores. So, when you execute a command, generally the command uses whatever

percentage it requires of one core until the task is complete. Doing something that requires significant amounts of processing can take a long time.

GNU `parallel` is a shell tool for executing jobs in parallel across multiple processors, cores, or even multiple connected computers. The jobs can be simple single commands or even scripts, as long as they are executables or functions. The tool is powerful and complex and deserves a book of its own. For more information, see the official documentation at www.gnu.org/software/parallel/ and the official tutorial at www.gnu.org/software/parallel/parallel_tutorial.html.

Using Environment Variables

A number of in-memory variables are assigned and loaded by default when you log in. These variables, known as *environment variables*, can be used by various commands to get information about your environment, such as the type of system you are running, your `/home` directory, and the shell in use. Environment variables are used to help tailor the computing environment of your system and include helpful specifications and setup, such as default locations of executable files and software libraries. If you begin writing shell scripts, you might use environment variables in your scripts. Until then, you need to be aware only of what environment variables are and do.

The following list includes a number of environment variables, along with descriptions of how the shell uses them:

- ▶ **PWD**—Provides the full path to your current working directory, used by the `pwd` command, such as `/home/matthew/Documents`
- ▶ **USER**—Declares the user's name, such as `matthew`
- ▶ **LANG**—Sets the default language, such as `English`
- ▶ **SHELL**—Declares the name and location of the current shell, such as `/bin/bash`
- ▶ **PATH**—Sets the default locations of executable files, such as `/bin`, `/usr/bin`, and so on
- ▶ **TERM**—Sets the type of terminal in use, such as `vt100`, which can be important when using screen-oriented programs, such as text editors

You can print the current value of any environment variable by using `echo $VARIABLE-NAME`, like this:

```
matthew@seymour:~$ echo $USER
matthew
matthew@seymour:~$
```

NOTE

Each shell can have its own feature set and language syntax, as well as a unique set of default environment variables.

You can use the `env` or `printenv` command to display all environment variables, as follows:

```
matthew@seymour:~$ env
ORBIT_SOCKETDIR=/tmp/orbit-matthew
SSH_AGENT_PID=1729
TERM=xterm
SHELL=/bin/bash
WINDOWID=71303173
GNOME_KEYRING_CONTROL=/tmp/keyring-qTEFTw
GTK_MODULES=canberra-gtk-module
USER=matt
hew
SSH_AUTH_SOCK=/tmp/keyring-qTEFTw/ssh
DEFAULTS_PATH=/usr/share/gconf/gnome.default.path
SESSION_MANAGER=local/seymour:/tmp/.ICE-unix/1695
USERNAME=matthew
XDG_CONFIG_DIRS=/etc/xdg/xdg-gnome:/etc/xdg
DESKTOP_SESSION=gnome
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/matthew
hew
GDM_KEYBOARD_LAYOUT=us
LANG=en_US.utf8
GNOME_KEYRING_PID=1677
MANDATORY_PATH=/usr/share/gconf/gnome.mandatory.path
GDM_LANG=en_US.utf8
GDMSESSION=gnome
HISTCONTROL=ignoreboth
SPEECHD_PORT=7560
SHLVL=1
HOME=/home/matthew
hew
LOGNAME=matthew
hew
LESSOPEN=| /usr/bin/lesspipe %s
DISPLAY=:0.0
LESSCLOSE=/usr/bin/lesspipe %s %s
XAUTHORITY=/var/run/gdm/auth-for-matthew-PzcGqF/database
COLORTERM=gnome-terminal
OLDPWD=/var/lib/mlocate
_=/usr/bin/env
```

This abbreviated list shows some of the common variables, which are set by configuration, or *resource*, files contained in `/etc` and `/etc/skel` and are copied to the user's `/home` directory when it is created. You can find default settings for `bash`, for example, in `/etc/`

profile and `/etc/bashrc` as well as `.bashrc` or `.bash_profile` files in your `/home` directory. Read the man page for `bash` for details about using these configuration files.

One of the most important environment variables is `$PATH`, which defines the location of executable files. For example, if, as a regular user, you try to use a command that is not located in your `$PATH` (such as the imaginary `command` command), you see something like this:

```
matthew@seymour:~$ command
-bash: command: command not found
```

If the command you're trying to execute exists in the Ubuntu software repositories but is not yet installed on your system, Ubuntu responds with the correct command to install the command:

```
matthew@seymour:~$ command
```

The program 'command' is currently not installed. You can install it by typing:

```
sudo apt-get install command
```

However, you might know that the command is definitely installed on your system, and you can verify it by using the `whereis` command, like this:

```
matthew@seymour:~$ whereis command
command: /sbin/command
```

You can also run the command by typing its full pathname or complete directory specification, as follows:

```
matthew@seymour:~$ sudo /sbin/command
```

As you can see in this example, the `command` command is indeed installed. What happened is that by default, the `/sbin` directory is not in your `$PATH`. One of the reasons for this is that commands under the `/sbin` directory are normally intended to be run only by root. You can add `/sbin` to your `$PATH` by editing the file `.bash_profile` in your `/home` directory (if you use the `bash` shell by default, as most Linux users). Look for the following line:

```
PATH=$PATH:$HOME/bin
```

You can then edit this file, perhaps using one of the text editors discussed later in this chapter, to add the `/sbin` directory, like so:

```
PATH=$PATH:/sbin:$HOME/bin
```

Save the file. The next time you log in, the `/sbin` directory is in your `$PATH`. One way to use this change right away is to read in the new settings in `.bash_profile` by using the `bash` shell's `source` command, as follows:

```
matthew@seymour:~$ source .bash_profile
```

You can now run commands located in the `/sbin` directory without the need to explicitly type the full pathname.

Some Linux commands also use environment variables to acquire configuration information (such as a communications program looking for a variable such as `BAUD_RATE`, which might denote a default modem speed).

To experiment with the environment variables, you can modify the `PS1` variable to manipulate the appearance of your shell prompt. If you are working with `bash`, you can use its built-in `export` command to change the shell prompt. For example, say that your default shell prompt looks like this:

```
matthew@seymour:~$
```

You can change its appearance by using the `PS1` variable, like this:

```
matthew@seymour:~$ export PS1='$OSTYPE r00lz ->'
```

After you press Enter, you see the following:

```
linux-gnu r00lz ->
```

NOTE

See the `bash` man page for other variables you can use for prompt settings.

Using Common Text Editors

Linux distributions include a number of applications known as *text editors* that you can use to create text files or edit system configuration files. Text editors are similar to word processing programs but generally have fewer features, work only with text files, and might or might not support spell checking or formatting. Text editors range in features and ease of use and are found on nearly every Linux distribution. The number of editors installed on a system depends on what software packages you've installed on the system.

The most popular console-based text editors include the following:

- ▶ **emacs**—The comprehensive GNU `emacs` editing environment, which is much more than an editor; see the section “Working with `emacs`,” later in this chapter
- ▶ **nano**—A simple text editor similar to the classic `pico` text editor that was included with the once-common `pine` email program
- ▶ **vim**—An improved compatible version of the `vi` text editor (which we call `vi` in the rest of this chapter because it has a symbolic link named `vi` and a symbolically linked manual page)

Note that not all text editors are *screen oriented*, meaning designed for use from a terminal. The following are some of the text editors designed to run from a graphical desktop and that provides a graphical interface with menu bars, buttons, scrollbars, and so on:

- ▶ **gedit**—A GUI text editor for GNOME, which is installed by default with Ubuntu
- ▶ **kate**—A simple KDE text editor
- ▶ **kedit**—Another simple KDE text editor

A good reason to learn how to use a text-based editor, such as `vi` or `nano`, is that system maintenance and recovery operations almost never take place during GUI sessions, negating the use of a GUI editor. Many larger, more complex and capable editors do not work when Linux is booted to its single-user or maintenance mode. If anything goes wrong with your system, and you can't log in to the GUI, knowledge and experience of using both the command line and text editors will be very important. Make a point of opening some of the editors and playing around with them. You never know; you might just thank us someday.

Another reason to learn how to use a text-based editor under the Linux console mode is so that you can edit text files through remote shell sessions because many servers do not host graphical desktops.

NOTE

Before you take the time to get familiar with a nonstandard text editor, consider this: All three of the editors discussed here are readily available and common. Two of them, `nano` and `vi`, are almost universally installed. If you spend your time learning a nonstandard editor, you will find yourself having to install it on every system or fighting against the software that is already there instead of using your time productively. Feel free to use any text editor you prefer, but we strongly recommend that you make sure you have at least a basic working knowledge of these standard editors so that you can walk up to any system and start working when necessary.

Working with `nano`

We discuss `nano` first because it has the easiest learning curve. It is neither the most powerful nor the most “guru approved,” but `nano` is a respectable text editor that you can run from the command line, and it's often perfect for quick tasks such as editing configuration files.

Learning how to use `nano` is quick and easy. You might need to edit files on a Linux system with a minimal install or a remote server without a more extensive offering of installed text editors. Chances are nearly 100 percent that `nano` will be available.

You can start an editing session by using the `nano` command like this:

```
matthew@seymour:~$ nano file.txt
```

When you first start editing, you see the text on the screen with a title bar across the top and a list of simple commands across the bottom. The editor is simple enough that you can use it without any instruction. Here are the basic commands, just so you can compare them with other editors discussed here:

- ▶ **Cursor movement**—Arrow keys (left, down, up, and right), Page Up and Page Down keys, or Ctrl+Y and Ctrl+V page up and down
- ▶ **Add characters**—Type at the cursor location
- ▶ **Delete character**—Backspace or Delete
- ▶ **Exit**—Ctrl+X (prompts to ask whether to save changes)
- ▶ **Get Help**—Ctrl+G

NOTE

`nano` really is very easy to use, but this does not mean it cannot be used by power users. Take a little time and read the contents of Help to discover some of the interesting and powerful capabilities of this editor.

Working with `vi`

The one editor found on nearly every UNIX and Linux system is the `vi` editor, originally written by Bill Joy. This simple-to-use but incredibly capable editor features a somewhat cryptic command set, but you can put it to use with only a few commands. Although many experienced UNIX and Linux users use `vi` extensively during computing sessions, many users who do only quick and simple editing might not need all its power and might prefer an easier-to-use text editor such as `nano`. Diehard GNU fans and programmers often use `emacs` for pretty much everything.

However, learning how to use `vi` is a good idea. You might need to edit files on a Linux system with a minimal install or on a remote server without a more extensive offering of installed text editors. Chances are nearly 100 percent that `vi` will be available.

You can start an editing session by using the `vi` command like this:

```
matthew@seymour:~$ vi file.txt
```

The `vi` command works by using an insert (or editing) mode and a viewing (or command) mode.

When you first start editing, you are in the viewing mode. You can use your arrow or other navigation keys (as shown later) to scroll through the text. To start editing, press the `i` key to insert text or the `a` key to append text. When you're finished, use the Esc key to toggle out of the insert or append modes and into the viewing (or command) mode. To enter a command, type a colon (:), followed by the command, such as `w` to write the file, and press Enter.

Although `vi` supports many complex editing operations and numerous commands, you can accomplish work by using a few basic commands. These basic `vi` commands are as follows:

- ▶ **Cursor movement**—`h, j, k, l` (left, down, up, and right)
- ▶ **Delete character**—`x`
- ▶ **Delete line**—`dd`
- ▶ **Mode toggle**—`Esc`, Insert (or `i`)
- ▶ **Quit**—`:q`
- ▶ **Quit without saving**—`:q!`
- ▶ **Run a shell command**—`:sh` (use `'exit'` to return)
- ▶ **Save file**—`:w`
- ▶ **Text search**—`/`

NOTE

Use the `vimtutor` command to quickly learn how to use `vi`'s keyboard commands. The tutorial takes less than 30 minutes, and it teaches new users how to start or stop the editor; navigate files; insert and delete text; and perform search, replace, and insert operations.

Working with `emacs`

Richard M. Stallman's GNU `emacs` editor, like `vi`, is available with Ubuntu and nearly every other Linux distribution. Unlike other UNIX and Linux text editors, `emacs` is much more than a simple text editor. It's an editing environment, and you can use it to compile and build programs and act as an electronic diary, appointment book, and calendar. Use it to compose and send email, read Usenet news, and even play games. The reason for this capability is that `emacs` contains a built-in language interpreter that uses the Lisp (`emacs` LISP) programming language. `emacs` is not installed in Ubuntu by default. To use `emacs`, the package you need to install is called `emacs`. See Chapter 9, "Managing Software."

You can start an `emacs` editing session like this:

```
matthew@seymour:~$ emacs file.txt
```

TIP

If you start `emacs` when using X11, the editor launches in its own floating window. To force `emacs` to display inside a terminal window instead of its own window (which can be useful if the window is a login at a remote computer), use the `-nw` command-line option like this: `emacs -nw file.txt`.

The `emacs` editor uses an extensive set of keystroke and named commands, but you can work with it by using a basic command subset. Many of these basic commands require you to hold down the `Ctrl` key, or to first press a *meta* key (generally mapped to the `Alt` key). The basic `emacs` commands are listed as follows:

- ▶ **Cursor left**—`Ctrl+B`
- ▶ **Cursor down**—`Ctrl+N`
- ▶ **Cursor right**—`Ctrl+F`
- ▶ **Cursor up**—`Ctrl+P`
- ▶ **Delete character**—`Ctrl+D`
- ▶ **Delete line**—`Ctrl+K`
- ▶ **Go to start of line**—`Ctrl+A`
- ▶ **Go to end of line**—`Ctrl+E`
- ▶ **Help**—`Ctrl+H`
- ▶ **Quit**—`Ctrl+X`, `Ctrl+C`
- ▶ **Save as**—`Ctrl+X`, `Ctrl+W`
- ▶ **Save file**—`Ctrl+X`, `Ctrl+S`
- ▶ **Search backward**—`Ctrl+R`
- ▶ **Search forward**—`Ctrl+S`
- ▶ **Start tutorial**—`Ctrl+H`, `T`
- ▶ **Undo**—`Ctrl+X`, `U`

One of the best reasons to learn how to use `emacs` is that you can use nearly all the same keystrokes to edit commands on the `bash` shell command line, although it is possible to change the default to use `vi` key bindings. Another reason is that like `vi`, `emacs` is universally available for installation on nearly every UNIX and Linux system, including Apple's macOS.

Working with `sed` and `awk`

`sed`, which is short for *stream editor*, is a command that is used to perform transformations on text. It works from the command line and processes text via standard in and standard out. It does not modify the original input and does not save the output unless you redirect that output to a file. It is most useful this way or when piped between other commands.

`awk` is a small programming language for processing strings. It takes in text, transforms it in whatever way you tell it to, and then outputs the transformed text. It doesn't do as much as other languages, but what it does do it does with elegance and speed.

Both `sed` and `awk` run from the command line. There is a lot to them—more than can be covered in a short section of one chapter of a long book—but you can learn enough about each in just a few minutes to find them useful. If you want to continue learning about them, great resources are available online and in the aptly named book *sed & awk* by Dale Dougherty and Arnold Robbins.

`sed` and `awk` aren't used much anymore, at least not by people who have entered the profession in the twenty-first century, but they are beloved by those who take the time to learn them. For this edition of the book, we are including only a brief mention and a couple quick examples—certainly not enough to really learn to use either in a large capacity. If there is significant interest, we may break this out and expand it into a separate chapter in a future edition.

You use `sed` with `sed` commands, like this:

```
matthew@seymour:~$ sed sedcommand inputfile
matthew@seymour:~$ sed -e sedcommand inputfile
matthew@seymour:~$ sed -e sedcommand -e anothersedcommand inputfile
```

The second example does the same thing as the first example, except that it specifically denotes the command to run. This is optional when there is only one command, but it is useful when you want to run other commands, as in the third example.

Let's say you want to change every instance of *camel* in the text file `transportation.txt` to *dune buggy*. Here is how you do that:

```
matthew@seymour:~$ sed -e 's/camel/dune buggy/g' transportation.txt
```

The `s` command stands for *substitution*. It is surrounded by `'` marks to prevent confusion with spaces or other characters in the strings it contains. The `s` is followed by `/` (slash), which is a delimiter separating the parts of the command from one another. Next is the pattern to match, followed by what it will be changed to. Finally, the letter `g` means to replace it globally, or everywhere in the text file that *camel* occurs.

You can process text based on line numbers in the file. If you wanted to delete lines 4 through 17 in the file `longtext.txt`, you would do this:

```
matthew@seymour:~$ sed -e '4,17d' longtext.txt
```

The characters used for `sed` commands are generally obvious, like the `d` in this example standing for *delete*. You can use a regular expression in place of the line numbers. You can also use other commands besides substitute and delete. You can use `sed` in scripts and chain commands together.

The most common use of `awk` is to manipulate files that consist of fields separated by delimiters, such as a comma-separated values (CSV) file output from a spreadsheet program or a configuration file that assigns default values to program variables.

You define the delimiter that `awk` will look for, and it then assigns an internal `awk` variable to each item on a line. That is, if you have a set of parameters and values in a file where

each parameter is listed, followed by an equals sign as the delimiter, then a value, you define this for `awk` in your command, and the parameter will be assigned, for example, as `$1` and the value as `$2`.

Most files contain more than lists of parameters and values, though. What if you had a comma-delimited file containing names of things on your desk, a category for each, a color for each, and a date corresponding to the last time you picked up each item? That is four columns: name, category, color, and date. If you only really cared about the names and dates, you could use `awk` to process the file quickly and list just these, like this:

```
matthew@seymour:~$ awk -F',' '{print $1, "was last picked up on", $4}' desk-
stuff.txt
```

The output would be displayed on the screen (but could be redirected to a file) and would contain a list of only the information you wanted. In the preceding command, `-F` defines the delimiter, which is placed in `'` marks, and the pair of `{ }` within a set of `'` marks defines what to output—first variable 1, then the text `was last picked up on`, followed by variable 4. At the end, the text file to process is named.

You can define multiple delimiters by using `[]`, like this: `-F'[;,-]'`. You can adjust how to format the text that is output. You can output placeholders when there are blank variables. You can place several `awk` statements in a file and then run it as if it were a shell script.

To close this introduction, say that you have a large text file containing delimited lists of data, and that file contains far more information than you need. After you extract the data you need from the file, you want to replace a subset of that data with a different value. Give yourself a minute to think at a high level about how you might be able to process the file through `awk`, then pipe it into `sed`, and then redirect the output to a file. Think about how long that would take you to perform by hand or even with most programming languages, like Python or Perl. Consider how long those programs would be and, in the case of Perl, how difficult it might be to read it later. Now you know why people who know them love `sed` and `awk`.

Working with Compressed Files

Another file management operation is compression and decompression of files, or the creation, listing, and expansion of file and directory archives. Linux distributions usually include several compression utilities you can use to create, compress, expand, or list the contents of compressed files and archives. These commands include the following:

- ▶ **bunzip2**—Expands a compressed file
- ▶ **bzip2**—Compresses or expands files and directories
- ▶ **gunzip**—Expands a compressed file
- ▶ **gzip**—Compresses or expands files and directories

- ▶ **tar**—Creates, expands, or lists the contents of compressed or uncompressed file or directory archives known as *tape archives* or *tarballs*
- ▶ **xz**—Creates or expands files or *directories*

Most of these commands are easy to use. However, the `tar` command, which is the most commonly used of the bunch, has a somewhat complex set of command-line options and syntax. This flexibility and power are part of its popularity: You can quickly learn to use `tar` by remembering a few of the simple command-line options. For example, to create a compressed archive of a directory, use `tar`'s `czf` options, like this:

```
matthew@seymour:~$ tar czf compressedfilename.tgz directoryname
```

The result is a compressed archive (a file ending in `.tgz`) of the specified directory (and all files and directories under it). Add the letter `v` to the preceding options to view the list of files added during compression and archiving while the archive is being created. To list the contents of the compressed archive, substitute the `c` option with the letter `t`, like this:

```
matthew@seymour:~$ tar tzf archive
```

However, if many files are in the archive, a better invocation (to easily read or scroll through the output) is this:

```
matthew@seymour:~$ tar tzf archive | less
```

To expand the contents of a compressed archive, use `tar`'s `zxf` options, as follows:

```
matthew@seymour:~$ tar zxf archive
```

The `tar` utility decompresses the specified archive and extracts the contents in the current directory.

Using Multiple Terminals with `byobu`

Many Linux veterans have enjoyed and use the GNU `screen` command, which was designed to enable you to use one terminal to control several terminal sessions easily (see www.gnu.org/software/screen/ for more info). Although `screen` has been and is a welcome and useful tool, you should consider `byobu`, which is an enhanced version of `screen`. *Byobu* is a Japanese term for decorative, multipanel, vertically folding screens that are often used as room dividers.

Picture this scene: You connect to a server via *Secure Shell (SSH)* and are working at the remote shell. You need to open another shell window so you can have the two running side by side; perhaps you want the output from `top` in one window while you're typing in another. What do you do? Most people would open another SSH connection, but that is both wasteful and unnecessary. Like `screen`, `byobu` is a terminal multiplexer, which is a fancy term for a program that enables you to run multiple terminals inside one terminal.

The best way to learn `byobu` is to try it yourself. So, open a console, type `byobu`, and then press Enter. Your display blinks momentarily and is then replaced with a new console with new information in a panel at the bottom. Now do something with that terminal: Run `top` and leave it running for the time being. Press F2. Your prompt clears again, leaving you able to type. Run the `uptime` command.

Pop quiz: What happened to the old terminal running `top`? It is still running, of course. You can press F3 to return to it. Press F4 to go back to your `uptime` terminal. While you are viewing other terminals, the commands in the other terminals carry on running as normal so you can multitask. Here are some of the basic commands in `byobu`:

- ▶ **F2**—Creates a new window
- ▶ **F3**—Goes to the previous window
- ▶ **F4**—Goes to the next window
- ▶ **F9**—Opens the `byobu` menu for help and configuration

To close a terminal within `byobu`, simply log out of it normally, using `exit` or Ctrl+D. When you exit the last terminal session that is open in `byobu`, the program closes as well and drops you to the regular terminal session you used to start `byobu`.

However, there are two alternatives to quitting a `byobu` session: locking and disconnecting. The first, activated with F12, locks access to your screen data until you enter your system password.

The second is the most powerful feature of `screen` and also works beautifully in `byobu`: You can exit it and do other things for a while and then reconnect later; both `screen` and `byobu` allow you to pick up where you left off. For example, you could be typing at your desk, detach from a session and go home, reconnect, and carry on as if nothing had changed. What's more, all the programs you ran from `screen` or `byobu` carry on running even while `screen` or `byobu` is disconnected. They even automatically disconnect for you if someone closes your terminal window while it is in a locked state (with Ctrl+A+X).

To disconnect, press F6. You are returned to the prompt from which you launched `screen` or `byobu` and can carry on working, close the terminal you had opened, or even log out completely. When you want to reconnect, run the command `screen -r` or `byobu -r`. You can, in the meantime, just run `screen` or `byobu` and start a new session without resuming the previous one, but that is not wise if you value your sanity. You can disconnect and reconnect the same session as many times you want, which potentially means you need never lose your session again.

Although this has been a mere taste of what `byobu` and `screen` can do, hopefully you can see how useful they can be. Check the man pages for each to learn more. You can also find `byobu` documentation at <https://byobu.co> and <https://help.ubuntu.com/community/Byobu>.

Doing a Polite System Reset Using REISUB

Sometimes computer systems freeze. We're not talking about the times when one program starts acting weird and the program freezes and everything else works fine. In those cases, you can use `kill` to terminate the program and move on, as described in the section "Printing Resource Usage with `top`," earlier in this chapter or you can use the `kill` command as described in Chapter 16, "System-Monitoring Tools." The freezing we're talking about is when nothing will work. Nothing responds to any keyboard or other input, not even a Ctrl+Alt+Del key combination. What then? The absolute worst-case scenario is to perform a power cycle, which is a fancy way of saying "Turn it off and back on again." The problem is that power cycling can cause you to lose data because it can corrupt the file system. This doesn't always happen, but it is a large enough risk that you want to avoid performing a power cycle unless absolutely necessary. Instead, you can try using REISUB.

NOTE

Before you can use the REISUB feature, it must be enabled. The feature is enabled when the value of `/proc/sys/kernel/sysrq` is set to 1. You must have this enabled before you encounter a problem in order for REISUB to work.

To check the current value of this variable, run the following:

```
matthew@seymour:~$ cat /proc/sys/kernel/sysrq
```

To change the value, first edit the `/etc/sysctl.conf` file by uncommenting the line in the file by removing the `#` in front of it and saving to set `kernel.sysrq=1`. Then run the following:

```
matthew@seymour:~$ sudo sysctl -p /etc/sysctl.conf
```

The Linux kernel has a set of key combination commands that are built in at the kernel level. These are referred to using the name of one of the keys, the SysRq key, often labeled PrtScr. The *Magic SysRq Key* combinations send commands directly to the kernel, bypassing any programs running on top of the kernel, including your window manager and probably anything that is frozen. To use these commands, press SysRq+Alt+ one other key. Here we focus on the six keys that are most useful to most people; you can find full lists of available commands at www.kernel.org/doc/html/latest/admin-guide/sysrq.html.

REISUB is an acronym used as a mnemonic device to help users remember the Magic SysRq Key sequence that is best to use when trying to restart a frozen system without risking damage to the file system. You hold down SysRq+Alt and press the R, E, I, S, U, B keys one at a time, in order. This performs the following series of actions, listed in order here, with the letter corresponding to the command capitalized:

- **unRaw**—Takes control of the keyboard back from the X server
- **tErminate**—Sends a SIGTERM command to all processes, which allows time for the processes to terminate gracefully

- ▶ **kill**—Sends a SIGKILL to all processes, forcing any that are still running to terminate immediately
- ▶ **Sync**—Flushes data from memory to disk
- ▶ **Unmount**—Unmounts and remounts all file systems as read-only
- ▶ **reBoot**—Turns off and back on again, restarting the computer

If you have to use REISUB, allow several seconds for each step. Be patient. Doing it this way can save you from the heartache of lost data.

Fixing an Ubuntu System That Will Not Boot

Although it's uncommon, it happens: Sometimes a system will not boot. There are many reasons a system won't boot. The goal here is to help you discover one that may help you recover your system. The ideas in this section are for computers that have had a working Ubuntu installation; however, they may also be useful if you attempted an install that did not work. The options described here are not going to help with troubleshooting computers running Windows or other operating systems.

Checking BIOS

If your computer is unable to boot at all—not even from a known-good bootable USB drive or live DVD—there are two options. It is possible that you accidentally reset the boot devices and/or order in your system BIOS. If making sure those settings are correct does not help, you may have a hardware problem.

Checking GRUB

If you are able to turn on the computer on and get past the initial BIOS startup, then you should consider whether you can access GRUB. As discussed in greater detail in Chapter 15, “The Boot Process,” the GRUB boot loader takes over after the BIOS has completed its initial work. Hold down the Shift key after the BIOS part is done to bring up the GRUB menu. If GRUB does not appear, then perhaps it has been overwritten, in which case the next section will help. If GRUB is working fine, skip to the “Using Recovery Mode” section.

Reinstalling GRUB

To restore GRUB, follow these steps:

NOTE

If you have a dual-boot system, you must be extremely careful with the steps in this section because the details in step 2 may differ depending on the other operating system residing on your system. Troubleshooting dual-boot systems is beyond the scope of this book.

1. Boot Ubuntu from a live DVD or bootable USB drive that has the same Ubuntu release as your system, such as 16.04.
2. Determine the boot drive on your system:
 - a. Open a terminal and use `sudo fdisk -l` to list the drives attached to the system.
 - b. Look for an entry in the output with an `*` in the `Boot` column. This is your boot device. It will look something like `/dev/sda1` or `/dev/nvmen0p1`.
3. Mount the Ubuntu partition at `/mnt` by using this command, replacing `/dev/sda1` with the information you just found:


```
matthew@seymour:~$ sudo mount /dev/sda1 /mnt
```
4. Reinstall GRUB with this command, again replacing `/dev/sda1` with what you found earlier:


```
matthew@seymour:~$ sudo grub-install --boot-directory=/mnt/boot /dev/sda1
```
5. Restart the computer, and Ubuntu should boot properly.

Using Recovery Mode

If GRUB is already working but you are unable to access Ubuntu, you may be able to use recovery mode. Press Shift after the BIOS is done to access the GRUB menu. Select Advanced Options for Ubuntu. From the new menu, select an entry with the words *recovery mode*. This boots into a recovery menu with options to automatically fix several possible problems, or at least it lets you boot into a minimal recovery-mode version of Ubuntu with only the most necessary processes loaded. From here, you may be able to fix disks, check file systems, drop to a root prompt to fix file permissions, and so on. If you don't understand the entries in this menu, they aren't likely to help you much, and you should consider the next option.

Reinstalling Ubuntu

If you are able to boot using a live DVD or bootable USB drive using the same Ubuntu release or one just newer than the one on the hard drive, and if there are no hardware problems with your system, you can usually recover all your files by reinstalling Ubuntu. Boot from the install medium and select Install Ubuntu. Make sure you are paying attention. The installer will detect an existing Ubuntu installation and give you the option to reinstall Ubuntu. When you do this, it should not overwrite existing files in your `/home` directory. Note that we said *should* not—not *will* not—and you should consider this an option of last resort.

Tips and Tricks

This last section is a motley collection of useful command-line tidbits that don't really fit well in the other categories but that are worth sharing. Enjoy.

Running the Previous Command

You can rerun the previous command with the up arrow and Enter. You can also rerun it with `!!` (referred to as “bang bang”). This is especially useful for times when you typed a command correctly but forgot to preface it with `sudo`, as shown here:

```
matthew@seymour:~$ apt-get update
```

```
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denied)
E: Unable to lock directory /var/lib/apt/lists/
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
matthew@seymour:~$ sudo !!
```

This runs `sudo apt-get update`.

12

Running Any Previous Command

You found a neat trick and ran it, but you can't remember it. That is frustrating. You know you can use the up and down arrows to search through your command history and try to find it, but you last ran it earlier in the week, and you aren't sure how long that will take. No worries. You can search your command history.

Type `Ctrl+R` at the command line to start what is called a “reverse-i search” and begin typing. Whatever you type will be matched to previously run commands, so if you know it was a cool combination of commands piped together that had `sort` in the middle, start typing “sort” and watch as the displayed commands from your history appear. When you find it, press Enter and run it again.

Running a Previous Command That Started with Specific Letters

Say that you are listing the contents of a directory that is several layers deep, and you last ran it about nine commands ago, but you don't feel like scrolling. No sweat. Use `!` (exclamation point) and the letters that make up that command or its beginning, as shown here:

```
matthew@seymour:~$ !ls
```

This runs the most recently run command that started with `ls`.

Running the Same Thing You Just Ran with a Different First Word

Say that you just used `ls` to list a file, and you have confirmed that it is present. Now you want to use `nano` to edit the file. Use `!*` (exclamation point asterisk):

```
matthew@seymour:~$ ls stuff/article.txt
article.txt
matthew@seymour:~$ nano !*
```

Viewing Your History and More

By default, the previous 1,000 commands you have run are saved in your `/home` directory in a file called `.bash_history`. You can edit this file. You can delete this file. You can change the number of commands saved in your history by editing this line in the `.bashrc` file in your `/home` directory to whatever number you want:

```
HISTSIZE=1000
```

Doing Two or More Things

There are a few ways you can do two or more things on one line.

Separating commands with a `;` (semicolon) causes the second command to execute after the first command is complete, regardless of the result of the first command. Here is an example:

```
matthew@seymour:~$ command1 ; command2
```

If you want the second command to be run only if the first command exited with no errors, use `&&` (two ampersands):

```
matthew@seymour:~$ command1 && command2
```

If you want the second command to be run only if the first command fails, use `||` (two pipes):

```
matthew@seymour:~$ command1 || command2
```

Using Shortcuts

We all make typing errors while entering commands. When you notice an error in a long line of text, just before you press Enter, it is frustrating to use the Backspace key to erase the entire line one character at a time. There are faster ways.

- ▶ **Ctrl+U**—Erases the entire line
- ▶ **Ctrl+W**—Erases word by word
- ▶ **Left and right arrow keys**—Move along the line to where the error is
- ▶ **Ctrl+A**—Moves the cursor to the beginning of the line
- ▶ **Ctrl+E**—Moves the cursor to the end of the line
- ▶ **Ctrl+K**—Erases everything to the right of the cursor's position
- ▶ **Ctrl+Y**—Restores something you just deleted but shouldn't have

Confining a Script to a Directory

Sometimes you want to isolate a process from the rest of the system, such as when you want to test a script you have written, but you also want to make sure the script is only able to affect what you want it to and not anything else. To do this, you can set up what is called a `chroot jail`. Really, all you are doing is creating a new directory, copying the files you need for the process to run into that directory, and then using the `chroot` command to change the root directory to the base of this new file system tree. Explained differently, you are making the system act temporarily as if the directory you just named is root, when in reality nothing in the file system has changed.

For example, let's say you have a simple file system like this:

```
/
├─etc
├─home
│   └─testing
│       └─fakeetc
│           └─www
└─var
    └─www
```

If you enter this:

```
matthew@seymour:~$ chroot testing
```

And follow it with this:

```
matthew@seymour:~$ ls /
```

You receive this output:

```
/
├─fakeetc
└─www
```

TIP

It is possible for processes that run as root to “break out” of a `chroot` environment, so maybe `chroot jail` is not the most accurate term, but it is commonly used. It is better to think of this as a means to do some isolated testing but not truly secure testing.

Using Coreutils

You have already learned about some of the contents of a package of useful command-line tools called GNU Coreutils. It includes some of the most commonly used commands, like `ls`, `mv`, `cp`, `rm`, and `cat`. It also contains a ton of lesser-known but incredibly useful tools. This package is installed by default. Few people ever make use of its richness. You will want to explore it more deeply. Coreutils contains so much, it is worthy of a chapter and maybe a book of its own. What we can do here is point you to the GNU website entry for Coreutils, at www.gnu.org/software/coreutils/, and also the info page at `info coreutils` (there is no man page for Coreutils).

Coreutils includes so many useful commands, many of which are covered across this book's command line chapters. For a version of the Coreutils manual that is easier to read than the `info` page, see <https://www.gnu.org/software/coreutils/manual/coreutils.html>.

Reading the Contents of the Kernel Ring Buffer with `dmesg`

Although it sounds fancy and ominous, the *kernel ring buffer* is actually quite simple—at least conceptually. It records a limited set of messages related to the operation of the Linux kernel. When it reaches a certain number of messages, it deletes the oldest message every time a new message is written to the list. Looking at the contents of the kernel ring buffer with `dmesg` can be helpful in determining what is happening with your system. If you enter the following, you will receive an incredibly long and scary-looking list of data:

```
matthew@seymour:~$ dmesg
```

This is too much information to be really helpful. The best use of `dmesg`, then, is to combine it with a filter and look for specific entries in the output. You might search for a part of the system, like “memory,” or a specific vendor, like “nvidia,” to read messages containing that text, like so:

```
matthew@seymour:~$ dmesg | grep nvidia
```

References

- ▶ **www.vim.org**—Home page for the `vim` (vi clone) editor included with Linux distributions, which provides updates, bug fixes, and news about this editor
- ▶ **www.gnu.org/software/emacs/emacs.html**—Home page for the FSF’s GNU `emacs` editing environment, where you can find additional documentation and links to the source code for the latest version
- ▶ **www.nano-editor.org**—Home page for the GNU `nano` editor environment
- ▶ ***sed & awk* by Dale Dougherty and Arnold Robbins**—The standard book for learning `sed` and `awk`
- ▶ ***The UNIX CD Bookshelf***—Seven highly respected books in one, which retails for more than \$120 but is incomparable in its depth and breadth of coverage