General overview of the Linux file system

Files

General

A simple description of the UNIX system, also applicable to Linux, is this:

“On a UNIX system, everything is a file; if something is not a file, it is a process.”

This statement is true because there are special files that are more than just files (named pipes and sockets, for instance), but to keep things simple, saying that everything is a file is an acceptable generalization. A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files. For now we will use this image of the tree, but we will find out later why this is not a fully accurate image.

Sorts of files

Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.

While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in `/dev`, we will discuss them later.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree. We will talk about links in detail.
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

The `-l` option to **ls** displays the file type, using the first character of each input line:

```
jaime:~/Documents> ls -l
```

```
total 80
-rw-rw-r--   1 jaime   jaime   31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r--   1 jaime   jaime   41472 Feb 21 17:56 Linux.doc
drwxrwxr-x   2 jaime   jaime    4096 Feb 25 11:50 course
```

This table gives an overview of the characters determining the file type:

Table 3.1. File types in a long list

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Special file
s	Socket
p	Named pipe
b	Block device

In order not to always have to perform a long listing for seeing the file type, a lot of systems by default don't issue just **ls**, but **ls `-F`**, which suffixes file names with one of the characters "/=*|@" to indicate the file type. To make it extra easy on the beginning user, both the `-F` and `--color` options are usually combined, see the section called "More about ls". We will use **ls `-F`** throughout this document for better readability.

As a user, you only need to deal directly with plain files, executable files, directories and links. The special file types are there for making your system do what you demand from it and are dealt with by system administrators and programmers.

Now, before we look at the important files and directories, we need to know more about partitions.

About partitioning

Why partition?

Most people have a vague knowledge of what partitions are, since every operating system has the ability to create or remove them. It may seem strange that Linux uses more than one partition on the same disk, even when using the standard installation procedure, so some explanation is called for.

One of the goals of having different partitions is to achieve higher data security in case of disaster. By dividing the hard disk in partitions, data can be grouped and separated.

When an accident occurs, only the data in the partition that got the hit will be damaged, while the data on the other partitions will most likely survive.

This principle dates from the days when Linux didn't have journaled file systems and power failures might have lead to disaster. The use of partitions remains for security and robustness reasons, so a breach on one part of the system doesn't automatically mean that the whole computer is in danger. This is currently the most important reason for partitioning. A simple example: a user creates a script, a program or a web application that starts filling up the disk. If the disk contains only one big partition, the entire system will stop functioning if the disk is full. If the user stores the data on a separate partition, then only that (data) partition will be affected, while the system partitions and possible other data partitions keep functioning.

Mind that having a journaled file system only provides data security in case of power failure and sudden disconnection of storage devices. This does not protect your data against bad blocks and logical errors in the file system. In those cases, you should use a RAID (Redundant Array of Inexpensive Disks) solution.

Partition layout and types

There are two kinds of major partitions on a Linux system:

- *data partition*: normal Linux system data, including the *root partition* containing all the data to start up and run the system; and
- *swap partition*: expansion of the computer's physical memory, extra memory on hard disk.

Most systems contain a root partition, one or more data partitions and one or more swap partitions. Systems in mixed environments may contain partitions for other system data, such as a partition with a FAT or VFAT file system for MS Windows data.

Most Linux systems use **fdisk** at installation time to set the partition type. As you may have noticed during the exercise from Chapter 1, this usually happens automatically. On some occasions, however, you may not be so lucky. In such cases, you will need to select the partition type manually and even manually do the actual partitioning. The standard Linux partitions have number 82 for swap and 83 for data, which can be journaled (ext3) or normal (ext2, on older systems). The **fdisk** utility has built-in help, should you forget these values.

Apart from these two, Linux supports a variety of other file system types, such as the relatively new Reiser file system, JFS, NFS, FATxx and many other file systems natively available on other (proprietary) operating systems.

The standard root partition (indicated with a single forward slash, */*) is about 100-500 MB, and contains the system configuration files, most basic commands and server

programs, system libraries, some temporary space and the home directory of the administrative user. A standard installation requires about 250 MB for the root partition.

Swap space (indicated with *swap*) is only accessible for the system itself, and is hidden from view during normal operation. Swap is the system that ensures, like on normal UNIX systems, that you can keep on working, whatever happens. On Linux, you will virtually never see irritating messages like *Out of memory, please close some applications first and try again*, because of this extra memory. The swap or virtual memory procedure has long been adopted by operating systems outside the UNIX world by now.

Using memory on a hard disk is naturally slower than using the real memory chips of a computer, but having this little extra is a great comfort. We will learn more about swap when we discuss processes in Chapter 4, *Processes*.

Linux generally counts on having twice the amount of physical memory in the form of swap space on the hard disk. When installing a system, you have to know how you are going to do this. An example on a system with 512 MB of RAM:

- 1st possibility: one swap partition of 1 GB
- 2nd possibility: two swap partitions of 512 MB
- 3rd possibility: with two hard disks: 1 partition of 512 MB on each disk.

The last option will give the best results when a lot of I/O is to be expected.

Read the software documentation for specific guidelines. Some applications, such as databases, might require more swap space. Others, such as some handheld systems, might not have any swap at all by lack of a hard disk. Swap space may also depend on your kernel version.

The kernel is on a separate partition as well in many distributions, because it is the most important file of your system. If this is the case, you will find that you also have a */boot* partition, holding your kernel(s) and accompanying data files.

The rest of the hard disk(s) is generally divided in data partitions, although it may be that all of the non-system critical data resides on one partition, for example when you perform a standard workstation installation. When non-critical data is separated on different partitions, it usually happens following a set pattern:

- a partition for user programs (*/usr*)
- a partition containing the users' personal data (*/home*)
- a partition to store temporary data like print- and mail-queues (*/var*)
- a partition for third party and extra software (*/opt*)

Once the partitions are made, you can only add more. Changing sizes or properties of existing partitions is possible but not advisable.

The division of hard disks into partitions is determined by the system administrator. On larger systems, he or she may even spread one partition over several hard disks, using the appropriate software. Most distributions allow for standard setups optimized for workstations (average users) and for general server purposes, but also accept customized partitions. During the installation process you can define your own partition layout using either your distribution specific tool, which is usually a straight forward graphical interface, or **fdisk**, a text-based tool for creating partitions and setting their properties.

A workstation or client installation is for use by mainly one and the same person. The selected software for installation reflects this and the stress is on common user packages, such as nice desktop themes, development tools, client programs for E-mail, multimedia software, web and other services. Everything is put together on one large partition, swap space twice the amount of RAM is added and your generic workstation is complete, providing the largest amount of disk space possible for personal use, but with the disadvantage of possible data integrity loss during problem situations.

On a server, system data tends to be separate from user data. Programs that offer services are kept in a different place than the data handled by this service. Different partitions will be created on such systems:

- a partition with all data necessary to boot the machine
- a partition with configuration data and server programs
- one or more partitions containing the server data such as database tables, user mails, an ftp archive etc.
- a partition with user programs and applications
- one or more partitions for the user specific files (home directories)
- one or more swap partitions (virtual memory)

Servers usually have more memory and thus more swap space. Certain server processes, such as databases, may require more swap space than usual; see the specific documentation for detailed information. For better performance, swap is often divided into different swap partitions.

Mount points

All partitions are attached to the system via a mount point. The mount point defines the place of a particular data set in the file system. Usually, all partitions are connected through the *root* partition. On this partition, which is indicated with the slash (/), directories are created. These empty directories will be the starting point of the partitions that are attached to them. An example: given a partition that holds the following directories:

```
videos/        cd-images/     pictures/
```

We want to attach this partition in the filesystem in a directory called `/opt/media`. In order to do this, the system administrator has to make sure that the directory `/opt/media` exists on the system. Preferably, it should be an empty directory. How this is done is explained later in this chapter. Then, using the **mount** command, the administrator can attach the partition to the system. When you look at the content of the formerly empty directory `/opt/media`, it will contain the files and directories that are on the mounted medium (hard disk or partition of a hard disk, CD, DVD, flash card, USB or other storage device).

During system startup, all the partitions are thus mounted, as described in the file `/etc/fstab`. Some partitions are not mounted by default, for instance if they are not constantly connected to the system, such like the storage used by your digital camera. If well configured, the device will be mounted as soon as the system notices that it is connected, or it can be user-mountable, i.e. you don't need to be system administrator to attach and detach the device to and from the system. There is an example in the section called “Using rsync”.

On a running system, information about the partitions and their mount points can be displayed using the **df** command (which stands for *disk full* or *disk free*). In Linux, **df** is the GNU version, and supports the `-h` or *human readable* option which greatly improves readability. Note that commercial UNIX machines commonly have their own versions of **df** and many other commands. Their behavior is usually the same, though GNU versions of common tools often have more and better features.

The **df** command only displays information about active non-swap partitions. These can include partitions from other networked systems, like in the example below where the home directories are mounted from a file server on the network, a situation often encountered in corporate environments.

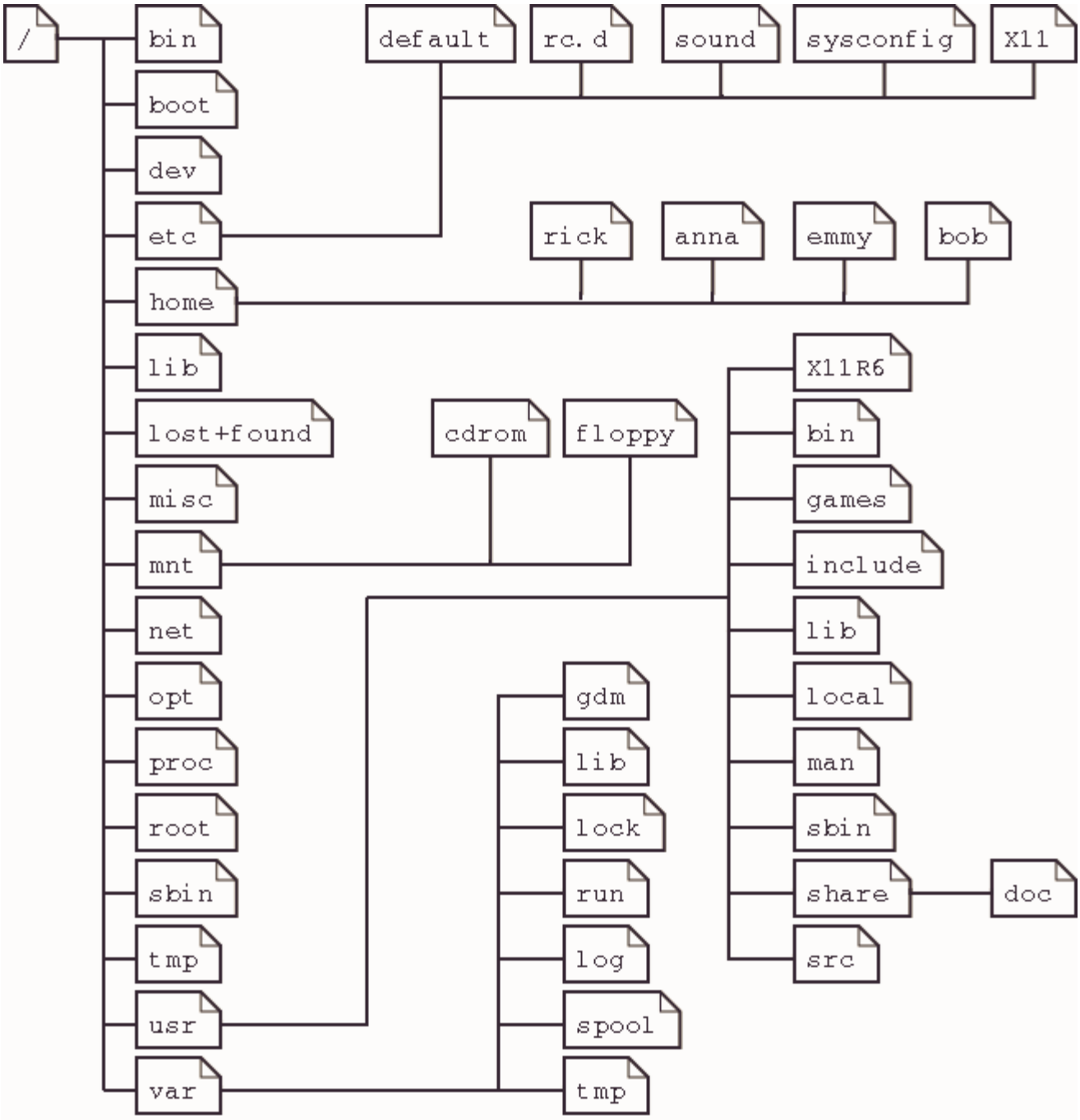
```
freddy:~> df -h
Filesystem          Size  Used Avail Use% Mounted on
/dev/hda8           496M  183M  288M  39% /
/dev/hda1           124M  8.4M  109M   8% /boot
/dev/hda5            19G   15G  2.7G  85% /opt
/dev/hda6           7.0G  5.4G  1.2G  81% /usr
/dev/hda7           3.7G  2.7G  867M  77% /var
fs1:/home           8.9G  3.7G  4.7G  44% /.automount/fs1/root/home
```

More file system layout

Visual

For convenience, the Linux file system is usually thought of in a tree structure. On a standard Linux system you will find the layout generally follows the scheme presented below.

Figure 3.1. Linux file system layout



This is a layout from a RedHat system. Depending on the system admin, the operating system and the mission of the UNIX machine, the structure may vary, and directories may be left out or added at will. The names are not even required; they are only a convention.

The tree of the file system starts at the trunk or *slash*, indicated by a forward slash (/). This directory, containing all underlying directories and files, is also called the *root directory* or “the root” of the file system.

Directories that are only one level below the root directory are often preceded by a slash, to indicate their position and prevent confusion with other directories that could have the same name. When starting with a new system, it is always a good idea to take a look in the root directory. Let's see what you could run into:

```
emmy:~> cd /
emmy:/> ls
bin/   dev/  home/    lib/         misc/  opt/     root/  tmp/  var/
boot/  etc/  initrd/  lost+found/  mnt/   proc/    sbin/  usr/
```

Table 3.2. Subdirectories of the root directory

Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, `vmlinuz`. In some recent distributions also `grub` data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in `/etc`, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting. Do not remove!
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a `lost+found` in its upper directory. Files that were saved during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in `proc` is obtained by entering the command **man *proc*** in a terminal window. The file `proc.txt` discusses the virtual file system in detail.
/root	The administrative user's home directory. Mind the difference between /, the

Directory	Content
	root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

How can you find out which partition a directory is on? Using the **df** command with a dot (.) as an option shows the partition the current directory belongs to, and informs about the amount of space used on this partition:

```
sandra:/lib> df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda7             980M  163M  767M  18% /
```

As a general rule, every directory under the root directory is on the root partition, unless it has a separate entry in the full listing from **df** (or **df** `-h` with no other options).

Read more in **man** ***hier***.

The file system in reality

For most users and for most common system administration tasks, it is enough to accept that files and directories are ordered in a tree-like structure. The computer, however, doesn't understand a thing about trees or tree-structures.

Every partition has its own file system. By imagining all those file systems together, we can form an idea of the tree-structure of the entire system, but it is not as simple as that. In a file system, a file is represented by an *inode*, a kind of serial number containing information about the actual data that makes up the file: to whom this file belongs, and where is it located on the hard disk.

Every partition has its own set of inodes; throughout a system with multiple partitions, files with the same inode number can exist.

Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data. When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created. This

number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition. We typically count on having 1 inode per 2 to 8 kilobytes of storage.

At the time a new file is created, it gets a free inode. In that inode is the following information:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file the section called “Access rights: Linux's first line of defense”
- Date and time of creation, last read and change.
- Date and time this information has been changed in the inode.
- Number of links to this file (see later in this chapter).
- File size
- An address defining the actual location of the file data.

The only information not included in an inode, is the file name and directory. These are stored in the special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. Users can display inode numbers using the `-i` option to ls. The inodes have their own separate space on the disk.

Orientation in the file system

The path

When you want the system to execute a command, you almost never have to give the full path to that command. For example, we know that the **ls** command is in the `/bin` directory (check with **which `-a ls`**), yet we don't have to enter the command **/bin/ls** for the computer to list the content of the current directory.

The `PATH` environment variable takes care of this. This variable lists those directories in the system where executable files can be found, and thus saves the user a lot of typing and memorizing locations of commands. So the path naturally contains a lot of directories containing `bin` somewhere in their names, as the user below demonstrates. The **echo** command is used to display the content (“$”) of the variable `PATH`:

```
rogier:> echo $PATH
/opt/local/bin:/usr/X11R6/bin:/usr/bin:/usr/sbin/:/bin
```

In this example, the directories `/opt/local/bin`, `/usr/X11R6/bin`, `/usr/bin`, `/usr/sbin` and `/bin` are subsequently searched for the required program. As soon as a match is found, the search is stopped, even if not every directory in the path has been searched. This can lead to strange situations. In the first example below, the user knows there is a program called **sendsms** to send an SMS message, and another user on the same

system can use it, but she can't. The difference is in the configuration of the `PATH` variable:

```
[jenny@blob jenny]$ sendsms
bash: sendsms: command not found
[jenny@blob jenny]$ echo $PATH
/bin:/usr/bin:/usr/bin/X11:/usr/X11R6/bin:/home/jenny/bin
[jenny@blob jenny]$ su - tony
Password:
tony:~>which sendsms
sendsms is /usr/local/bin/sendsms

tony:~>echo $PATH
/home/tony/bin.Linux:/home/tony/bin:/usr/local/bin:/usr/local/sbin:\
/usr/X11R6/bin:/usr/bin:/usr/sbin:/bin:/sbin
```

Note the use of the **su** (switch user) facility, which allows you to run a shell in the environment of another user, on the condition that you know the user's password.

A backslash indicates the continuation of a line on the next, without an **Enter** separating one line from the other.

In the next example, a user wants to call on the **wc** (word count) command to check the number of lines in a file, but nothing happens and he has to break off his action using the **Ctrl**+**C** combination:

```
jumper:~> wc -l test

(Ctrl-C)
jumper:~> which wc
wc is hashed (/home/jumper/bin/wc)

jumper:~> echo $PATH
/home/jumper/bin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:\
/usr/bin:/usr/sbin:/bin:/sbin
```

The use of the **which** command shows us that this user has a `bin`-directory in his home directory, containing a program that is also called **wc**. Since the program in his home directory is found first when searching the paths upon a call for **wc**, this “home-made” program is executed, with input it probably doesn't understand, so we have to stop it. To resolve this problem there are several ways (there are always several ways to solve a problem in UNIX/Linux): one answer could be to rename the user's **wc** program, or the user can give the full path to the exact command he wants, which can be found by using the `-a` option to the **which** command.

If the user uses programs in the other directories more frequently, he can change his path to look in his own directories last:

```
jumper:~> export PATH=/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:\
/usr/bin:/usr/sbin:/bin:/sbin:/home/jumper/bin
```

Changes are not permanent!

Note that when using the **export** command in a shell, the changes are temporary and only valid for this session (until you log out). Opening new sessions, even while the current one is still running, will not result in a new path in the new session. We will see in the section called “Your text environment” how we can make these kinds of changes to the environment permanent, adding these lines to the shell configuration files.

Absolute and relative paths

A path, which is the way you need to follow in the tree structure to reach a given file, can be described as starting from the trunk of the tree (the / or root directory). In that case, the path starts with a slash and is called an absolute path, since there can be no mistake: only one file on the system can comply.

In the other case, the path doesn't start with a slash and confusion is possible between `~/bin/wc` (in the user's home directory) and `bin/wc` in `/usr`, from the previous example. Paths that don't start with a slash are always relative.

In relative paths we also use the . and .. indications for the current and the parent directory. A couple of practical examples:

- When you want to compile source code, the installation documentation often instructs you to run the command **./configure**, which runs the *configure* program located in the current directory (that came with the new code), as opposed to running another configure program elsewhere on the system.
- In HTML files, relative paths are often used to make a set of pages easily movable to another place:

```
  <img alt="Garden with trees" src="../images/garden.jpg">
```

- Notice the difference one more time:

```
  theo:~> ls /mp3
  ls: /mp3: No such file or directory
  theo:~>ls mp3/
  oriental/  pop/  sixties/
```

The most important files and directories

The kernel

The kernel is the heart of the system. It manages the communication between the underlying hardware and the peripherals. The kernel also makes sure that processes and daemons (server processes) are started and stopped at the exact right times. The kernel has a lot of other important tasks, so many that there is a special kernel-development mailing list on this subject only, where huge amounts of information are

shared. It would lead us too far to discuss the kernel in detail. For now it suffices to know that the kernel is the most important file on the system.

The shell

What is a shell?

When I was looking for an appropriate explanation on the concept of a *shell*, it gave me more trouble than I expected. All kinds of definitions are available, ranging from the simple comparison that “the shell is the steering wheel of the car”, to the vague definition in the Bash manual which says that “bash is an sh-compatible command language interpreter,” or an even more obscure expression, “a shell manages the interaction between the system and its users”. A shell is much more than that.

A shell can best be compared with a way of talking to the computer, a language. Most users do know that other language, the point-and-click language of the desktop. But in that language the computer is leading the conversation, while the user has the passive role of picking tasks from the ones presented. It is very difficult for a programmer to include all options and possible uses of a command in the GUI-format. Thus, GUIs are almost always less capable than the command or commands that form the backend.

The shell, on the other hand, is an advanced way of communicating with the system, because it allows for two-way conversation and taking initiative. Both partners in the communication are equal, so new ideas can be tested. The shell allows the user to handle a system in a very flexible way. An additional asset is that the shell allows for task automation.

Shell types

Just like people know different languages and dialects, the computer knows different shell types:

- **sh** or Bourne Shell: the original shell still used on UNIX systems and in UNIX related environments. This is the basic shell, a small program with few features. When in POSIX-compatible mode, **bash** will emulate this shell.
- **bash** or Bourne Again SHell: the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, **bash** is the standard shell for common users. This shell is a so-called *superset* of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again SHell is compatible with the Bourne shell: commands that work in **sh**, also work in **bash**. However, the reverse is not always the case. All examples and exercises in this book use **bash**.
- **csh** or C Shell: the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.

- **tcsh** or Turbo C Shell: a superset of the common C Shell, enhancing user-friendliness and speed.
- **ksh** or the Korn shell: sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

The file `/etc/shells` gives an overview of known shells on a Linux system:

```
mia:~> cat /etc/shells
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

Fake Bourne shell

Note that `/bin/sh` is usually a link to Bash, which will execute in Bourne shell compatible mode when called on this way.

Your default shell is set in the `/etc/passwd` file, like this line for user *mia*:

```
mia:L2NOfqdlPrHwE:504:504:Mia Maya:/home/mia:/bin/bash
```

To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the `PATH` settings, and since a shell is an executable file (program), the current shell activates it and it gets executed. A new prompt is usually shown, because each shell has its typical appearance:

```
mia:~> tcsh
[mia@post21 ~]$
```

Which shell am I using?

If you don't know which shell you are using, either check the line for your account in `/etc/passwd` or type the command

```
echo $SHELL
```

Your home directory

Your home directory is your default destination when connecting to the system. In most cases it is a subdirectory of `/home`, though this may vary. Your home directory may be located on the hard disk of a remote file server; in that case your home directory may be found in `/nethome/your_user_name`. In another case the system administrator may have opted for a less comprehensible layout and your home directory may be on `/disk6/HU/07/jgillard`.

Whatever the path to your home directory, you don't have to worry too much about it. The correct path to your home directory is stored in the `HOME` environment variable, in case some program needs it. With the **echo** command you can display the content of this variable:

```
orlando:~> echo $HOME
/nethome/orlando
```

You can do whatever you like in your home directory. You can put as many files in as many directories as you want, although the total amount of data and files is naturally limited because of the hardware and size of the partitions, and sometimes because the system administrator has applied a quota system. Limiting disk usage was common practice when hard disk space was still expensive. Nowadays, limits are almost exclusively applied in large environments. You can see for yourself if a limit is set using the **quota** command:

```
pierre@lamaison:/> quota -v
Diskquotas for user pierre (uid 501): none
```

In case quotas have been set, you get a list of the limited partitions and their specific limitations. Exceeding the limits may be tolerated during a grace period with fewer or no restrictions at all. Detailed information can be found using the **info** ***quota*** or **man** ***quota*** commands.

No Quota?

If your system can not find the **quota**, then no limitation of file system usage is being applied.

Your home directory is indicated by a tilde (~), shorthand for `/path_to_home/user_name`. This same path is stored in the `HOME` variable, so you don't have to do anything to activate it. A simple application: switch from `/var/music/albums/arno/2001` to `images` in your home directory using one elegant command:

```
rom:/var/music/albums/arno/2001> cd ~/images

rom:~/images> pwd
/home/rom/images
```

Later in this chapter we will talk about the commands for managing files and directories in order to keep your home directory tidy.

The most important configuration files

As we mentioned before, most configuration files are stored in the `/etc` directory. Content can be viewed using the **cat** command, which sends text files to the standard output (usually your monitor). The syntax is straight forward:

```
cat file1 file2 ... fileN
```

In this section we try to give an overview of the most common configuration files. This is certainly not a complete list. Adding extra packages may also add extra configuration files in `/etc`. When reading the configuration files, you will find that they are usually quite well commented and self-explanatory. Some files also have man pages which contain extra documentation, such as **man *group***.

Table 3.3. Most common configuration files

File	Information/service
`aliases`	Mail aliases file for use with the Sendmail and Postfix mail server. Running a mail server on each and every system has long been common use in the UNIX world, and almost every Linux distribution still comes with a Sendmail package. In this file local user names are matched with real names as they occur in E-mail addresses, or with other local addresses.
`apache`	Config files for the Apache web server.
`bashrc`	The system-wide configuration file for the Bourne Again SHell. Defines functions and aliases for all users. Other shells may have their own system-wide config files, like `cshrc`.
`crontab` and the `cron.*` directories	Configuration of tasks that need to be executed periodically - backups, updates of the system databases, cleaning of the system, rotating logs etc.
`default`	Default options for certain commands, such as **useradd**.
`filesystems`	Known file systems: ext3, vfat, iso9660 etc.
`fstab`	Lists partitions and their *mount points*.
`ftp*`	Configuration of the ftp-server: who can connect, what parts of the system are accessible etc.
`group`	Configuration file for user groups. Use the shadow utilities **groupadd**, **groupmod** and **groupdel** to edit this file. Edit manually only if you really know what you are doing.
`hosts`	A list of machines that can be contacted using the network, but without the need for a domain name service. This has nothing to do with the system's network configuration, which is done in `/etc/sysconfig`.

File	Information/service
`inittab`	Information for booting: mode, number of text consoles etc.
`issue`	Information about the distribution (release version and/or kernel info).
`ld.so.conf`	Locations of library files.
`lilo.conf`, `silo.conf`, `aboot.conf` etc.	Boot information for the LInux LOader, the system for booting that is now gradually being replaced with GRUB.
`logrotate.*`	Rotation of the logs, a system preventing the collection of huge amounts of log files.
`mail`	Directory containing instructions for the behavior of the mail server.
`modules.conf`	Configuration of modules that enable special features (drivers).
`motd`	Message Of The Day: Shown to everyone who connects to the system (in text mode), may be used by the system admin to announce system services/maintenance etc.
`mtab`	Currently mounted file systems. It is advised to never edit this file.
`nsswitch.conf`	Order in which to contact the name resolvers when a process demands resolving of a host name.
`pam.d`	Configuration of authentication modules.
`passwd`	Lists local users. Use the shadow utilities **useradd**, **usermod** and **userdel** to edit this file. Edit manually only when you really know what you are doing.
`printcap`	Outdated but still frequently used printer configuration file. Don't edit this manually unless you really know what you are doing.
`profile`	System wide configuration of the shell environment: variables, default properties of new files, limitation of resources etc.
`rc*`	Directories defining active services for each run level.
`resolv.conf`	Order in which to contact DNS servers (Domain Name Servers only).

File	Information/service
`sendmail.cf`	Main config file for the Sendmail server.
`services`	Connections accepted by this machine (open ports).
`sndconfig` or `sound`	Configuration of the sound card and sound events.
`ssh`	Directory containing the config files for secure shell client and server.
`sysconfig`	Directory containing the system configuration files: mouse, keyboard, network, desktop, system clock, power management etc. (specific to RedHat)
`X11`	Settings for the graphical server, X. RedHat uses XFree, which is reflected in the name of the main configuration file, XFree86Config. Also contains the general directions for the window managers available on the system, for example **gdm**, **fvwm**, **twm**, etc.
`xinetd.*` or `inetd.conf`	Configuration files for Internet services that are run from the system's (extended) Internet services daemon (servers that don't run an independent daemon).

Throughout this guide we will learn more about these files and study some of them in detail.

The most common devices

Devices, generally every peripheral attachment of a PC that is not the CPU itself, is presented to the system as an entry in the `/dev` directory. One of the advantages of this UNIX-way of handling devices is that neither the user nor the system has to worry much about the specification of devices.

Users that are new to Linux or UNIX in general are often overwhelmed by the amount of new names and concepts they have to learn. That is why a list of common devices is included in this introduction.

Table 3.4. Common devices

Name	Device

Name	Device
`cdrom`	CD drive
`console`	Special entry for the currently used console.
`cua*`	Serial ports
`dsp*`	Devices for sampling and recording
`fd*`	Entries for most kinds of floppy drives, the default is `/dev/fd0`, a floppy drive for 1.44 MB floppies.
`hd[a-t][1-16]`	Standard support for IDE drives with maximum amount of partitions each.
`ir*`	Infrared devices
`isdn*`	Management of ISDN connections
`js*`	Joystick(s)
`lp*`	Printers
`mem`	Memory
`midi*`	midi player
`mixer*` and `music`	Idealized model of a mixer (combines or adds signals)
`modem`	Modem
`mouse` (also msmouse, logimouse, psmouse, input/mice, psaux)	All kinds of mouses
`null`	Bottomless garbage can
`par*`	Entries for parallel port support
`pty*`	Pseudo terminals
`radio*`	For Radio Amateurs (HAMs).

Name	Device
ram*	boot device
sd*	SCSI disks with their partitions
sequencer	For audio applications using the synthesizer features of the sound card (MIDI-device controller)
tty*	Virtual consoles simulating vt100 terminals.
usb*	USB card and scanner
video*	For use with a graphics card supporting video.

The most common variable files

In the /var directory we find a set of directories for storing specific non-constant data (as opposed to the ls program or the system configuration files, which change relatively infrequently or never at all). All files that change frequently, such as log files, mailboxes, lock files, spoolers etc. are kept in a subdirectory of /var.

As a security measure these files are usually kept in separate parts from the main system files, so we can keep a close eye on them and set stricter permissions where necessary. A lot of these files also need more permissions than usual, like /var/tmp, which needs to be writable for everyone. A lot of user activity might be expected here, which might even be generated by anonymous Internet users connected to your system. This is one reason why the /var directory, including all its subdirectories, is usually on a separate partition. This way, there is for instance no risk that a mail bomb, for instance, fills up the rest of the file system, containing more important data such as your programs and configuration files.

/var/tmp and /tmp

Files in /tmp can be deleted without notice, by regular system tasks or because of a system reboot. On some (customized) systems, also /var/tmp might behave unpredictably. Nevertheless, since this is not the case by default, we advise to use the /var/tmp directory for saving temporary files. When in doubt, check with your system administrator. If you manage your own system, you can be reasonably sure that this is a safe place if you did not consciously change settings on /var/tmp (as root, a normal user can not do this).

> Whatever you do, try to stick to the privileges granted to a normal user - don't go saving files directly under the root (/) of the file system, don't put them in `/usr` or some subdirectory or in another reserved place. This pretty much limits your access to safe file systems.

One of the main security systems on a UNIX system, which is naturally implemented on every Linux machine as well, is the log-keeping facility, which logs all user actions, processes, system events etc. The configuration file of the so-called *syslogdaemon* determines which and how long logged information will be kept. The default location of all logs is `/var/log`, containing different files for access log, server logs, system messages etc.

In `/var` we typically find server data, which is kept here to separate it from critical data such as the server program itself and its configuration files. A typical example on Linux systems is `/var/www`, which contains the actual HTML pages, scripts and images that a web server offers. The FTP-tree of an FTP server (data that can be downloaded by a remote client) is also best kept in one of `/var`'s subdirectories. Because this data is publicly accessible and often changeable by anonymous users, it is safer to keep it here, away from partitions or directories with sensitive data.

On most workstation installations, `/var/spool` will at least contain an `at` and a `cron` directory, containing scheduled tasks. In office environments this directory usually contains `lpd` as well, which holds the print queue(s) and further printer configuration files, as well as the printer log files.

On server systems we will generally find `/var/spool/mail`, containing incoming mails for local users, sorted in one file per user, the user's “inbox”. A related directory is `mqueue`, the spooler area for unsent mail messages. These parts of the system can be very busy on mail servers with a lot of users. News servers also use the `/var/spool` area because of the enormous amounts of messages they have to process.

The `/var/lib/rpm` directory is specific to RPM-based (RedHat Package Manager) distributions; it is where RPM package information is stored. Other package managers generally also store their data somewhere in `/var`.

Manipulating files

Viewing file properties

More about ls

Besides the name of the file, **ls** can give a lot of other information, such as the file type, as we already discussed. It can also show permissions on a file, file size, inode number, creation date and time, owners and amount of links to the file. With the `-a` option to **ls**,

files that are normally hidden from view can be displayed as well. These are files that have a name starting with a dot. A couple of typical examples include the configuration files in your home directory. When you've worked with a certain system for a while, you will notice that tens of files and directories have been created that are not automatically listed in a directory index. Next to that, every directory contains a file named just dot (.) and one with two dots (..), which are used in combination with their inode number to determine the directory's position in the file system's tree structure.

You should really read the Info pages about **ls**, since it is a very common command with a lot of useful options. Options can be combined, as is the case with most UNIX commands and their options. A common combination is **ls -al**; it shows a long list of files and their properties as well as the destinations that any symbolic links point to. **ls -latr** displays the same files, only now in reversed order of the last change, so that the file changed most recently occurs at the bottom of the list. Here are a couple of examples:

```
krissie:~/mp3> ls
Albums/  Radio/  Singles/  gene/  index.html

krissie:~/mp3> ls -a
./   .thumbs  Radio     gene/
../  Albums/  Singles/  index.html

krissie:~/mp3> ls -l Radio/
total 8
drwxr-xr-x    2 krissie krissie  4096 Oct 30  1999 Carolina/
drwxr-xr-x    2 krissie krissie  4096 Sep 24  1999 Slashdot/

krissie:~/mp3> ls -ld Radio/
drwxr-xr-x    4 krissie krissie  4096 Oct 30  1999 Radio/

krissie:~/mp3> ls -ltr
total 20
drwxr-xr-x    4 krissie krissie  4096 Oct 30  1999 Radio/
-rw-r--r--    1 krissie krissie   453 Jan  7  2001 index.html
drwxrwxr-x   30 krissie krissie  4096 Oct 20 17:32 Singles/
drwxr-xr-x    2 krissie krissie  4096 Dec  4 23:22 gene/
drwxrwxr-x   13 krissie krissie  4096 Dec 21 11:40 Albums/
```

On most Linux versions **ls** is *aliased* to color-ls by default. This feature allows to see the file type without using any options to **ls**. To achieve this, every file type has its own color. The standard scheme is in `/etc/DIR_COLORS`:

Table 3.5. Color-ls default color scheme

Color	File type
blue	directories

Color	File type
red	compressed archives
white	text files
pink	images
cyan	links
yellow	devices
green	executables
flashing red	broken links

More information is in the man page. The same information was in earlier days displayed using suffixes to every non-standard file name. For mono-color use (like printing a directory listing) and for general readability, this scheme is still in use:

Table 3.6. Default suffix scheme for ls

Character	File type
nothing	regular file
/	directory
*	executable file
@	link
=	socket
|	named pipe

A description of the full functionality and features of the **ls** command can be read with **info** ***coreutils ls***.

More tools

To find out more about the kind of data we are dealing with, we use the **file** command. By applying certain tests that check properties of a file in the file system, magic numbers and language tests, **file** tries to make an educated guess about the format of a file. Some examples:

```
mike:~> file Documents/
Documents/: directory

mike:~> file high-tech-stats.pdf
high-tech-stats.pdf: PDF document, version 1.2

mike:~> file Nari-288.rm
Nari-288.rm: RealMedia file

mike:~> file bijlage10.sdw
bijlage10.sdw: Microsoft Office Document

mike:~> file logo.xcf
logo.xcf: GIMP XCF image data, version 0, 150 x 38, RGB Color

mike:~> file cv.txt
cv.txt: ISO-8859 text

mike:~> file image.png
image.png: PNG image data, 616 x 862, 8-bit grayscale, non-interlaced

mike:~> file figure
figure: ASCII text

mike:~> file me+tux.jpg
me+tux.jpg: JPEG image data, JFIF standard 1.01, resolution (DPI),
            "28 Jun 1999", 144 x 144

mike:~> file 42.zip.gz
42.zip.gz: gzip compressed data, deflated, original filename,
          `42.zip', last modified: Thu Nov  1 23:45:39 2001, os: Unix

mike:~> file vi.gif
vi.gif: GIF image data, version 89a, 88 x 31

mike:~> file slide1
slide1: HTML document text

mike:~> file template.xls
template.xls: Microsoft Office Document

mike:~> file abook.ps
abook.ps: PostScript document text conforming at level 2.0

mike:~> file /dev/log
/dev/log: socket

mike:~> file /dev/hda
/dev/hda: block special (3/0)
```

The **file** command has a series of options, among others the `-z` option to look into compressed files. See **info** ***file*** for a detailed description. Keep in mind that the results of **file** are not absolute, it is only a guess. In other words, **file** can be tricked.

Why all the fuss about file types and formats?

Shortly, we will discuss a couple of command-line tools for looking at *plain text files*. These tools will not work when used on the wrong type of files. In the worst case, they will crash your terminal and/or make a lot of beeping noises. If this happens to you, just close the terminal session and start a new one. But try to avoid it, because it is usually very disturbing for other people.

Creating and deleting files and directories

Making a mess...

... Is not a difficult thing to do. Today almost every system is networked, so naturally files get copied from one machine to another. And especially when working in a graphical environment, creating new files is a piece of cake and is often done without the approval of the user. To illustrate the problem, here's the full content of a new user's directory, created on a standard RedHat system:

```
[newuser@blob user]$ ls -al
total 32
drwx------   3 user    user        4096 Jan 16 13:32 .
drwxr-xr-x   6 root     root        4096 Jan 16 13:32 ..
-rw-r--r--   1 user    user          24 Jan 16 13:32 .bash_logout
-rw-r--r--   1 user    user         191 Jan 16 13:32 .bash_profile
-rw-r--r--   1 user    user         124 Jan 16 13:32 .bashrc
drwxr-xr-x   3 user    user        4096 Jan 16 13:32 .kde
-rw-r--r--   1 user    user        3511 Jan 16 13:32 .screenrc
-rw-------   1 user    user          61 Jan 16 13:32 .xauthDqztLr
```

On first sight, the content of a “used” home directory doesn't look that bad either:

```
olduser:~> ls
app-defaults/ crossover/   Fvwm@     mp3/      OpenOffice.org638/
articles/     Desktop/     GNUstep/  Nautilus/ staroffice6.0/
bin/          Desktop1/    images/   nqc/      training/
brol/         desktoptest/ Machines@ ns_imap/  webstart/
C/            Documents/   mail/     nsmail/   xml/
closed/       Emacs@       Mail/     office52/ Xrootenv.0
```

But when all the directories and files starting with a dot are included, there are 185 items in this directory. This is because most applications have their own directories and/or files, containing user-specific settings, in the home directory of that user. Usually these files are created the first time you start an application. In some cases you will be

notified when a non-existent directory needs to be created, but most of the time everything is done automatically.

Furthermore, new files are created seemingly continuously because users want to save files, keep different versions of their work, use Internet applications, and download files and attachments to their local machine. It doesn't stop. It is clear that one definitely needs a scheme to keep an overview on things.

In the next section, we will discuss our means of keeping order. We only discuss text tools available to the shell, since the graphical tools are very intuitive and have the same look and feel as the well known point-and-click MS Windows-style file managers, including graphical help functions and other features you expect from this kind of applications. The following list is an overview of the most popular file managers for GNU/Linux. Most file managers can be started from the menu of your desktop manager, or by clicking your home directory icon, or from the command line, issuing these commands:

- **nautilus**: The default file manager in Gnome, the GNU desktop. Excellent documentation about working with this tool can be found at http://www.gnome.org.
- **konqueror**: The file manager typically used on a KDE desktop. The handbook is at http://docs.kde.org.
- **mc**: Midnight Commander, the Unix file manager after the fashion of Norton Commander. All documentation available from http://gnu.org/directory/ or a mirror, such as http://www.ibiblio.org.

These applications are certainly worth giving a try and usually impress newcomers to Linux, if only because there is such a wide variety: these are only the most popular tools for managing directories and files, and many other projects are being developed. Now let's find out about the internals and see how these graphical tools use common UNIX commands.

The tools

Creating directories

A way of keeping things in place is to give certain files specific default locations by creating directories and subdirectories (or folders and sub-folders if you wish). This is done with the **mkdir** command:

```
richard:~> mkdir archive

richard:~> ls -ld archive
drwxrwxrwx  2 richard richard          4096 Jan 13 14:09 archive/
```

Creating directories and subdirectories in one step is done using the `-p` option:

```
richard:~> cd archive

richard:~/archive> mkdir 1999 2000 2001

richard:~/archive> ls
1999/  2000/  2001/

richard:~/archive> mkdir 2001/reports/Restaurants-Michelin/
mkdir: cannot create directory `2001/reports/Restaurants-Michelin/':
No such file or directory

richard:~/archive> mkdir -p 2001/reports/Restaurants-Michelin/

richard:~/archive> ls 2001/reports/
Restaurants-Michelin/
```

If the new file needs other permissions than the default file creation permissions, the new access rights can be set in one move, still using the **mkdir** command, see the Info pages for more. We are going to discuss access modes in the next section on file security.

The name of a directory has to comply with the same rules as those applied on regular file names. One of the most important restrictions is that you can't have two files with the same name in one directory (but keep in mind that Linux is, like UNIX, a case sensitive operating system). There are virtually no limits on the length of a file name, but it is usually kept shorter than 80 characters, so it can fit on one line of a terminal. You can use any character you want in a file name, although it is advised to exclude characters that have a special meaning to the shell. When in doubt, check with Appendix C, *Shell Features*.

Moving files

Now that we have properly structured our home directory, it is time to clean up unclassified files using the **mv** command:

```
richard:~/archive> mv ../report[1-4].doc reports/Restaurants-Michelin/
```

This command is also applicable when renaming files:

```
richard:~> ls To_Do
-rw-rw-r--    1 richard richard      2534 Jan 15 12:39 To_Do

richard:~> mv To_Do done

richard:~> ls -l done
-rw-rw-r--    1 richard richard      2534 Jan 15 12:39 done
```

It is clear that only the name of the file changes. All other properties remain the same.

Detailed information about the syntax and features of the **mv** command can be found in the man or Info pages. The use of this documentation should always be your first reflex when confronted with a problem. The answer to your problem is likely to be in the system documentation. Even experienced users read man pages every day, so beginning users should read them all the time. After a while, you will get to know the most common options to the common commands, but you will still need the documentation as a primary source of information. Note that the information contained in the HOWTOs, FAQs, man pages and other sources is slowly being merged into the Info pages, which are today the most up-to-date source of online (as in readily available on the system) documentation.

Copying files

Copying files and directories is done with the **cp** command. A useful option is recursive copy (copy all underlying files and subdirectories), using the `-R` option to **cp**. The general syntax is

```
cp [-R] fromfile tofile
```

As an example the case of user *newguy*, who wants the same Gnome desktop settings user *oldguy* has. One way to solve the problem is to copy the settings of *oldguy* to the home directory of *newguy*:

```
victor:~> cp -R ../oldguy/.gnome/ .
```

This gives some errors involving file permissions, but all the errors have to do with private files that *newguy* doesn't need anyway. We will discuss in the next part how to change these permissions in case they really are a problem.

Removing files

Use the **rm** command to remove single files, **rmdir** to remove empty directories. (Use **ls -a** to check whether a directory is empty or not). The **rm** command also has options for removing non-empty directories with all their subdirectories, read the Info pages for these rather dangerous options.

How empty can a directory be?

It is normal that the directories . (dot) and .. (dot-dot) can't be removed, since they are also necessary in an empty directory to determine the directories ranking in the file system hierarchy.

On Linux, just like on UNIX, there is no garbage can - at least not for the shell, although there are plenty of solutions for graphical use. So once removed, a file is really gone, and there is generally no way to get it back unless you have backups, or you are really fast and have a real good system administrator. To protect the beginning user from this

malice, the interactive behavior of the **rm**, **cp** and **mv** commands can be activated using the `-i` option. In that case the system won't immediately act upon request. Instead it will ask for confirmation, so it takes an additional click on the **Enter** key to inflict the damage:

```
mary:~> rm -ri archive/
rm: descend into directory `archive'? y
rm: descend into directory `archive/reports'? y
rm: remove directory `archive/reports'? y
rm: descend into directory `archive/backup'? y
rm: remove `archive/backup/sysbup200112.tar'? y
rm: remove directory `archive/backup'? y
rm: remove directory `archive'? y
```

We will discuss how to make this option the default in Chapter 7, *Home sweet /home*, which discusses customizing your shell environment.

Finding files

Using shell features

In the example on moving files we already saw how the shell can manipulate multiple files at once. In that example, the shell finds out automatically what the user means by the requirements between the square braces “[” and “]”. The shell can substitute ranges of numbers and upper or lower case characters alike. It also substitutes as many characters as you want with an asterisk, and only one character with a question mark.

All sorts of substitutions can be used simultaneously; the shell is very logical about it. The Bash shell, for instance, has no problem with expressions like **ls *dirname/*/*/*[2-3]***.

In other shells, the asterisk is commonly used to minimize the efforts of typing: people would enter **cd *dir**** instead of **cd `directory`**. In Bash however, this is not necessary because the GNU shell has a feature called file name completion. It means that you can type the first few characters of a command (anywhere) or a file (in the current directory) and if no confusion is possible, the shell will find out what you mean. For example in a directory containing many files, you can check if there are any files beginning with the letter A just by typing **ls *A*** and pressing the **Tab** key twice, rather than pressing **Enter**. If there is only one file starting with “A”, this file will be shown as the argument to **ls** (or any shell command, for that matter) immediately.

Which

A very simple way of looking up files is using the **which** command, to look in the directories listed in the user's search path for the required file. Of course, since the search path contains only paths to directories containing executable programs, **which** doesn't work for ordinary files. The **which** command is useful when troubleshooting

“Command not Found” problems. In the example below, user *tina* can't use the **acroread** program, while her colleague has no troubles whatsoever on the same system. The problem is similar to the PATH problem in the previous part: Tina's colleague tells her that he can see the required program in `/opt/acroread/bin`, but this directory is not in her path:

```
tina:~> which acroread
/usr/bin/which: no acroread in (/bin:/usr/bin:/usr/bin/X11)
```

The problem can be solved by giving the full path to the command to run, or by re-exporting the content of the PATH variable:

```
tina:~> export PATH=$PATH:/opt/acroread/bin

tina:~> echo $PATH
/bin:/usr/bin:/usr/bin/X11:/opt/acroread/bin
```

Using the **which** command also checks to see if a command is an alias for another command:

```
gerrit:~> which -a ls
ls is aliased to `ls -F --color=auto'
ls is /bin/ls
```

If this does not work on your system, use the **alias** command:

```
tille@www:~/mail$ alias ls
alias ls='ls --color'
```

Find and locate

These are the real tools, used when searching other paths beside those listed in the search path. The **find** tool, known from UNIX, is very powerful, which may be the cause of a somewhat more difficult syntax. GNU **find**, however, deals with the syntax problems. This command not only allows you to search file names, it can also accept file size, date of last change and other file properties as criteria for a search. The most common use is for finding file names:

```
find <path> -name <searchstring>
```

This can be interpreted as “Look in all files and subdirectories contained in a given path, and print the names of the files containing the search string in their name” (not in their content).

Another application of **find** is for searching files of a certain size, as in the example below, where user *peter* wants to find all files in the current directory or one of its subdirectories, that are bigger than 5 MB:

```
peter:~> find . -size +5000k
psychotic_chaos.mp3
```

If you dig in the man pages, you will see that **find** can also perform operations on the found files. A common example is removing files. It is best to first test without the `-exec` option that the correct files are selected, after that the command can be rerun to delete the selected files. Below, we search for files ending in `.tmp`:

```
peter:~>  find . -name "*.tmp" -exec rm {} \;

peter:~>
```

Optimize!

This command will call on **rm** as many times as a file answering the requirements is found. In the worst case, this might be thousands or millions of times. This is quite a load on your system.

A more realistic way of working would be the use of a pipe (|) and the **xargs** tool with **rm** as an argument. This way, the **rm** command is only called when the command line is full, instead of for every file. See Chapter 5, *I/O redirection* for more on using I/O redirection to ease everyday tasks.

Later on (in 1999 according to the man pages, after 20 years of **find**), **locate** was developed. This program is easier to use, but more restricted than **find**, since its output is based on a file index database that is updated only once every day. On the other hand, a search in the **locate** database uses less resources than **find** and therefore shows the results nearly instantly.

Most Linux distributions use **slocate** these days, security enhanced locate, the modern version of **locate** that prevents users from getting output they have no right to read. The files in *root*'s home directory are such an example, these are not normally accessible to the public. A user who wants to find someone who knows about the C shell may issue the command **locate `.cshrc`**, to display all users who have a customized configuration file for the C shell. Supposing the users *root* and *jenny* are running C shell, then only the file `/home/jenny/.cshrc` will be displayed, and not the one in *root*'s home directory. On most systems, **locate** is a symbolic link to the **slocate** program:

```
billy:~> ls -l /usr/bin/locate
lrwxrwxrwx 1 root slocate  7 Oct 28 14:18 /usr/bin/locate -> slocate*
```

User *tina* could have used **locate** to find the application she wanted:

```
tina:~> locate acroread
/usr/share/icons/hicolor/16x16/apps/acroread.png
/usr/share/icons/hicolor/32x32/apps/acroread.png
/usr/share/icons/locolor/16x16/apps/acroread.png
/usr/share/icons/locolor/32x32/apps/acroread.png
/usr/local/bin/acroread
```

```
/usr/local/Acrobat4/Reader/intellinux/bin/acroread
/usr/local/Acrobat4/bin/acroread
```

Directories that don't contain the name `bin` can't contain the program - they don't contain executable files. There are three possibilities left. The file in `/usr/local/bin` is the one *tina* would have wanted: it is a link to the shell script that starts the actual program:

```
tina:~> file /usr/local/bin/acroread
/usr/local/bin/acroread: symbolic link to ../Acrobat4/bin/acroread

tina:~> file /usr/local/Acrobat4/bin/acroread
/usr/local/Acrobat4/bin/acroread: Bourne shell script text executable

tina:~> file /usr/local/Acrobat4/Reader/intellinux/bin/acroread
/usr/local/Acrobat4/Reader/intellinux/bin/acroread: ELF 32-bit LSB
executable, Intel 80386, version 1, dynamically linked (uses
shared libs), not stripped
```

In order to keep the path as short as possible, so the system doesn't have to search too long every time a user wants to execute a command, we add `/usr/local/bin` to the path and not the other directories, which only contain the binary files of one specific program, while `/usr/local/bin` contains other useful programs as well.

Again, a description of the full features of **find** and **locate** can be found in the Info pages.

The grep command

General line filtering

A simple but powerful program, **grep** is used for filtering input lines and returning certain patterns to the output. There are literally thousands of applications for the **grep** program. In the example below, *jerry* uses **grep** to see how he did the thing with **find**:

```
jerry:~> grep -a find .bash_history
find . -name userinfo
man find
find ../ -name common.cfg
```

Search history

Also useful in these cases is the search function in **bash**, activated by pressing **Ctrl**+**R** at once, such as in the example where we want to check how we did that last **find** again:

```
thomas ~> ^R
(reverse-i-search)`find': find `/home/thomas` -name *.xml
```

Type your search string at the search prompt. The more characters you type, the

more restricted the search gets. This reads the command history for this shell session (which is written to `.bash_history` in your home directory when you quit that session). The most recent occurrence of your search string is shown. If you want to see previous commands containing the same string, type **Ctrl**+**R** again.

See the Info pages on **bash** for more.

All UNIXes with just a little bit of decency have an online dictionary. So does Linux. The dictionary is a list of known words in a file named `words`, located in `/usr/share/dict`. To quickly check the correct spelling of a word, no graphical application is needed:

```
william:~> grep pinguin /usr/share/dict/words

william:~> grep penguin /usr/share/dict/words
penguin
penguins
```

Dictionary vs. word list

Some distributions offer the **dict** command, which offers more features than simply searching words in a list.

Who is the owner of that home directory next to mine? Hey, there's his telephone number!

```
lisa:~> grep gdbruyne /etc/passwd
gdbruyne:x:981:981:Guy Debruyne, tel 203234:/home/gdbruyne:/bin/bash
```

And what was the E-mail address of Arno again?

```
serge:~/mail> grep -i arno *
sent-mail: To: <Arno.Hintjens@celeb.com>
sent-mail: On Mon, 24 Dec 2001, Arno.Hintjens@celeb.com wrote:
```

find and **locate** are often used in combination with **grep** to define some serious queries. For more information, see Chapter 5, *I/O redirection* on I/O redirection.

Special characters

Characters that have a special meaning to the shell have to be *escaped*. The escape character in Bash is backslash, as in most shells; this takes away the special meaning of the following character. The shell knows about quite some special characters, among the most common /, ., ? and *. A full list can be found in the Info pages and documentation for your shell.

For instance, say that you want to display the file “*” instead of all the files in a directory, you would have to use

```
less \*
```

The same goes for filenames containing a space:

```
cat This\ File
```

More ways to view file content

General

Apart from **cat**, which really doesn't do much more than sending files to the standard output, there are other tools to view file content.

The easiest way of course would be to use graphical tools instead of command line tools. In the introduction we already saw a glimpse of an office application, OpenOffice.org. Other examples are the GIMP (start up with **gimp** from the command line), the GNU Image Manipulation Program; **xpdf** to view Portable Document Format files (PDF); GhostView (**gv**) for viewing PostScript files; Mozilla/FireFox, **links** (a text mode browser), Konqueror, Opera and many others for web content; XMMS, CDplay and others for multimedia file content; AbiWord, Gnumeric, KOffice etc. for all kinds of office applications and so on. There are thousands of Linux applications; to list them all would take days.

Instead we keep concentrating on shell- or text-mode applications, which form the basics for all other applications. These commands work best in a text environment on files containing text. When in doubt, check first using the **file** command.

So let's see what text tools we have that are useful to look inside files.

Font problems

Plain text tools such as the ones we will now be discussing, often have problems with “plain” text files because of the font encoding used in those files. Special characters, such as accented alphabetical characters, Chinese characters and other characters from languages using different character sets than the default *en_US* encoding and so on, are then displayed the wrong way or replaced by unreadable rubbish. These problems are discussed in the section called “Region specific settings”.

“less is more”

Undoubtedly you will hear someone say this phrase sooner or later when working in a UNIX environment. A little bit of UNIX history explains this:

- First there was **cat**. Output was streamed in an uncontrollable way.

- Then there was **pg**, which may still be found on older UNIXes. This command puts text to the output one page at the time.
- The **more** program was a revised version of **pg**. This command is still available on every Linux system.
- **less** is the GNU version of more and has extra features allowing highlighting of search strings, scrolling back etc. The syntax is very simple:

```
less name_of_file
```

More information is located in the Info pages.

You already know about pagers by now, because they are used for viewing the man pages.

The head and tail commands

These two commands display the n first/last lines of a file respectively. To see the last ten commands entered:

```
tony:~> tail -10 .bash_history
locate configure | grep bin
man bash
cd
xawtv &
grep usable /usr/share/dict/words
grep advisable /usr/share/dict/words
info quota
man quota
echo $PATH
frm
```

head works similarly. The **tail** command has a handy feature to continuously show the last n lines of a file that changes all the time. This `-f` option is often used by system administrators to check on log files. More information is located in the system documentation files.

Linking files

Link types

Since we know more about files and their representation in the file system, understanding links (or shortcuts) is a piece of cake. A link is nothing more than a way of matching two or more file names to the same set of file data. There are two ways to achieve this:

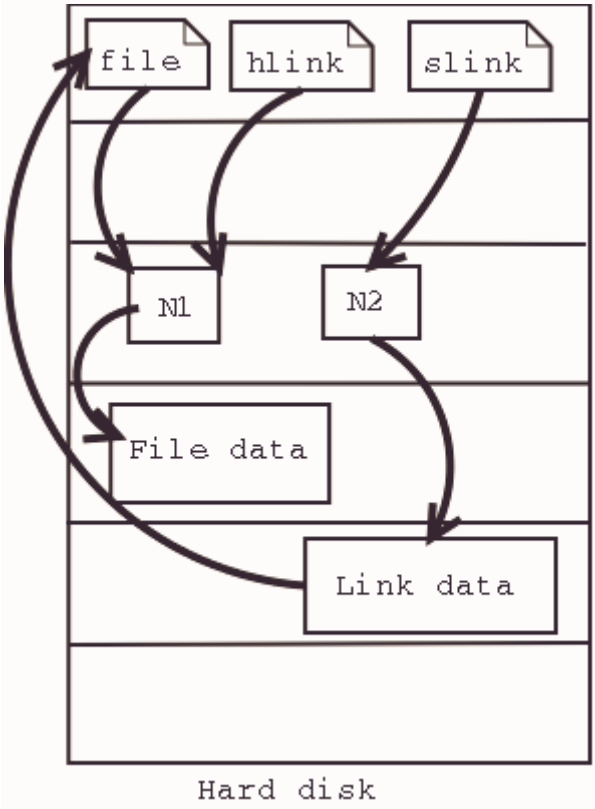
- Hard link: Associate two or more file names with the same inode. Hard links share the same data blocks on the hard disk, while they continue to behave as independent files.

  There is an immediate disadvantage: hard links can't span partitions, because inode numbers are only unique within a given partition.

- Soft link or symbolic link (or for short: symlink): a small file that is a pointer to another file. A symbolic link contains the path to the target file instead of a physical location on the hard disk. Since inodes are not used in this system, soft links can span across partitions.

The two link types behave similar, but are not the same, as illustrated in the scheme below:

Figure 3.2. Hard and soft link mechanism



Note that removing the target file for a symbolic link makes the link useless.

Each regular file is in principle a hardlink. Hardlinks can not span across partitions, since they refer to inodes, and inode numbers are only unique within a given partition.

It may be argued that there is a third kind of link, the *user-space* link, which is similar to a shortcut in MS Windows. These are files containing meta-data which can only be interpreted by the graphical file manager. To the kernel and the shell these are just normal files. They may end in a *.desktop* or *.lnk* suffix; an example can be found in `~/.gnome-desktop`:

```
[dupont@boulot .gnome-desktop]$ cat La\ Maison\ Dupont
[Desktop Entry]
Encoding=Legacy-Mixed
Name=La Maison Dupont
Type=X-nautilus-home
X-Nautilus-Icon=temp-home
URL=file:///home/dupont
```

This example is from a KDE desktop:

```
[lena@venus Desktop]$ cat camera
[Desktop Entry]
Dev=/dev/sda1
FSType=auto
Icon=memory
MountPoint=/mnt/camera
Type=FSDevice
X-KDE-Dynamic-Device=true
```

Creating this kind of link is easy enough using the features of your graphical environment. Should you need help, your system documentation should be your first resort.

In the next section, we will study the creation of UNIX-style symbolic links using the command line.

Creating symbolic links

The symbolic link is particularly interesting for beginning users: they are fairly obvious to see and you don't need to worry about partitions.

The command to make links is **ln**. In order to create symlinks, you need to use the `-s` option:

```
ln -s targetfile linkname
```

In the example below, user *freddy* creates a link in a subdirectory of his home directory to a directory on another part of the system:

```
freddy:~/music> ln -s /opt/mp3/Queen/ Queen

freddy:~/music> ls -l
lrwxrwxrwx  1 freddy  freddy  17 Jan 22 11:07 Queen -> /opt/mp3/Queen
```

Symbolic links are always very small files, while hard links have the same size as the original file.

The application of symbolic links is widespread. They are often used to save disk space, to make a copy of a file in order to satisfy installation requirements of a new program that expects the file to be in another location, they are used to fix scripts that suddenly have to run in a new environment and can generally save a lot of work. A system admin may decide to move the home directories of the users to a new location, `disk2` for instance, but if he wants everything to work like before, like the `/etc/passwd` file, with a minimum of effort he will create a symlink from `/home` to the new location `/disk2/home`.

File security

Access rights: Linux's first line of defense

The Linux security model is based on the one used on UNIX systems, and is as rigid as the UNIX security model (and sometimes even more), which is already quite robust. On a Linux system, every file is owned by a user and a group user. There is also a third category of users, those that are not the user owner and don't belong to the group owning the file. For each category of users, read, write and execute permissions can be granted or denied.

We already used the *long* option to list files using the **ls `-l`** command, though for other reasons. This command also displays file permissions for these three user categories; they are indicated by the nine characters that follow the first character, which is the file type indicator at the beginning of the file properties line. As seen in the examples below, the first three characters in this series of nine display access rights for the actual user that owns the file. The next three are for the group owner of the file, the last three for other users. The permissions are always in the same order: read, write, execute for the user, the group and the others. Some examples:

```
marise:~> ls -l To_Do
-rw-rw-r--    1 marise  users      5 Jan 15 12:39 To_Do
marise:~> ls -l /bin/ls
-rwxr-xr-x    1 root    root   45948 Aug  9 15:01 /bin/ls*
```

The first file is a regular file (first dash). Users with user name *marise* or users belonging to the group *users* can read and write (change/move/delete) the file, but they can't execute it (second and third dash). All other users are only allowed to read this file, but they can't write or execute it (fourth and fifth dash).

The second example is an executable file, the difference: everybody can run this program, but you need to be *root* to change it.

The Info pages explain how the **ls** command handles display of access rights in detail, see the section *What information is listed*.

For easy use with commands, both access rights or modes and user groups have a code. See the tables below.

Table 3.7. Access mode codes

Code	Meaning
0 or -	The access right that is supposed to be on this place is not granted.
4 or r	read access is granted to the user category defined in this place
2 or w	write permission is granted to the user category defined in this place
1 or x	execute permission is granted to the user category defined in this place

Table 3.8. User group codes

Code	Meaning
u	user permissions
g	group permissions
o	permissions for others

This straight forward scheme is applied very strictly, which allows a high level of security even without network security. Among other functions, the security scheme takes care of user access to programs, it can serve files on a need-to-know basis and protect sensitive data such as home directories and system configuration files.

You should know what your user name is. If you don't, it can be displayed using the **id** command, which also displays the default group you belong to and eventually other groups of which you are a member:

```
tilly:~> id
uid=504(tilly) gid=504(tilly) groups=504(tilly),100(users),2051(org)
```

Your user name is also stored in the environment variable USER:

```
tilly:~> echo $USER
tilly
```

The tools

The chmod command

A normal consequence of applying strict file permissions, and sometimes a nuisance, is that access rights will need to be changed for all kinds of reasons. We use the **chmod**

command to do this, and eventually *to chmod* has become an almost acceptable English verb, meaning the changing of the access mode of a file. The **chmod** command can be used with alphanumeric or numeric options, whatever you like best.

The example below uses alphanumeric options in order to solve a problem that commonly occurs with new users:

```
asim:~> ./hello
bash: ./hello: bad interpreter: Permission denied

asim:~> cat hello
#!/bin/bash
echo "Hello, World"

asim:~> ls -l hello
-rw-rw-r--    1 asim    asim    32 Jan 15 16:29 hello

asim:~> chmod u+x hello

asim:~> ./hello
Hello, World

asim:~> ls -l hello
-rwxrw-r--   1 asim    asim    32 Jan 15 16:29 hello*
```

The **+** and **-** operators are used to grant or deny a given right to a given group. Combinations separated by commas are allowed. The Info and man pages contain useful examples. Here's another one, which makes the file from the previous example a private file to user *asim*:

```
asim:~> chmod u+rwx,go-rwx hello

asim:~> ls -l hello
-rwx------    1 asim    asim    32 Jan 15 16:29 hello*
```

The kind of problem resulting in an error message saying that permission is denied somewhere is usually a problem with access rights in most cases. Also, comments like, “It worked yesterday,” and “When I run this as root it works,” are most likely caused by the wrong file permissions.

When using **chmod** with numeric arguments, the values for each granted access right have to be counted together per group. Thus we get a 3-digit number, which is the symbolic value for the settings **chmod** has to make. The following table lists the most common combinations:

Table 3.9. File protection with chmod

Command	Meaning
chmod *400* file	To protect a file against accidental overwriting.

Command	Meaning
chmod *500* directory	To protect yourself from accidentally removing, renaming or moving files from this directory.
chmod *600* file	A private file only changeable by the user who entered this command.
chmod *644* file	A publicly readable file that can only be changed by the issuing user.
chmod *660* file	Users belonging to your group can change this file, others don't have any access to it at all.
chmod *700* file	Protects a file against any access from other users, while the issuing user still has full access.
chmod *755* directory	For files that should be readable and executable by others, but only changeable by the issuing user.
chmod *775* file	Standard file sharing mode for a group.
chmod *777* file	Everybody can do everything to this file.

If you enter a number with less than three digits as an argument to **chmod**, omitted characters are replaced with zeros starting from the left. There is actually a fourth digit on Linux systems, that precedes the first three and sets special access modes. Everything about these and many more are located in the Info pages.

Logging on to another group

When you type **id** on the command line, you get a list of all the groups that you can possibly belong to, preceded by your user name and ID and the group name and ID that you are currently connected with. However, on many Linux systems you can only be actively logged in to one group at the time. By default, this active or *primary group* is the one that you get assigned from the `/etc/passwd` file. The fourth field of this file holds users' primary group ID, which is looked up in the `/etc/group` file. An example:

```
asim:~> id
uid=501(asim) gid=501(asim) groups=100(users),501(asim),3400(web)

asim:~> grep asim /etc/passwd
asim:x:501:501:Asim El Baraka:/home/asim:/bin/bash

asim:~> grep 501 /etc/group
asim:x:501:
```

The fourth field in the line from `/etc/passwd` contains the value “501”, which represents the group *asim* in the above example. From `/etc/group` we can get the name matching this group ID. When initially connecting to the system, this is the group that *asim* will belong to.

User private group scheme

In order to allow more flexibility, most Linux systems follow the so-called *user private group scheme*, that assigns each user primarily to his or her own group. This group is a group that only contains this particular user, hence the name “private group”. Usually this group has the same name as the user login name, which can be a bit confusing.

Apart from his own private group, user *asim* can also be in the groups *users* and *web*. Because these are secondary groups to this user, he will need to use the **newgrp** to log into any of these groups (use **gpasswd** for setting the group password first). In the example, *asim* needs to create files that are owned by the group *web*.

```
asim:/var/www/html> newgrp web

asim:/var/www/html> id
uid=501(asim) gid=3400(web) groups=100(users),501(asim),3400(web)
```

When *asim* creates new files now, they will be in group ownership of the group *web* instead of being owned by the group *asim*:

```
asim:/var/www/html> touch test

asim:/var/www/html> ls -l test
-rw-rw-r--  1 asim web   0 Jun 10 15:38 test
```

Logging in to a new group prevents you from having to use **chown** (see the section called “Changing user and group ownership”) or calling your system administrator to change ownerships for you.

See the manpage for **newgrp** for more information.

The file mask

When a new file is saved somewhere, it is first subjected to the standard security procedure. Files without permissions don't exist on Linux. The standard file permission is determined by the *mask* for new file creation. The value of this mask can be displayed using the **umask** command:

```
bert:~> umask
0002
```

Instead of adding the symbolic values to each other, as with **chmod**, for calculating the permission on a new file they need to be subtracted from the total possible access rights. In the example above, however, we see 4 values displayed, yet there are only 3 permission categories: *user*, *group* and *other*. The first zero is part of the special file attributes settings, which we will discuss in the section called “Changing user and group ownership” and the section called “SUID and SGID”. It might just as well be that this first

zero is not displayed on your system when entering the **umask** command, and that you only see 3 numbers representing the default file creation mask.

Each UNIX-like system has a system function for creating new files, which is called each time a user uses a program that creates new files, for instance, when downloading a file from the Internet, when saving a new text document and so on. This function creates both new files and new directories. Full read, write and execute permission is granted to everybody when creating a new directory. When creating a new file, this function will grant read and write permissions for everybody, but set execute permissions to none for all user categories. This, before the mask is applied, a directory has permissions *777* or *rwxrwxrwx*, a plain file *666* or *rw-rw-rw-*.

The *umask* value is subtracted from these default permissions after the function has created the new file or directory. Thus, a directory will have permissions of *775* by default, a file *664*, if the mask value is *(0)002*. This is demonstrated in the example below:

```
bert:~> mkdir newdir

bert:~> ls -ld newdir
drwxrwxr-x    2 bert    bert          4096 Feb 28 13:45 newdir/

bert:~> touch newfile

bert:~> ls -l newfile
-rw-rw-r--    1 bert    bert             0 Feb 28 13:52 newfile
```

Files versus directories

A directory gets more permissions by default: it always has the *execute* permission. If it wouldn't have that, it would not be accessible. Try this out by chmodding a directory 644!

If you log in to another group using the **newgrp** command, the mask remains unchanged. Thus, if it is set to *002*, files and directories that you create while being in the new group will also be accessible to the other members of that group; you don't have to use **chmod**.

The *root* user usually has stricter default file creation permissions:

```
[root@estoban root]# umask
022
```

These defaults are set system-wide in the shell resource configuration files, for instance `/etc/bashrc` or `/etc/profile`. You can change them in your own shell configuration file, see Chapter 7, *Home sweet /home* on customizing your shell environment.

Changing user and group ownership

When a file is owned by the wrong user or group, the error can be repaired with the **chown** (change owner) and **chgrp** (change group) commands. Changing file ownership is a frequent system administrative task in environments where files need to be shared in a group. Both commands are very flexible, as you can find out by using the `--help` option.

The **chown** command can be applied to change both user and group ownership of a file, while **chgrp** only changes group ownership. Of course the system will check if the user issuing one of these commands has sufficient permissions on the file(s) she wants to change.

In order to only change the user ownership of a file, use this syntax:

```
chown newuser file
```

If you use a colon after the user name (see the Info pages), group ownership will be changed as well, to the primary group of the user issuing the command. On a Linux system, each user has his own group, so this form can be used to make files private:

```
jacky:~> id
uid=1304(jacky) gid=(1304) groups=1304(jacky),2034(pproject)

jacky:~> ls -l my_report
-rw-rw-r--  1 jacky   project       29387 Jan 15 09:34 my_report

jacky:~> chown jacky: my_report

jacky:~> chmod o-r my_report

jacky:~> ls -l my_report
-rw-rw----  1 jacky   jacky         29387 Jan 15 09:34 my_report
```

If *jacky* would like to share this file, without having to give everybody permission to write it, he can use the **chgrp** command:

```
jacky:~> ls -l report-20020115.xls
-rw-rw---- 1 jacky   jacky   45635 Jan 15 09:35 report-20020115.xls

jacky:~> chgrp project report-20020115.xls

jacky:~> chmod o= report-20020115.xls

jacky:~> ls -l report-20020115.xls
-rw-rw---- 1 jacky   project 45635 Jan 15 09:35 report-20020115.xls
```

This way, users in the group *project* will be able to work on this file. Users not in this group have no business with it at all.

Both **chown** and **chgrp** can be used to change ownership recursively, using the `-R` option. In that case, all underlying files and subdirectories of a given directory will belong to the given user and/or group.

Restrictions

On most systems, the use of the **chown** and **chgrp** commands is restricted for non-privileged users. If you are not the administrator of the system, you can not change user nor group ownerships for security reasons. If the usage of these commands would not be restricted, malicious users could assign ownership of files to other users and/or groups and change behavior of those users' environments and even cause damage to other users' files.

Special modes

For the system admin to not be bothered solving permission problems all the time, special access rights can be given to entire directories, or to separate programs. There are three special modes:

- Sticky bit mode: After execution of a job, the command is kept in the system memory. Originally this was a feature used a lot to save memory: big jobs are loaded into memory only once. But these days memory is inexpensive and there are better techniques to manage it, so it is not used anymore for its optimizing capabilities on single files. When applied to an entire directory, however, the sticky bit has a different meaning. In that case, a user can only change files in this directory when she is the user owner of the file or when the file has appropriate permissions. This feature is used on directories like `/var/tmp`, that have to be accessible for everyone, but where it is not appropriate for users to change or delete each other's data. The sticky bit is indicated by a *t* at the end of the file permission field:

```
mark:~> ls -ld /var/tmp
drwxrwxrwt   19 root     root         8192 Jan 16 10:37 /var/tmp/
```

  The sticky bit is set using the command **chmod *o+t* directory**. The historic origin of the “t” is in UNIX' *save Text access* feature.

- SUID (set user ID) and SGID (set group ID): represented by the character *s* in the user or group permission field. When this mode is set on an executable file, it will run with the user and group permissions on the file instead of with those of the user issuing the command, thus giving access to system resources. We will discuss this further in Chapter 4, *Processes*.
- SGID (set group ID) on a directory: in this special case every file created in the directory will have the same group owner as the directory itself (while normal behavior would be that new files are owned by the users who create them). This way, users don't need to worry about file ownership when sharing directories:

```
mimi:~> ls -ld /opt/docs
drwxrws---  4 root    users          4096 Jul 25 2001 docs/
```

```
mimi:~> ls -l /opt/docs
-rw-rw----  1 mimi    users        345672 Aug 30 2001-Council.doc
```

This is the standard way of sharing files in UNIX.

Existing files are left unchanged!

Files that are being moved to a SGID directory but were created elsewhere keep their original user and group owner. This may be confusing.

Summary

On UNIX, as on Linux, all entities are in some way or another presented to the system as files with the appropriate file properties. Use of (predefined) paths allows the users and the system admin to find, read and manipulate files.

We've made our first steps toward becoming an expert: we discussed the real and the fake structure of the file system, and we know about the Linux file security model, as well as several other security precautions that are taken on every system by default.

The shell is the most important tool for interaction with the system. We learned several shell commands in this chapter, which are listed in the table below.

Table 3.10. New commands in chapter 3: Files and the file system

Command	Meaning
bash	GNU shell program.
cat `file(s)`	Send content of file(s) to standard output.
cd `directory`	Enter `directory`. **cd** is a **bash** built-in command.
chgrp *`newgroup`* `file(s)`	Change the group ownership of `file(s)` to *newgroup*
chmod *`mode`* `file(s)`	Change access permissions on `file(s)`
chown *`newowner`*[:[newgroup]] `file(s)`	Change file owner and group ownership.
cp `sourcefile targetfile`	Copy `sourcefile` to `targetfile`.
df `file`	Reports on used disk space on the partition containing `file`.
echo *`string`*	Display a line of text
export	Part of **bash** that announces variables and their values to the system.
file `filename`	Determine file type of `filename`.
find *`path expression`*	Find files in the file system hierarchy

Command	Meaning
grep ***PATTERN*** **file**	Print lines in `file` containing the search pattern.
head `file`	Send the first part of `file` to standard output
id	Prints real and effective user name and groups.
info ***command***	Read documentation about **command**.
less `file`	View `file` with a powerful viewer.
ln `targetfile linkname`	Make a link with name `linkname` to `targetfile`.
locate ***searchstring***	Print all accessible files matching the search pattern.
ls `file(s)`	Prints directory content.
man ***command***	Format and display online (system) manual pages for **command**.
mkdir `newdir`	Make a new empty directory.
mv `oldfile newfile`	Rename or move `oldfile`.
newgrp ***groupname***	Log in to a new group.
pwd	Print the present or current working directory.
quota	Show disk usage and limits.
rm `file`	Removes files and directories.
rmdir `file`	Removes directories.
tail `file`	Print the last part of `file`.
umask [*value*]	Show or change new file creation mode.
wc `file`	Counts lines, words and characters in `file`.
which ***command***	Shows the full path to **command**.

We also stressed the fact that you should READ THE MAN PAGES. This documentation is your first-aid kit and contains the answers to many questions. The above list contains the basic commands that you will use on a daily basis, but they can do much more than the tasks we've discussed here. Reading the documentation will give you the control you need.

Last but not least, a handy overview of file permissions:

Table 3.11. File permissions

Who\What	r(ead)	w(rite)	(e)x(ecute)
u(ser)	4	2	1
g(roup)	4	2	1

Who\What	r(ead)	w(rite)	(e)x(ecute)
o(ther)	4	2	1