Reference Manual

Mandrakelinux 10.1

http://www.mandrakesoft.com

Chapter 1. Basic UNIX System Concepts

The name "UNIX®" may be familiar to some of you. You may even use a UNIX® system at work, in which case this chapter may not be very interesting.

For those of you who have never used a UNIX® system, reading this chapter is absolutely necessary. Understanding the concepts which will be introduced here will answer a surprisingly large number of questions commonly asked by beginners in the **GNU/Linux** world. Similarly some of these concepts will likely answer most of the problems you may encounter in the future.

1.1. Users and Groups

Since they have a direct influence on all other concepts, this chapter will introduce the concepts of users and groups which are extremely important.

Linux is a true ***multiuser*** system, and in order to use your GNU/Linux machine, you must have an ***account*** on the machine. When you created a user during installation, you actually created a account. In case you don't remember, you were prompted for the following items:

- the user's "real name" (which could actually be whatever you want)
- a ***login*** name
- and a ***password***.

The two most important parameters here are the login name (commonly abbreviated to login) and the password. You must have both of these in order to access the system.

When you create a user, a default group is also created. Later on, we will see that groups are useful when you want to share files with other people. A group may contain as many users as you wish, and it is very common to see such a separation in large systems. For example, at a university, you could have one group per department, another group for teachers, and so on. The opposite is also true: a user may be a member of one or more groups. A math teacher, for example, could be a member of the teachers' group and also of his math students' group.

Now that we've covered the background information, let us look at how to actually log in.

If the graphical interface (X) is started automatically on boot up, your start-up screen will look similar to figure 1-1.

Figure 1-1. Graphical Mode Login Session

In order to log in, you must first select your account from the list. A new dialog will be displayed, prompting you for your password . Note that you will have to type in your password blindly, because the characters will be echoed on screen as stars (*) instead of the characters you type in the password field. You may also choose your session type (window manager). Once you're ready, press the Login button.

If you are in console or "text" mode, you will be presented with something similar to the following:

```
Mandrakelinux Release 10.1 (CodeName) for i586
Kernel 2.6.8-3mdk on an i686 / tty1
```

```
[machine_name] login:
```

To log in, type your login name at the Login: prompt and press **Enter**. Next, the login program (login) will display a Password: prompt and wait for you to enter your password. Like the graphic mode login, the console login will not echo the characters you are typing on the screen.

Note that you can log in several times with the same account on additional ***consoles*** and under X. Each session you open is independent of the others, and it is even possible to open several X sessions at the same time (although this is not recommended since it consumes a lot of resources). By default, Mandrakelinux has six ***virtual consoles*** in addition to the one reserved for the graphical interface. You can switch to any of them by pressing the **Ctrl-Alt-F<n>** key sequence, where <n> is the number of the console that you want to switch to. By default, the graphical interface is on console number 7. Therefore, to switch to the second console, you would press the Ctrl, Alt and F2 keys.

During the installation, DrakX also prompted you for the password of a very special user: root. This is the system administrator who will most likely be yourself. For your system's security, it is very important for the root account to be always protected by a good and hard-to-guess password!

If you regularly log in as root, it can be very easy to make a mistake which could render your system unusable: one single mistake can do it. In particular, if you did not set a password for the root account, then **any** user can alter **any** part of your system (and even other operating systems on your machine!). Obviously this is not a good idea.

It is worth mentioning that internally, the system does not identify you by your login name. Instead, it uses a unique number assigned to the name: the *User ID* (UID) . Similarly every group is identified by its *Group ID* (GID) and not by its name.

1.2. File Basics

Compared to Windows® and most other ***operating systems***, files are handled very differently under GNU/Linux. In this section we will cover the most obvious differences. For more information, please read "*The Linux File System*", page 57.

The major differences result directly from the fact that Linux is a multiuser system: every file is the exclusive property of one user and one group. One thing we didn't mention about users and groups is that every one of them possesses a personal directory (called the ***home directory***). The user is the owner of this directory and of all files created in it.

However, this would not be very useful if that were the only notion of file ownership. As the file owner, a user may set **permissions** on files. These permissions distinguish between three user categories: the **owner** of the file, every user who is a member of the **group** associated with the file (also called the ***owner group***) but who is not the owner, and **others**, which includes every other user who is neither the owner nor a member of the owner's group.

There are three different permissions:

1. **Read** permission (r): enables a user to read the contents of a file. For a directory, the user can list its contents (i.e. the files in this directory).
2. **Write** permission (w): allows the modification of a file's content. For a directory, the write permission allows a user to add or remove files from this directory, even if he is not the owner of these files.
3. **eXecute** permission (x): enables a file to be executed (normally only executable files have this permission set). For a directory, it allows a user to ***traverse*** it, which means going into or through that directory. Note that this is different from the read access: you may be able to traverse a directory but still be unable to read its content!

Every permission combination is possible. For example, you can allow only yourself to read the file and forbid access to all other users. As the file owner, you can also change the owner group (if and only if you're a member of the new group).

Let's take the example of a file and a directory. The display below represents entering the `ls -l` command from the ***command line***:

```
$ ls -l
total 1
-rw-r-----   1 queen    users           0 Jul  8 14:11 a_file
drwxr-xr--   2 peter    users        1024 Jul  8 14:11 a_directory/
$
```

The results of the `ls -l` command are (from left to right):

- The first ten characters represent the file's type and the permissions associated with it. The first character is the file's type: if it's a regular file, you will see a dash (`-`). If it's a directory, the leftmost character will be a `d`. There are other file types, which we'll discuss later on. The next nine characters represent the permissions associated with that file. The nine characters are actually three groups of three permissions. The first group represents the rights associated with the file owner; the next three apply to all users belonging to the owner group; and the last three apply to others. A dash (`-`) means that the permission is not set.
- Next comes the number of links for the file. Later on we'll see that the unique identifier of a file is not its name, but a number (the ***inode number***), and that it's possible for one file on disk to have several names. For a directory, the number of links has a special meaning, which will also be discussed a bit further.
- The next piece of information is the name of the file owner and the name of the owner group.
- Finally, the size of the file (in ***bytes***) and its last modification time are displayed, with the name of the file or directory itself as the last item on the line.

Let's take a closer look at the permissions associated with each of these files. First of all, we must strip off the first character representing the type, and for the file `a_file`, we get the following rights: `rw-r-----`. Here's a breakdown of the permissions.

- the first three characters (`rw-`) are the owner's rights, which in this case is queen. Therefore, queen has the right to read the file (`r`), to modify its content (`w`) but not to execute it (`-`).
- the next three characters (`r--`) apply to any user who is not queen but who is a member of the `users` group. They will be able to read the file (`r`), but will not be able to write nor execute it (`--`).
- the last three characters (`---`) apply to any user who is not queen and is not a member of the `users` group. Those users don't have any rights on the file at all.

For the `a_directory` directory, the rights are `rwxr-xr--`, so:

- peter, as the directory owner, can list files contained inside (`r`), add to or remove files from that directory (`w`), and may traverse it (`x`).
- Each user who isn't peter, but is a member of the `users` group, will be able to list files in this directory (`r`), but not remove or add files (`-`), and will be able to traverse it (`x`).
- Every other user will only be able to list the contents of this directory (`r`). Because they don't have `wx` permissions, they won't be able to write files or enter the directory.

There is **one** exception to these rules: `root`. `root` can change attributes (permissions, owner and group owner) of all files, even if he's not the owner, and could therefore grant ownership of the file to himself! `root` can read files on which he has no read permissions, traverse directories which he would normally have no access to, and so on. And if `root` lacks a permission, he only has to add it. `root` has complete control over the system, which involves a certain amount of trust in the person wielding the `root` password.

Lastly, it's worth noting the differences between file names in the UNIX® and the Windows® worlds. For one, UNIX® allows for a much greater flexibility and has fewer limitations.

- A file name may contain any character, including non-printable ones, except for the ASCII character 0, which denotes the end of a string, and `/`, which is the directory separator. Moreover, because UNIX® is case sensitive, the files `readme` and `Readme` are different, because `r` and `R` are considered two **different** characters on UNIX®-based systems.
- As you may have noticed, a file name does not have to include an extension, unless that's the way you prefer to name your files. File extensions don't identify the content of files under GNU/Linux, nor almost any other operating system. So-called "file extensions" are quite convenient though. The period (`.`) under UNIX® is

just one character among others, but it also has one special meaning. Under UNIX®, file names beginning with a period are "hidden files"[1], which also includes directories whose names start with a `.`

However it's worth noting that many graphical applications (file managers, office applications, etc.) actually use file extensions to recognize their files. It is therefore a good idea to use file-name extensions for those applications which support them.

1.3. Processes

A ***process*** defines an instance of a program being executed and its ***environment***. We will only mention the most important differences between GNU/Linux and Windows® here (please refer to "*Process Control*", page 45 for more information).

The most important difference is directly related to the **user** concept: each process is executed with the rights of the user who launched it. Internally, the system identifies processes with a unique number, called the *process ID*, or PID. From this PID, the system knows who (that is, which user) has launched the process and a number of other pieces of information, and the system only needs to verify the process' validity. Let's take our `a_file` example. peter will be able to open this file in ***read-only mode***, but not in ***read-write mode*** because the permissions associated with the file forbid it. Once again the exception to this rule is `root`.

Because of this, GNU/Linux is virtually immune to viruses. In order to operate, viruses must infect executable files. As a user, you don't have write access to vulnerable system files, so the risk is greatly reduced. Generally speaking, viruses are very rare in the UNIX® world. There are only a few known viruses for Linux, and they are harmless when executed by a normal user. Only one user can damage a system by activating these viruses: `root`.

Interestingly enough, anti-virus software does exist for GNU/Linux, but mostly for DOS/Windows® files! Why are there anti-virus programs running on GNU/Linux which focus on DOS/Windows®? More and more often, you will see GNU/Linux systems acting as file servers for Windows® machines with the help of the Samba software package (see the Sharing Files and Printers chapter of the *Server Administration Guide*).

Linux makes it easy to control processes. One way is through "signals", which allow you to suspend or kill a process by sending it the corresponding signal. However, you are limited to sending signals to your own processes. With the exception of `root`, UNIX® does not allow you to send signals to a process launched by any other user. In "*Process Control*", page 45, you will learn how to obtain the PID of a process and to send it signals.

1.4. A Short Introduction to the Command Line

The command line is the most direct way to send commands to your machine. If you use the GNU/Linux command line, you will soon find that it is much more powerful and capable than other command prompts you may have encountered previously. This power is available because you have access, not only to all X applications, but also to thousands of other utilities in console mode (as opposed to graphical mode) which don't have graphical equivalents, with their many options and possible combinations, which would be hard to access in the form of buttons or menus.

Admittedly most people require a little help to get started. If you're not already working in console mode and are using the graphical interface, the first thing to do is to launch a terminal emulator. Access the main menu of GNOME, KDE or any other window manager you might be using and you will find a number of emulators in the System+Terminals menu. Choose the one you want, for example Konsole or XTerm. Depending on your user interface, there may also be an icon which clearly identifies it on the panel (figure 1-2).

1. By default, hidden files won't be displayed in a file manager, unless you tell it to. In a terminal, you must type the `ls -a` command to see all hidden files. Essentially, they hold configuration information. From your `home/` directory, take a look at `.mozilla` or `.openoffice` to see an example.

Figure 1-2. The Terminal Icon on the KDE Panel

When you launch this terminal emulator, you are actually using a shell. This is the name of the program which you interact with. You will find yourself in front of the ***prompt***:

```
[queen@localhost queen]$
```

This assumes that your user name is queen and that your machine's name is `localhost` (which is the case if your machine is not part of an existing network). Following the prompt there is space for you to type your commands. Note that when you're `root`, the prompt's `$` character becomes a `#` (this is true only in the default configuration, since you may customize all such details in GNU/Linux). In order to become `root`, type `su` after launching a shell.

```
# Enter the root password; (it will not appear on the screen)
[queen@localhost queen]$ su
Password:
# exit (or Ctrl-D) will take you back to your normal user account
[root@localhost queen]# exit
[queen@localhost queen]$
```

Everywhere else in this book, the prompt will be symbolically represented by a `$`, whether you are a normal user or `root`. You will be told when you have to be `root` to execute a command, so please remember the `su` command.

When you ***launch*** a shell for the first time, you normally find yourself in your `home/` directory. To display the name of the directory you are currently in, type `pwd` (which stands for *Print Working Directory*):

```
$ pwd
/home/queen
```

Next we will look at a few basic commands which are very useful.

1.4.1. cd: Change Directory

The `cd` command is just like the DOS one, with extras. It does just what its acronym states, changes the working directory. You can use `.` and `..`, which respectively stand for the current and parent directories. Typing `cd` alone will take you back to your home directory. Typing `cd -` will take you back to the last directory you visited. And lastly, you can specify peter's home directory by typing `cd ~peter` (`~` on its own means your own `home/` directory). Note that as a normal user, you cannot usually get into another user's `home/` directory (unless they explicitly authorized it or if this is the default configuration on the system), unless you are `root`, so let's become `root` and practice:

```
$ pwd
/root
# cd /usr/share/doc/HOWTO
# pwd
/usr/share/doc/HOWTO
# cd ../FAQ-Linux
# pwd
/usr/share/doc/FAQ-Linux
# cd ../../../lib
# pwd
/usr/lib
# cd ~peter
# pwd
/home/peter
# cd
# pwd
/root
```

Now, go back to being a normal user again by typing `exit` (or pressing **Ctrl**-**D**).

1.4.2. Some Environment Variables and the echo Command

All processes have their ***environment variables*** and the shell allows you to view them directly with the `echo` command. Some interesting variables are:

1. `HOME`: this environment variable contains a string which represents your home directory.
2. `PATH`: it contains the list of all directories in which the shell should look for executables when you type a command. Note that unlike DOS, by default, a shell will **not** look for commands in the current directory!
3. `USERNAME`: this variable contains your login name.
4. `UID`: this one contains your user ID.
5. `PS1`: it determines what your prompt will display, and is often a combination of special sequences. You may read the bash(1) ***manual page*** for more information by typing `man bash` in a terminal.

To have the shell print a variable's value, you must put a `$` in front of its name. Here's an example with the `echo` command:

```
$ echo Hello
Hello
$ echo $HOME
/home/queen
$ echo $USERNAME
queen
$ echo Hello $USERNAME
Hello queen
$ cd /usr
$ pwd
/usr
$ cd $HOME
$ pwd
/home/queen
```

As you can see, the shell substitutes the variable's value before it executes the command. Otherwise, our `cd $HOME` example would not have worked. In fact, the shell first replaced `$HOME` by its value (`/home/queen`) so the line became `cd /home/queen`, which is what we wanted. The same thing happened with the `echo $USERNAME` example.

If one of your environment variables doesn't exist, you can create them temporarily by typing `export ENV_VAR_NAME=value`. Once this is done, you can verify it has been created:

```
$ export USERNAME=queen $ echo $USERNAME queen
```

1.4.3. cat: Print the Contents of One or More Files to the Screen

Nothing much to say, this command does just that: it prints the contents of one or more files to the standard output, normally the screen:

```
$ cat /etc/fstab
/dev/hda5 / ext2 defaults 1 1
/dev/hda6 /home ext2 defaults 1 2
/dev/hda7 swap swap defaults 0 0
/dev/hda8 /usr ext2 defaults 1 2
/dev/fd0 /mnt/floppy auto sync,user,noauto,nosuid,nodev 0 0
none /proc proc defaults 0 0
none /dev/pts devpts mode=0620 0 0
/dev/cdrom /mnt/cdrom auto user,noauto,nosuid,exec,nodev,ro 0 0
$ cd /etc
$ cat modules.conf shells
alias parport_lowlevel parport_pc
pre-install plip modprobe parport_pc ; echo 7 > /proc/parport/0/irq
#pre-install pcmcia_core /etc/rc.d/init.d/pcmcia start
```

```
#alias char-major-14 sound
alias sound esssolo1
keep
/bin/zsh
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
/bin/ash
/bin/bsh
/usr/bin/zsh
```

1.4.4. less: a Pager

The name is a play on words related to the first pager ever used under UNIX® called `more`. A ***pager*** is a program which allows a user to view long files page by page (more accurately, screen by screen). The reason that we discuss `less` rather than `more` is that `less` is more intuitive. You should use `less` to view large files which will not fit on a single screen. For example:

```
less /etc/termcap
```

To browse through this file, use the up and down arrow keys. Press **Q** to quit. `less` can do far more than just that: press **H** for help to display the various options available.

1.4.5. ls: Listing Files

The `ls` (*LiSt*) command is equivalent to the `dir` command in DOS, but it can do much much more. In fact, this is largely because files can do more too. The command syntax for `ls` is:

```
ls [options] [file|directory] [file|directory...]
```

If no file or directory is specified on the command line, `ls` will list files in the current directory. Its options are numerous, so we will only describe a few of them:

- `-a`: lists all files, including ***hidden files***. Remember that in UNIX®, hidden files are those whose names begin with a `.`; the `-A` option lists "almost" all files, which means every file the `-a` option would print except for "." and ".."
- `-R`: lists recursively, i.e. all files and subdirectories of directories entered on the command line.
- `-s`: prints the file size in kilobytes next to each file.
- `-l`: prints additional information about the files such as the permissions associated to it, the owner and owner group, the file's size and the last-access time.
- `-i`: prints the inode number (the file's unique number in the file system, see "*The Linux File System*", page 57) next to each file.
- `-d`: treats directories on the command line as if they were normal files rather than listing their contents.

Here are some examples:

- `ls -R`: recursively lists the contents of the current directory.
- `ls -is images/ ..`: lists the inode number and the size in kilobytes for each file in the `images/` directory as well as in the parent directory of the current one.
- `ls -l images/*.png`: lists all files in the `images/` directory whose names end with `.png`, including the file `.png`, if it exists.

1.4.6. Useful Keyboard Shortcuts

There are a number of shortcuts available, their primary advantage being that they save you a lot of typing time. This section assumes you're using the default shell provided with Mandrakelinux, bash, but these keystrokes might work with other shells too.

First: the arrow keys. bash maintains a history of previous commands which you can view with the up and down arrow keys. You can scroll up to a maximum number of lines defined in the `HISTSIZE` environment variable. In addition, the history is persistent from one session to another, so you won't lose all the commands you typed in previous sessions.

The left and right arrow keys move the cursor left and right on the current line, allowing you to edit your commands. But there's more to editing than just moving one character at a time: **Ctrl-A** and **Ctrl-E**, for example, will take you to the beginning and the end of the current line. The **Backspace** and **Del** keys work as expected. **Backspace** and **Ctrl-H** are equivalent. **Del** and **Ctrl-D** can also be used interchangeably. **Ctrl-K** will delete from the position of the cursor to the end of line, and **Ctrl-W** will delete the word before the cursor (so will **Alt-Backspace**).

Typing **Ctrl-D** on a blank line will let you close the current session, which is much shorter than having to type `exit`. **Ctrl-C** will interrupt the currently running command, except if you were in the process of editing your command line, in which case it will cancel the editing and get you back to the prompt. **Ctrl-L** clears the screen. **Ctrl-Z** temporarily stops a task, it suspends it. This shortcut is very useful when you forget to type the "&" character after typing a command. For instance:

```
$ xpdf MyDocument.pdf
```

Hence you cannot use your shell anymore since the foreground task is allocated to the xpdf process. To put that task in the background and restore your shell, simply type **Ctrl-Z** and then `bg`.

Finally, there are **Ctrl-S** and **Ctrl-Q**, which are used to suspend and restore output to the screen. They are not used often, but you might type **Ctrl-S** by mistake (after all, **S** and **D** are close to each other on the keyboard). So, if you get into the situation where you're typing but you can't see any characters appearing on the Terminal, try **Ctrl-Q**. Note that all the characters you typed between the unwanted **Ctrl-S** and **Ctrl-Q** will be printed to the screen all at once.

Chapter 3. Introduction to the Command Line

In the chapter "*Basic UNIX System Concepts*", page 7, you were shown how to launch a shell. In this chapter, we will show you how to work with it.

The shell's main asset is the number of existing utilities: there are thousands of them, and each one is devoted to a particular task. We will only look at a (very) small number of them here. One of UNIX®'s greatest assets is the ability to combine these utilities, as we shall see later.

3.1. File-Handling Utilities

In this context, file handling means copying, moving and deleting files. Later, we will look at ways of changing file attributes (owner, permissions).

3.1.1. mkdir, touch: Creating Empty Directories and Files

`mkdir` (*MaKe DIRectory*) is used to create directories. Its syntax is simple:

```
mkdir [options] <directory> [directory ...]
```

Only one option is worth noting: the `-p` option. It does two things:

1. it will create parent directories if they did not exist previously. Without this option, `mkdir` would just fail, complaining that the said parent directories do not exist;
2. it will return silently if the directory you wanted to create already exists. Similarly, if you did not specify the `-p` option, `mkdir` will send back an error message, complaining that the directory already exists.

Here are some examples:

- `mkdir foo`: creates a directory `foo` in the current directory;
- `mkdir -p images/misc docs`: creates the `misc` directory in the `images` directory. First, it creates the latter if it does not exist (`-p`); it also creates a directory named `docs` in the current directory.

Initially, the `touch` command was not intended for creating files but for updating file access and modification times[1]. However, `touch` will create the files listed as empty files if they do not exist. The syntax is:

```
touch [options] file [file...]
```

So running the command:

```
touch file1 images/file2
```

will create an empty file called `file1` in the current directory and an empty file `file2` in directory `images`, if the files did not previously exist.

3.1.2. rm: Deleting Files or Directories

The `rm` command (*ReMove*) replaces the DOS commands `del` and `deltree`, and adds more options. Its syntax is as follows:

```
rm [options] <file|directory> [file|directory...]
```

Options include:

- `-r`, or `-R`: delete recursively. This option is **mandatory** for deleting a directory, empty or not. However, you can also use `rmdir` to delete empty directories.

1. In UNIX®, there are three distinct timestamps for each file: the last file access date (`atime`), i.e. the last date when the file was opened for read or write; the last date when the inode attributes were modified (`mtime`); and finally, the last date when the **content** of the file were modified (`ctime`).

- `-i`: request confirmation before each deletion. Note that by default in Mandrakelinux, `rm` is an ***alias*** to `rm -i`, for safety reasons (similar aliases exist for `cp` and `mv`). Your mileage may vary as to the usefulness of these aliases. If you want to remove them, you can create an empty `~/.alias` file which will prevent setting system wide aliases. Alternatively you can edit your `~/.bashrc` file to disable some of the system wide aliases by adding this line: `unalias rm cp mv`
- `-f`, the opposite of `-i`, forces deletion of the files or directories, even if the user has no write access on the files[2].

Some examples:

- `rm -i images/*.jpg file1`: deletes all files with names ending in `.jpg` in the `images` directory and deletes `file1` in the current directory, requesting confirmation for each file. Answer **y** to confirm deletion, **n** to cancel.
- `rm -Rf images/misc/ file*`: deletes, without requesting confirmation, the whole directory `misc/` in the `images/` directory, together with all files in the current directory whose names begin with `file`.

Using `rm` deletes files **irrevocably**. There is no way to restore them! (Well, actually there are several ways to do this but it is no trivial task.) Do not hesitate to use the `-i` option to ensure that you do not delete something by mistake.

3.1.3. mv: Moving or Renaming Files

The syntax of the `mv` (*MoVe*) command is as follows:

```
mv [options] <file|directory> [file|directory ...] <destination>
```

Some options:

- `-f`: forces operation — no warning if an existing file is overwritten.
- `-i`: the opposite. Asks the user for confirmation before overwriting an existing file.
- `-v`: ***verbose*** mode, report all changes and activity.

Some examples:

- `mv -i /tmp/pics/*.png .`: move all files in the `/tmp/pics/` directory whose names end with `.png` to the current directory (`.`), but request confirmation before overwriting any files already there.
- `mv foo bar`: rename file `foo` to `bar`. If a `bar` directory already existed, the effect of this command would be to move file `foo` or the whole directory (the directory itself plus all files and directories in it, recursively) into the `bar` directory.
- `mv -vf file* images/ trash/`: move, without requesting confirmation, all files in the current directory whose names begin with `file`, together with the entire `images/` directory to the `trash/` directory, and show each operation carried out.

3.1.4. cp: Copying Files and Directories

`cp` (*CoPy*) replaces the DOS commands `copy` and `xcopy` and adds more options. Its syntax is as follows:

```
cp [options] <file|directory> [file|directory ...] <destination>
```

`cp` has a lot of options. Here are the most common:

- `-R`: recursive copy; **mandatory** for copying a directory, even an empty directory.

2. It is enough for the user to have write access to the **directory** to be able to delete files in it, even if he is not the owner of the files.

- `-i`: request confirmation before overwriting any files which might be overwritten.
- `-f`: the opposite of `-i`, replaces any existing files without requesting confirmation.
- `-v`: verbose mode, displays all actions performed by `cp`.

Some examples:

- `cp -i /timages/* images/`: copies all files in the `/timages/` directory to the `images/` directory located in the current directory. It requests confirmation if a file is going to be overwritten.
- `cp -vR docs/ /shared/mp3s/* mystuff/`: copies the whole `docs` directory, plus all files in the `/shared/mp3s` directory to the `mystuff` directory.
- `cp foo bar`: makes a copy of the `foo` file with the name `bar` in the current directory.

3.2. Handling File Attributes

The series of commands shown here are used to change the owner or owner group of a file or its permissions. We looked at the different permissions in chapter `Basic UNIX System Concepts`.

3.2.1. chown, chgrp: Change the Owner and Group of One or More Files

The syntax of the `chown` (*CHange OWNer*) command is as follows:

```
chown [options] <user[:group]> <file|directory> [file|directory...]
```

The options include:

- `-R`: recursive. To change the owner of all files and subdirectories in a given directory.
- `-v`: verbose mode. Displays all actions performed by `chown`; reports which files have changed ownership as a result of the command and which files have not been changed.
- `-c`: like `-v`, but only reports which files have been changed.

Some examples:

- `chown nobody /shared/book.tex`: changes the owner of the `/shared/book.tex` file to `nobody`.
- `chown -Rc queen:music *.mid concerts/`: changes the ownership of all files in the current directory whose name ends with `.mid` and all files and subdirectories in the `concerts/` directory to user `queen` and group `music`, reporting only files affected by the command.

The `chgrp` (*CHange GRouP*) command lets you change the group ownership of a file (or files); its syntax is very similar to that of `chown`:

```
chgrp [options] <group> <file|directory> [file|directory...]
```

The options for this command are the same as for `chown`, and it is used in a very similar way. Thus, the command:

```
chgrp disk /dev/hd*
```

changes the ownership of all files in directory `/dev/` with names beginning with `hd` to group `disk`.

3.2.2. chmod: Changing Permissions on Files and Directories

The `chmod` (*CHange MODe*) command has a very distinct syntax. The general syntax is:

```
chmod [options] <change mode> <file|directory> [file|directory...]
```

but what distinguishes it is the different forms that the mode change can take. It can be specified in two ways:

1. in octal. The owner user permissions then correspond to figures with the form `<x>00`, where `<x>` corresponds to the permission assigned: 4 for read permission, 2 for write permission and 1 for execute permission. Similarly, the owner group permissions take the form `<x>0` and permissions for "others" the form `<x>`. Then, all you need to do is add together the assigned permissions to get the right mode. Thus, the permissions `rwxr-xr--` correspond to 400+200+100 (owner permissions, `rwx`) +40+10 (group permissions, `r-x`) +4 (others' permissions, `r--`) = 754; in this way, the permissions are expressed in absolute terms. This means that previous permissions are unconditionally replaced;
2. with expressions. Here permissions are expressed by a sequence of expressions separated by commas. Hence an expression takes the following form: `[category]<+|-|=><permissions>`.

   The category may be one or more of:

   - `u` (*User*, permissions for owner);
   - `g` (*Group*, permissions for owner group);
   - `o` (*Others*, permissions for "others").

   If no category is specified, changes will apply to all categories. A `+` sets a permission, a `-` removes the permission and a `=` sets the permission. Finally, the permission is one or more of the following:

   - `r` (*Read*);
   - `w` (*Write*) or;
   - `x` (*eXecute*).

The main options are quite similar to those of `chown` and `chgrp`:

- `-R`: changes permissions recursively.
- `-v`: verbose mode. Displays the actions carried out for each file.
- `-c`: like `-v` but only shows files affected by the command.

Examples:

- `chmod -R o-w /shared/docs`: recursively removes write permission for `others` on all files and subdirectories in the `/shared/docs/` directory.
- `chmod -R og-w,o-x private/`: recursively removes write permission for `group` and `others` for the whole `private/` directory, and removes the execution permission for `others`.
- `chmod -c 644 misc/file*`: changes permissions of all files in the `misc/` directory whose names begin with `file` to `rw-r--r--` (i.e. read permission for everyone and write permission only for the owner), and reports only files affected by this command.

3.3. Shell Globbing Patterns

You probably already use ***globbing*** characters without knowing it. When you specify a file in Windows® or when you look for a file, you use * to match a random string. For example, *.txt matches all files with names ending with .txt. We also used it heavily in the last section. But there is more to globbing than just *.

When you type a command like ls *.txt and press Enter, the task of finding which files match the *.txt pattern is not done by the ls command, but by the shell itself. This requires a little explanation about how a command line is interpreted by the shell. When you type:

```
$ ls *.txt
      readme.txt  recipes.txt
```

the command line is first split into words (ls and *.txt in this example). When the shell sees a * in a word, it will interpret the whole word as a globbing pattern and will replace it with the names of all matching files. Therefore, the command, just before the shell executes it, has become ls readme.txt recipe.txt, which gives the expected result. Other characters make the shell react this way too:

- ?: matches one and only one character, regardless of what that character is;
- [...]: matches any character found in the brackets. Characters can be referred to either as a range of characters (i.e. 1-9) or ***discrete values***, or even both. Example: [a-zBE5-7] will match all characters between a and z, a B, an E, a 5, a 6 or a 7;
- [!...]: matches any character **not** found in the brackets. [!a-z], for example, will match any character which is not a lowercase letter[3];
- {c1,c2}: matches c1 or c2, where c1 and c2 are also globbing patterns, which means you can write {[0-9]*,[acr]} for example.

Here are some patterns and their meanings:

- /etc/*conf: all files in the /etc directory with names ending in conf. It can match /etc/inetd.conf, /etc/conf.linuxconf, **and also** /etc/conf if such a file exists. Remember that * can also match an empty string.
- image/{cars,space[0-9]}/*.jpg: all file names ending with .jpg in directories image/cars, image/space0, (...), image/space9, if those directories exist.
- /usr/share/doc/*/README: all files named README in all of /usr/share/doc's immediate subdirectories. This will make /usr/share/doc/mandrake/README match, for example, but not /usr/share/doc/myprog/doc/README.
- *[!a-z]: all files with names which do **not** end with a lowercase letter in the current directory.

3.4. Redirections and Pipes

3.4.1. A Little More About Processes

To understand the principle of redirections and pipes, we need to explain a notion about processes which has not yet been introduced. Most UNIX® processes (this also includes graphical applications but excludes most daemons) use a minimum of three file descriptors: standard input, standard output and standard error. Their respective numbers are 0, 1 and 2. In general, these three descriptors are associated with the terminal from which the process was started, with the input being the keyboard. The aim of redirections and pipes is to redirect these descriptors. The examples in this section will help you better understand this concept.

3. Beware! While this is true for most languages, this may not be true for your own language setting (locale). This depends on the **collating order**. On some language configurations, [a-z] will match a, A, b, B, (...), z. And we do not even mention the fact that some languages have accentuated characters...

3.4.2. Redirections

Imagine, for example, that you wanted a list of files ending with `.png`[4] in the `images` directory. This list is very long, so you may want to store it in a file in order to look through it at your leisure. You can enter the following command:

```
$ ls images/*.png 1>file_list
```

This means that the standard output of this command (`1`) is redirected (`>`) to the file named `file_list`. The `>` operator is the output redirection operator. If the redirection file does not exist, it is created, but if it does exist, its previous contents are overwritten. However, the default descriptor redirected by this operator is the standard output and does not need to be specified on the command line. So you can write more simply:

```
$ ls images/*.png >file_list
```

and the result will be exactly the same. Then you could look at the file using a text file viewer such as `less`.

Now, imagine you want to know how many of these files exist. Instead of counting them by hand, you can use the utility called `wc` (*Word Count*) with the `-l` option, which writes on the standard output the number of lines in the file. One solution is as follows:

```
wc -l 0<file_list
```

and this gives the desired result. The `<` operator is the input redirection operator, and the default redirected descriptor is the standard input one, i.e. `0`, and you simply need to write the line:

```
wc -l <file_list
```

Now suppose you want to remove all the file "extensions" and put the result in another file. One tool for doing this is `sed` (*Stream EDitor*). You simply redirect the standard input of `sed` to the `file_list` file and redirect its output to the result file, i.e. `the_list`:

```
sed -e 's/\.png$//g' <file_list >the_list
```

and your list is created, ready for you to view at your leisure with any viewer.

It can also be useful to redirect standard errors. For example, you want to know which directories in `/shared` you cannot access: one solution is to list this directory recursively and to redirect the errors to a file, while not displaying the standard output:

```
ls -R /shared >/dev/null 2>errors
```

which means that the standard output will be redirected (`>`) to `/dev/null`, a special file in which everything you write is discarded (i.e. the standard output is not displayed) and the standard error channel (`2`) is redirected (`>`) to the `errors` file.

3.4.3. Pipes

Pipes are in some ways a combination of input and output redirections. The principle is that of a physical pipe, hence the name: one process sends data into one end of the pipe and another process reads the data at the other end. The pipe operator is `|`. Let us go back to the file list example above. Suppose you want to find out directly how many corresponding files there are without having to store the list in a temporary file, you would then use the following command:

```
ls images/*.png | wc -l
```

which means that the standard output of the `ls` command (i.e. the list of files) is redirected to the standard input of the `wc` command. This then gives you the desired result.

You can also directly put together a list of files "without extensions" using the following command:

```
ls images/*.png | sed -e 's/\.png$//g' >the_list
```

4. You might think it is crazy to say "files ending with `.png`" rather than "PNG images". However, once again, files under UNIX® only have an extension by convention: extensions in no way define a file type. A file ending with `.png` could perfectly well be a JPEG image, an application file, a text file or any other type of file. The same is true under Windows® as well!

or, if you want to consult the list directly without storing it in a file:

```
ls images/*.png | sed -e 's/\.png$//g' | less
```

Pipes and redirections are not restricted solely to text which can be read by human beings. For example, the following command sent from a Terminal:

```
xwd -root | convert - ~/my_desktop.png
```

will send a screen shot of your desktop to the `my_desktop.png` [5] file in your personal directory.

3.5. Command-Line Completion

Completion is a very handy function, and all modern shells (including bash) have it. Its role is to give the user as little work to do as possible. The best way to illustrate completion is to give an example.

3.5.1. Example

Suppose your personal directory contains the `file_with_very_long_name_impossible_to_type` file, and you want to look at it. Suppose you also have, in the same directory, another file called `file_text`. You are in your personal directory, so type the following sequence:

```
$ less fi<TAB>
```

(i.e., type `less fi` and then press the `TAB` key). The shell will then expand the command line for you:

```
$ less file_
```

and also give the list of possible choices (in its default configuration, which can be customized). Then type the following key sequence:

```
less file_w<TAB>
```

and the shell will extend the command line to give you the result you want:

```
less file_with_very_long_name_impossible_to_type
```

All you need to do then is press the `Enter` key to confirm and read the file.

3.5.2. Other Completion Methods

The TAB key is not the only way to activate completion, although it is the most common one. As a general rule, the word to be completed will be a command name for the first word of the command line (`nsl<TAB>` will give `nslookup`), and a file name for all the others, unless the word is preceded by a "magic" character like `~`, `@` or `$`, in which case the shell will try to complete a user name, a machine name or an environment variable name respectively[6]. There is also a magic character for completing a file name (`/`) and a command to recall a command from the history (`!`).

The other two ways to activate completion are the sequences `Esc-<x>` and `Ctrl+x <x>`, where `<x>` is one of the magic characters already mentioned. `Esc-<x>` will attempt to come up with a unique completion. If it fails, it will complete the word with the largest possible substring in the choice list. A ***beep*** means either that the choice is not unique, or simply that there is no corresponding choice. The sequence `Ctrl+x <x>` displays the list of possible choices without attempting any completion. Pressing the `TAB` key is the same as successively pressing `Esc-<x>` and `Ctrl+x <x>`, where the magic character depends on the context.

Thus, one way to see all the environment variables defined is to type the sequence `Ctrl+x $` on a blank line. Another example: if you want to see the man page for the `nslookup` command, you simply type `man nsl` then `Esc-!`, and the shell will automatically complete the command to `man nslookup`.

5. Yes, it will indeed be a PNG image (however, the ImageMagick package needs to be installed...).
6. Remember: UNIX® differentiates between uppercase and lowercase. The `HOME` environment variable and the `home` variable are not the same.

3.6. Starting and Handling Background Processes: Job Control

You have probably noticed that when you enter a command from a Terminal, you normally have to wait for the command to finish before the shell returns control to you. This means that you have sent the command in the ***foreground***. However, there are occasions when this is not desirable.

Suppose, for example, that you decide to copy a large directory recursively to another. You have also decided to ignore errors, so you redirect the error channel to `/dev/null`:

```
cp -R images/ /shared/ 2>/dev/null
```

Such a command can take several minutes until it is fully executed. You then have two solutions: the first one is violent, and means stopping (killing) the command and then doing it again when you have the time. To do this, press `Ctrl+c`: this will terminate the process and take you back to the prompt. But wait, don't do it yet! Read on.

Suppose you want the command to run while you do something else. The solution is then to put the process into the ***background***. To do this, press `Ctrl+z` to suspend the process:

```
$ cp -R images/ /shared/ 2>/dev/null
# Type C-z here
[1]+  Stopped                 cp -R images/ /shared/ 2>/dev/null
$
```

and there you are again at the prompt. The process is then on standby, waiting for you to restart it (as shown by the `Stopped` keyword). That, of course, is what you want to do, but in the background. Type `bg` (for *Back-Ground*) to get the desired result:

```
$ bg
      [1]+ cp -R images/ /shared/ 2>/dev/null &
      $
```

The process will then start running again as a background task, as indicated by the `&` (ampersand) sign at the end of the line. You will then be back at the prompt and able to continue working. A process which runs as a background task, or in the background, is called a background ***job***.

Of course, you can start processes directly as background tasks by adding an **&** character at the end of the command. For example, you can start the command to copy the directory in the background by writing:

```
cp -R images/ /shared/ 2>/dev/null &
```

If you want, you can also restore this process to the foreground and wait for it to finish by typing `fg` (*ForeGround*). To put it into the background again, type the sequence `Ctrl+z`, `bg`.

You can start several jobs this way: each command will then be given a job number. The shell command `jobs` lists all the jobs associated with the current shell. The job preceded by a + sign indicates the last process begun as a background task. To restore a particular job to the foreground, you can then type `fg <n>` where `<n>` is the job number, i.e. `fg 5`.

Note that you can also suspend or start ***full-screen*** applications this way, such as `less` or a text editor like Vi, and restore them to the foreground when you want.

3.7. A Final Word

As you can see, the shell is very comprehensive and using it effectively is a matter of practice. In this relatively long chapter, we have only mentioned a few of the available commands: Mandrakelinux has thousands of utilities, and even the most experienced users do not make use of them all.

There are utilities for all tastes and purposes: you have utilities for handling images (like `convert` mentioned above, but also GIMP ***batch*** mode and all ***pixmap*** handling utilities), sound (Ogg Vorbis encoders, audio CD players), CD writing, e-mail clients, FTP clients and even web browsers (like `lynx` or `links`), not to mention all the administration tools.

Even if graphical applications with equivalent functions do exist, they are usually graphical interfaces built over these very same utilities. In addition, command-line utilities have the advantage of being able to operate in non-interactive mode: you can start writing a CD and then log off the system with the confidence that the writing will take place (see the nohup(1) man page or the screen(1) man page).

Chapter 4. Text Editing: Emacs and VI

As stated in the introduction, text editing[1] is a fundamental feature when using a UNIX® system. The two editors we are going to take a quick look at are a little difficult to use initially, but once you understand the basics, each one can prove to be a powerful tool. This is particularly because of the numerous edit modes available which provide specific editing features for a great variety of file types (perl, C++, XML, etc.).

4.1. Emacs

Emacs is probably the most powerful text editor in existence. It can do absolutely everything and is infinitely extensible through its built-in lisp-based programming language. With Emacs, you can move around the web, read your mail, take part in Usenet newsgroups, make coffee, and so on. This is not to say that you will learn how to do all of that in this chapter, but you will get a good start with opening Emacs, editing one or more files, saving them and quitting Emacs.

If, after reading this, you wish to learn more about Emacs, you can have a look at this Tutorial Introduction to GNU Emacs (`http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html`).

4.1.1. Short Presentation

Invoking Emacs is done as follows:

```
emacs [file] [file...]
```

Emacs will open every file entered as an argument into a separate buffer, with a maximum of two buffers visible at a time. If you start Emacs without specifying any files on the command line you will be placed into a buffer called `*scratch*`. If you are in X, menus will be available, but in this chapter we will concentrate on working strictly with the keyboard.

4.1.2. Getting Started

It's now time to get some hands-on experience. For our example, let us start by opening two files, `file1` and `file2`. If these files do not exist, they will be created as soon as you write something in them:

```
$ emacs file1 file2
```

By typing that command, the following window will be displayed:

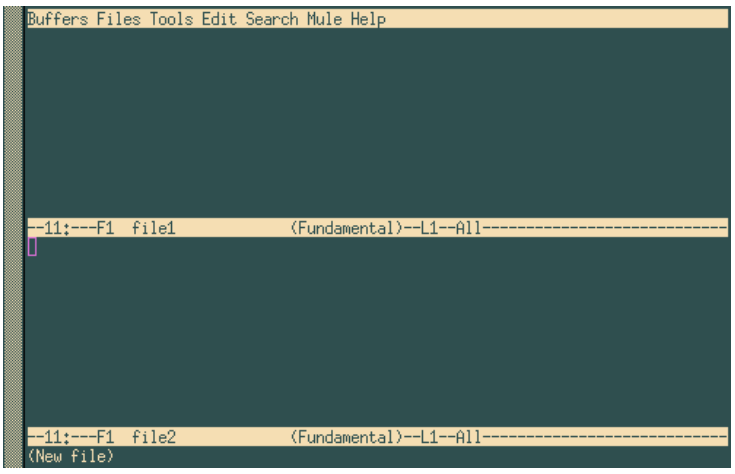

Figure 4-1. Editing Two Files at Once

1. "To edit text" means to modify the content of a file containing only letters, digits, and punctuation symbols. It contains no layout information such as fonts, quadding, etc. Such files may be e-mail messages, source code, documents, or even configuration files.

As you can see, one buffer has been created. A third one is also present at the bottom of the screen (where you see `(New file)`). That is the mini-buffer. You cannot access this buffer directly. You must be invited by Emacs during interactive entries. To change the current buffer, type `Ctrl+x o`. You type text just as in a "normal" editor, deleting characters with the `DEL` or `Backspace` key.

To move around, you can use the arrow keys, or you could use the following key combinations: `Ctrl+a` to go to the beginning of the line, `Ctrl+e` to go to the end of the line, `Alt+<` to go to the beginning of the buffer and `Alt+>` to go to the end of the buffer. There are many other combinations, even ones for each of the arrow keys [2].

Once you are ready to save your changes to disk, type `Ctrl+x Ctrl+s`, or if you want to save the contents of the buffer to another file, type `Ctrl+x Ctrl+w`. Emacs will ask you for the name of the file that the contents of the buffer should be written to. You can use ***completion*** to do this.

4.1.3. Handling buffers

If you want, you can switch to displaying a single buffer on the screen. There are two ways of doing this:

- If you are in the buffer that you want to hide: type `Ctrl+x 0`
- If you are in the buffer which you want to keep on the screen: type `Ctrl+x 1`.

There are two ways of restoring a buffer back to the screen:

- type `Ctrl+x b` and enter the name of the buffer you want, or
- type `Ctrl+x Ctrl+b`. This will open a new buffer called `*Buffer List*`. You can move around this buffer using the sequence `Ctrl+x o`, then select the buffer you want and press the `Enter` key, or else type the name of the buffer in the mini-buffer. The buffer `*Buffer List*` returns to the background once you have made your choice.

If you have finished with a file and you want to get rid of the associated buffer, type `Ctrl+x k`. Emacs will then ask you which buffer it should close. By default, this will be the buffer you are currently in. If you want to get rid of a buffer other than the one suggested, enter its name directly or press `TAB`: Emacs will open yet another buffer called `*Completions*` giving the list of possible choices. Confirm the choice with the `Enter` key.

You can also restore two visible buffers to the screen at any time. To do this type `Ctrl+x 2`. By default, the new buffer created will be a copy of the current buffer (which enables you, for example, to edit a large file in several places "at the same time"). To move between buffers, use the commands that were previously mentioned.

You can open other files at any time, using `Ctrl+x Ctrl+f`. Emacs will prompt you for the file name and you can again use completion if you find it more convenient.

4.1.4. Copy, Cut, Paste, Search

Suppose you find yourself in the following situation: figure 4-2.

2. Emacs has been designed to work on a great variety of machines, some of which do not have arrow keys on their keyboards. This is even more true of Vi.

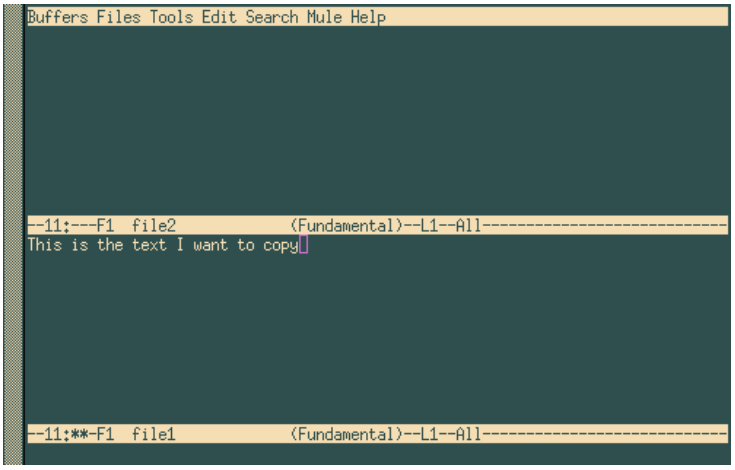

Figure 4-2. Emacs, before copying the text block

First off, you will need to select the text you want to copy. In this example we want to copy the entire sentence. The first step is to place a mark at beginning of the area. Assuming the cursor is in the position where it is in figure 4-2, the command sequence would be `Ctrl+ SPACE` (`Control` + space bar). Emacs will display the message `Mark set` in the mini-buffer. Next, move to the beginning of the line with `Ctrl+a`. The area selected for copying or cutting is the entire area located between the mark and the cursor's current position, so in this case it will be the entire line of text. There are two command sequences available: `Alt+w` (to copy) or `Ctrl+w` (to cut). If you copy, Emacs will briefly return to the mark position so that you can view the selected area.

Finally, go to the buffer where you want the text to end up and type `Ctrl+y`. This will give you the following result:

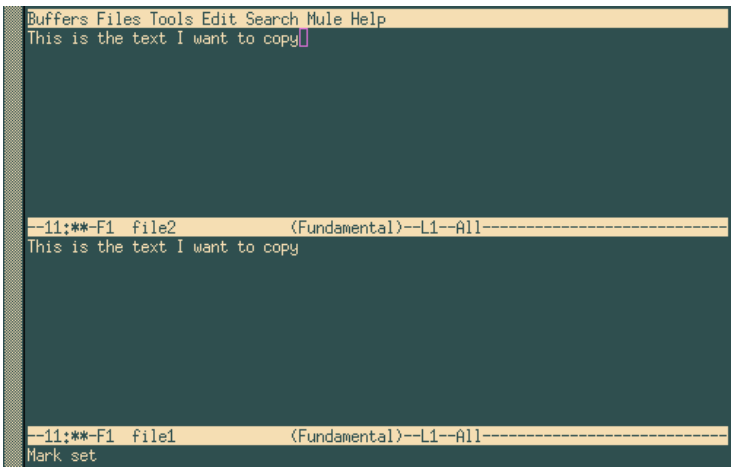

Figure 4-3. Copying Text with emacs

In fact, what you've done is copy text to Emacs's ***kill ring***. This kill ring contains all of the areas copied or cut since Emacs was started. **Any** area just copied or cut is placed at the top of the kill ring. The `Ctrl+y` sequence only "pastes" the area at the top. If you want to access any of the other areas, press `Ctrl+y` then `Alt+y` until you get to the desired text.

To search for text, go to the desired buffer and type `Ctrl+s`. Emacs will ask you what string it should search for. To continue a search with the same string in the current buffer, just type `Ctrl+s` again. When Emacs reaches the end of the buffer and finds no more occurrences, you can type `Ctrl+s` again to restart the search from the beginning of the buffer. Pressing the `Enter` key ends the search.

To search and replace, type `Alt+%`. Emacs asks you what string to search for, what to replace it with, and asks for confirmation for each occurrence it finds.

To Undo, type `Ctrl+x u` which will undo the previous operation. You can undo as many operations as you want.

4.1.5. Quit emacs

The shortcut to quit Emacs is `Ctrl+x Ctrl+c`. If you have not saved your changes, Emacs will ask you whether you want to save your buffers or not.

4.2. Vi: the ancestor

Vi was the first full-screen editor in existence. It is one of the main programs UNIX® detractors point to, but also one of the better arguments of its defenders: while it is complicated to learn, it is also an extremely powerful tool once you get into the habit of using it. With a few keystrokes, a Vi user can move mountains, and other than Emacs, few text editors can make the same claims.

The version supplied with Mandrakelinux is in fact Vim, for *VI iMproved*, but we will refer to it as Vi throughout this chapter.

If you wish to learn more about Vi, you can have a look at this Hands-On Introduction to the Vi Editor (`http://www.library.yale.edu/wsg/docs/vi_hands_on/`) or at the Vim home page (`http://www.vim.org/`).

4.2.1. Insert Mode, Command Mode, ex Mode...

To begin using Vi we use the same sort of command line as we did with Emacs. So let us go back to our two files and type:

```
$ vi file1 file2
```

At this point, you will find yourself looking at a window resembling the following one:

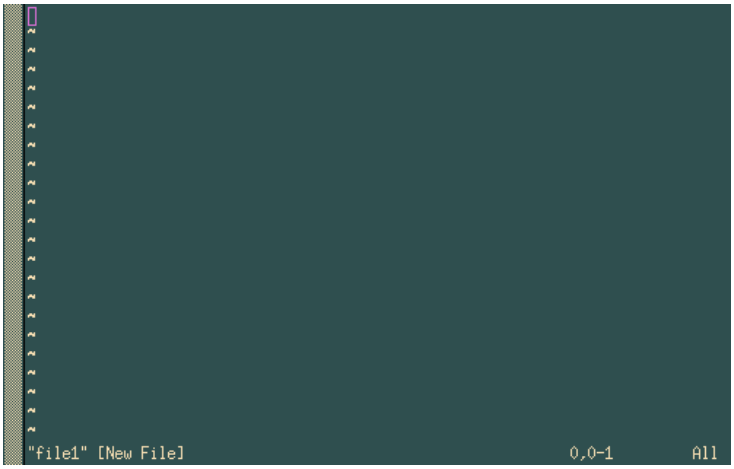

Figure 4-4. Starting position in VIM

You are now in ***command mode*** in front of the first opened file. In this mode you cannot insert text into a file. To do so you must switch to ***insert mode***.

Here are some shortcuts to inserting text:

- **a** and **i**: to insert text after and before the cursor (**A** and **I** insert text at the end and at the beginning of the current line);
- **o** and **O** : to insert text below and above the current line.

In insert mode, you will see the string `--INSERT--` appear at the bottom of the screen (so you know which mode you are in). This is the only mode which will allow you to insert text. To return to command mode, press the `Esc` key.

In insert mode, you can use the `Backspace` and `DEL` keys to delete text as you go along. The arrow keys will allow you to move around within the text in Command mode and Insert mode. In command mode, there are also other key combinations which we will look at later.

`ex` mode is accessed by pressing the **:** key in command mode. A **:** will appear at the bottom left of the screen with the cursor positioned after it. Vi will consider everything you type up to the `Enter` key as an `ex` command. If you delete the command and the **:** you typed in, you will be returned to command mode and the cursor will go back to its original position in the text.

To save changes to a file type `:w` in command mode. If you want to save the contents of the buffer to another file, type `:w <file_name>`.

4.2.2. Handling Buffers

To move, in the same buffer, between the files whose names were passed on the command line, type `:next` to move to the next file and `:prev` to move to the previous file. You can also use `:e <file_name>`, which allows you to either change to the desired file if it is already open, or to open another file. You may also use completion.

As with Emacs, you can have several buffers displayed on the screen. To do this, use the `:split` command.

To change buffers, type `Ctrl+w j` to go to the buffer below or `Ctrl+w k` to go to the buffer above. You can also use the up and down arrow keys instead of **j** or **k**. The `:close` command hides a buffer and the `:q` command closes it.

You should be aware that if you try to hide or close a buffer without saving the changes, the command will not be carried out and Vi will display this message:

```
No write since last change (use ! to override)
```

In this case, do as you are told and type `:q!` or `:close!`.

4.2.3. Editing Text and Move Commands

Apart from the `Backspace` and `DEL` keys in edit mode, Vi has many other commands for deleting, copying, pasting, and replacing text in command mode. All the commands shown hereafter are in fact separated into two parts: the action to be performed and its effect. The action may be:

- **c**: to replace (*Change*). The editor deletes the requested text and goes back into insert mode after this command.
- **d**: to delete (*Delete*);
- **y**: to copy ("Yank"). We will look at this in the next section.
- **.**: repeats last action.

The effect defines which group of characters the command acts upon.

- **h**, **j**, **k**, **l**: one character left, down, up, right[3]
- **e**, **b**, **w**: to the end, beginning of the current word and the beginning of the next word.
- **^**, **0**, **$**: to the first non-blank character of the current line, the beginning of the current line, and the end of current line.
- `f<x>`: go to next occurrence of character `<x>`. For example, `fe` moves the cursor to the next occurrence of the character **e**.
- `/<string>`, `?<string>`: to the next and previous occurrence of string or regexp `<string>`. For example, `/foobar` moves the cursor to the next occurrence of the word `foobar`.
- {, }: to the beginning, to the end of current paragraph;
- **G**, **H**: to end of file, to beginning of screen.

3. A shortcut for `dl` (delete one character forward) is **x**; a shortcut for `dh` is **X**; `dd` deletes the current line.

Each of these "effect" characters or move commands can be preceded by a repetition number. For **G**, ("Go") this references the line number in the file. Based on this information, you can make all sorts of combinations.

Here are some examples:

- `6b`: moves 6 words backwards;
- `c8fk`: delete all text until the eighth occurrence of the character **k** then go into insert mode;
- `91G`: goes to line 91 of the file;
- `d3$`: deletes up to the end of the current line plus the next two lines.

While many of these commands are not very intuitive, the best method to remember the commands is to practice them. But you can see that the expression "move mountains with a few keys" is not much of an exaggeration.

4.2.4. Cut, Copy, Paste

Vi contains a command which we have already seen for copying text: the **y** command. To cut text, simply use the **d** command. There are 27 memories or buffers for storing text: an anonymous memory and 26 memories named after the 26 lowercase letters of the alphabet.

To use the anonymous memory you enter the command "as is". So the command `y12w` will copy the 12 words after the cursor into anonymous memory[4]. Use `d12w` if you want to cut this area.

To use one of the 26 named memories, enter the sequence `"<x>` before the command, where `<x>` gives the name of the memory. Therefore, to copy the same 12 words into the memory **k**, you would write `"ky12w`, or `"kd12w` to cut them.

To paste the contents of the anonymous memory, use the commands **p** or **P** (for *Paste*), to insert text after or before the cursor. To paste the contents of a named memory, use `"<x>p` or `"<x>P` in the same way (for example `"dp` will paste the contents of memory `d` after the cursor).

Let us look at an example:.

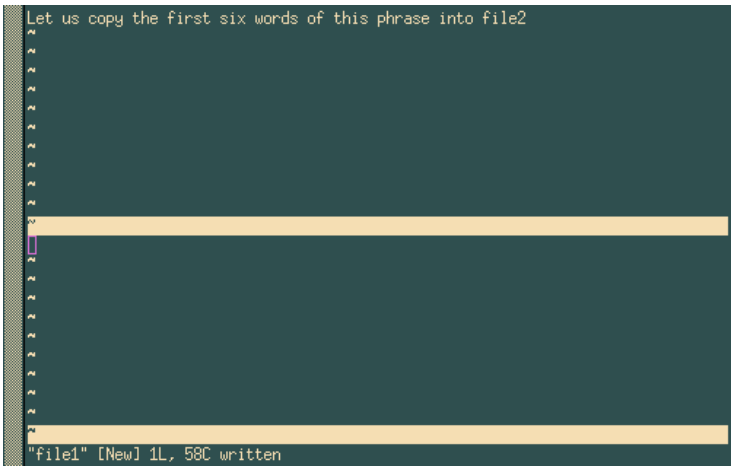

Figure 4-5. VIM, before copying the text block

To carry out this action, we will:

- recopy the first 6 words of the sentence into memory **r** (for example): `"ry6w`[5];
- go into the buffer `file2`, which is located below: `Ctrl+w j`;
- paste the contents of memory **r** before the cursor: `"rp`.

We get the expected result, as shown in figure 4-6.

4. But only if the cursor is positioned at the beginning of the first word!
5. `y6w` literally means: "Yank 6 words".

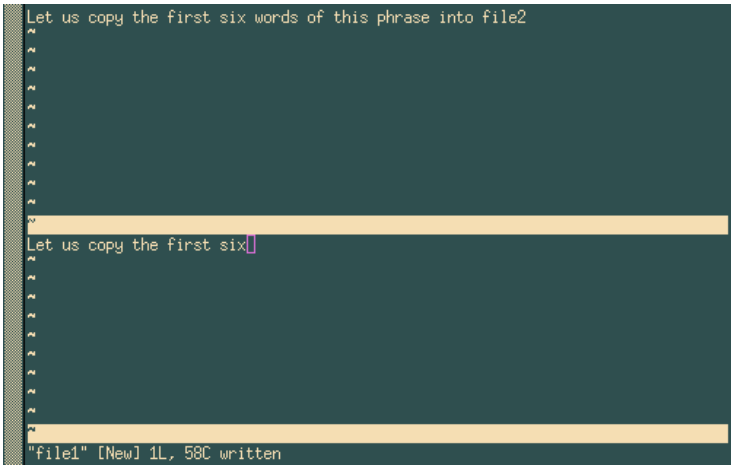

Figure 4-6. VIM, after having copied the text block

Searching for text is very simple: in command mode, you simply type **/** followed by the string to search for, and then press the `Enter` key. For example, `/party` will search for the string `party` from the current cursor position. Pressing **n** takes you to the next occurrence, and if you reach the end of the file, the search will start again from the beginning. To search backwards, use **?** instead of **/**.

4.2.5. Quit Vi

The command to quit is `:q` (in fact, this command actually closes the active buffer, as we have already seen, but if it is the only buffer open, you will quit Vi). There is a shortcut: most of the time you edit only one file. So to quit, you will use:

- `:wq` to save changes and quit (a quicker solution is `ZZ`), or
- `:q!` to quit without saving.

You should note that if you have several buffers, `:wq` will only write the active buffer and then close it.

4.3. A last word...

Of course, we have said much more here than was necessary (after all, the first aim was to edit a text file), but hopefully we have been able to show you some of the possibilities of each of these editors. There is a great deal more to be said about them, as witnessed by the number of books dedicated to each of these editors.

Take the time to absorb all this information, opt for one of them, or learn only as much as you think necessary. But at least you know that when you want to go further, you can.

Chapter 5. Command-Line Utilities

The purpose of this chapter is to introduce a number of command-line tools that may prove useful for everyday use. Of course, you may skip this chapter if you only intend to use a graphical environment, but a quick glance may change your opinion.

Each command will be illustrated by an example, but this chapter is meant as an exercise in order for you to fully grasp their function and use.

5.1. File Operations and Filtering

Most command-line work is done on files. In this section we discuss how to watch and filter file content, take required information from files using a single command, and to sort a file's content.

5.1.1. cat, tail, head, tee: File Printing Commands

These commands have almost the same syntax: `command_name [option(s)] [file(s)]`, and may be used in a pipe. All of them are used to print part of a file according to certain criteria.

The `cat` utility concatenates files printing the results to standard output. This is one of the most widely used commands. You can use:

```
# cat /var/log/mail/info
```

to print, for example, the content of a mailer daemon log file to standard output[1]. The `cat` command has a very useful option (`-n`) which allows you to print the line numbers.

Some files, like daemon log files (if they are running) are usually huge in size[2] and printing them completely on the screen is not very useful. Often you need to see only some lines of the file. You can use the `tail` command to do so. The following command will print, by default, the last 10 lines of the file `/var/log/mail/info`:

```
# tail /var/log/mail/info
```

You can use the `-n` option to display the last N^{th} lines of a file. For example, to display the last 2 lines, you would issue:

```
# tail -n2 /var/log/mail/info
```

The `head` command is similar to `tail`, but it prints the first lines of a file. The following command will print, by default, the first 10 lines of the `/var/log/mail/info` file:

```
# head /var/log/mail/info
```

As with `tail` you can use the `-n` option to specify the number of lines to be printed. For example, to print the first 2, you issue:

```
# head -n2 /var/log/mail/info
```

You can also use these commands together. For example, if you wish to display only lines 9 and 10, you can use a command where the `head` command will select the first 10 lines from a file and pass them through a pipe to the `tail` command.

```
# head /var/log/mail/info | tail -n2
```

The last part will then select the last 2 lines and will print them to the screen. In the same way you can select line number 20, counted from the end of a file:

```
# tail -n20 /var/log/mail/info |head -n1
```

1. Some examples in this section are based on real work and server log files (services, daemons). Make sure syslogd (allows daemon's logging), and the corresponding daemon (in our case Postfix) are running, and that you work as `root`. Of course, you can always apply our examples to other files.
2. For example, the `/var/log/mail/info` file contains information about all sent mails, messages about fetching mail by users with the POP protocol, etc.

In this example we tell `tail` to select the file's last 20 lines and pass them through a pipe to `head`. Then the `head` command prints to the screen the first line from the obtained data.

Let's suppose we want to print the result of the last example to the screen and save it to the file `results.txt` at the same time. The `tee` utility can help us. Its syntax is:

```
tee [option(s)] [file]
```

Now we can change the previous command this way:

```
# tail -n20 /var/log/mail/info |head -n1|tee results.txt
```

Let's take yet another example. We want to select the last 20 lines, save them to `results.txt`, but print on screen only the first of the 20 selected lines. Then we should type:

```
# tail -n20 /var/log/mail/info |tee results.txt |head -n1
```

The `tee` command has a useful option (`-a`) which allows you to append data to an existing file.

Let's go back to the `tail` command. Files such as logs usually vary dynamically because the daemon associated to that log constantly adds actions and events to the log file. So, if you want to interactively watch the changes to the log file you can take advantage of the `-f` option:

```
# tail -f /var/log/mail/info
```

In this case all changes in the `/var/log/mail/info` file will be printed on screen immediately. Using the `tail` command with option `-f` is very helpful when you want to know how your system works. For example, looking through the `/var/log/messages` log file, you can keep up with system messages and various daemons.

In the next section we will see how we can use grep as a filter to separate Postfix messages from messages coming from other services.

5.1.2. grep: Locate Strings in Files

Neither the name nor the acronym ("General Regular Expression Parser") is very intuitive, but what it does and its use are simple: grep looks for a pattern given as an argument in one or more files. Its syntax is:

```
grep [options] <pattern> [one or more file(s)]
```

If several files are mentioned, their names will precede each matching line displayed in the result. Use the `-h` option to prevent the display of these names; use the `-l` option to get nothing but the matching filenames. The pattern is a regular expression, even though most of the time it consists of a simple word. The most frequently used options are the following:

- `-i`: make a case insensitive search (i.e. ignore differences between lower and uppercase);
- `-v`: invert search. display lines which do **not** match the pattern;
- `-n`: display the line number for each line found;
- `-w`: tells grep that the pattern should match a whole word.

So let's go back to analyze the mailer daemon's log file. We want to find all lines in the file `/var/log/mail/info` which contain the "postfix" pattern. Then we type this command:

```
# grep postfix /var/log/mail/info
```

The grep command can be used in a pipe. Thus we can get the same result as in the previous example by doing this:

```
# cat /var/log/mail/info | grep postfix
```

If we want to find all lines not containing the "postfix" pattern, we would use the `-v` option:

```
# grep -v postfix /var/log/mail/info
```

Let's suppose we want to find all messages about successfully sent mails. In this case we have to filter all lines which were added into the log file by the mailer daemon (contains the "postfix" pattern) and they must contain a message about successful sending ("status=sent"):

```
# grep postfix /var/log/mail/info |grep status=sent
```

In this case grep is used twice. It is allowed, but not very elegant. We can get the same result by using the `fgrep` utility. First, we need to create a file containing patterns written out in a column. Such a file can be created this way (we use `patterns.txt` as the file name):

```
# echo -e 'status=sent\npostfix' >./patterns.txt
```

Then we call the next command where we use the `patterns.txt` file with a list of patterns and the `fgrep` utility instead of the "double calling" of grep:

```
# fgrep -f ./patterns.txt /var/log/mail/info
```

The file `./patterns.txt` may contain as many patterns as you wish. Each of them has to be typed as a single line. For example, to select messages about successfully sent mails to `peter@mandrakesoft.com`, it would be enough to add this email into our `./patterns.txt` file by running this command:

```
# echo 'peter@mandrakesoft.com' >>./patterns.txt
```

It is clear that you can combine grep with `tail` and `head`. If we want to find messages about the last but one email sent to `peter@mandrakesoft.com` we type:

```
# fgrep -f ./patterns.txt /var/log/mail/info | tail -n2 | head -n1
```

Here we apply the filter described above and place the result in a pipe for the `tail` and `head` commands. They select the last but one value from the data.

5.1.3. wc: Count Elements in Files

The `wc` command (*Word Count*) is used to count the number of strings and words in files. It is also helpful to count bytes, characters and the length of the longest line. Its syntax is:

```
wc [option(s)] [file(s)]
```

The following options are useful:

- `-l`: print the number of new lines;
- `-w`: print the number of words;
- `-m`: print the total number of characters;
- `-c`: print the number of bytes;
- `-L`: print the length of the longest line in the obtained text.

The `wc` command prints the number of newlines, words and characters by default. Here some usage examples:

If we want to find the number of users in our system, we can type:

```
$wc -l /etc/passwd
```

If we want to know the number of CPU's in our system, we write:

```
$grep "model name" /proc/cpuinfo |wc -l
```

In the previous section we obtained a list of messages about successfully sent mails to e-mail addresses listed in our `./patterns.txt` file. If we want to know the number of such messages, we can redirect our filter's results in a pipe for the `wc` command:

```
# fgrep -f ./patterns.txt /var/log/mail/info | wc -l
```

5.1.4. sort: Sorting File Content

Here is the syntax of this powerful sorting utility[3]:

```
sort [option(s)] [file(s)]
```

Let's consider sorting on part of the `/etc/passwd` file. As you can see this file is not sorted:

```
$ cat /etc/passwd
```

If we want to sort it by `login` field. Then we type:

```
$ sort /etc/passwd
```

The `sort` command sorts data in ascending order starting by the first field (in our case, the `login` field) by default. If we want to sort data in descending order, we use the option `-r`:

```
$ sort -r /etc/passwd
```

Every user has his own `UID` written in the `/etc/passwd` file. Let's sort a file in ascending order using the `UID` field:

```
$ sort /etc/passwd -t":" -k3 -n
```

Here we use the following `sort` options:

- `-t":"`: tells `sort` that the field separator is the `":"` symbol;
- `-k3`: means that sorting must be done on the third column;
- `-n`: says that the sort is to occur on numerical data, not alphabetical.

The same can be done in reverse:

```
$ sort /etc/passwd -t":" -k3 -n -r
```

Note that `sort` has two other important options:

- `-u`: perform a strict ordering: duplicate sort fields are discarded;
- `-f`: ignore case (treat lowercase characters the same way as uppercase ones).

Finally, if we want to find the user with the highest `UID` we can use this command:

```
$ sort /etc/passwd -t":" -k3 -n |tail -n1
```

where we sort the `/etc/passwd` file in ascending order according to the `UID` column, and redirect the result through a pipe to the `tail` command which will print out the first value of the sorted list.

5.2. find: Find Files According to Certain Criteria

`find` is a long-standing UNIX® utility. Its role is to recursively scan one or more directories and find files which match a certain set of criteria in those directories. Even though it is very useful, the syntax is truly obscure, and using it requires a little practice. The general syntax is:

```
find [options] [directories] [criterion] [action]
```

If you do not specify any directory, `find` will search the current directory. If you do not specify criteria, this is equivalent to "true", thus all files will be found. The options, criteria and actions are so numerous that we will only mention a few of each here. Let's start with options:

- `-xdev`: do not search on directories located on other file systems.

3. We only discuss `sort` briefly here because whole books can be written about its features.

- `-mindepth <n>`: descend at least `n` levels below the specified directory before searching for files.
- `-maxdepth <n>`: search for files which are located at most `n` levels below the specified directory.
- `-follow`: follow symbolic links if they link to directories. By default, `find` does not follow links.
- `-daystart`: when using tests related to time (see below), take the beginning of current day as a timestamp instead of the default (24 hours before current time).

A criteria may be one or more of several ***atomic*** tests. Some useful tests are:

- `-type <file_type>`: search for a given type of file. `file_type` can be one of: `f` (regular file), `d` (directory), `l` (symbolic link), `s` (socket), `b` (block mode file), `c` (character mode file) or `p` (named pipe).
- `-name <pattern>`: find files whose names match the given pattern. With this option, the pattern is treated as a ***shell globbing*** pattern (see *Shell Globbing Patterns*, page 22).
- `-iname <pattern>`: like `-name`, but ignore case.
- `-atime <n>`, `-amin <n>`: find files that have last been accessed `n` days ago (`-atime`) or `n` minutes ago (`-amin`). You can also specify `<+n>` or `<-n>`, in which case the search will be done for files accessed at most or at least `n` days/minutes ago.
- `-anewer <a_file>`: find files which have been accessed more recently than file `a_file`.
- `-ctime <n>`, `-cmin <n>`, `-cnewer <file>`: same as for `-atime`, `-amin` and `-anewer`, but applies to the last time that the contents of the file were modified.
- `-regex <pattern>`: same as `-name`, but `pattern` is treated as a ***regular expression***
- `-iregex <pattern>`: same as `-regex`, but ignore case.

There are many other tests, refer to `find`(1) for more details. To combine tests, you can use one of:

- `<c1> -a <c2>`: true if both `c1` and `c2` are true; `-a` is implicit, therefore you can type `<c1> <c2> <c3>` if you want all tests `c1`, `c2` and `c3` to match.
- `<c1> -o <c2>`: true if either `c1` or `c2` are true, or both. Note that `-o` has a lower ***precedence*** than `-a`, therefore if you want to match files which match criteria `c1` or `c2` and match criterion `c3`, you will have to use parentheses and write `( <c1> -o <c2> ) -a <c3>`. You must ***escape*** (deactivate) parentheses, as otherwise they will be interpreted by the shell!
- `-not <c1>`: inverts test `c1`, therefore `-not <c1>` is true if `c1` is false.

Finally, you can specify an action for each file found. The most frequently used are:

- `-print`: just prints the name of each file on the standard output. This is the default action.
- `-ls`: prints on the standard output the equivalent of `ls -ilds` for each file found.
- `-exec <command_line>`: execute command `command_line` on each file found. The command line `command_line` must end with a `;`, which you must escape so that the shell does not interpret it; the file position is marked with `{}`. See the usage examples.
- `-ok <command>`: same as `-exec` but asks for confirmation for each command.

The best way to consolidate all of the options and parameters is with some examples. Let's say you want to find all directories in the `/usr/share` directory. You would type:

```
find /usr/share -type d
```

Suppose you have an HTTP server, all your HTML files are in `/var/www/html`, which is also your current directory. You want to find all files whose contents have not been modified for a month. Because you have pages from several writers, some files have the `html` extension and some have the `htm` extension. You want to link these files in directory `/var/www/obsolete`. You would type[4]:

```
find \( -name "*.htm" -o -name "*.html" \) -a -ctime -30 \
-exec ln {} /var/www/obsolete \;
```

This is a fairly complex example, and requires a little explanation. The criterion is this:

4. Note that this example requires that `/var/www` and `/var/www/obsolete` be on the same file system!

```
\( -name "*.htm" -o -name "*.html" \) -a -ctime -30
```

which does what we want: it finds all files whose names end either in `.htm` or `.html` " `\( -name "*.htm" -o -name "*.html" \)` ", **and** (`-a`) which have not been modified in the last 30 days, which is roughly a month (`-ctime -30`). Note the parentheses: they are necessary here, because `-a` has a higher precedence. If there weren't any, all files ending with `.htm` would have been found, plus all files ending with `.html` and which haven't been modified for a month, which is not what we want. Also note that parentheses are escaped from the shell: if we had put `( .. )` instead of `\( .. \)`, the shell would have interpreted them and tried to execute `-name "*.htm" -o -name "*.html"` in a sub-shell... Another solution would have been to put parentheses between double quotes or single quotes, but a backslash here is preferable as we only have to isolate one character.

And finally, there is the command to be executed for each file:

```
-exec ln {} /var/www/obsolete \;
```

Here too, you have to escape the `;` from the shell, because otherwise the shell interprets it as a command separator. If you happen to forget, `find` will complain that `-exec` is missing an argument.

A last example: you have a huge directory (`/shared/images`) containing all kinds of images. Regularly, you use the `touch` command to update the times of a file named `stamp` in this directory, so that you have a time reference. You want to find all ***JPEG*** images which are newer than the `stamp` file, but because you got the images from various sources, these files have extensions `jpg`, `jpeg`, `JPG` or `JPEG`. You also want to avoid searching in the `old` directory. You want this file list to be mailed to you, and your user name is `peter`:

```
find /shared/images -cnewer     \
     /shared/images/stamp       \
     -a -iregex ".*\.jpe?g"     \
     -a -not -regex ".*/old/.*" \
       | mail peter -s "New images"
```

Of course, this command is not very useful if you have to type it each time, and you would like it to be executed regularly. A simple way to have the command run periodically is to use the cron daemon as shown in the next section.

5.3. Commands Startup Scheduling

5.3.1. crontab: reporting or editing your crontab file

`crontab` is a command which allows you to execute commands at regular time intervals, with the added bonus that you don't have to be logged in. `crontab` will have the output of your command mailed to you. You can specify the intervals in minutes, hours, days, and even months. Depending on the options, `crontab` will act differently:

- `-l`: Print your current `crontab` file.
- `-e`: Edit your `crontab` file.
- `-r`: Remove your current `crontab` file.
- `-u <user>`: Apply one of the above options for user `<user>`. Only `root` can do this.

Let's start by editing a `crontab`. If you type `crontab -e`, you will be in front of your favorite text editor if you have set the `EDITOR` or `VISUAL` environment variable, otherwise Vi will be used. A line in a `crontab` file is made of six fields. The first five fields are time intervals for minutes, hours, days in the month, months and days in the week. The sixth field is the command to be executed. Lines beginning with a `#` are considered to be comments and will be ignored by `crond` (the program which is responsible for executing `crontab` files). Here is an example of `crontab`:

in order to print this out in a readable font, we had to break up long lines. Therefore, some chunks must be typed on a single line. When the \ character ends a line, it means that line is to be continued. This convention works in `Makefile` files and in the shell as well as in other contexts.

```
# If you don't want to be sent mail, just comment
#   out the following line
#MAILTO="your_email_address"
#
# Report every 2 days about new images at 2 pm,
#   from the example above - after that, "retouch"
#   the "stamp" file. The "%" is treated as a
#   newline, this allows you to put several
#   commands in a same line.
0 14 */2 * *  find /shared/images              \
  -cnewer /shared/images/stamp                 \
  -a -iregex ".*\.jpe?g"                       \
  -a -not -regex                               \
    ".*/old/.*"%touch /shared/images/stamp
#
# Every Christmas, play a melody :)
0 0 25 12 * mpg123 $HOME/sounds/merryxmas.mp3
#
# Every Tuesday at 5pm, print the shopping list...
0 17 * * 2 lpr $HOME/shopping-list.txt
```

There are several ways to specify intervals other than the ones shown in this example. For example, you can specify a set of ***discrete values*** separated by commas (`1,14,23`) or a range (`1-15`), or even combine both of them (`1-10,12-20`), optionally with a step (`1-12,20-27/2`). Now it's up to you to find useful commands to put in it!

5.3.2. at: schedule a command, but only once

You may also want to launch a command at a given day, but not regularly. For example, you want to be reminded of an appointment, today at 6pm. You run X, the `X11R6-contrib` package is installed, and you would like to be notified at 5:30pm, for example, that you must go. `at` is what you want here:

```
$ at 5:30pm
  # You're now in front of the "at" prompt
at> xmessage "Time to go now! Appointment at 6pm"
  # Type CTRL-d to exit
at> <EOT>
$
```

You can specify the time in different ways:

- `now + <interval>`: Means, well, now, plus an interval (Optional. No interval specified means just now). The syntax for the interval is `<n> (minutes|hours|days|weeks|months)`. For example, you can specify `now + 1 hour` (an hour from now), `now + 3 days` (three days from now) and so on.
- `<time> <day>`: Fully specify the date. The `<time>` parameter is mandatory. `at` is very liberal in what it accepts: you can for example type `0100`, `04:20`, `2am`, `0530pm`, `1800`, or one of three special values: `noon`, `teatime` (4pm) or `midnight`. The `<day>` parameter is optional. You can specify this in different ways as well: `12/20/2001` for example, which stands for December 20th, 2001, or, the European way, `20.12.2001`. You may omit the year, but then only the European notation is accepted: `20.12`. You can also specify the month in full letters: `Dec 20` or `20 Dec` are both valid.

`at` also accepts different options:

- `-l`: Prints the list of currently queued jobs; the first field is the job number. This is equivalent to the `atq` command.
- `-d <n>`: Remove job number `<n>` from the queue. You can obtain job numbers from `atq`. This is equivalent to `atrm <n>`.

As usual, see the at(1) `manpage` for more options.

5.4. Archiving and Data Compression

5.4.1. tar: Tape ARchiver

Just like `find`, `tar` is a long standing UNIX® utility, so its syntax is a bit special. The syntax is:

```
tar [options] [files...]
```

Here is a list of some options. Note that all of them have an equivalent long option, but you will have to refer to `tar`'s `man` page, as we will not list them here.

the initial dash (`-`) of short options is now deprecated with `tar`, except after a long option.

- `c`: this option is used in order to create new archives.
- `x`: this option is used in order to extract files from an existing archive.
- `t`: lists files from an existing archive.
- `v`: lists the files which are added to an archive or extracted from an archive, or, in conjunction with the `t` option (see above), it outputs a long listing of files instead of a short one.
- `f <file_name>`: create archive with name `file_name`, extract from archive `file_name` or list files from archive `file_name`. If this parameter is omitted, the default file will be `/dev/rmt0`, which is generally the special file associated with a ***streamer***. If the file parameter is `-` (a dash), the input or output (depending on whether you create an archive or extract from one) will be associated to the standard input or standard output.
- `z`: tells `tar` that the archive to create should be compressed with `gzip`, or that the archive to extract from is compressed with `gzip`.
- `j`: same as `z`, but the program used for compression is `bzip2`.
- `p`: when extracting files from an archive, preserve all file attributes, including ownership, last access time and so on. Very useful for file system dumps.
- `r`: append the list of files given on the command line to an existing archive. Note that the archive to which you want to append files must **not** be compressed!
- `A`: append archives given on the command line to the one submitted with the `f` option. Similar to `r`, the archives must not be compressed in order for this to work.

There are many, many, many other options, so you may want to refer to tar(1) for the entire list. See, for example, the `d` option. Let's proceed with an example. Say you want to create an archive of all images in `/shared/images`, compressed with `bzip2`, named `images.tar.bz2` and located in your home directory. You would then type:

```
 #
 # Note: you must be in the directory from which
 #   you want to archive files!
 #
$ cd /shared
$ tar cjf ~/images.tar.bz2 images/
```

As you can see, we used three options here: `c` told `tar` we wanted to create an archive, `j` to compress it with `bzip2`, and `f ~/images.tar.bz2` that the archive was to be created in our home directory, and its name will be `images.tar.bz2`. We may want to check if the archive is valid now. We can do this by listing its files:

```
 #
 # Get back to our home directory
 #
$ cd
$ tar tjvf images.tar.bz2
```

Here, we told `tar` to list (`t`) files from archive `images.tar.bz2` (`f images.tar.bz2`), warned that this archive was compressed with `bzip2` (`j`), and that we wanted a long listing (`v`). Now, say you have erased the images

directory. Fortunately, your archive is intact, and you now want to extract it back to its original place, in `/shared`. But as you don't want to break your `find` command for new images, you need to preserve all file attributes:

```
 #
 # cd to the directory where you want to extract
 #
$ cd /shared
$ tar jxpf ~/images.tar.bz2
```

And here you are!

Now, let's say you want to extract the directory `images/cars` from the archive, and nothing else. Then you can type this:

```
$ tar jxf ~/images.tar.bz2 images/cars
```

If you try to back up special files, `tar` will take them as what they are, special files, and will not dump their contents. So yes, you can safely put `/dev/mem` in an archive. It also deals correctly with links, so do not worry about this either. For symbolic links, also look at the `h` option in the `manpage`.

5.4.2. bzip2 and gzip: Data Compression Programs

You can see that we have already talked of these two programs when dealing with `tar`. Unlike WinZip® in Windows®, archiving and compressing are done using two separate utilities — `tar` for archiving, and the two programs which we will now introduce for compressing data: `bzip2` and `gzip`. You might also use a different compression tool, programs like `zip`, `arj` or `rar` also exist for GNU/Linux (but they are rarely used).

At first, `bzip2` was written as a replacement for `gzip`. Its compression ratios are generally better, but on the other hand, it requires more RAM while working. However `gzip` is still used for compatibility with older systems.

Both commands have a similar syntax:

```
gzip [options] [file(s)]
```

If no filename is given, both `gzip` and `bzip2` will wait for data from the standard input and send the result to the standard output. Therefore, you can use both programs in pipes. Both programs also have a set of common options:

- `-1`, ..., `-9`: set the compression ratio. The higher the number, the better the compression, but better also means slower.
- `-d`: uncompress file(s). This is equivalent to using `gunzip` or `bunzip2`.
- `-c`: dump the result of compression/decompression of files given as parameters to the standard output.

By default, both `gzip` and `bzip2` erase the file(s) that they have compressed (or uncompressed) if you don't use the `-c` option. You can avoid doing this in `bzip2` by using the `-k` option. `gzip` has no equivalent option.

Now some examples. Let's say you want to compress all files ending with `.txt` in the current directory using `bzip2` with maximum compression. You would type:

```
$ bzip2 -9 *.txt
```

Let's say you want to share your image archives with someone, but he does not have `bzip2`, only `gzip`. You don't need to uncompress the archive and re-compress it, you can just uncompress to the standard output, use a pipe, compress from standard input and redirect the output to the new archive. Like this:

```
bzip2 -dc images.tar.bz2 | gzip -9 >images.tar.gz
```

You could have typed `bzcat` instead of `bzip2 -dc`. There is an equivalent for `gzip` but its name is `zcat`, not **gzcat**. You also have `bzless` for `bzip2` file and `zless` for `gzip` if you want to view compressed files directly

instead of having to uncompress them first. As an exercise, try and find the command you would have to type in order to view compressed files without uncompressing them, and without using `bzless` or `zless`.

5.5. Many, many more...

There are so many commands that a comprehensive book about them would be the size of an encyclopedia. This chapter hasn't even covered a tenth of the subject, yet you can do much with what you learned here. If you wish, you may read some `manual` pages: sort(1), sed(1) and zip(1) (yes, that's what you think: you can extract or make `.zip` archives with GNU/Linux), convert(1), and so on. The best way to get accustomed to these tools is to practice and experiment with them, and you will probably find a lot of uses for them, even quite unexpected ones. Have fun!

Chapter 6. Process Control

We already seen in *Processes*, page 10 what a process is. We will now learn how to list processes and their characteristics, and how to "manipulate" them.

6.1. More About Processes

It is possible to monitor processes and to "ask" them to terminate, pause, continue, etc. To understand the examples we're going to examine, it is helpful to know a bit more about processes.

6.1.1. The Process Tree

As with files, all processes that run on a GNU/Linux system are organized in the form of a tree. The root of this tree is `init`, a system-level process which is started at boot time. The system assigns a number (PID, *Process ID*) to each process in order to be able to uniquely identify processes. Processes also inherit the PID of their parent process (PPID, *Parent Process ID*). `init` is its own father: the PID and PPID of `init` is 1.

6.1.2. Signals

Every process in UNIX® can react to signals sent to it. There are 64 different signals which are identified either by their number (starting from `1`) or by their symbolic names (`SIGx`, where `x` is the signal's name). The 32 "higher" signals (`33` to `64`) are real-time signals and are beyond the scope of this chapter. For each of these signals, the process can define its own behavior, except for two signals: signal number 9 (`KILL`) and number 19 (`STOP`).

Signal `9` terminates a process irrevocably without giving it the time to terminate properly. This is the signal you send to a process which is stuck or exhibits other problems. A full list of signals is available using the `kill -l` command.

6.2. Information on Processes: ps and pstree

These two commands display a list of processes currently running on the system, according to criteria you set.

6.2.1. ps

Running `ps` without arguments will show only processes initiated by you and attached to the terminal you are using:

```
$ ps
      PID TTY          TIME CMD
    18614 pts/3    00:00:00 bash
    20173 pts/3    00:00:00 ps
```

As with many UNIX® utilities, `ps` has a handful of options, the most common of which are:

- `a`: displays processes started by other users;
- `x`: displays processes with no control terminal or with a control terminal different to the one you are using;
- `u`: displays for each process the name of the user who started it and the time at which it was started.

There are many other options. Refer to the ps(1) manual page for more information.

The output of this command is divided into different fields: the one that will interest you the most is the `PID` field which contains the process identifier. The `CMD` field contains the name of the executed command. A very common way of invoking `ps` is as follows:

```
$ ps ax | less
```

This gives you a list of all processes currently running so that you can identify one or more processes which are causing problems, and subsequently terminate them.

6.2.2. pstree

The `pstree` command displays processes in the form of a tree structure. One advantage is that you can immediately see the processes' parents: when you want to kill a whole series of processes and if they are all parents and children, you can simply kill the parent. You will want to use the `-p` option to display the PID of each process, and the `-u` option to show the name of the user who started the process. Because the tree structure is generally quite long, you will want to invoke `pstree` in the following way:

```
$ pstree -up | less
```

This gives you an overview of the whole process tree structure.

6.3. Sending Signals to Processes: kill, killall and top

6.3.1. kill, killall

These two commands are used to send signals to processes. The `kill` command requires a process number as an argument, while `killall` requires a process name.

Both of these commands can optionally receive the signal number of the signal to be sent as an argument. By default, they both send the signal 15 (`TERM`) to the relevant process(es). For example, if you want to kill the process with PID 785, you enter the command:

```
$ kill 785
```

If you want to send it signal 19 (`STOP`), you enter:

```
$ kill -19 785
```

Suppose instead that you want to kill a process where you know the command name. Instead of finding the process number using `ps`, you can kill the process by its name:

```
$ killall -9 mozilla
```

Whatever happens, you will only kill your own processes (unless you are `root`) so you do not need to worry about your "neighbor's" processes if you're running on a multi-user system, since they will not be affected.

6.3.2. Mixing ps and kill: top

`top` is a program that simultaneously fulfills the functions of `ps` and `kill` and is also used to monitor processes in real-time giving information about CPU and memory usage, running time, etc. as shown in figure 6-1.

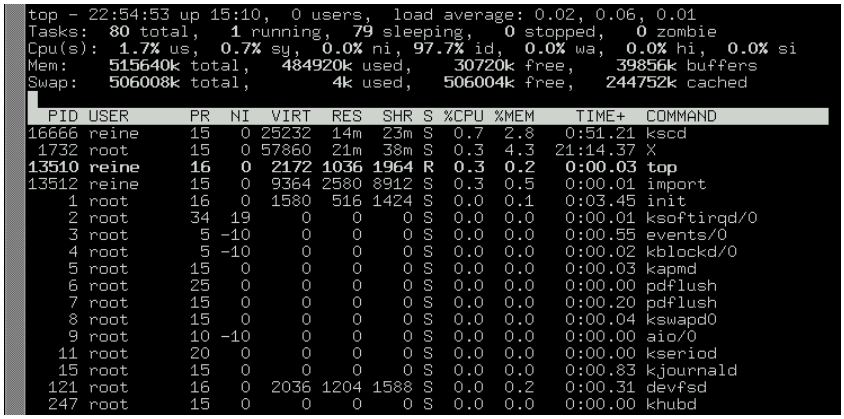

Figure 6-1. Monitoring Processes with top

The `top` utility is entirely keyboard controlled. You can access help by pressing **h**. Its most useful commands are the following:

- **k**: this command is used to send a signal to a process. `top` will then ask you for the process' PID followed by the number or the name of the signal to be sent (`TERM` — or `15` — by default);
- **M**: this command is used to sort processes by the amount of memory they take up (field `%MEM`);
- **P**: this command is used to sort processes by the CPU time they take up (field `%CPU`; this is the default sorting method);
- **u**: this one is used to display a given user's processes, `top` will ask you which one. You need to enter the user's **name**, not his UID. If you do not enter any name, all processes will be displayed;
- **i**: by default, all processes, even sleeping ones, are displayed. This command ensures that only processes currently running are displayed (processes whose `STAT` field shows `R`, *Running*) and not the others. Using this command again takes you back to showing all processes.
- **r**: this command is used to change the priority of selected process.

6.4. Setting Priority to Processes: nice, renice

Every process in the system is running with defined priorities, also called "nice value", which may vary from -20 (highest priority) to 19 (lowest priority). If not defined, every process will run with a default priority of 0 (the "base" scheduling priority). Processes with greater priority (lower nice value, down to -20) will be scheduled to run more often than others which have less priority (up to 19), thus granting them more processor cycles. Users other than the super-user may only lower the priority of processes they own within a range of 0 to 19. The super-user (`root`) may set the priority of any process to any value.

6.4.1. renice

If one or more processes use too many system resources, you can change their priorities instead of killing them. To do so, the `renice` command is used. Its syntax is as follows:

```
renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]
```

Where `priority` is the value of the priority, `pid` (use option `-p` for multiple processes) is the process ID, `pgrp` (introduced by `-g` if various) is the process group ID, and `user` (`-u` for more than one) is the user name of the process' owner.

Let's suppose you have a process running with PID 785, which makes a long scientific operation, and while it is working you want to play a game for which you need to free system resources. Then you would type:

```
$ renice +15 785
```

In this case your process could potentially take longer to complete but will not take CPU time from other processes.

If you are the system administrator and you see that some user is running too many processes and they use too many system resources, you can change that user's processes priority with a single command:

```
# renice +20 -u peter
```

After this, all of peter's processes will have the lowest priority and will not obstruct any other user's processes.

6.4.2. nice

Now that you know that you can change the priority of processes, you may wish to run a command with a defined priority. For this, use the `nice` command.

In this case you need to specify your command as an option to `nice`. Option `-n` is used to set priority value. By default `nice` sets a priority of 10.

For example, you want to create an ISO image of a Mandrakelinux installation CD-ROM:

```
$ dd if=/dev/cdrom of=~/mdk1.iso
```

On some systems with a standard IDE CD-ROM, the process of copying large volumes of information can use too many system resources. To prevent the copying from blocking other processes, you can start the process with a lowered priority by using this command:

```
$ nice -n 19 dd if=/dev/cdrom of=~/mdk1.iso
```

and continue with what you were doing.

Chapter 9. The Linux File System

Naturally, your GNU/Linux system is contained on your hard disk within a file system. Here, we will discuss the various aspects of file systems available on GNU/Linux, as well as the possibilities they offer.

9.1. Comparing a Few File Systems

During installation, you can choose different **file systems** for your partitions, so they will be formatted using different algorithms.

Unless you are a specialist, choosing a file system is not obvious. We will take a quick look at a few current file systems, all of which are available with Mandrakelinux.

9.1.1. Different Usable File Systems

9.1.1.1. Ext2

The **Second Extended File System** (its abbreviated form is ext2 or simply `ext2`) has been GNU/Linux's default file system for many years. It replaced the Extended File System (that's where the "Second" comes from). ext2 correcting certain problems and limitations of its predecessor.

ext2 respects the usual standards for UNIX®-type file systems. Since its inception, it was destined to evolve while still offering great robustness and good performance.

9.1.1.2. Ext3

As its name suggests, the Third Extended File System is ext2's successor. It is compatible with the latter but enhanced by incorporating **journaling**.

One of the major flaws of "traditional" file systems like ext2 is their low tolerance to abrupt system breakdowns (power failure or crashing software). Generally speaking, once the system is restarted, these sorts of events involve a very long examination of the file system's structure and attempts to correct errors, which sometimes result in an extended corruption. This corruption could cause partial or total loss of saved data.

Journaling answers this problem. To simplify, let's say that what we are doing is storing the actions (such as the saving of a file) **before** really doing it. We could compare this functionality to that of a boat captain who uses a log book to note daily events. The result: an always coherent file system. And if problems occur, the verification is very rapid and the eventual repairs, very limited. The time spent in verifying a file system is thus proportional to its actual use and not related to its size.

So, ext3 offers journal file system technology while keeping ext2's structure, ensuring excellent compatibility. This makes it very easy to switch from ext2 to ext3 and back again.

9.1.1.3. ReiserFS

Unlike ext3, `reiserfs` was written from scratch. It is a journalized file system like ext3, but its internal structure is radically different because it uses binary-tree concepts inspired by database software.

9.1.1.4. JFS

JFS is the journalized file system designed and used by IBM. Proprietary and closed at first, IBM decided to open it to access by the free software movement. Its internal structure is similar to that of `reiserfs`.

9.1.1.5. XFS

XFS is the journalized file system designed by SGI and also used with the Irix operating system. Proprietary and closed at first, SGI decided to open it to access by the free software movement. Its internal structure has lots of different features, such as support for real-time bandwidth, extents, and clustered file systems (but not in the free version).

9.1.2. Differences Between the File Systems

	Ext2	Ext3	ReiserFS	JFS	XFS
Stability	Excellent	Good	Good	Medium	Good
Tools to restore erased files	Yes (complex)	Yes (complex)	No	No	No
Reboot time after crash	Long, even very long	Fast	Very fast	Very fast	Very fast
Status of the data in case of a crash	Generally speaking, good, but high risk of partial or total data loss	Very good	Medium[a]	Very good	Very good
ACL support	Yes	Yes	No	No	Yes
Notes: a. It is possible to improve results on crash recovery by journaling the **data** and not just the **metadata**, adding the option *data=journal* to /etc/fstab.					

Table 9-1. File System Characteristics

The maximum size of a file depends on many parameters (e.g. the block size for ext2/ext3), and is likely to evolve depending on the kernel version and architecture. According to the file system limits, the current maximum size is currently near or greater than 2 TeraBytes (TB, 1 TB=1024 GB) and for JFS can go up to 4 PetaBytes (PB, 1 Pb=1024 TB). Unfortunately, these values are also limited to maximum block device size, which in the current 2.4.X kernel is limited (for X86 arch only) to 2TB[1] even in RAID mode. In kernel 2.6.X this block device limit could be extended using a kernel compiled with Large Block Device support enabled (CONFIG_LBD=y). For more information, consult Adding Support for Arbitrary File Sizes to the Single UNIX Specification (http://ftp.sas.com/standards/large.file/x_open.20Mar96.html), Large File Support in Linux (http://www.suse.com/~aj/linux_lfs.html), and Large Block Devices (http://www.gelato.unsw.edu.au/IA64wiki/LargeBlockDevices).

9.1.3. And Performance Wise?

It is always very difficult to compare performance between file systems. All tests have their limitations and the results must be interpreted with caution. Nowadays, ext2 is very mature but its development is slow; ext3 and reiserfs are quite mature at this point. New features for reiserfs are included in reiserfs4[2]. On the other hand XFS has a lot of features, and as time passes more of the advanced features work better on Linux. JFS took a different approach here, and they are integrating on Linux feature by feature. This makes the process slower, but they are also finishing with a very clean code base. Comparisons done a couple of months or weeks ago are already too old. Let us not forget that today's material (specially concerning hard drive capacities) has greatly leveraged the differences between them. XFS has the advantage that just now it is the better performer with large streaming files.

1. You may wonder how to achieve such capacities with hard drives that barely store 320-400GB. Using for instance one RAID card with 8 * 250GB drives in RAID-striping, you can achieve 2TB of storage. Combining the storage of several RAID cards using Linux software RAID, or using LVM (Logical Volume Manager) it should be possible to go even beyond (block size permitting) the 2TB limit.
2. At the time of writing, reiserfs4 was not included in kernel 2.6.X

Each system offers advantages and disadvantages. In fact, it all depends on how you use your machine. A simple desktop machine will be happy with ext2. For a server, a journalized file system like ext3 is preferred. `reiserfs`, perhaps because of its genesis, is more suited to a database server. JFS is preferred in cases were file system throughput is the main issue. XFS is interesting if you need any of its advanced features.

For "normal" use, the four file systems give approximately the same results. `reiserfs` allows you to access small files rapidly, but it is fairly slow in manipulating large files (many megabytes). In most cases, the advantages brought by `reiserfs`' journaling capabilities outweigh its drawbacks. Notice that by default `reiserfs` is mounted with the `notail` option. That means that there is no optimization for small files and that big files run at normal speed.

9.2. Everything is a File

The *Starter Guide* introduced the file ownership and permissions access concepts, but really understanding the UNIX® ***file system*** (and this also applies to Linux's file systems) requires that we redefine the concept of "What is a file.".

Here, "everything" **really** means everything. A hard disk, a partition on a hard disk, a parallel port, a connection to a web site, an Ethernet card: all these are files. Even directories are files. Linux recognizes many types of files in addition to the standard files and directories. Note that by file type here, we do not mean the type of **content** of a file: for GNU/Linux and any UNIX® system, a file, whether it be a PNG image, a binary file or whatever, is just a stream of bytes. Differentiating files according to their contents is left to applications.

9.2.1. The Different File Types

If you remember, when you do `ls -l`, the character before the access rights identifies the type of a file. We have already seen two types of files: regular files (`-`) and directories (`d`). You can also find other types if you wander through the file tree and list the contents of directories:

1. ***Character mode files***: these files are either special system files (such as `/dev/null`, which we have already discussed), or peripherals (serial or parallel ports), which share the trait that their contents (if they have any) are not ***buffered*** (meaning they are not kept in memory). Such files are identified by the letter `c`.
2. ***Block mode files***: these files are peripherals, and unlike character files, their contents **are** buffered. For example, some files in this category are: hard disks, partitions on a hard disk, floppy drives, CD-ROM drives and so on. Files `/dev/hda`, `/dev/sda5` are example of block mode files. In `ls -l` output, these are identified by the letter `b`.
3. ***Symbolic links***: these files are very common, and heavily used in the Mandrakelinux system startup procedure (see chapter "*The Start-Up Files: init sysv*", page 71). As their name implies, their purpose is to link files in a symbolic way, which means that they are files whose content is the path to a different file. They may not point to an existing file. They are very frequently called ***"soft links"***, and are identified by an "l".
4. ***Named pipes***: in case you were wondering, yes, these are very similar to pipes used in shell commands, but with the difference that these actually have names. Read on to learn more. They are very rare, however, and it's not likely that you will see one during your journey into the file tree. Just in case you do, the letter identifying them is `p`. To learn more, have a look at *"Anonymous" Pipes and Named Pipes*, page 61.
5. ***Sockets***: this is the file type for all network connections, but only a few of them have names. What's more, there are different types of sockets and only one can be linked, but this is way beyond the scope of this book. Such files are identified by the letter `s`.

Here is a sample of each file:

```
$ ls -l /dev/null /dev/sda  /etc/rc.d/rc3.d/S20random /proc/554/maps \
/tmp/ssh-queen/ssh-510-agent
crw-rw-rw-    1 root     root       1,   3 May  5  1998 /dev/null
brw-rw----    1 root     disk       8,   0 May  5  1998 /dev/sda
lrwxrwxrwx    1 root     root           16 Dec  9 19:12 /etc/rc.d/rc3.d/
  S20random -> ../init.d/random*
pr--r--r--    1 queen  queen         0 Dec 10 20:23 /proc/554/maps|
srwx------    1 queen  queen         0 Dec 10 20:08 /tmp/ssh-queen/
  ssh-510-agent=
$
```

9.2.2. Inodes

Inodes are, along with the "Everything Is a File" paradigm, a fundamental part of any UNIX® file system. The word ***inode*** is short for "Information NODE".

Inodes are stored on disk in an **inode table**. They exist for all types of files which may be stored on a file system, including directories, named pipes, character mode files and so on. Which leads to this other famous sentence: "The inode is the file". Inodes are how UNIX® identifies a file in a unique way.

No, you didn't misread that: in UNIX®, you **do not identify a file by its name**, but by its inode number[3]. The reason for this is that the same file may have several names, or even no name. In UNIX®, a file name is just an entry in a directory inode. Such an entry is called a **link**. Let us look at links in more detail.

9.3. Links

The best way to understand what links are is to look at an example. Let us create a (regular) file:

```
$ pwd
/home/queen/example
   $ ls
   $ touch a
   $ ls -il a
   32555 -rw-rw-r--    1 queen     queen            0 Dec 10 08:12 a
```

The `-i` option of the `ls` command prints the inode number, which is the first field on the output. As you can see, before we created file `a`, there were no files in the directory. The other field of interest is the third one, which is the number of file links (well, inode links, in fact).

The `touch a` command can be separated into two distinct actions:

- creation of an inode, to which the operating system has given the number 32555, and whose type is the one of a regular file;
- and creation of a link to this inode, named `a`, in the current directory, `/home/queen/example`. Therefore, the `/home/queen/example/a` file is a link to the inode numbered 32555, and it's currently the only one: the link counter shows `1`.

But now, if we type:

```
$ ln a b
   $ ls -il a b
   32555 -rw-rw-r--    2 queen     queen            0 Dec 10 08:12 a
   32555 -rw-rw-r--    2 queen     queen            0 Dec 10 08:12 b
   $
```

We create another link to the same inode. As you can see, we didn't create a file named `b`. Instead, we just added another link to the inode numbered 32555 in the same directory, and attributed the name `b` to this new link. You can see on the `ls -l` output that the link counter for the inode is now 2 rather than 1.

Now, if we do:

```
$ rm a
   $ ls -il b
   32555 -rw-rw-r--    1 queen     queen            0 Dec 10 08:12 b
   $
```

We see that even though we deleted the "original file", the inode still exists. But now, the only link to it is the file named `/home/queen/example/b`.

Therefore, a file in UNIX® has no name; instead, it has one or more ***link***(s) in one or more directories.

3. Important: note that inode numbers are unique **per file system**, which means that an inode with the same number can exist on another file system. This leads to the difference between on-disk inodes and in-memory inodes. While two on-disk inodes may have the same number if they are on two different file systems, in-memory inodes have a unique number right across the system. One solution to obtain uniqueness, for example, is to hash the on-disk inode number against the block device identifier.

Directories themselves are also stored in inodes, their link count coincides with the number of subdirectories within them. This is due to the fact that there are at least two links per directory: the directory itself (`.`) and its parent directory (`..`).

Typical examples of files which are not linked (i.e.: have no name) are network connections; you will never see the file corresponding to your connection to the Mandrakelinux web site (`www.mandrakelinux.com`) in your file tree, no matter which directory you look in. Similarly, when you use a ***pipe*** in the shell, the inode corresponding to the pipe exists, but it is not linked. Other use of inodes without names is temporary files. You create a temporary file, open it, and then remove it. The file exists while it's open, but nobody else can open it (as there is no name for opening it). This way, if the application crashes, the temporary file is removed automatically.

9.4. "Anonymous" Pipes and Named Pipes

Let's get back to the example of pipes, as it is quite interesting and is also a good illustration of the links notion. When you use a pipe in a command line, the shell creates the pipe for you and operates so that the command before the pipe writes to it, while the command after the pipe reads from it. All pipes, whether they be anonymous (like the ones used by the shells) or named (see below) act like FIFOs ("*First In, First Out*"). We've already seen examples of how to use pipes in the shell, but let's take another look for the sake of our demonstration:

```
$ ls -d /proc/[0-9] | head -5
/proc/1/
/proc/2/
/proc/3/
/proc/4/
/proc/5/
```

One thing that you will not notice in this example (because it happens too fast for one to see) is that writes on pipes are blocking. This means that when the `ls` command writes to the pipe, it is blocked until a process at the other end reads from the pipe. In order to visualize the effect, you can create named pipes, which unlike the pipes used by shells, have names (i.e.: they are linked, whereas shell pipes are not)[4]. The command to create a named pipe is `mkfifo`:

```
$ mkfifo a_pipe
$ ls -il
total 0
    169 prw-rw-r--    1 queen     queen        0 Dec 10 14:12 a_pipe|
  #
  # You can see that the link counter is 1, and that the output shows
  # that the file is a pipe ('p').
  #
  # You can also use ln here:
  #
$ ln a_pipe the_same_pipe
$ ls -il
total 0
    169 prw-rw-r--    2 queen     queen        0 Dec 10 15:37 a_pipe|
    169 prw-rw-r--    2 queen     queen        0 Dec 10 15:37 the_same_pipe|
$ ls -d /proc/[0-9] >a_pipe
  #
  # The process is blocked, as there is no reader at the other end.
  # Type Control Z to suspend the process...
  #
zsh: 3452 suspended  ls -d /proc/[0-9] > a_pipe
  #
  # ...Then put in into the background:
  #
$ bg
[1]  + continued  ls -d /proc/[0-9] > a_pipe
  #
  # now read from the pipe...
  #
$ head -5 <the_same_pipe
  #
  # ...the writing process terminates
  #
[1]  + 3452 done       ls -d /proc/[0-9] > a_pipe
```

4. Other differences exist between the two kinds of pipes, but they are beyond the scope of this book.

```
/proc/1/
/proc/2/
/proc/3/
/proc/4/
/proc/5/
#
```

Similarly, reads are also blocking. If we execute the above commands in the reverse order, we will see that `head` blocks, waiting for some process to give it something to read:

```
$ head -5 <a_pipe # #
  Program blocks, suspend it: C-z # zsh: 741 suspended head -5 <
  a_pipe # # Put it into the background...  # $ bg [1] + continued
  head -5 < a_pipe # # ...And give it some food :) # $ ls -d
  /proc/[0-9] >the_same_pipe $ /proc/1/ /proc/2/ /proc/3/ /proc/4/
  /proc/5/ [1] + 741 done head -5 < a_pipe $
```

You can also see an undesired effect in the previous example: the `ls` command has terminated before the `head` command took over. The consequence is that you were immediately returned to the prompt, but `head` executed later and you only saw its output after returning.

9.5. Special Files: Character Mode and Block Mode Files

As already stated, such files are either created by the system or peripherals on your machine. We also mentioned that the contents of block mode character files were buffered, while character mode files were not. In order to illustrate this, insert a floppy into the drive and type the following command twice:

```
$ dd if=/dev/fd0 of=/dev/null
```

You should have observed the following: the first time the command was launched, the entire content of the floppy was read. The second time you executed the command, there was no access to the floppy drive at all. This is because the content of the floppy was buffered the first time you launched the command — and you did not change anything on the floppy between the two instances.

But now, if you want to print a big file this way (yes it will work):

```
$ cat /a/big/printable/file/somewhere >/dev/lp0
```

The command will take as much time, whether you launch it once, twice or fifty times. This is because `/dev/lp0` is a character mode file, and its contents are not buffered.

The fact that block mode files are buffered has a nice side effect: not only are reads buffered, but writes are buffered too. This allows for writes to the disks to be asynchronous: when you write a file on disk, the write operation itself is not immediate. It will only occur when the Linux kernel decides to execute the write to the hardware.

Finally, each special file has a ***major*** and ***minor*** number. On a `ls -l` output, they appear in place of the size, as the size for such files is irrelevant:

```
ls -l /dev/ide/host0/bus0/target0/lun0/disc /dev/printers/0
brw-------  1 root root 3, 0 dic 31  1969 /dev/ide/host0/bus0/target0/lun0/disc
crw-rw----  1 lp   sys  6, 0 dic 31  1969 /dev/printers/0
```

Here, the major and minor of `/dev/ide/host0/bus0/target0/lun0/disc` are 3 and 0, whereas for `/dev/printers/0`, they are 6 and 0. Note that these numbers are unique per file category, which means that there can be a character mode file with major 3 and minor 0 (this file actually exists: `/dev/pty/s0`), and similarly, there can be a block mode file with major 6 and minor 0. These numbers exist for a simple reason: it allows the kernel to associate the correct operations to these files (that is, to the peripherals these files refer to): you don't handle a floppy drive the same way as, say, a SCSI hard drive.

9.6. Symbolic Links, Limitation of "Hard" Links

Here we have to face a very common misconception, even among UNIX® users, which is mainly due to the fact that links as we have seen them so far (wrongly called "hard" links) are only associated with regular files (and we have seen that it is not the case — since even symbolic links are "linked"). But this requires that we first explain what symbolic links ("soft" links, or even more often "symlinks") are.

Symbolic links are files of a particular type whose sole content is an arbitrary string, which may or may not point to an existing file. When you mention a symbolic link on the command line or in a program, in fact, you access the file it points to, if it exists. For example:

```
$ echo Hello >myfile
$ ln -s myfile mylink
$ ls -il
total 4
    169 -rw-rw-r--    1 queen     queen            6 Dec 10 21:30 myfile
    416 lrwxrwxrwx    1 queen     queen            6 Dec 10 21:30 mylink
-> myfile
$ cat myfile
Hello
$ cat mylink
Hello
```

You can see that the file type for `mylink` is `l`, for symbolic *Link*. The access rights for a symbolic link are not significant: they will always be `rwxrwxrwx`. You can also see that it **is** a different file from `myfile`, as its inode number is different. But it refers to it symbolically, therefore when you type `cat mylink`, you will in fact print the contents of the `myfile` file. To demonstrate that a symbolic link contains an arbitrary string, we can do the following:

```
$ ln -s "I'm no existing file" anotherlink
$ ls -il anotherlink
    418 lrwxrwxrwx    1 queen     queen           20 Dec 10 21:43 anotherlink
-> I'm no existing file
$ cat anotherlink
cat: anotherlink: No such file or directory
$
```

But symbolic links exist because they overcome several limitations encountered by normal ("hard") links:

- You cannot create a link to an inode in a directory which is on a different file system than the said inode. The reason is simple: the link counter is stored in the inode itself, and inodes cannot be shared between file systems. Symlinks allow this;
- You cannot link directories to avoid creating cycles in the file system. But you can make a symlink point to a directory and use it as if it were actually a directory.

Symbolic links are therefore very useful in several circumstances, and very often, people tend to use them to link files together even when a normal link could be used instead. One advantage of normal linking, though, is that you do not lose the file if you delete the "original one".

Lastly, if you observed carefully, you know what the size of a symbolic link is: it is simply the size of the string.

9.7. File Attributes

The same way that FAT has file attributes (archive, system file, invisible), a GNU/Linux file systems has its own, but they are different. We will briefly go over them here for the sake of completeness, but they are very seldom used. However, if you really want a secure system, read on.

There are two commands for manipulating file attributes: `lsattr(1)` and `chattr(1)`. You probably guessed it, `lsattr` "LiSts" attributes, whereas `chattr` "CHanges" them. These attributes can only be set on directories and regular files. The following attributes are possible:

1. `A` ("no **A**ccess time"): if a file or directory has this attribute set, whenever it is accessed, either for reading or for writing, its last access time won't be updated. This can be useful, for example, on files or directories

which are often accessed for reading, especially since this parameter is the only one which changes on an inode when it is open read-only.

2. `a` ("**a**ppend only"): if a file has this attribute set and is open for writing, the only operation possible will be to append data to its previous contents. For a directory, this means that you can only add files to it, but not rename or delete any existing file. Only `root` can set or clear this attribute.
3. `d` ("no **d**ump"): `dump (8)` is the standard UNIX® utility for backups. It dumps any file system for which the dump counter is 1 in `/etc/fstab` (see chapter "*File Systems and Mount Points*", page 53). But if a file or directory has this attribute set, unlike others, it will not be taken into account when a dump is in progress. Note that for directories, this also includes all subdirectories and files under it.
4. `i` ("**i**mmutable"): a file or directory with this attribute set can not be modified at all: it cannot be renamed, no further link can be created to it[5] and it cannot be removed. Only `root` can set or clear this attribute. Note that this also prevents changes to access time, therefore you don't need to set the `A` attribute when `i` is set.
5. `s` ("**s**ecure deletion"): when a file or directory with this attribute is deleted, the blocks it was occupying on disk are overwritten with zeroes.
6. `S` ("**S**ynchronous mode"): when a file or directory has this attribute set, all modifications on it are synchronous and written to the disk immediately.

For example, you may want to set the `i` attribute on essential system files in order to avoid bad surprises. Also, consider the `A` attribute on man pages: this prevents a lot of disk operations and, in particular, can save some battery life on laptops.

5. Be sure to understand what "adding a link" means, both for a file and a directory!