# 5 – Working with Commands

Up to this point, we have seen a series of mysterious commands, each with its own mysterious options and arguments. In this chapter, we will attempt to remove some of that mystery and even create our own commands. The commands introduced in this chapter are:

- `type` – Indicate how a command name is interpreted
- `which` – Display which executable program will be executed
- `help` – Get help for shell builtins
- `man` – Display a command's manual page
- `apropos` – Display a list of appropriate commands
- `info` – Display a command's info entry
- `whatis` – Display one-line manual page descriptions
- `alias` – Create an alias for a command

## What Exactly Are Commands?

A command can be one of four different things:

1. **An executable program** like all those files we saw in `/usr/bin`. Within this category, programs can be *compiled binaries* such as programs written in C and C++, or programs written in *scripting languages* such as the shell, Perl, Python, Ruby, and so on.
2. **A command built into the shell itself**. `bash` supports a number of commands internally called *shell builtins*. The `cd` command, for example, is a shell builtin.
3. **A shell function.** Shell functions are miniature shell scripts incorporated into the *environment*. We will cover configuring the environment and writing shell functions in later chapters, but for now, just be aware that they exist.
4. **An alias.** Aliases are commands that we can define ourselves, built from other commands.

## Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of ways to find out.

### `type` – Display a Command's Type

The `type` command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
type command
```

where “command” is the name of the command we want to examine. Here are some examples:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Here we see the results for three different commands. Notice the one for `ls` (taken from a Fedora system) and how the `ls` command is actually an alias for the `ls` command with the “--color=tty” option added. Now we know why the output from `ls` is displayed in color!

### `which` – Display an Executable's Location

Sometimes there is more than one version of an executable program installed on a system. While this is not common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the `which` command is used.

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs. When we try to use `which` on a shell builtin for example, `cd`, we either get no response or get an error message:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/usr/local/bin:/usr/bin:/bin:/usr/local
/games:/usr/games)
```

This response is a fancy way of saying “command not found.”

## Getting a Command's Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

### `help` – Get Help for Shell Builtins

`bash` has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. Here is an example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Change the shell working directory.

    Change the current directory to DIR.  The default DIR is the
    value of the HOME shell variable.

    The variable CDPATH defines the search path for the directory
    containing DIR.  Alternative directory names in CDPATH are
    separated by a colon (:). A null directory name is the same as
    the current directory.  If DIR begins with a slash (/), then
    CDPATH is not used.

    If the directory is not found, and the shell option `cdable_vars'
    is set, the word is assumed to be  a variable name.  If that
    variable has a value, its value is used for DIR.

    Options:
       -L    force symbolic links to be followed: resolve symbolic
             links in DIR after processing instances of `..'
       -P    use the physical directory structure without following
             symbolic links: resolve symbolic links in DIR before
             processing instances of `..'
       -e    if the -P option is supplied, and the current working
             directory cannot be determined successfully, exit with
             a non-zero status
```

```
       -@    on systems that support it, present a file with extended
             attributes as a directory containing the file attributes


     The default is to follow symbolic links, as if `-L' were
    specified. `..' is processed by removing the immediately previous
    pathname component back to a slash or the beginning of DIR.

    Exit Status:
    Returns 0 if the directory is changed, and if $PWD is set
    successfully when -P is used; non-zero otherwise.
```

**A note on notation:** When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the `cd` command above:

```
cd [-L|[-P[-e]]] [dir]
```

This notation says that the command `cd` may be followed optionally by either a “-L” or a “-P” and further, if the “-P” option is specified the “-e” option may also be included followed by the optional argument “dir”.

While the output of `help` for the `cd` commands is concise and accurate, it is by no means tutorial and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

## `--help` – Display Usage Information

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options. For example:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options
too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx – umask
  -p, --parents     no error if existing, make parent directories as
                    needed
  -v, --verbose     print a message for each created directory
      --help        display this help and exit
```

```
      --version     output version information and exit
Report bugs to <bug-coreutils@gnu.org>.
```

Some programs don't support the “--help” option, but try it anyway. Often it results in an error message that will reveal the same usage information.

## man – Display a Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called `man` is used to view them. It is used like this:

```
man program
```

where “program” is the name of the command to view.

Man pages vary somewhat in format but generally contain the following:

- A title (the page’s name)
- A synopsis of the command's syntax
- A description of the command's purpose
- A listing and description of each of the command's options

Man pages, however, do not usually include examples, and are intended as a reference, not a tutorial. As an example, let's try viewing the man page for the `ls` command:

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page, so all of the familiar `less` commands work while displaying the page.

The “manual” that `man` displays is broken into sections and covers not only user commands but also system administration commands, programming interfaces, file formats and more. Table 5-1 describes the layout of the manual.

*Table 5-1: Man Page Organization*

| Section | Contents |
|---|---|
| 1 | User commands |

| | |
|---|---|
| 2 | Programming interfaces for kernel system calls |
| 3 | Programming interfaces to the C library |
| 4 | Special files such as device nodes and drivers |
| 5 | File formats |
| 6 | Games and amusements such as screen savers |
| 7 | Miscellaneous |
| 8 | System administration commands |

Sometimes we need to refer to a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. Without specifying a section number, we will always get the first instance of a match, probably in section 1. To specify a section number, we use `man` like this:

```
man section search_term
```

Here's an example:

```
[me@linuxbox ~]$ man 5 passwd
```

This will display the man page describing the file format of the `/etc/passwd` file.

## `apropos` – Display Appropriate Commands

It is also possible to search the list of man pages for possible matches based on a search term. It's crude but sometimes helpful. Here is an example of a search for man pages using the search term *partition*:

```
[me@linuxbox ~]$ apropos partiton
addpart (8)          - simple wrapper around the "add partition"...
all-swaps (7)        - event signalling that all swap partitions...
cfdisk (8)           - display or manipulate disk partition table
cgdisk (8)           - Curses-based GUID partition table (GPT)...
delpart (8)          - simple wrapper around the "del partition"...
fdisk (8)            - manipulate disk partition table
fixparts (8)         - MBR partition table repair utility
```

```
gdisk (8)            - Interactive GUID partition table (GPT)...
mpartition (1)       - partition an MSDOS hard disk
partprobe (8)        - inform the OS of partition table changes
partx (8)            - tell the Linux kernel about the presence...
resizepart (8)       - simple wrapper around the "resize partition...

sfdisk (8)           - partition table manipulator for Linux
sgdisk (8)           - Command-line GUID partition table (GPT)...
```

The first field in each line of output is the name of the man page, and the second field shows the section. Note that the `man` command with the "-k" option performs the same function as `apropos`.

## `whatis` – Display One-line Manual Page Descriptions

The `whatis` program displays the name and a one-line description of a man page matching a specified keyword:

```
[me@linuxbox ~]$ whatis ls
ls                   (1)  - list directory contents
```

### The Most Brutal Man Page Of Them All

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read, but I think that the grand prize for difficulty has got to go to the man page for `bash`. As I was doing research for this book, I gave the `bash` man page careful review to ensure that I was covering most of its topics. When printed, it's more than 80 pages long and extremely dense, and its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate and concise, as well as being extremely complete. So check it out if you dare and look forward to the day when you can read it and it all makes sense.

## `info` – Display a Program's Info Entry

The GNU Project provides an alternative to man pages for their programs, called "info."

Info manuals are displayed with a reader program named, appropriately enough, `info`. Info pages are *hyperlinked* much like web pages. Here is a sample:

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,
Up: Directory listing
10.1 `ls': List directory contents
==================================
The `ls' program lists information about files (of any type,
including directories).  Options and file arguments can be intermixed
arbitrarily, as usual.
   For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the filename.  If no non-option
argument is specified, `ls' operates on the current directory, acting
as if it had been invoked with a single argument of `.'.
   By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top---------
```

The `info` program reads *info files*, which are tree structured into individual *nodes*, each containing a single topic. Info files contain hyperlinks that can move the reader from node to node. A hyperlink can be identified by its leading asterisk and is activated by placing the cursor upon it and pressing the Enter key.

To invoke `info`, type `info` followed optionally by the name of a program. Table 5-2 describes the commands used to control the reader while displaying an info page.

*Table 5-2: `info` Commands*

| Command | Action |
|---|---|
| `?` | Display command help |
| `PgUp` or `Backspace` | Display previous page |
| `PgDn` or `Space` | Display next page |
| `n` | Next - Display the next node |
| `p` | Previous - Display the previous node |
| `u` | Up - Display the parent node of the currently displayed node, usually a menu |
| `Enter` | Follow the hyperlink at the cursor location |

| | |
|---|---|
| q | Quit |

Most of the command line programs we have discussed so far are part of the GNU Project's *coreutils* package, so typing the following:

```
[me@linuxbox ~]$ info coreutils
```

will display a menu page with hyperlinks to each program contained in the coreutils package.

### README and Other Program Documentation Files

Many software packages installed on our system have documentation files residing in the `/usr/share/doc` directory. Most of these are stored in plain text format and can be viewed with `less`. Some of the files are in HTML format and can be viewed with a web browser. We may encounter some files ending with a ".gz" extension. This indicates that they have been compressed with the `gzip` compression program. The gzip package includes a special version of `less` called `zless` that will display the contents of gzip-compressed text files.

## Creating Our Own Commands with `alias`

Now for our first experience with programming! We will create a command of our own using the `alias` command. But before we start, we need to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon. It works like this:

```
command1; command2; command3...
```

Here's the example we will use:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games  include  lib  local  sbin  share  src
/home/me
[me@linuxbox ~]$
```

As we can see, we have combined three commands on one line. First we change directory to `/usr` then list the directory and finally return to the original directory (by using `'cd`

`-'`) so we end up where we started. Now let's turn this sequence into a new command using `alias`. The first thing we have to do is dream up a name for our new command. Let's try “test”. Before we do that, it would be a good idea to find out if the name “test” is already being used. To find out, we can use the `type` command again:

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

Oops! The name `test` is already taken. Let's try `foo`:

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

Great! “foo” is not taken. So let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Notice the structure of this command shown here:

```
alias name='string'
```

After the command `alias`, we give alias a name followed immediately (no whitespace allowed) by an equal sign, followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, we can use it anywhere the shell would expect a command. Let's try it:

```
[me@linuxbox ~]$ foo
bin  games  include  lib  local  sbin  share  src
/home/me
[me@linuxbox ~]$
```

We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls; cd -'
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
```

To see all the aliases defined in the environment, use the `alias` command without arguments. Here are some of the aliases defined by default on a Fedora system. Try to figure out what they all do:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

There is one tiny problem with defining aliases on the command line. They vanish when our shell session ends. In Chapter 11, "The Environment", we will see how to add our own aliases to the files that establish the environment each time we log on, but for now, enjoy the fact that we have taken our first, albeit tiny, step into the world of shell programming!

## Summing Up

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them!

## Further Reading

There are many online sources of documentation for Linux and the command line. Here are some of the best:

- The *Bash Reference Manual* is a reference guide to the `bash` shell. It's still a reference work but contains examples and is easier to read than the `bash` man page.
  http://www.gnu.org/software/bash/manual/bashref.html
- The *Bash FAQ* contains answers to frequently asked questions regarding `bash`. This list is aimed at intermediate to advanced users, but contains a lot of good information.
  http://mywiki.wooledge.org/BashFAQ
- The GNU Project provides extensive documentation for its programs, which form the core of the Linux command line experience. You can see a complete list here:
  http://www.gnu.org/manual/manual.html
- Wikipedia has an interesting article on man pages:
  http://en.wikipedia.org/wiki/Man_page

# 9 – Permissions

Operating systems in the Unix tradition differ from those in the MS-DOS tradition in that they are not only *multitasking* systems, but also *multi-user* systems.

What exactly does this mean? It means that more than one person can be using the computer at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via `ssh` (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display. The X Window System supports this as part of its basic design.

The multiuser capability of Linux is not a recent "innovation," but rather a feature that is deeply embedded into the design of the operating system. Considering the environment in which Unix was created, this makes perfect sense. Years ago, before computers were "personal," they were large, expensive, and centralized. A typical university computer system, for example, consisted of a large central computer located in one building and terminals that were located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

To make this practical, a method had to be devised to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

In this chapter we will look at this essential part of system security and introduce the following commands:

- `id` – Display user identity
- `chmod` – Change a file's mode
- `umask` – Set the default file permissions
- `su` – Run a shell as another user
- `sudo` – Execute a command as another user
- `chown` – Change a file's owner

- `chgrp` – Change a file's group ownership
- `passwd` – Change a user's password

## Owners, Group Members, and Everybody Else

When we were exploring the system in Chapter 3, we may have encountered a problem when trying to examine a file such as `/etc/shadow`:

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

The reason for this error message is that, as regular users, we do not have permission to read this file.

In the Unix security model, a user may *own* files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a *group* consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as the *world*. To find out information about your identity, use the `id` command.

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

Let's look at the output. When user accounts are created, users are assigned a number called a *user ID* (*uid*) which is then, for the sake of the humans, mapped to a username. The user is assigned a *primary group ID* (*gid*) and may belong to additional groups. The above example is from a Fedora system. On other systems, such as Ubuntu, the output may look a little different:

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(v
ideo),46(plugdev),108(lpadmin),114(admin),1000(me)
```

As we can see, the uid and gid numbers are different. This is simply because Fedora starts its numbering of regular user accounts at 500, while Ubuntu starts at 1000. We can also

see that the Ubuntu user belongs to a lot more groups. This has to do with the way Ubuntu manages privileges for system devices and services.

So where does this information come from? Like so many things in Linux, it comes from a couple of text files. User accounts are defined in the `/etc/passwd` file and groups are defined in the `/etc/group` file. When user accounts and groups are created, these files are modified along with `/etc/shadow` which holds information about the user's password. For each user account, the `/etc/passwd` file defines the user (login) name, uid, gid, account's real name, home directory, and login shell. If we examine the contents of `/etc/passwd` and `/etc/group`, we notice that besides the regular user accounts, there are accounts for the superuser (uid 0) and various other system users.

In the next chapter, when we cover processes, we will see that some of these other "users" are, in fact, quite busy.

While many Unix-like systems assign regular users to a common group such as "users", modern Linux practice is to create a unique, single-member group with the same name as the user. This makes certain types of permission assignment easier.

## Reading, Writing, and Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the `ls` command, we can get some clue as to how this is implemented:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2016-03-06 14:52 foo.txt
```

The first 10 characters of the listing are the *file attributes*. The first of these characters is the *file type*. Table 9-1 describes the file types we are most likely to see (there are other, less common types too):

*Table 9-1: File Types*

| Attribute | File Type |
|---|---|
| `-` | A regular file. |
| `d` | A directory. |
| `l` | A symbolic link. Notice that with symbolic links, the remaining file attributes are always "rwxrwxrwx" and are dummy values. The real file attributes are those of the file the symbolic link points to. |

| | |
|---|---|
| `c` | A *character special file*. This file type refers to a device that handles data as a stream of bytes, such as a terminal or `/dev/null`. |
| `b` | A *block special file*. This file type refers to a device that handles data in blocks, such as a hard drive or DVD drive. |

The remaining nine characters of the file attributes, called the *file mode*, represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else.

| Owner | Group | World |
|---|---|---|
| `rwx` | `rwx` | `rwx` |

Table 9-2 describes the effect the `r`, `w`, and `x` mode attributes have on files and directories:

*Table 9-2: Permission Attributes*

| Attribute | Files | Directories |
|---|---|---|
| `r` | Allows a file to be opened and read. | Allows a directory's contents to be listed if the execute attribute is also set. |
| `w` | Allows a file to be written to or truncated, however this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes. | Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set. |
| `x` | Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed. | Allows a directory to be entered, e.g., cd *directory*. |

Table 9-3 provides some examples of file attribute settings:

*Table 9-3: Permission Attribute Examples*

| File Attributes | Meaning |
|---|---|
| `-rwx------` | A regular file that is readable, writable, and executable by the file's owner. No one else has any access. |
| `-rw-------` | A regular file that is readable and writable by the file's owner. No one else has any access. |
| `-rw-r--r--` | A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world-readable. |
| `-rwxr-xr-x` | A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else. |
| `-rw-rw----` | A regular file that is readable and writable by the file's owner and members of the file's group owner only. |
| `lrwxrwxrwx` | A symbolic link. All symbolic links have “dummy” permissions. The real permissions are kept with the actual file pointed to by the symbolic link. |
| `drwxrwx---` | A directory. The owner and the members of the owner group may enter the directory and create, rename and remove files within the directory. |
| `drwxr-x---` | A directory. The owner may enter the directory and create, rename, and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete, or rename files. |

## `chmod` – Change File Mode

To change the mode (permissions) of a file or directory, the `chmod` command is used. Be aware that only the file’s owner or the superuser can change the mode of a file or directory. `chmod` supports two distinct ways of specifying mode changes: octal number representation, or symbolic representation. We will cover octal number representation first.

## What the Heck is Octal?

*Octal* (base 8), and its cousin, *hexadecimal* (base 16) are number systems often used to express numbers on computers. We humans, owing to the fact that we (or at least most of us) were born with 10 fingers, count using a base 10 number system. Computers, on the other hand, were born with only one finger and thus do all their counting in *binary* (base 2). Their number system has only two numerals, 0 and 1. So, in binary, counting looks like this:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...

In octal, counting is done with the numerals zero through seven, like so:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...

Hexadecimal counting uses the numerals zero through nine plus the letters “A” through “F”:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

While we can see the sense in binary (since computers have only one finger), what are octal and hexadecimal good for? The answer has to do with human convenience. Many times, small portions of data are represented on computers as *bit patterns*. Take for example an RGB color. On most computer displays, each pixel is composed of three color components: eight bits of red, eight bits of green, and eight bits of blue. A lovely medium blue would be a 24 digit number:

010000110110111111001101

How would you like to read and write those kinds of numbers all day? I didn't think so. Here's where another number system would help. Each digit in a hexadecimal number represents four digits in binary. In octal, each digit represents three binary digits. So our 24 digit medium blue could be condensed to a six-digit hexadecimal number:

436FCD

Since the digits in the hexadecimal number “line up” with the bits in the binary number, we can see that the red component of our color is 43, the green 6F, and the blue CD.

These days, hexadecimal notation (often spoken as “hex”) is more common than octal, but as we will soon see, octal's ability to express three bits of binary will be very useful...

With octal notation, we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the

scheme used to store the file mode. Table 9-4 shows what we mean.

*Table 9-4: File Modes in Binary and Octal*

| Octal | Binary | File Mode |
|---|---|---|
| 0 | 000 | --- |
| 1 | 001 | --x |
| 2 | 010 | -w- |
| 3 | 011 | -wx |
| 4 | 100 | r-- |
| 5 | 101 | r-x |
| 6 | 110 | rw- |
| 7 | 111 | rwx |

By using three octal digits, we can set the file mode for the owner, group owner, and world.

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2016-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me   0 2016-03-06 14:52 foo.txt
```

By passing the argument “600”, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though remembering the octal to binary mapping may seem inconvenient, we will usually have only to use a few common ones: 7 (`rwx`), 6 (`rw-`), 5 (`r-x`), 4 (`r--`), and 0 (`---`).

`chmod` also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts.

- Who the change will affect
- Which operation will be performed
- What permission will be set.

To specify who is affected, a combination of the characters “u”, “g”, “o”, and “a” is used as shown in Table 9-5.

*Table 9-5: `chmod` Symbolic Notation*

| Symbol | Meaning |
|---|---|
| `u` | Short for “user” but means the file or directory owner. |
| `g` | Group owner. |
| `o` | Short for “others” but means world. |
| `a` | Short for “all.” This is the combination of “u”, “g”, and “o”. |

If no character is specified, “all” will be assumed. The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the “r”, “w”, and “x” characters. Table 9-6 provides some examples of symbolic notation:

*Table 9-6: `chmod` Symbolic Notation Examples*

| Notation | Meaning |
|---|---|
| `u+x` | Add execute permission for the owner. |
| `u-x` | Remove execute permission from the owner. |
| `+x` | Add execute permission for the owner, group, and world. This is equivalent to `a+x`. |
| `o-rw` | Remove the read and write permissions from anyone besides the owner and group owner. |
| `go=rw` | Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or the world previously had execute permission, it is removed. |
| `u+x,go=rx` | Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas. |

Some people prefer to use octal notation, and some folks really like the symbolic. Symbolic notation does offer the advantage of allowing us to set a single attribute without disturbing any of the others.

Take a look at the `chmod` man page for more details and a list of options. A word of caution regarding the “--recursive” option: it acts on both files and directories, so it's not as

useful as we would hope since we rarely want files and directories to have the same permissions.

## Setting File Mode with the GUI

Now that we have seen how the permissions on files and directories are set, we can better understand the permission dialogs in the GUI. In both Files (GNOME) and Dolphin (KDE), right-clicking a file or directory icon will expose a properties dialog. Here is an example from GNOME:

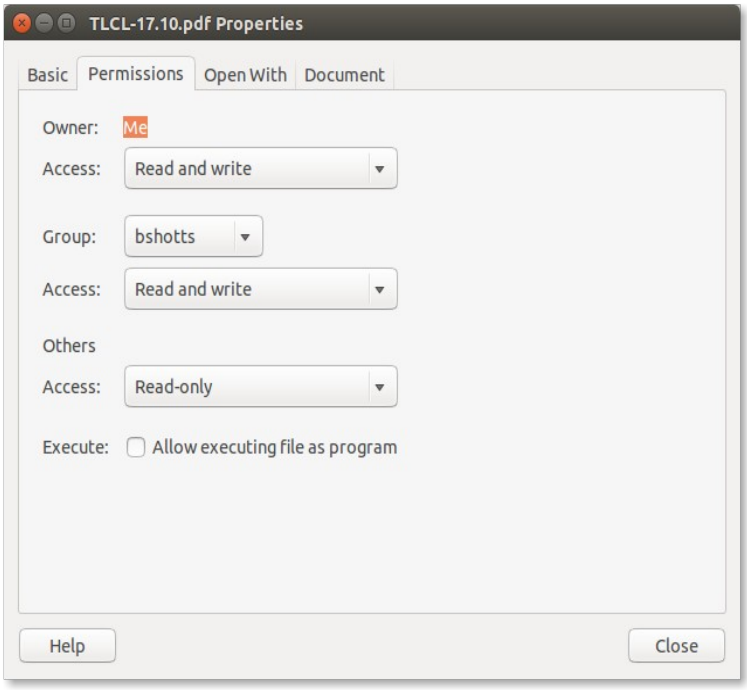


*Figure 2: GNOME file permissions dialog*

Here we can see the settings for the owner, group, and world.

## `umask` – Set Default Permissions

The `umask` command controls the default permissions given to a file when it is created. It uses octal notation to express a *mask* of bits to be removed from a file's mode attributes. Let's take a look.

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2018-03-06 14:53 foo.txt
```

We first removed any old copy of `foo.txt` to make sure we were starting fresh. Next, we ran the `umask` command without an argument to see the current value. It responded with the value `0002` (the value `0022` is another common default value), which is the octal representation of our mask. We next create a new instance of the file `foo.txt` and observe its permissions.

We can see that both the owner and group get read and write permission, while everyone else only gets read permission. The reason that world does not have write permission is because of the value of the mask. Let's repeat our example, this time setting the mask ourselves.

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me    me   0 2018-03-06 14:58 foo.txt
```

When we set the mask to `0000` (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we take the mask, expand it into binary, and then compare it to the attributes we can see what happens.

| **Original file mode** | --- rw- rw- rw- |
|---|---|
| **Mask** | 000 000 000 010 |
| **Result** | --- rw- rw- r-- |

Ignore for the moment the leading zeros (we'll get to those in a minute) and observe that where the 1 appears in our mask, an attribute was removed—in this case, the world write permission. That's what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of `0022`, we can see what it does.

| Original file mode | --- rw- rw- rw- |
|---|---|
| Mask | 000 000 010 010 |
| Result | --- rw- r-- r-- |

Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some sevens) to get used to how this works. When you're done, remember to clean up.

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

Most of the time we won't have to change the mask; the default provided by the distribution will be fine. In some high-security situations, however, we will want to control it.

## Some Special Permissions

Though we usually see an octal permission mask expressed as a three-digit number, it is more technically correct to express it in four digits. Why? Because, in addition to read, write, and execute permission, there are some other, less used, permission settings.

The first of these is the *setuid bit* (octal 4000). When applied to an executable file, it sets the *effective user ID* from that of the real user (the user actually running the program) to that of the program's owner. Most often this is given to a few programs owned by the superuser. When an ordinary user runs a program that is "*setuid root*" , the program runs with the effective privileges of the superuser. This allows the program to access files and directories that an ordinary user would normally be prohibited from accessing. Clearly, because this raises security concerns, the number of setuid programs must be held to an absolute minimum.

The second less-used setting is the *setgid bit* (octal 2000), which, like the setuid bit, changes the *effective group ID* from the *real group ID* of the real user to that of the file owner. If the setgid bit is set on a directory, newly created files in the directory will be given the group ownership of the directory rather the group ownership of the file's creator. This is useful in a shared directory when members of a common group need access to all the files in the directory, regardless of the file owner's primary group.

The third is called the *sticky bit* (octal 1000). This is a holdover from ancient Unix, where it was possible to mark an executable file as "not swappable." On

files, Linux ignores the sticky bit, but if applied to a directory, it prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser. This is often used to control access to a shared directory, such as `/tmp`.

Here are some examples of using `chmod` with symbolic notation to set these special permissions. Here's an example of assigning setuid to a program:

```
chmod u+s program
```

Next, here's and example of assigning setgid to a directory:

```
chmod g+s dir
```

Finally, here's an example of assigning the sticky bit to a directory:

```
chmod +t dir
```

When viewing the output from `ls`, you can determine the special permissions. Here are some examples. First, an example of a program that is setuid:

```
-rwsr-xr-x
```

Here's an example of a directory that has the setgid attribute:

```
drwxrwsr-x
```

Here's an example of a directory with the sticky bit set:

```
drwxrwxrwt
```

## Changing Identities

At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to "become" another regular user for such things as testing an account. There are three ways to take on an alternate identity.

1. Log out and log back in as the alternate user.
2. Use the `su` command.
3. Use the `sudo` command.

We will skip the first technique since we know how to do it and it lacks the convenience of the other two. From within our own shell session, the `su` command allows us to assume the identity of another user and either start a new shell session with that user's ID, or to issue a single command as that user. The `sudo` command allows an administrator to set up a configuration file called `/etc/sudoers` and define specific commands that

particular users are permitted to execute under an assumed identity. The choice of which command to use is largely determined by which Linux distribution you use. Your distribution probably includes both commands, but its configuration will favor either one or the other. We'll start with `su`.

## su – Run a Shell with Substitute User and Group IDs

The `su` command is used to start a shell as another user. The command syntax looks like this:

```
su [-[l]] [user]
```

If the "-l" option is included, the resulting shell session is a *login shell* for the specified user. This means the user's environment is loaded and the working directory is changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the `-l` may be abbreviated as `-`, which is how it is most often used. To start a shell for the superuser, we would do this:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]#
```

After entering the command, we are prompted for the superuser's password. If it is successfully entered, a new shell prompt appears indicating that this shell has superuser privileges (the trailing # rather than a `$`), and the current working directory is now the home directory for the superuser (normally `/root`). Once in the new shell, we can carry out commands as the superuser. When finished, enter `exit` to return to the previous shell.

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

It is also possible to execute a single command rather than starting a new interactive command by using `su` this way.

```
su -c 'command'
```

Using this form, a single command line is passed to the new shell for execution. It is im-

portant to enclose the command in quotes, as we do not want expansion to occur in our shell, but rather in the new shell.

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw------- 1 root root     754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

## sudo – Execute a Command as Another User

The sudo command is like su in many ways but has some important additional capabilities. The administrator can configure sudo to allow an ordinary user to execute commands as a different user (usually the superuser) in a controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of sudo does not require access to the superuser's password. To authenticating using sudo, requires the user's own password. Let's say, for example, that sudo has been configured to allow us to run a fictitious backup program called "backup_script", which requires superuser privileges. With sudo it would be done like this:

```
[me@linuxbox ~]$ sudo backup_script
Password:
System Backup Starting...
```

After entering the command, we are prompted for our password (not the superuser's) and once the authentication is complete, the specified command is carried out. One important difference between su and sudo is that sudo does not start a new shell, nor does it load another user's environment. This means that commands do not need to be quoted any differently than they would be without using sudo. Note that this behavior can be overridden by specifying various options. Note, too, that sudo can be used to start an interactive superuser session (much like su -) by using the -i option. See the sudo man page for details.

To see what privileges are granted by sudo, use the -l option to list them:

```
[me@linuxbox ~]$ sudo -l
```

```
User me may run the following commands on this host:
    (ALL) ALL
```

### Ubuntu and sudo

One of the recurrent problems for regular users is how to perform certain tasks that require superuser privileges. These tasks include installing and updating software, editing system configuration files, and accessing devices. In the Windows world, this is often done by giving users administrative privileges. This allows users to perform these tasks. However, it also enables programs executed by the user to have the same abilities. This is desirable in most cases, but it also permits *malware* (malicious software) such as viruses to have free rein of the computer.

In the Unix world, there has always been a larger division between regular users and administrators, owing to the multiuser heritage of Unix. The approach taken in Unix is to grant superuser privileges only when needed. To do this, the `su` and `sudo` commands are commonly used.

Up until a few of years ago, most Linux distributions relied on `su` for this purpose. `su` didn't require the configuration that `sudo` required, and having a root account is traditional in Unix. This introduced a problem. Users were tempted to operate as root unnecessarily. In fact, some users operated their systems as the root user exclusively, since it does away with all those annoying “permission denied” messages. This is how you reduce the security of a Linux system to that of a Windows system. Not a good idea.

When Ubuntu was introduced, its creators took a different tack. By default, Ubuntu disables logins to the root account (by failing to set a password for the account) and instead uses `sudo` to grant superuser privileges. The initial user account is granted full access to superuser privileges via `sudo` and may grant similar powers to subsequent user accounts.

## `chown` – Change File Owner and Group

The `chown` command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of `chown` looks like this:

```
chown [owner][:[group]] file...
```

`chown` can change the file owner and/or the file group owner depending on the first argument of the command. Table 9-7 provides some examples.

*Table 9-7: `chown` Argument Examples*

| Argument | Results |
|---|---|
| `bob` | Changes the ownership of the file from its current owner to user `bob`. |
| `bob:users` | Changes the ownership of the file from its current owner to user `bob` and changes the file group owner to group `users`. |
| `:admins` | Changes the group owner to the group `admins`. The file owner is unchanged. |
| `bob:` | Changes the file owner from the current owner to user `bob` and changes the group owner to the login group of user `bob`. |

Let's say we have two users; `janet`, who has access to superuser privileges and `tony`, who does not. User `janet` wants to copy a file from her home directory to the home directory of user `tony`. Since user `janet` wants `tony` to be able to edit the file, `janet` changes the ownership of the copied file from `janet` to `tony`.

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 root  root  root 2018-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 tony  tony  tony 2018-03-20 14:30 /home/tony/myfile.txt
```

Here we see user `janet` copy the file from her directory to the home directory of user `tony`. Next, `janet` changes the ownership of the file from `root` (a result of using `sudo`) to `tony`. Using the trailing colon in the first argument, `janet` also changed the group ownership of the file to the login group of `tony`, which happens to be group `tony`.

Notice that after the first use of `sudo`, `janet` was not prompted for her password. This is because `sudo`, in most configurations, “trusts” us for several minutes until its timer

runs out.

### `chgrp` – Change Group Ownership

In older versions of Unix, the `chown` command only changed file ownership, not group ownership. For that purpose, a separate command, `chgrp` was used. It works much the same way as `chown`, except for being more limited.

## Exercising Our Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We are going to demonstrate the solution to a common problem—setting up a shared directory. Let's imagine that we have two users named “bill” and “karen.” They both have music collections and want to set up a shared directory, where they will each store their music files as Ogg Vorbis or MP3. User `bill` has access to superuser privileges via `sudo`.

The first thing that needs to happen is a group needs to be created that will have both `bill` and `karen` as members. Using the graphical user management tool, `bill` creates a group called `music` and adds users `bill` and `karen` to it:

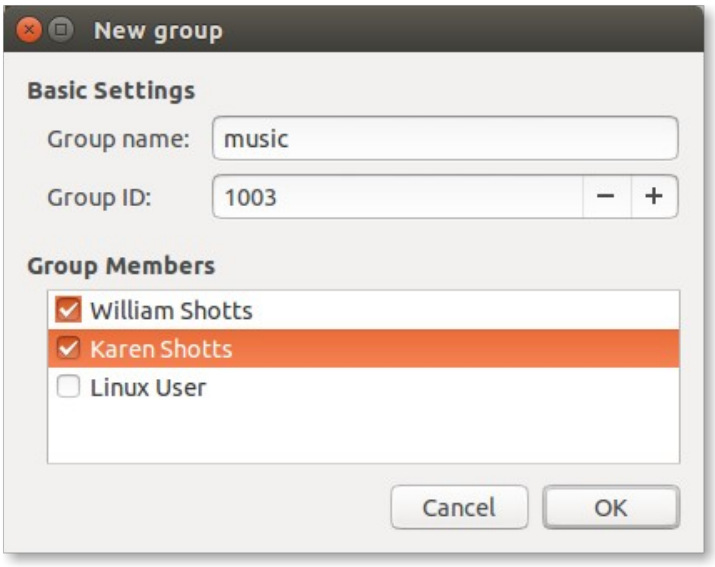


*Figure 3: Creating a new group with GNOME*

Next, `bill` creates the directory for the music files.

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Password:
```

Since `bill` is manipulating files outside his home directory, superuser privileges are required. After the directory is created, it has the following ownerships and permissions:

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2018-03-21 18:05 /usr/local/share/Music
```

As we can see, the directory is owned by `root` and has permission mode 755. To make this directory sharable, `bill` needs to change the group ownership and the group permissions to allow writing.

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2018-03-21 18:05 /usr/local/share/Music
```

What does this all mean? It means that we now have a directory, `/usr/local/share/Music` that is owned by `root` and allows read and write access to group `music`. Group `music` has members `bill` and `karen`; thus, `bill` and `karen` can create files in directory `/usr/local/share/Music`. Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the `Music` directory will have the normal permissions of the users `bill` and `karen`.

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill   bill   0 2018-03-24 20:03 test_file
```

Actually there are two problems. First, the default `umask` on this system is 0022, which prevents group members from writing files belonging to other members of the group. This would not be a problem if the shared directory contained only files, but since this directory will store music, and music is usually organized in a hierarchy of artists and albums, members of the group will need the ability to create files and directories inside directories created by other members. We need to change the `umask` used by `bill` and

`karen` to 0002 instead.

Second, each file and directory created by one member will be set to the primary group of the user rather than the group `music`. This can be fixed by setting the setgid bit on the directory.

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2018-03-24 20:03 /usr/local/share/Music
```

Now we test to see whether the new permissions fix the problem. `bill` sets his `umask` to 0002, removes the previous test file, and creates a new test file and directory:

```
[bill@linuxbox ~]$ umask 0002
[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill   music 4096 2018-03-24 20:24 test_dir
-rw-rw-r-- 1 bill   music    0 2018-03-24 20:22 test_file
[bill@linuxbox ~]$
```

Both files and directories are now created with the correct permissions to allow all members of the group `music` to create files and directories inside the `Music` directory.

The one remaining issue is `umask`. The necessary setting only lasts until the end of session and must be reset. In Chapter 11, we'll look at making the change to `umask` permanent.

## Changing Your Password

The last topic we'll cover in this chapter is setting passwords for yourself (and for other users if you have access to superuser privileges). To set or change a password, the `passwd` command is used. The command syntax looks like this:

```
passwd [user]
```

To change your password, just enter the `passwd` command. You will be prompted for your old password and your new password.

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
```

The `passwd` command will try to enforce use of “strong” passwords. This means it will refuse to accept passwords that are too short, are too similar to previous passwords, are dictionary words, or are too easily guessed.

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
BAD PASSWORD: is too similar to the old one
New UNIX password:
BAD PASSWORD: it is WAY too short
New UNIX password:
BAD PASSWORD: it is based on a dictionary word
```

If you have superuser privileges, you can specify a username as an argument to the `passwd` command to set the password for another user. Other options are available to the superuser to allow account locking, password expiration, and so on. See the `passwd` man page for details.

## Summing Up

In this chapter we saw how Unix-like systems such as Linux manage user permissions to allow the read, write, and execution access to files and directories. The basic ideas of this system of permissions date back to the early days of Unix and have stood up pretty well to the test of time. But the native permissions mechanism in Unix-like systems lacks the fine granularity of more modern systems.

## Further Reading

- Wikipedia has a good article on malware: http://en.wikipedia.org/wiki/Malware

There are number of command line programs used to create and maintain users and groups. For more information, see the man pages for the following commands:

- `adduser`
- `useradd`
- `groupadd`

# 10 – Processes

Modern operating systems are usually *multitasking*, meaning they create the illusion of doing more than one thing at once by rapidly switching from one executing program to another. The Linux kernel manages this through the use of *processes*. Processes are how Linux organizes the different programs waiting for their turn at the CPU.

Sometimes a computer will become sluggish or an application will stop responding. In this chapter, we will look at some of the tools available at the command line that let us examine what programs are doing and how to terminate processes that are misbehaving.

This chapter will introduce the following commands:

- `ps` – Report a snapshot of current processes
- `top` – Display tasks
- `jobs` – List active jobs
- `bg` – Place a job in the background
- `fg` – Place a job in the foreground
- `kill` – Send a signal to a process
- `killall` – Kill processes by name
- `shutdown` – Shutdown or reboot the system

## How a Process Works

When a system starts up, the kernel initiates a few of its own activities as processes and launches a program called `init`. `init`, in turn, runs a series of shell scripts (located in `/etc`) called *init scripts*, which start all the system services. Many of these services are implemented as *daemon programs*, programs that just sit in the background and do their thing without having any user interface. So, even if we are not logged in, the system is at least a little busy performing routine stuff.

The fact that a program can launch other programs is expressed in the process scheme as a *parent process* producing a *child process*.

The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a *process ID* (*PID*). PIDs are assigned in ascending order, with `init` always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well as the processes' readiness to resume execution. Like files, processes also have owners and user IDs, effective user IDs, etc.

## Viewing Processes

The most commonly used command to view processes (there are several) is `ps`. The `ps` program has a lot of options, but in its simplest form it is used like this:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 5198 pts/1    00:00:00 bash
10129 pts/1    00:00:00 ps
```

The result in this example lists two processes, process 5198 and process 10129, which are `bash` and `ps` respectively. As we can see, by default, `ps` doesn't show us very much, just the processes associated with the current terminal session. To see more, we need to add some options, but before we do that, let's look at the other fields produced by `ps`. `TTY` is short for “teletype,” and refers to the *controlling terminal* for the process. Unix is showing its age here. The `TIME` field is the amount of CPU time consumed by the process. As we can see, neither process makes the computer work very hard.

If we add an option, we can get a bigger picture of what the system is doing.

```
[me@linuxbox ~]$ ps x
  PID TTY      STAT   TIME COMMAND
 2799 ?        Ssl    0:00 /usr/libexec/bonobo-activation-server –ac
 2820 ?        Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?        Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?        S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?        Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 –pr
15774 ?        Ss     0:02 /usr/bin/gpg-agent -s –daemon
15793 ?        S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?        Ss     0:00 kdeinit Running...
15797 ?        S      0:00 dcopserver –nosid
and many more...
```

Adding the “x” option (note that there is no leading dash) tells `ps` to show all of our pro-

cesses regardless of what terminal (if any) they are controlled by. The presence of a “?” in the TTY column indicates no controlling terminal. Using this option, we see a list of every process that we own.

Since the system is running a lot of processes, ps produces a long list. It is often helpful to pipe the output from ps into less for easier viewing. Some option combinations also produce long lines of output, so maximizing the terminal emulator window may be a good idea, too.

A new column titled STAT has been added to the output. STAT is short for “state” and reveals the current status of the process, as shown in Table 10-1.

*Table 10-1: Process States*

| State | Meaning |
|---|---|
| R | Running. This means that the process is running or ready to run. |
| S | Sleeping. The process is not running; rather, it is waiting for an event, such as a keystroke or network packet. |
| D | Uninterruptible sleep. The process is waiting for I/O such as a disk drive. |
| T | Stopped. The process has been instructed to stop. More on this later in the chapter. |
| Z | A defunct or “zombie” process. This is a child process that has terminated but has not been cleaned up by its parent. |
| < | A high-priority process. It's possible to grant more importance to a process, giving it more time on the CPU. This property of a process is called *niceness*. A process with high priority is said to be less *nice* because it's taking more of the CPU's time, which leaves less for everybody else. |
| N | A low-priority process. A process with low priority (a “nice” process) will get processor time only after other processes with higher priority have been serviced. |

The process state may be followed by other characters. These indicate various exotic process characteristics. See the ps man page for more detail.

Another popular set of options is “aux” (without a leading dash). This gives us even more information.

```
[me@linuxbox ~]$ ps aux
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1  0.0  0.0   2136   644 ?        Ss   Mar05   0:31 init
root      2  0.0  0.0      0     0 ?        S<   Mar05   0:00 [kt]
root      3  0.0  0.0      0     0 ?        S<   Mar05   0:00 [mi]
root      4  0.0  0.0      0     0 ?        S<   Mar05   0:00 [ks]
root      5  0.0  0.0      0     0 ?        S<   Mar05   0:06 [wa]
root      6  0.0  0.0      0     0 ?        S<   Mar05   0:36 [ev]
root      7  0.0  0.0      0     0 ?        S<   Mar05   0:00 [kh]
and many more...
```

This set of options displays the processes belonging to every user. Using the options without the leading dash invokes the command with “BSD style” behavior. The Linux version of `ps` can emulate the behavior of the `ps` program found in several different Unix implementations. With these options, we get the additional columns shown in Table 10-2.

*Table 10-2: BSD Style `ps` Column Headers*

| Header | Meaning |
|---|---|
| `USER` | User ID. This is the owner of the process. |
| `%CPU` | CPU usage in percent. |
| `%MEM` | Memory usage in percent. |
| `VSZ` | Virtual memory size. |
| `RSS` | Resident set size. This is the amount of physical memory (RAM) the process is using in kilobytes. |
| `START` | Time when the process started. For values over 24 hours, a date is used. |

## Viewing Processes Dynamically with `top`

While the `ps` command can reveal a lot about what the machine is doing, it provides only a snapshot of the machine's state at the moment the `ps` command is executed. To see a more dynamic view of the machine's activity, we use the `top` command:

```
[me@linuxbox ~]$ top
```

The `top` program displays a continuously updating (by default, every three seconds) display of the system processes listed in order of process activity. The name *top* comes from the fact that the `top` program is used to see the "top" processes on the system. The `top` display consists of two parts: a system summary at the top of the display, followed by a table of processes sorted by CPU activity:

```
top - 14:59:20 up  6:30,  2 users,  load average: 0.07, 0.02, 0.00
Tasks: 109 total,   1 running, 106 sleeping,   0 stopped,   2 zombie
Cpu(s):  0.7%us,  1.0%sy,  0.0%ni, 98.3%id,  0.0%wa,  0.0%hi,  0.0%si
Mem:    319496k total,   314860k used,     4636k free,    19392k buff
Swap:   875500k total,   149128k used,   726372k free,   114676k cach

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6244 me        39  19 31752 3124 2188 S  6.3  1.0  16:24.42 trackerd
11071 me        20   0  2304 1092  840 R  1.3  0.3   0:00.14 top
 6180 me        20   0  2700 1100  772 S  0.7  0.3   0:03.66 dbus-dae
 6321 me        20   0 20944 7248 6560 S  0.7  2.3   2:51.38 multiloa
 4955 root      20   0  104m 9668 5776 S  0.3  3.0   2:19.39 Xorg
    1 root      20   0  2976  528  476 S  0.0  0.2   0:03.14 init
    2 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migratio
    4 root      15  -5     0    0    0 S  0.0  0.0   0:00.72 ksoftirq
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.04 watchdog
    6 root      15  -5     0    0    0 S  0.0  0.0   0:00.42 events/0
    7 root      15  -5     0    0    0 S  0.0  0.0   0:00.06 khelper
   41 root      15  -5     0    0    0 S  0.0  0.0   0:01.08 kblockd/
   67 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kseriod
  114 root      20   0     0    0    0 S  0.0  0.0   0:01.62 pdflush
  116 root      15  -5     0    0    0 S  0.0  0.0   0:02.44 kswapd0
```

The system summary contains a lot of good stuff. Here's a rundown:

*Table 10-3: `top` Information Fields*

| Row | Field | Meaning |
|---|---|---|
| 1 | `top` | The name of the program. |
| | `14:59:20` | The current time of day. |
| | `up 6:30` | This is called *uptime*. It is the amount of time since the machine was last booted. In this example, the system has been up for six-and-a-half hours. |

| | | |
|---|---|---|
| | `2 users` | There are two users logged in. |
| | `load average:` | *Load average* refers to the number of processes that are waiting to run, that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values less than 1.0 indicate that the machine is not busy. |
| 2 | `Tasks:` | This summarizes the number of processes and their various process states. |
| 3 | `Cpu(s):` | This row describes the character of the activities that the CPU is performing. |
| | `0.7%us` | 0.7 percent of the CPU is being used for *user processes*. This means processes outside the kernel. |
| | `1.0%sy` | 1.0 percent of the CPU is being used for *system* (kernel) processes. |
| | `0.0%ni` | 0.0 percent of the CPU is being used by “nice” (low-priority) processes. |
| | `98.3%id` | 98.3 percent of the CPU is idle. |
| | `0.0%wa` | 0.0 percent of the CPU is waiting for I/O. |
| 4 | `Mem:` | This shows how physical RAM is being used. |
| 5 | `Swap:` | This shows how swap space (virtual memory) is being used. |

The `top` program accepts a number of keyboard commands. The two most interesting are `h`, which displays the program's help screen, and `q`, which quits `top`.

Both major desktop environments provide graphical applications that display information similar to `top` (in much the same way that Task Manager in Windows works), but `top` is better than the graphical versions because it is faster and it consumes far fewer system resources. After all, our system monitor program shouldn't be the source of the system slowdown that we are trying to track.

## Controlling Processes

Now that we can see and monitor processes, let's gain some control over them. For our experiments, we're going to use a little program called `xlogo` as our guinea pig. The `xlogo` program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on our display go), which simply displays a re-sizable window containing the X logo. First, we'll get to know our test subject.

```
[me@linuxbox ~]$ xlogo
```

After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` may print a warning message, but it may be safely ignored.

> **Tip:** If your system does not include the `xlogo` program, try using `gedit` or `kwrite` instead.

We can verify that `xlogo` is running by resizing its window. If the logo is redrawn in the new size, the program is running.

Notice how our shell prompt has not returned? This is because the shell is waiting for the program to finish, just like all the other programs we have used so far. If we close the `xlogo` window, the prompt returns.


*Figure 4: The xlogo program*

### Interrupting a Process

Let's observe what happens when we run `xlogo` again. First, enter the `xlogo` command

and verify that the program is running. Next, return to the terminal window and press `Ctrl-c`.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

In a terminal, pressing `Ctrl-c`, *interrupts* a program. This means we are politely asking the program to terminate. After we pressed `Ctrl-c`, the `xlogo` window closed and the shell prompt returned.

Many (but not all) command-line programs can be interrupted by using this technique.

## Putting a Process in the Background

Let's say we wanted to get the shell prompt back without terminating the `xlogo` program. We can do this by placing the program in the *background*. Think of the terminal as having a *foreground* (with stuff visible on the surface like the shell prompt) and a background (with stuff hidden behind the surface). To launch a program so that it is immediately placed in the background, we follow the command with an ampersand (&) character.

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

After entering the command, the `xlogo` window appeared and the shell prompt returned, but some funny numbers were printed too. This message is part of a shell feature called *job control*. With this message, the shell is telling us that we have started job number 1 (`[1]`) and that it has PID 28236. If we run `ps`, we can see our process.

```
[me@linuxbox ~]$ ps
 PID TTY          TIME CMD
10603 pts/1    00:00:00 bash
28236 pts/1    00:00:00 xlogo
28239 pts/1    00:00:00 ps
```

The shell's job control facility also gives us a way to list the jobs that have been launched from our terminal. Using the `jobs` command, we can see this list:

```
[me@linuxbox ~]$ jobs
[1]+  Running                 xlogo &
```

The results show that we have one job, numbered 1, that it is running, and that the command was `xlogo &`.

## Returning a Process to the Foreground

A process in the background is immune from terminal keyboard input, including any attempt to interrupt it with `Ctrl-c`. To return a process to the foreground, use the `fg` command in this way:

```
[me@linuxbox ~]$ jobs
[1]+  Running                 xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

The `fg` command followed by a percent sign and the job number (called a *jobspec*) does the trick. If we only have one background job, the jobspec is optional. To terminate `xlogo`, press `Ctrl-c`.

## Stopping (Pausing) a Process

Sometimes we'll want to stop a process without terminating it. This is often done to allow a foreground process to be moved to the background. To stop a foreground process and place it in the background, press `Ctrl-z`. Let's try it. At the command prompt, type `xlogo`, press the `Enter` key, and then press `Ctrl-z`:

```
[me@linuxbox ~]$ xlogo
[1]+  Stopped                 xlogo
[me@linuxbox ~]$
```

After stopping `xlogo`, we can verify that the program has stopped by attempting to resize the `xlogo` window. We will see that it appears quite dead. We can either continue the program's execution in the foreground, using the `fg` command, or resume the program's execution in the background with the `bg` command:

```
[me@linuxbox ~]$ bg %1
```

```
[1]+ xlogo &
[me@linuxbox ~]$
```

As with the `fg` command, the jobspec is optional if there is only one job.

Moving a process from the foreground to the background is handy if we launch a graphical program from the command line, but forget to place it in the background by appending the trailing `&`.

Why would we want to launch a graphical program from the command line? There are two reasons.

- The program we want to run might not be listed on the window manager's menus (such as `xlogo`).
- By launching a program from the command line, we might be able to see error messages that would otherwise be invisible if the program were launched graphically. Sometimes, a program will fail to start up when launched from the graphical menu. By launching it from the command line instead, we may see an error message that will reveal the problem. Also, some graphical programs have interesting and useful command line options.

## Signals

The `kill` command is used to "kill" processes. This allows us to terminate programs that need killing (that is, some kind of pausing or termination). Here's an example:

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+  Terminated              xlogo
```

We first launch `xlogo` in the background. The shell prints the jobspec and the PID of the background process. Next, we use the `kill` command and specify the PID of the process we want to terminate. We could have also specified the process using a jobspec (for example, `%1`) instead of a PID.

While this is all very straightforward, there is more to it than that. The `kill` command doesn't exactly "kill" processes: rather it sends them *signals*. Signals are one of several ways that the operating system communicates with programs. We have already seen signals in action with the use of `Ctrl-c` and `Ctrl-z`. When the terminal receives one of these keystrokes, it sends a signal to the program in the foreground. In the case of `Ctrl-c`, a signal called `INT` (interrupt) is sent; with `Ctrl-z`, a signal called `TSTP` (terminal

stop) is sent. Programs, in turn, “listen” for signals and may act upon them as they are received. The fact that a program can listen and act upon signals allows a program to do things such as save work in progress when it is sent a termination signal.

## Sending Signals to Processes with `kill`

The `kill` command is used to send signals to programs. Its most common syntax looks like this:

```
kill [-signal] PID...
```

If no signal is specified on the command line, then the `TERM` (terminate) signal is sent by default. The `kill` command is most often used to send the following signals:

*Table 10-4: Common Signals*

| Number | Name | Meaning |
|---|---|---|
| 1 | HUP | Hangup. This is a vestige of the good old days when terminals were attached to remote computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has “hung up.” The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate.<br><br>This signal is also used by many daemon programs to cause a reinitialization. This means that when a daemon is sent this signal, it will restart and reread its configuration file. The Apache web server is an example of a daemon that uses the `HUP` signal in this way. |
| 2 | INT | Interrupt. This performs the same function as a `Ctrl-c` sent from the terminal. It will usually terminate a program. |
| 9 | KILL | Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways, including ignoring them all together, the `KILL` signal is never actually sent to |

| | | |
|---|---|---|
| | | the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to "clean up" after itself or save its work. For this reason, the KILL signal should be used only as a last resort when other termination signals fail. |
| 15 | TERM | Terminate. This is the default signal sent by the kill command. If a program is still "alive" enough to receive signals, it will terminate. |
| 18 | CONT | Continue. This will restore a process after a STOP or TSTP signal. This signal is sent by the bg and fg commands. |
| 19 | STOP | Stop. This signal causes a process to pause without terminating. Like the KILL signal, it is not sent to the target process, and thus it cannot be ignored. |
| 20 | TSTP | Terminal stop. This is the signal sent by the terminal when Ctrl-z is pressed. Unlike the STOP signal, the TSTP signal is received by the program, but the program may choose to ignore it. |

Let's try out the kill command:

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+  Hangup                  xlogo
```

In this example, we start the xlogo program in the background and then send it a HUP signal with kill. The xlogo program terminates, and the shell indicates that the background process has received a hangup signal. We may need to press the Enter key a couple of times before the message appears. Note that signals may be specified either by number or by name, including the name prefixed with the letters SIG.

```
[me@linuxbox ~]$ xlogo &
```

```
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+  Interrupt               xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+  Interrupt               xlogo
```

Repeat the example above and try the other signals. Remember, we can also use jobspecs in place of PIDs.

Processes, like files, have owners, and you must be the owner of a process (or the superuser) to send it signals with `kill`.

In addition to the list of signals above, which are most often used with `kill`, there are other signals frequently used by the system as listed in Table 10-5.

*Table 10-5: Other Common Signals*

| Number | Name | Meaning |
|---|---|---|
| 3 | QUIT | Quit. |
| 11 | SEGV | Segmentation violation. This signal is sent if a program makes illegal use of memory, that is, if it tried to write somewhere it was not allowed to write. |
| 28 | WINCH | Window change. This is the signal sent by the system when a window changes size. Some programs , such as `top` and `less` will respond to this signal by redrawing themselves to fit the new window dimensions. |

For the curious, a complete list of signals can be displayed with the following command:

```
[me@linuxbox ~]$ kill -l
```

## Sending Signals to Multiple Processes with `killall`

It's also possible to send signals to multiple processes matching a specified program or username by using the `killall` command. Here is the syntax:

```
killall [-u user] [-signal] name...
```

To demonstrate, we will start a couple of instances of the `xlogo` program and then terminate them.

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]-  Terminated              xlogo
[2]+  Terminated              xlogo
```

Remember, as with `kill`, we must have superuser privileges to send signals to processes that do not belong to us.

## Shutting Down the System

The process of shutting down the system involves the orderly termination of all the processes on the system, as well as performing some vital housekeeping chores (such as syncing all of the mounted file systems) before the system powers off. There are four commands that can perform this function. They are `halt`, `poweroff`, `reboot`, and `shutdown`. The first three are pretty self-explanatory and are generally used without any command line options. Here's an example:

```
[me@linuxbox ~]$ sudo reboot
```

The `shutdown` command is a bit more interesting. With it, we can specify which of the actions to perform (halt, power down, or reboot) and provide a time delay to the shutdown event. Most often it is used like this to halt the system:

```
[me@linuxbox ~]$ sudo shutdown -h now
```

or like this to reboot the system:

```
[me@linuxbox ~]$ sudo shutdown -r now
```

The delay can be specified in a variety of ways. See the `shutdown` man page for details. Once the `shutdown` command is executed, a message is "broadcast" to all logged-in users warning them of the impending event.

## More Process-Related Commands

Since monitoring processes is an important system administration task, there are a lot of commands for it. Table 10-6 lists some to play with:

*Table 10-6: Other Process Related Commands*

| Command | Description |
|---|---|
| `pstree` | Outputs a process list arranged in a tree-like pattern showing the parent-child relationships between processes. |
| `vmstat` | Outputs a snapshot of system resource usage including, memory, swap, and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. Here's an example: `vmstat 5`. Terminate the output with `Ctrl-c`. |
| `xload` | A graphical program that draws a graph showing system load over time. |
| `tload` | Similar to the `xload` program but draws the graph in the terminal. Terminate the output with `Ctrl-c`. |

## Summing Up

Most modern systems feature a mechanism for managing multiple processes. Linux provides a rich set of tools for this purpose. Given that Linux is the world's most deployed server operating system, this makes a lot of sense. However, unlike some other systems, Linux relies primarily on command line tools for process management. Though there are graphical process tools for Linux, the command line tools are greatly preferred because of their speed and light footprint. While the GUI tools may look pretty, they often create a lot of system load themselves, which somewhat defeats the purpose.