Chapter Three

Routine administration

- Editing files
- Virtual terminals
- Security basics
- Users on the system
- Processes
- Redirection and pipes
- Manipulating text files
- Date and time
- System logs

> "And are you not," said Fook leaning anxiously forward, "a greater analyst than the Googleplex Star Thinker in the Seventh Galaxy of Light and Ingenuity which can calculate the trajectory of every single dust particle throughout a five-week Dangrabad Beta sand blizzard?"

Editing files

As a Linux administrator, the one thing you will do more often than any other is to edit files. This is so basic and fundamental an action that it has to be the first topic taken up in the course after the intro tutorial.

You use a text editor to edit text files – but you probably already know that. There are a great many text editors available on Linux, and from these there will be at least one that you like and fits your needs.

The ultimate in editors is of course `emacs`. This baby can do just about anything, including reading your email for you. It is highly configurable, and comes with it's own programming language built in that allows you to create new functionality. `emacs` come from the GNU stable, and was written by Richard Stallman himself. It is one of two editors that are standard on most Linux systems. But because of the sheer size and complexity of `emacs`, it probably is not the best choice to start your editing experience.

The other common editor is `vi`. It has been expanded and improved far beyond the original, and is now actually known as `vim` (vi improved), but the command `vi` will also work – it ends up at the same place as `vim`.

`vi` dates back to the early days of Unix, and was very advanced for it's time. Nowadays it's user interface can best be described as, um, quaint[1].

So why learn `vi`? Because it will be installed on just about any system you ever come across. Sometimes it may be the only editor on the system so you have to use it. To do that, you have to know how to use it.

`vi` operates in one of two modes – command mode or insert mode. Command mode is where you tell `vi` what you want it to do, insert mode is where to type text that will go into the file.

Why two modes? Because `vi` was developed in the days of stripped down keyboards, with no F-keys and such fancy things. If you want to delete an entire line for example, you use the "`d`" key, which is obviously the same key used to enter a "`d`" in the text.

I could give a several page tutorial on `vi` at this point, but I decided not to because someone else has already written one. It's called `vimtutor` and leads you through all the basic `vi` functions that you need to know.

You should run this tutorial now by entering `vimtutor` on the command line and pressing `Enter`.

List of vi commands

Complete Documentation

The vi editor is a common editor for unix systems in that it makes use of a regular keyboard with an escape key. On the DECstation, the escape key is the F11 key. It therefore works on all unix computers. Complete documentation is available by typing

```
man vi at the unix prompt.
```

Starting an Editing Session

```
vi filename
where filename is the name of the file to be edited.
```

Undo Command

u
: undo the last command.

1 **quaint**: cute or attractive because it's unusual or old-fashioned

Screen Commands

CTL/l
: Reprints current screen.

CTL/L
: Exposes one more line at top of screen.

CTL/E
: Exposes one more line at bottom of screen.

CTL/F
: Pages forward one screen.

CTL/B
: Pages back one screen.

CTL/D
: Pages down half screen.

CTL/U
: Pages up half screen.

Cursor Positioning Commands

j
: Moves cursor down one line, same column.

k
: Moves cursor up one line, same column.

h
: Moves cursor back one character.

l
: Moves cursor forward one character.

RET
: Moves cursor to beginning of next line.

0
: Moves cursor to beginning of current line.

$
: Moves cursor to end of current line.

SPACE
: Moves cursor forward one character.

*n*G
: Moves cursor to beginning of line *n*. Default is last line of file.

0
: Moves the cursor to the first character of the line.

:*n*
: Moves cursor to beginning of line *n*.

b
: Moves the cursor backward to the beginning of the previous word.

e
: Moves the cursor backward to the end of the previous word.

w
: Moves the cursor forward to the next word.

/pattern
: Moves cursor forward to next occurrence of *pattern*.

?*pattern*
: Moves cursor backward to next occurrence of *pattern*.

n
: Repeats last / or ? pattern search.

Text Insertion Commands

a
: Appends text after cursor. Terminated by escape key.

A
: Appends text at the end of the line. Terminated the escape key.

i
: Inserts text before cursor. Terminated by the escape key.

I
: Inserts text at the beginning of the line. Terminated by the escape key.

o
: Opens new line below the current line for text insertion. Terminated by the escape key.

O
: Opens new line above the current line for text insertion. Terminated by the escape key.

DEL
: Overwrites last character during text insertion.

ESC
: Stops text insertion. The escape key on the DECstations is the F11 key.

Text Deletion Commands

x
: Deletes current character.

dd
: Deletes current line.

dw
: Deletes the current word.

d)
: Deletes the rest of the current sentence.

D, d$
: Deletes from cursor to end of line.

P
: Puts back text from the previous delete.

Changing Commands

cw
: Changes characters of current word until stopped with escape key.

c$
: Changes text up to the end of the line.

C, cc
: Changes remaining text on current line until stopped by pressing the escape key.

~
: Changes case of current character.

xp
: Transposes current and following characters.

J
: Joins current line with next line.

s
: Deletes the current character and goes into the insertion mode.

r*x*
: Replaces current character with *x*.

R
: Replaces the following characters until terminated with the escape key.

Cut and Paste Commands

yy
: Puts the current line in a buffer. Does not delete the line from its current position.

p
: Places the line in the buffer after the current position of the cursor.

Appending Files into Current File

:R *filename*
: Inserts the file *filename* where the cursor was before the ``:'' was typed.

Exiting vi

ZZ

Exits vi and saves changes.

:wq

Writes changes to current file and quits edit session.

:q!

Quits edit session (no changes made).

Virtual terminals

Linux allows you to be logged in more than once at the same time. This is how it accomplishes multi-tasking – running more than one program at one time. To do this, press `Alt-F2` at the command prompt. You will see the normal Linux log-in screen. Log in, and `ls` the `/` directory. Press `Alt-F1` and you will see the previous screen you were working on is still there. Use `Alt-F1` and `Alt-F2` to switch between these screens. You can log in on up to 6 different sessions (`Alt-F1` to `Alt-F6`) simultaneously.

Each of these sessions is called a *virtual terminal*[2] (VT) because each acts like a completely separate terminal with keyboard and screen. Of course, there is only one real keyboard and screen, the 6 are simulated by the software, hence the description “virtual”.

You might ask the question, “can more than one person log in with a separate physical terminal?” Yes you can, you plug the other machine into one of the sockets on the back. You have to set it up to do this, and some software is needed, but this software comes standard with Linux.

Security basics

So far, you have learned the essential basics of getting the computer to do something useful. You have learned how to create and delete directories, list files in those directories and change directories. Now it's time to move onto the user security features of Linux.

Users

Linux security is built around the concept of a *user* - a person with an account on the system. The account has a password, and the system checks that the password matches the user name before allowing access. Each user is given a place on the disk drive to store their own private files, and initially the only person that can access those files is the user. The user can later allow other users to have access to his files if he wants – you will often come across files that have to be shared between more than one user. For example a group of programmers working on one project might all need to be able to access all files for the project.

By default, all users must have a password. It is possible to create an account that does not have a password – in other words, anyone can log in as that user. This practice is not recommended, but it does have uses in certain cases – an information directory for the general public would be an example.

Passwords should be difficult to guess. There are many published tips regarding dos and dont s of passwords- here is a quick summary:

- Don't use your name, or the name of a family member, pet or some other significant (to you) person or thing for your password. These are easy to guess.
- Don't use telephone numbers, addresses, significant dates as the basis for your password.
- Don't use words in the dictionary for your password. There are about 80,000 words in the English language, and programs exist to try them all in an effort to crack your password.
- Do use a password that is at leasts 8 characters long, and is a mixture of upper and lower case characters, numbers and punctuation.
- Don't write your password down, and paste it on the side of your monitor – don't laugh, this happens more often than you think.
- Don't give your password to others – this is like giving out the access code on your bank account.

2 **virtual terminal**: a terminal that is not a separate physical terminal, but is set up in software to behave like one

- The best passwords are those that are completely random and have no actual meaning. But these are difficult to remember, so an equally good choice would be the first or last letters of each word of an easy to remember sentence.

A user name can be any name that the administrator chooses to assign to the user, but must be unique. Two users cannot share the same user name. Linux does not impose any special rules about user names – there is nothing to stop you from creating a user with the name 123 for example. But if your name is Bill, then `bill` is a better choice.

There is one particular special user account – `root`. We have already encountered this user name – it is for the system administrator. `root` has certain privileges that no other user has, such as adding and modifying other user accounts. All access controls are switched off for the `root` user – `root` has complete access to every file on the system.

Routinely logging on as the `root` user is a very bad idea! This might be fine for one stand-alone machine for home use, but in all other cases full user accounts should be used. It is very easy to mistype a command and cause damage to the data on your system if `root` is the current user. Rather create a regular account for everyday work, and switch to root mode when you really need it – there is a command to temporarily do this, which we will cover later.

Thus far we have been operating as the `root` user (and breaking an important rule in doing so). We had to do this because there were a few things to learn about before getting into user security. From now on you should always log into the system using the normal account we are about to create.

useradd

The command to create a new user is `useradd`. Most systems also have the equivalent command `adduser` – they do the same thing. The format is:

```
useradd [username]
```

You must be logged in as `root` to use this command. Enter it now, entering your own choice of user for `[username]`. The system will create this user account, and create a directory for the user to store their personal files.

This directory has the same name as the user name, and is in the **/home** directory. These directories are collectively called *home directories*[3]. It does this by copying the existing **/etc/skel** directory to **/home** and renaming it to the same name as the new user. This is more than just a convenient way to create a new directory – any files that are in **/etc/skel** are also copied over. Certain programs look in a user's home directory for configuration files specific to that user, so **/etc/skel** (skel stands for skeleton) is a good place to put default versions of these files, which the user can then customize to his liking. Enter:

```
ls /home
```

to see this new directory.

The system provides a convenient shortcut for the name of the home directory - ~. This character is called the *tilde*[4]. If the system sees a tilde when it expects a directory name, it replaces ~ with the full name of the home directory of the current user. If user `alan` is logged in, ~ expands to /**home/alan**; if user `bill` is logged in, ~ expands to **/home/bill**.

This new user account does not have a password yet. To enter the initial password, root can either do it immediately, or the user can do it the first time they log in. In both cases, the relevant command is `passwd`

passwd

This command updates the password for a user account. The format is:

```
passwd <username>
```

The user name portion is optional, and only root can do this. The system will ask for the new password for `<username>` to be entered twice – this prevents typing errors when entering the password. Note that the characters typed are not printed on the screen – a security feature.

Users can also enter their initial password themselves. Log in as the new user, and note that the system does not ask for a password (because there isn't one yet). Enter the command

```
passwd
```

3 **home directory**: a directory in /home set aside for a specific user's private use
4 **tilde**: the ~ character.

and enter the new password as above. Normally this command first asks you to enter the current password, then has you enter the new password twice. This is another security feature and prevents other users from changing a user's password if the user steps away from the terminal for a short while without logging out. But in this case there is no current password, so the first step is omitted.

su

From now on you should always log in as a regular user. But what if you need to do something that only `root` can do? You can't do it if logged in as a regular user.

You do this with the `su` command, short for *superuser* (another name for `root`). If you use the `–` or `-l` option, the system will duplicate the entire login process all over again. Without these options, you are only asked for a password. `su` also needs to know which user you wish to change to, supply this as an argument on the command line. If no user is supplied, `root` is assumed. Use the command like this:

```
su – root
```

which is the same as

```
su -
```

You can `su` to any other user (if you have the password) like this

```
su – [username]
```

And finally, `root` can `su` to any user at all on the system without giving a password.

userdel

This is the command to delete a user. The format is:

```
userdel [username]
```

Only `root` can do this. After using `userdel`, the user's home directories are not deleted – they still exist. This is because there may well be files is those directories that should be preserved. To delete the user's home directories as well, use the `-r` switch like this:

```
userdel -r [username]
```

Groups

In practice, using only user names is very limited. Users often have to share files between themselves, so it makes sense to be able to assign permissions to entire *groups* of people. Linux uses the simplest possible solutionfor this:

A separate group account is created for each desired group, and a list of users that belong to that group is maintained in the file **/etc/group**. At any one time while logged in, a user is recorded as belonging to one of these groups, and can change their current group at any time.

It's important to understand how simple this scheme is. The one thing it does not do is keep a full list of each user's group in memory and give the user all access permissions for all groups all the time.

a way and certain groups of users need to do operations on the system that other groups of users can't do. For example, in a college each lecturer and student has a login account. Lecturers may run programs that update student exam marks, but students may not do this. It is possible to assign each lecturer and each student the correct permissions one by one, but this is prone to error and a real pain to keep updated.

The solution is to create two *groups* of users – one for lecturers and and one for students. Assign each user to one or other of the group, and assign *group permissions* to the files and programs having to do with exam marks. Administering such a system is much easier – you simply tell the system which group a user belongs to.

groupadd

This command adds a new group to the system. It does not add any users to the group – that is done separately. The format is:

```
groupadd [groupname]
```

groupdel

This command deletes a group. The format is:

```
groupdel [groupname]
```

usermod

This command allows you to change a user's settings, especially the groups the user belongs to. The format to do this is:

```
usermod -G [group][,group]... [username]
```

List the names of the groups the user is to belong to, separated by commas without any spaces between group names.

gpasswd

This command changes the password for a group. It operates similarly to the `passwd` command, you supply the group name on the command line:

```
gpasswd [groupname]
```

The system will issue all the necessary prompts for the password to be correctly changed.

groups

This command lists the groups a user is in. To use, enter the command followed by a user name. Here is a listing from my machine for user `root` and user `alan`:

```
[alan@notebook alan]$ groups root
root : root bin daemon sys adm disk wheel
[alan@notebook alan]$ groups alan
alan : alan
```

If you don't supply a user name, the command uses the currently logged in user.

Default groups

When a new user is added to the system, most Linux versions will also create a new group with the same name as the user. If the user is `bill`, a group named `bill` is also created. User `bill` is automatically a member of group `bill`.

This can be useful if a user wants to give another user the same access to his files that he has. If Bill has an assistant called Joe, whom he trusts completely, he could make Joe a member of the `bill` group. This may be easier to set up in some cases than a full-blown group. We will cover the topic of access rights to files shortly, and then all will become clear.

User and group IDs

Each user and group name on the system has a unique number ID assigned to it. These are called UID and GID. The computer internally uses only these IDs, the user and group name is there for the benefit of humans (we are more comfortable with user names than user numbers). When you log in, the computer finds your login name, reads the ID that goes with the name, and thereafter uses the ID. You can change the login name if you want.

To change a user's login name, use `usermod`:

```
usermod -l [newusername] [oldusername]
```

For this to work, the user must not be currently logged into the system, and no programs can be running that the user started.

There is a similar command for changing group names:

```
groupmod -n [newgroupname] [oldgroupname]
```

Note that the option is different – `usermod` uses a `-l` option, while `groupmod` uses a `-n` option.

IDs between 0 and 99 are normally reserved for system use. Regular users should be numbered from 100 and higher. Most distributions start numbering user IDs even higher – Red Hat starts at 500. It's not impossible for the system to want to create more than 100 user IDs

File ownership

All files and directories are marked as having an *owner*[5]. This is usually the user that created the file, but `root` can change this. Files and directories also have *group owners*[6], meaning the group that owns the file. By default the group owner is whichever group that the owner belongs to. If `bill` creates a new file, the owner is (user) `bill` and the group owner is (group) `bill`.

If you were wondering what the reason for default groups is while reading that section earlier – now you know. It simplifies the process of group ownership for new files.

Permissions

We briefly touched on permissions above without going into any detail. The general idea of permissions is obvious – users may or may not do certain things on the system, depending on whether they have the necessary permission to do so or not. Linux uses three permissions for three classes of users. The permission types are:

1. read A user with read permission may open and look inside a file
2. writeWrite permission allows the user to modify the file
3. execute If the file is a program, the user may execute the program. For directories, the user may open the directory (`cd` into it)

The classes of users are:

1. owner the user owner of the file
2. group the group owner of the file
3. otherall other users, who are neither the owner of the file, nor members of the group that owns the file.

Now that we have covered files, directories, users, groups and permissions, we can introduce a useful option to the `ls` command. The option is `-l` (for long). Enter the following:

```
ls -al /
```

The output from this command on my machine looks like this (details may differ on your machine but the format is the same):

```
drwxr-xr-x  19 root  root   4096 Mar 12 08:31 .
drwxr-xr-x  19 root  root   4096 Mar 12 08:31 ..
-rw-r--r--   1 root  root      0 Mar 12 08:31 .autofsck
drwxr-xr-x   2 root  root   4096 Mar 10 16:15 bin
drwxr-xr-x   4 root  root   1024 Mar 10 16:02 boot
drwxr-xr-x  21 root  root 118784 Mar 12 08:32 dev
drwxr-xr-x  67 root  root   8192 Mar 12 09:49 etc
-rw-r--r--   1 root  root     51 Mar 10 18:03 .fonts.cache-1
drwxr-xr-x   5 root  root   4096 Mar 12 09:49 home
drwxr-xr-x   2 root  root   4096 Oct  7 13:16 initrd
drwxr-xr-x   9 root  root   4096 Mar 12 09:40 lib
drwx------   2 root  root  16384 Mar 10 17:54 lost+found
drwxr-xr-x   2 root  root   4096 Sep  8  2003 misc
drwxr-xr-x   3 root  root   4096 Mar 10 17:57 mnt
drwxr-xr-x   2 root  root   4096 Oct  7 13:16 opt
dr-xr-xr-x  76 root  root      0 Mar 12 10:31 proc
drwxr-x---  23 root  root   4096 Mar 12 09:54 root
drwxr-xr-x   2 root  root   8192 Mar 10 19:42 sbin
drwxrwxrwt  23 root  root   4096 Mar 12 09:37 tmp
drwxr-xr-x  15 root  root   4096 Mar 10 16:00 usr
drwxr-xr-x  22 root  root   4096 Mar 10 19:40 var
```

The listing is divided up into 7 columns:

1. permissions for the file

5 **owner**: the user in charge of a file or directory. The owner is often the creator of the file, but this is not required.

6 **group owner**: a group that has special rights to a file or directory. Similar to owner, but applies to groups.

2. a number (which we will cover later but basically it is the number of existing copies of the file)
3. file owner
4. group owner
5. the size of the file on disk
6. date and time the file was last modified (omitting the year)
7. the file name

The permissions column consists of 10 entries – each usually a letter. The first letter indicates if the line refers to a file or a directory. Directories are indicated with a "d", regular files with a "-".

The 2nd, 3rd and 4th letters give the permissions for the owner of the file, in the order read, write, execute. If the action is allowed, the relevant r, w or x (for execute) is printed. If the permission is not allowed, a dash is printed.

The 5th, 6th and 7th letters give permission for the group owner of the file, listed the same way as user permissions.

The 8th, 9th and 10th letters give the permissions for all other users on the system, listed the same way as user permissions.

Lets examine closely two entries from the above listing (numbers have been added above the permission column to make reading easier):

```
1234567890
-rw-r--r--    1 root   root      0 Mar 12 08:31 .autofsck
drwxr-xr-x    2 root   root   4096 Mar 10 16:15 bin
```

The first line refers to a hidden file called **.autofsck**. The first letter is a dash, so it is not a directory. Letters 2-4 read "`rw-`" so the owner (`root`) may read and write to the file. Letters 5-7 read "`r--`", so other users who are members of group `root` may read the file, but not write to it. Letters 8-0 also read "`r--`", so for this file all other users may also read the file but not write to it. This file contains data and is not a program, so the executable columns (4, 7 and 10) all contain dashes. You could put x's in these columns, but this doesn't make sense and the system will probably give you an error if you try and execute the file, because it is not a program.

The second line refers to a directory called **bin**, the owner and group owner are `root`. `root` may read from, write to, and enter the directory, members of the group owner (group) `root` may read from and enter the directory, but not write to it, and all other users have the same permission as the group owners.

The "sticky" bit

One of the lines in the above sample from my machine shows an unusual permission. The line is:

```
drwxrwxrwt   23 root   root   4096 Mar 12 09:37 tmp
```

The tenth column under permissions lists a "t", meaning that the **/tmp** directory is marked *sticky*[7]. Sticky is typical Unix slang, and for a directory it means that for files in that directory, only the file owner or root may delete or rename the files. This is useful for directories like **/tmp** (which are accessible to all users) where programs write temporary files that will be deleted later – you wouldn't want some other user to delete your temporary files that your program is still using!

These days you will seldom see the sticky bit applied to a file. It dates back to the early days of Unix, and was intended to set important system programs to remina in memory so they could be available quickly when needed. These days memory management works very differently to back then and Linux has always totally ignored the sticky bit on files. Some other (non-Linux) systems use the sticky bit for their own purposes, and some systems only allow `root` to set the sticky bit.

The sticky bit is described here for completeness, it is not all that important, but setting the sticky bit for a directory may come up in your exam.

7 **sticky**: for files – an obsolete attribute having to do with swap space memory. For directories – only the owner of a file in the directory may delete or rename the file.

setuid and setgid bits

Jut as files have user and group owners, programs running in the computer's memory also have owners, called the *process owner*[8]. This is normally the user that started the program, as opposed to the owner of the file that contains the program on disk. But this can be changed. You may occasionally see the letter “s” in the executable permission column (columns 4 & 7). If present, these mean that when the program executes, the system will set the process owner to the owner of the disk file, not the user that started the program. There is a setting for the process user owner – *setuid*, and a setting for the process group owner – *setgid*.

Changing file permissions with chmod

Everything we have covered so far regarding permissions is very interesting, but we still don't have a way to change permissions on a file or directory. The command to do this is `chmod`. This command can accomplish the same thing in two different ways.

Text method

The first method is the text method. You enter three characters, specifying who the new permissions apply to, how to apply them, and which permissions to apply. The first character must be one of the following:

u
: the owner of the file name

g
: the group owner of the file

o
: all other users

a
: all users, i.e. u, g and o combined

The second character must be one of the following

+
: adds the new permission(s)

-
: removes the permission(s)

=
: set these permissions and remove all others

The third character must be one of the following

r
: read permission

w
: write permission

x
: execute permission

X
: execute permission, but only if the entry is for a directory or a file that already has an execute permission set

s
: set user or group id on execution

t
: sticky bit

u
: the permissions already assigned to the file's owner

8 **process owner**: Usually the ID of the owner that started a process. Sometimes it is the ID of the owner of the program's disk file.

g
: the permissions already assigned to the file's group owner

o
: the permissions already assigned to other users

For the most basic method of using `chmod`, you supply one character from each one of the above three groups. To give the owner of the file write permission on the file, enter

```
chmod  u+w  [file name]
```

To remove read permission to the file from all other users, enter

```
chmod  o-r  [file name]
```

To give the file's group owner only read and write permissions on a file, enter

To give the file's group owner the same permissions as the file owner, enter

```
chmod  g+u  [file name]
chmod  g=rw  [file name]
```

To give read permission and remove execute permission for the file to all other users, enter

```
chmod  o+r,o-w  [file name]
```

To give read permission to every user on the system and remove execute permission for all other users, enter

```
chmod  a+r,o-x  [file name]
```

The last three examples show how operations can be combined. To assign more than one permission type at a time, simply list them all after the operation sign (+, - or =). For other combinations, list them all separated by commas.

Take note of the difference between “o” and “a” in the first group of characters. “o” is users who are not the owner or in the owner group – columns 8 to 10 in an `ls -l` listing. “a”is every user on the system, columns 2 to 10 in an `ls -l` listing.

The third group of characters has an interesting option – X. A common mistake that can occur is to assign execute permissions to a file where this does not apply – it is not a program. X helps prevent this and will only apply the permission if the file name is a directory (where execute permission allows users to enter the directory) or one of the execute permissions is already set. The “s” and “t” permissions are seldom used, but are included here for completeness.

Numeric method

Using this method, permissions are given numerical values and added up for each different class of user.

- set user ID has the value 4
- set group ID has the value 2
- sticky bit has the value 1

Regular permissions have the following values:

- read permission has the value 4
- write permission has the value 2
- execute permission has the value 1

This form of the command is used as follows:

```
chmod [mode] [file name]
```

where [mode] is four digits. The first digit is the values of set user ID, set group ID and sticky bit. The second digit is the values of regular permissions for the file owner, the third is the values of regular permissions for the group owner, the fourth digit is the value of regular permissions for all other users. If any of these four digits are omitted, they are assumed to be leading zeros (added on the left).

To assign all permissions to the file owner, and read and write permissions only to group owner and other users, enter

```
chmod 0766 [file name]
```

For a data file that may be read by everyone in the owner's group (and no-one else), but only modified by the owner, you could use this command:

```
chmod 0640 [file name]
```

No execute permissions were set with this example.

The user ID, group ID and sticky bit options are seldom used, so the last example is often entered as:

```
chmod 640 [file name]
```

The first digit is omitted, so it is assumed to be zero.

When using `chmod`, you can specify multiple file names by listing them all separated by spaces. File globbing characters are also allowed.

The numeric method is often called the *octal* method, because it uses the values 1, 2 and 4 like a number system that has only 8 digits.

Students often find the text method easier to grasp at first, but the numeric method is the more common. But as soon as the student *groks*[9] the numeric method, it usually becomes second nature, then the syntax of the text method becomes difficult to remember!

Setting default permissions with umask

Setting permissions is all very well and fine – but what are the default permissions? The answer is – whatever you have set them to be. They are set with the `umask` command. Using this command is a little non-intuitive at first, because you specify the permissions that will *not* be set. So it operates as a mask, blanking out something that you don't want, hence the name.

A common default setting is 022. The order and value of the permissions is the same as for `chmod`. The group owner value is 2 – this is the group owner write permission. The other user value is also 2. What this means is that those permissions will *not* be allowed, all others will.

So finally we get to see what the default permissions are if we use a umask of 022 – the file owner may read form, write to and execute the file. The group owner and other users may read from and execute the file, but may not write to it.

If you are a super-secret-paranoid type of individual, and wanted to remove all permissions on your new files, except that other members of your group could at least read them, you would use the mask 037 – the write and execute values total 3, and read, write and execute values total 7.

Note that `umask` sets only the default permissions for new files. If you want a different set of permissions for a specific file, you have to change that file with `chmod`.

`umask` applies only to your own files – each user can start a session at the terminal with a different mask setting. Later in the course we will cover the method to use to automatically set this value each time you log in.

Using the `umask` command is easy – supply the mask value as an argument on the command line. With no arguments, umask prints the current mask, with the -S option, it prints the current mask in the chmod text style. Here are examples of all these usages:

```
[root@notebook /]# umask 022
[root@notebook /]# umask
0022
[root@notebook /]# umask -S
u=rwx,g=rx,o=rx
```

Changing a file's owner

The command that does this is `chown`. In it's most basic form, it is used liked this:

```
chown [newowner] [file name]
```

This will assign a new owner to the file. Multiple files can be specified like this:

```
chown [newowner] [file name] [file name]...
```

in this case the owner is changed for each listed file. Group ownership can be changed at the same time, like this:

9 **grok**: yet more Unix slang. It means “to understand” or “to grasp the meaning” of something.

```
chown [newowner]:[groupowner] [file name]
```

This changes the owner and the group owner at the same time. A dot can be used instead of a colon, but there must be no spaces on either side of the colon or dot.

The colon or dot can be entered, but group owner can be left off, like this

```
chown [newowner]: [file name]
```

this is the same as not entering a colon at all.

Lastly, the colon or dot can be given with a group name, but without a user name, like this

```
chown :[groupname] [file name]
```

this will change only the group owner and leave the user owner unchanged.

User and group IDs can be entered instead of user and group names.

Changing a file's group owner

The command that does this is `chgrp`. The format is

```
chgrp [groupname] [file name] [file name]...
```

This changes the group ownership of each listed file to `[groupname]`. This does exactly the same thing as the last example for `chown`.

User and password information

So where does the system store this password information? You might think perhaps they are stored in some deep dark hole that no-one can ever get to. Well, you'd be wrong. They are stored in open view to the entire world, in two files, both in the **/etc** directory. User information is in the file **/etc/passwd.** `less` this file to see what it looks like. A sample of the **passwd** file on my machine looks like:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
alan:x:500:500:Alan McKinnon:/home/alan:/bin/bash
temp:x:501:501::/home/temp:/bin/bash
```

This shows four different users: `root`, `bin`, `alan` and `temp`. `alan` is my everyday use account, `temp` is just that – a temporary account made during testing, and `bin` is a system account. The **passwd** file consists of one entry per line, and each entry is divided up into 7 fields, separated by colons:

1. username
2. password
3. user ID
4. group ID
5. extra information – intended to be the user's full name, etc. This field is seldom used in practice.
6. user's home directory
7. the shell program that will run after login. The shell is the program that displays the system prompt and accepts commands at the keyboard. There are many shell programs available, but **/bin/bash** is by far the most popular. You should leave this field as is, at least for now.

The various user and group commands we have covered so far all update these files. But there is no reason why you can't edit them directly yourself if you know what to do. After some practice, this is often easier and quicker than entering the command – especially if you just want to update one user.

So where is the password? Each entry above just shows an “x” in the password field. In the early days, the password was stored in the **passwd** file (hence the name), but this is not secure.

The actual user password itself is never stored anywhere on the system. Instead, the password is encrypted – it goes through a complex mathematical process that gives out a long string of letters and this string is what is stored. This process is one-way – it is easy to generate the long string, but if you have the long string, it is impossible to figure out what was the original password. Well, maybe it's not impossible, but so far no-one has figured out a way to do it and published the results. And the process is guaranteed to generate different long strings for different passwords.

To validate a password, the system passes the entered password through the same encryption process to generate a new long string, then compares that string with what is stored in the password field. If the two long strings match, the entered password is the same as the original password.

This looks fine, and the passwords seem to be safe, but there is a loophole. Any user and program on the system must be able to read the **passwd** file (but not write to it – only `root` can do that) to validate users, so anyone can get their hands on the encrypted strings. Programs exist that check every word in the dictionary against the encrypted password strings in the **passwd** file. If you have several hundred users on the system, it's a sure bet that at least a few of them used dictionary words as their password. Someone wanting to break system security could easily find those user's passwords. Adding a number to the start and end of a word doesn't help either – those cracking programs are smart enough to try that trick as well.

The solution is to remove the password out of the **passwd** file and put it some place safe. This is indicated with the "*" field in **passwd**, and the encrypted password is stored in **/etc/shadow**. This file has a very strict set of permissions – `root` may read it, everyone else has no permissions on the file at all. In making this change, the designers also made the encryption process even stronger and more difficult to crack[10], and added extra useful features to the system. These features are covered in the next section, password policies.

Password policies

Password policies implement sensible rules regarding passwords. There is nothing we can do about users who choose silly passwords, or give their password to the milk man, but we can enforce rules to make users change their passwords often, and we can also cause accounts to become expired or disabled.

Here is a sample of the `shadow` file on my system for the same 4 users as above:

```
root:$1$cWz1uM3k$saguncyHLwzgLlG8Ip3SR/:12487:0:99999:7:::
bin:*:12487:0:99999:7:::
alan:$1$UazZZkJZ$46qXv0v1b5Jbqyg1AuewR1:12487:0:99999:7:::
temp:!!:12488:0:99999:7:::
```

Again, there is one entry per line and each line consists of 9 fields, separated with ":"

1. user name – same as first entry in the **passwd** file
2. encrypted password
3. number of days since 1 Jan 1970 when password was last changed
4. number of days that must pass before password can be changed again
5. number of days after which password must be changed
6. number of days before password expiry that user is warned
7. number of days after password expiry that account will be disabled
8. number of days since Jan 1, 1970 that account will be disabled
9. reserved field

Password policies are entered by filling in fields 4, 5, 6 and 7. The system updates everything else. Point 4 requires some explanation – without it, users can bypass the rule of changing passwords every x number of days by changing their password to something temporary and immediately changing it back to their favourite password.

If a password has expired and that user logs in, the system will inform the user that the password has expired, ask for the old password, and once verified immediately ask for the new password. There is no way to bypass this, thus forcing a password change. The old and new password must of course be different. The system knows when each password will expire, so shortly before it does, the system will start reminding the user that the password is due to expire. The number of days before expiry when this starts is specified in field 6. If an account is not used for a period after the password expires, the system can be set to disable the account, and only root can re-enable it again. This grace period is set in field 7. Field 8 contains the date when the account will be disabled if the password has expired. Field 9 is reserved and is not for general use.

Most Linux distributions do not automatically set password policies – this would cause havoc if the person doing the installation was not aware of it. Instead, field 5 is set to a very high value 99999 days (more than 270 years), field 6 is set to 7 (a reasonable value, but will never be used) and fields 7-9 are left blank.

10 For those who are interested and know about these things, the encryption process used is MD5

The entries for `bin` and `temp` also need explanation. `bin` has a "*" in the password field – this is a system account used only by the system and no-one will ever log in at the terminal with this user name. Account `temp` has been disabled – indicated with the first letter of the password field being "!"

Password policies are implemented with several commands and options.

```
usermod -e [expiredate] [username]
```

sets the date when a password will expire. `[expiredate]` must be in the form YYYY-MM-DD

```
usermod -f [inactivedays] [username]
```

sets field 7 in the **shadow** file

```
usermod -L [username]
```

disables (locks) an account

```
usermod -U [username]
```

enables (unlocks) an account

The `useradd` command also has the identical `-e` and `-f` options for use when creating new accounts.

The master command that does everything is `chage`. Generally, only `root` can use this command. The format is:

```
chage <-m mindays> <-M maxdays> <-d lastday> <-I inactive>
      <-E expiredate> <-W warndays> [username]
```

`[username]` is required, but if none of the options is specified, the command acts interactively, displaying each current option and asking the user to update it. If the option doesn't need updating, simply press enter to accept the current value.

`-m mindays`
: updates field 4

`-M maxdays`
: updates field 5

`-d lastday`
: updates field 3. The system accepts date input in the form YYYY-MM-DD.

`-I inactive`
: updates field 7

`-E expiredate`
: updates field 8. The system accepts date input in the form YYYY-MM-DD.

`-W warndays`
: updates field 6

There is one option to `chage` that a regular user can use. This is:

```
chage -l [username]
```

which displays when the user's password is due to expire.

Group files

There are two files that contain information about groups. They are similar to the **/etc/passwd** and **/etc/shadow** files; these files are **/etc/group** and **/etc/gshadow** respectively.

The list of defined groups on the system is kept in the file **/etc/groups**. This file does the same thing for groups as **/etc/passwd** does for users. A sample of this file from my machine looks as follows:

```
root:x:0:root
bin:x:1:root,bin,daemon
alan:x:500:
```

The file consists of 4 fields, separated by colons. These fields are:

1. group name
2. encrypted group password
3. comma separated list of group administrators

4. comma-separated list of users that belong to this group

The **/etc/group** file suffers from the same security weakness as /**etc/passwd**, so it too has a shadow file - this is called **/etc/gshadow**. On my system, the file looks like this:

```
root:::root
bin:::root,bin,daemon
alan:!!::
```

This is a 4-field file, and the fields have the same format as **/etc/group**.

Converting between password formats

You may find yourself in a position where a system is using the old-style user and group files, and has not converted over to the better shadow format. Although you could do this manually, re-creating each user and group at the command line, there are four commands that automate the process. There are two each for users and groups, one to convert to shadow format, one to convert back to the old format.

These commands are:

`pwconv`
: creates or updates **/etc/shadow**

`pwunconv`
: converts back to **/etc/passwd** only format

`grpconv`
: creates or updates **/etc/gshadow**

`grpunconv`
: converts back to **/etc/group** format

None of these commands take any arguments, as there is no optional behaviour. The location of all files is known beforehand, and the command simply has to get on with doing it's job.
If the `pwconv` and `grpconv` commands find an existing shadow file, they will add to the file – not replace it. Necessary encrypted password are added to the shadow file, the password field in the main file is replaced with an “x”, and system defaults for the password policies are used for new entries to the shadow file.

The `pwunconv` and `grpunconv` commands move information out of the shadow file into the main file, and the shadow file is then deleted. Information that is only stored in the shadow file – like some password ageing fields – is lost in the process. There is nothing that can be done about this – the process is a conversion to a format with less detail.

Some odd entries in the files can cause these programs to fail in strange ways or go round and round in circles forever. There are two commands that check the files and interactively allow oddities to be corrected. These commands are:

`pwck`
: checks **/etc/passwd** and **/etc/shadow**

`grpck`
: checks **/etc/group** and **/etc/gshadow**

Without options, these commands automatically use the normal user, group and shadow files. You can tell them to use other files if you wish:

```
pwck [userfile] [shadowfile]
grpck [groupfile] [shadowfile]
```

This might be useful if you wanted to avoid making any disastrous changes to your user and group files. To simply check if the files are OK and not make any changes to them, use the `-r` option with either command – this opens the files read-only, the user is not offered the chance to correct any oddities, and the files are not altered at all.

Users on the system

You might want to know who is currently logged in and using the system. Several commands let you do this.

who

who lists the users logged in on the system. An output from running who on my system shows:

```
[alan@notebook alan]$ who
root     tty1         Mar 14 12:33
alan     tty2         Mar 14 12:33
alan     :0           Mar 12 19:14
```

user root is logged on virtual terminal 1, and user alan is logged in on virtual terminal 2. User alan is also logged in somewhere called ":0" - this is the graphical user interface that I am currently using to write this manual. The time each user logged in is also listed.

w

Sometimes who doesn't give enough information. You might want to know what each user is currently doing. w gives this information. Running w on my system shows:

```
 12:37:09  up 1 day, 17:27, 3 users, load: 0.01, 0.10, 0.09
USER     TTY      FROM     LOGIN@   IDLE   JCPU   PCPU   WHAT
root     tty1     -        12:33pm  3:49   0.10s  0.10s  top
alan     tty2     -        12:33pm  3:39   0.06s  0.06s  -bash
alan     :0       -        Fri 7pm  ?      0.00s  0.13s  /bin/sh
```

The first line tells me the current time, how long the system has been running since last reboot or restart (like most Linux systems, I let my machine run all the time, seldom switching it off),there are 3 users logged in, and the average load the system experienced in the last 1, 5 and 15 minutes (loads less than 1 are ideal).

For each user, I can see where they logged in from using the FROM column. In this example, the entries are all blank because all three users are logged in on this machine using virtual terminals. If other users were logged in over the network, their network address would show up here. The 5th to 7th columns show statistics regarding time usage of the system for each user, and the last column shows the command that each is currently running.

whoami

Forgot who you logged in as? whoami will remind you. It displays the name of the current user:

```
[alan@notebook alan]$ whoami
alan
```

This doesn't look like much use – why ask the system who you logged in as when you a) probably remember it anyway and b) can see it right there on the prompt.?

But a program doesn't know that, so it could use whoami to find it out.

id

Here's another useful little program. It shows the UserID, GroupID, and GroupID {*** FIXME – this is real and effective groups. Need to define these somewhere so it doesn't cause confusion} of all groups the current user belongs to. Here is what I get on my machine while logged in as alan:

```
[alan@notebook /]$ id
uid=500(alan) gid=500(alan) groups=500(alan)
```

And when I log in as root I get the following:

```
[root@notebook /]# id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),6(disk)
```

Processes

By now you know that Linux is a multi-task system – it runs more than one program (correctly called a *process*[11]) at a time[12]. So, what programs are running on the system? ps is the command that tells you this.

11 process: a program that is actually running in the computer's memory

12 Well, actually it only runs one program at any specific time but it jumps from one program to another so fast it looks like they all run at the same time

ps

```
alan@notebook alan]$ ps
  PID TTY          TIME CMD
14868 pts/1    00:00:00 bash
15064 pts/1    00:00:00 ps
```

Without any options, `ps` displays the processes for the current user only. We can see that this user is running `bash` – the standard shell displayed at the console. He is also running `ps` – the very command that displays itself.

The first column gives a unique process ID (or PID) for each process – this identifies the process and is used by commands that operates on running processes. The second column indicates which console the command was started from – in this case pts/1 which is the first virtual console. The third column shows how much cpu time the process has used so far – this is not the same thing as the elapsed time since the command was started. And the last column is the command that is being run.

To see processes running on all terminals, use the `-a` option

Linux runs many processes by itself in the background. These include programs that allow communication across the network, programs that log significant occurrences in the system logs, and more. To see these, use the `-x` option.

`ps` is often run with both the `-a` and `-x` options, like this:

```
[alan@notebook alan]$ ps -ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init [5]
    2 ?        SW     0:00 [keventd]
    3 ?        SW     0:00 [kapmd]
    4 ?        SWN    0:00 [ksoftirqd/0]
    6 ?        SW     0:00 [bdflush]
    5 ?        SW     0:06 [kswapd]
```

This command on my machine actually displayed much more information than this – several screens full actually. I deleted most of it, the important thing to note is the second column – the "?" indicates that the process is not being run from a terminal, the system itself is running it in the background. If you run this command on your system, use the `Shift-PgUp` and `Shift-PgDn` keys to move backwards through the listing.

There are many more options to the `ps` command, allowing you to specify exactly what type of processes to list, how to sort them, what information to display for each and in what format to display them – the options section of the man page is many screens long. Consult this page for further information if you are interested.

kill

Most of the commands we have looked at so far do their required action and exit immediately. Some commands carry on for longer than this, and some are interactive – meaning you keep giving them instructions (perhaps from a menu) and they keep doing something. Interactive programs usually have some kind of a menu option to get them to exit.

If you really can't figure out how to get a program to stop, you can normally force it to do so by pressing `Ctrl-C`. This makes the system give the command a standard signal to end itself.

But computers, being machines, don't always behave. What do you do if even `Ctrl-C` doesn't work? You use the `kill` command. Use `ps` to find the PID of the offending process, and issue the `kill` command like this:

```
kill [pid]
```

This should do the trick. Some systems will allow you to specify the process name instead of the PID.

The Linux system is designed so that the kernel can send a number of signals to running processes, these signals are given signal numbers, and names starting with "SIG". Two useful signals are SIGTERM and SIGKILL. The signal number for SIGTERM is 15, and the signal number for SIGKILL is 9. SIGTERM is the normal signal to stop a process, as it causes the process to shut down properly – it is common for processes to store information in memory and then write that information to disk when they shut down. The SIGTERM signal allows this to happen, sort of like the manager at a night club politely asking you to leave when you had too much. SIGKILL on the other hand, is more like the bouncer – the process is forced to end whether it wants to or not. This can result in lost data, but sometimes it is the only way to stop a process that has gone berserk.

There are several options for the `kill` command:

-s
: specifies the signal to send. This can be the signal name or number.

-p
: doesn't actually send the signal, just prints the PID. This can be useful in checking that the command will do what you want it to do before running it for real.

-l
: prints the full list of signal names and numbers – 63 in total. Here is the full list:

```
[alan@notebook tmp]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Process priority

It should be obvious that some processes are more important than other processes. If you are running a very intensive calculation – some engineering problem for example – and the time comes due for `updatedb` to do it's daily thing, you probably wouldn't want the engineering program to slow down so that `updatedb` can run. You set this up with process priorities. The cpu is jumping from one process to another many times a second giving each a chance to run, and it uses priorities to determine how large a fraction of cpu time each process actually gets.

A process' priority is a number between -20 and 19, with 0 being the default. But it works the wrong way round to what you expect. 19 is the lowest possible priority – such a process will only run when nothing else in the system wants to, and -20 is the highest. Only `root` can give a process a negative priority, and regular users are free to give their own processes any priority in the range 0 to 19.

When used properly, programs that don't need huge amounts of cpu time and have their priorities set low, can be seen to be *nice* to other programs that do need lots of cpu time. In fact they are so nice that the command to set priorities is called `nice`, and priorities are commonly called *niceness* or *nice value*.

nice

When using `nice`, specify the command you want to run on the command line, like this

```
nice updatedb
```

This will run the `updatedb` command with the default nice value of 10. The full syntax of the command is:

```
nice [option] [command [arg]...]
```

Where `[arg]` is the normal arguments for `[command]`. The option is:

`-n [adj]`
: sets the nice value to `[adj]`. The default is 10, and only `root` can set a niceness less than 0.

renice

`nice` sets the niceness of a command when it is started. You can also alter the niceness of a process while it is running, and the command for that is `renice`. Regular users can only decrease the niceness of their own processes, but once done they cannot increase it again – only `root` can do that.

The normal way to use `renice` is to state the new priority and the PID in that order on the command line. If `updatedb` was running and interfering with another process, you might want to set it's niceness to say 10 or 15. Lets assume that `updatedb` had a PID of 1234, you would then use this command:

```
renice 15 1234
```

`renice` can also adjust all processes owned by a user, and all processes owned by a group. The full syntax of the command is:

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

Here are the options:

-p
: renice the process with the following PID. this is the default, and you normally only use -p if one of the other two options were earlier used in the same command

-u
: renice all processes belonging to the following user

-g
: renice all processes with the following process group ID

Here is an example:

```
renice +1 987 -u daemon root -p 32
```

This will adjust the niceness (to 1) of PID 987 and 32, and all processes belonging to users `daemon` and `root`.

If user `bob` really irritated you this morning in the parking lot, here's a good way to take revenge:

```
renice 20 -u bob
```

Background processes

Normally, when the user runs a process, the process does it's work and sends it's output to the screen. The keyboard is tied up until this process completes. If this process is expected to take a long time, the user will probably want to continue with some other work while the first process does it's thing.

To do this, the user could open another virtual terminal and continue working there. Or, the process can be set to work in the *background*. For this to work correctly, the process must not require user input, as the keyboard will not be available. To cause a program to run in the background, you can do one of two things:

- append an ampersand (`&`) to the end of the command line used to run the program.
- Start the command normally from the command line, then press `Ctrl-Z`.

The `updatedb` command (which updates the database used by the `locate` command) can take a while to run, as it searches the entire hard disk for programs and files. To run this program in the background, enter the following as `root`:

```
updatedb &
```

On my machine this gives the following output:

```
[root@notebook root]# updatedb &
[1] 6442
```

The command returned two values – the process PID (covered earlier) – in this case it is 6442 – and a *job number*[13] – in this case 1. Job numbers are assigned to background processes in order starting at 1. They are easier to remember and type than PIDs.

fg

There can be many programs running in the background, but only one can run in the foreground, and this is the program that receives keyboard input. To bring a job to the foreground, use the `fg` command. To specify which program to bring to the foreground, you must supply either the job number or the PID. If you supply the job number, add a `%` sign before the job number to indicate it is a job number. If you supply the PID, only the PID number is required. For the `updatedb` command above, the following two commands are identical:

```
fg %1
fg 6442
```

If no job number or PID is supplied, the system will bring whatever job it considers to be the current job to the foreground.

13 **job number**: a unique identifying number assigned to a process running in the background

bg

This command is the complement to `fg` – it sends jobs to the background. Just like `fg`, to specify which jobs is to move to the background, specify a job number with a "%" character in front, or use the PID. If neither are supplied, the shell will send the current job to the background.

jobs

To get a listing of all current background jobs, use the `jobs` command. This is the output I get if I start `updatedb` in the background then run `jobs`:

```
[root@notebook root]# updatedb &
[1] 6442
[root@notebook root]# jobs
[1]+  Running                 updatedb &
```

To get rid of a command in the background, you can use the `kill` command with the job number:

```
kill %1
```

nohup

Some mathematical and engineering programs take ages to run, some take so long that they have to be left running overnight. But this opens a large security hole, because we assume that the engineer running the program would like to go home and sleep overnight too. What should he do? If he just leaves, he is still logged in to the system, so anyone can come along, move his program to the background, and do whatever they want on the system. If he logs out, the program stops running.

The solution is to be able to let the program continue running in the background *after* the user logs out. One of the signals that the kernel sends to processes is SIGHUP – it means "hangup", and is sent to all processes started at a terminal when that terminal is logged out. The `nohup` command allows a program to ignore the SIGHUP signal, which means it just keeps running after the user has logged out, and continues till it terminates normally.

When using `nohup`, there is no screen available to the program to send output to, so it has to send it to a *file*. `nohup` tries to append output to the file **nohup.out**. If that doesn't work, it tries **~/nohup.out**. If that doesn't work, it gives up and the command is not run. If `nohup` has to create a **nohup.out** file, it does so with no group and other user permissions at all. If such a file already exists, the permissions are not changed.

`nohup` does not automatically run the program in the background, you have to do that yourself by adding "`&`" to the end of the command line. `nohup` does not alter the priority or niceness of the program, you need to `nice` it yourself.

The syntax for `nohup` is very simple:

```
nohup [command [args]]
```

where `[command]` is the command you want to run in the background, and `[args]` are the regular arguments for that command.

free

Once you have a bunch of process running on your system, you may want to know how much memory they are using. The `free` command displays the total memory usage of the system as a whole. It looks like this:

```
[root@notebook etc]# free
             total       used       free     shared    buffers     cached
Mem:        190660     184876       5784          0      38084      78956
-/+ buffers/cache:      67836     122824
Swap:       192772      30880     161892
```

Each number is a total amount in KB. But first a quick introduction is needed to some topics that will be covered later in detail:

The hard disk is thousands of times slower at reading and writing data than RAM. To help out, the system includes a few tricks. Large amounts of data to be written to disk don't go there immediately – they are stored in spare RAM first and later written to disk as a background process.

If some data is read from the disk, there is a pretty good chance the data immediately after it in the file will be the next thing to be read. So when a process asks for data from disk, the system delivers it, and also reads the

next bunch of sectors from disk as well, and stores them in spare RAM. Hopefully the next few disk reads the process asks for will be in this RAM and can be delivered quicker than having to read the disk again.

These two ideas go under the name of *buffers* and *cache*. The Linux kernel is designed to use as much spare memory as it can for buffers and cache. If another process needs more memory than is free, a lot of the buffer and cache space can be freed up and used rapidly – it was only a temporary holding area after all.

Shared memory is exactly that – RAM memory that can be shared by more than one process. A good idea, but unfortunately few current programs actually use it.

You may already be familiar with the idea of virtual or swapped memory – they are different names for the same thing. When the system runs low on memory, it can take some memory data that hasn't been used for a while and write it to the hard disk. When it needs that data again, it can read it back in from the hard disk. This swapping of memory to and from the hard disk gives it the name *swap memory*, or *swap space*.

The `free` commands displays statistics about all these various kinds of memory. The first line of numbers displays the total amount of memory used in the various categories. The second line shows the amount of memory used by processes, and the amount used by the buffers/cache, in that order. The third line shows the swap space usage.

There are some options to `free`, you can specify what units to use for display (bytes, KB, MB, GB) and whether to display a running total or not. The normal usage is just `free` without any options for a one-time display.

top

The `ps` command shows you the status of processes as they were when the command was run. To see the system running dynamically, use the `top` command:

```
13:19:46 up 1 day, 18:10, 5 users, load: 0.12, 0.11, 0.09
65 processes: 64 sleeping, 1 running, 0 zombie, 0 stopped
CPU: cpu  user  nice  system  irq  softirq  iowait    idle
   total  1.9%  0.0%  0.9%    0.0% 0.0%     0.0%      97.1%
Mem: 190660k av, 184072k used, 6588k free, 0k shrd, 56992k buff
      90324k active,          79948k inactive
Swap:409648k av,  80008k used,329640k free          55572k cach

  PID USER PRI NI SIZE RSS  SHARE STAT %CPU %MEM TIME CPU CMD
15122 alan  17  0 1172 1172   908 R     2.4  0.6 0:00   0 top
15081 alan  15  0 1324 1324  1116 S     0.0  0.6 0:00   0 bash
15065 root  17  0 1172 1172   904 S     0.0  0.6 0:12   0 top
15001 alan  25  0 1316 1316  1116 S     0.0  0.6 0:00   0 bash
14946 root  15  0 1324 1324  1116 S     0.0  0.6 0:00   0 bash
14863 alan  16  0 2688  12M 10456 S     0.4  6.5 0:04   0 kdein
 7609 root  16  0 2068 2044  1616 S     0.0  1.0 0:00   0 cupsd
 7371 alan  16  0 6044 5560  5328 S     0.0  2.9 0:00   0 kdein
 7050 alan  15  0 4240  22M 12776 S     0.0 11.9 1:01   0 kdein
 2639 alan  15  0 4228  37M 34372 S     0.0 20.3 3:23   0 soffi
```

The above is a sample of `top` running on my machine (edited to fit on the printed page). The display is updated is real time, so if you watch this for a while you will see the display update every second or so. The display also updates every time you press the space bar.

`top` also includes information about the amount of memory in use, niceness, total amount of cpu time used so far, and more for those who are interested in that kind of thing.

w

Sometimes you want a quick display of the state of the system and who is logged in. The command `w` is often good for this (no, I don't know where the command name comes from!) Using `w` gives a display like this:

```
[alan@notebook alan]$ w
 23:06:11  up 4 days,  3:40,  7 users,  load average: 0.42, 0.30, 0.21
USER     TTY      FROM       LOGIN@   IDLE   JCPU   PCPU  WHAT
alan     pts/0    -         Wed 7pm  4days  0.00s  3.49s  kdeinit: kwrited
alan     pts/1    -         Wed 7pm 25:34   0.41s  0.41s  /bin/bash
alan     pts/2    -         Wed 7pm 25:00m  0.20s  0.20s  /bin/bash
alan     pts/3    -         Wed 7pm 24:25m  0.10s  0.10s  /bin/bash
alan     pts/4    -         Wed 7pm 24:25m  0.41s  0.41s  /bin/bash
alan     pts/5    -          1:13am 20:39m  0.20s  0.10s  /bin/bash
alan     pts/6    -         12:15pm  1.00s  0.12s  0.03s  w
```

`w` displays the following information:

A header showing (in this order) the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes.

`w` then displays the following 8 columns for each user:

1. login name
2. tty (console) name
3. remote host (in other words, if they logged in over a network, from which machine over the network – we cover this topic in a later chapter)
4. login time
5. idle time
6. JCPU - the time used by all processes attached to the tty
7. PCPU - the time used by the current process
8. the command line of their current process.

The syntax for `w` is as follows:

```
w - [-husfV] [user]
```

`w` supports the following command line options:

`-h`
: Don't print the header.

`-u`
: Ignores the username while figuring out the current process and cpu times. To demonstrate this, do a "`su`" and do a "`w`" and a "`w -u`".

`-s`
: Use the short format. Don't print the login time, JCPU or PCPU times.

`-f`
: Toggle printing the from (remote hostname) field. The default is for the from field to be printed, although your system administrator or distribution maintainer may have compiled a version in which the from field is not shown by default.

`-V`
: Display version information and exit.

`user`
: Show information about the specified user only.

uptime

Linux system administrators love to brag about how long it's been since they last rebooted their machine. `top` and `w` display this information, but the easiest way to get it is with the `uptime` command:

```
[alan@notebook alan]$ uptime
 23:14:59  up 4 days,  3:49,  7 users,  load average: 0.85, 0.50, 0.32
```

`uptime` has only one option: `-V` displays the version number.

wall

To send a message to all users on the system, use the `wall` command. This allows a message of up to 20 lines to be sent to all active terminals. The full message can be entered on the command line. Or, just enter the command `wall` then type the message - this allows you to break the message up into lines using the `Enter` key, which you can't do if the message is entered on the command line. End the message input by pressing `Ctrl-D`. The following command:

```
[root@notebook root]# wall
This is root speaking.
Would the person on terminal 3 who is downloading a 30MB
video clip please stop doing it.
Thank you.
[root@notebook root]#
```

produced the following output on another terminal:

```
[alan@notebook alan]$
Broadcast message from root (pts/1) (Sun May 23 23:32:01 2004):

This is root speaking.
Would the person on terminal 3 who is downloading a 30MB
video clip please stop doing it.
Thank you.
```

Daemons

The system automatically starts some programs running in the background when it reboots. These programs pretty much just sit there doing nothing most of the time, until something happens to make them wake up and respond. A web server is a typical example. While no-one out there is looking at your web site, the server does nothing. When a request for a page comes in, the system wakes up the web server, which delivers the page, then goes back to sleep. At least that's the theory – if your web server just happens to be running the **www.linux.org** site, it might never get a chance to go to sleep...

Such programs go by the delicious name of *daemons*[14]. The official line is that it means "d*isk and execution monitor*", but don't believe a word of it – the word daemon was used long before someone figured out a reasonable name that it was an abbreviation for. The truth is that it comes from Greek mythology and is similar to "demon" without the evil part – daemons hang around in the background saying and doing things on your behalf, and generally making life easier.

You normally control a daemon with the command `service` using the following syntax:

```
service [servicename] [start | stop | restart]
```

Remember this syntax well – you will be using it a lot. It is important!

I have a daemon installed on this notebook to control the *pcmcia*[15] cards which goes by the unlikely name of pcmcia. I seldom start this daemon myself, because the machine does this when it boots up. I also never stop the daemon myself, because I always want the pcmcia facility available. But my pcmcia network card occasionally just stops working and I found out the hard way that the solution is to simply restart the daemon:

```
[root@notebook /]# service pcmcia restart
Shutting down PCMCIA services: cardmgr modules.
Starting PCMCIA services: cardmgr.
```

This is a system command, so I have to be `root` to use it. There are many other daemon programs you can use, the print system is a typical example. We will return to this topic of daemons later in the chapter on networking, which makes extensive use of them.

Redirection and pipes

All computer programs do basically three things: get some input from somewhere, manipulate that input somehow, and do something with the output. It doesn't matter how complex or simple the program is, these three things are always present. So where can a program get input from? Here are some examples:

- the keyboard
- a modem
- a disk file

and some examples of where to put the output are:

- the screen
- a disk file
- a printer
- a modem

14 **daemon**: a process that usually starts when the machine boots, then runs in the background delivering system services.

15 **pcmcia**: {define this} – small credit card-sized modules that plug into portable notebooks. Common cards are modems, network cards, or extra memory

Note that some items appear in both lists. The keyboard is an input device, and the screen is an output device. Taken together they are often called the *console*[16], which is an input and output device.

A typical operation that you might want to do could be to read a list of words, sort them in alphabetical order and display the results. There are three actions here:

- read a list of words is “getting some input from somewhere”,
- sorting the list of words is “doing some operation on the data”, and
- displaying the results is “doing something with the output”.

Where might this program get it's input from? Perhaps the keyboard, or from a disk file, or maybe even from the modem.

Where could the output be displayed? On the screen, on a printer, or perhaps even through the modem.

In the bad old pre-Unix days, writing a program that did this involved the programmer getting low down and dirty with all the fine details of how to get data from the keyboard, how to read from a modem, how to read a disk file. At the nuts, bolts and wires level, these devices are all very different, and are programmed differently. The same applies to the output devices. If a programmer in those days didn't include a facility to read from and write to the modem, then using the modem could not be done with that program, and the user was stuck. This kind of problem kept coming up as new input and output devices were designed, so the designers of Unix decided to do something about it, to make the user's life easier.

The solution was in realizing that no matter how different input and output devices were from each other, they all had at least one thing in common - you could read data from them or write data to them. The next step was to *represent* each of these devices as nothing more than a place to read data from or a place to write data to. The key word is “represent” - all input devices are *represented* similarly, no matter how different they may be at a nuts and bolts level. The same applies to output devices.

The next bright idea was getting data into and out of a program. Imagine a stream of data – the list of words we want to sort flows like a stream from the input device into the program, then the sorted list of words flows out of the program to the output device.

In Unix, the system provides the input and output streams, freeing the program from having to worry about setting them up, and freeing the programmer from having to worry about how to write programs that cope with the latest new hardware – he can concentrate on writing a program that sorts words instead.

Earlier in the course we mentioned the two fundamental ideas that Unix is built on:

1. Everything is a file
2. The system consists of many small programs that do one thing well instead of a few large programs that try and do everything.

This should now make more sense – a file is how the system represents everything that inputs and/or outputs data. Note that “file” does not mean the same as “disk file” - a disk file is a certain kind of file, but a file is really just the abstract idea of “a place to get data from” and “a place to put data to”.

Using this idea of files, the user can now specify the source of input and destination of output instead of waiting for the programmer to write a program tuned to our specific input and output needs. The programmer can now write a program that does one thing really well, which is the whole idea of point 2.

Most programs read data from the keyboard and write information to the screen. Unless the user specifies otherwise, the system assumes this is what is wanted. The system defines three things to help with this:

- standard input. Normally the keyboard
- standard output. Normally the display screen
- standard error. Error messages, normally also the display screen

Unless the user says otherwise, the above devices are used. But you don't have to do it this way; you might not want error messages cluttering up the screen display, so send standard error to a printer instead. Some computers do not have keyboards and screens so all three might use the modem instead.

To use a different input or output device for one run of a program, the user specifies them on the command line. You do this using two symbols, called the redirection *operators*[17]:

---

16 **console**: the keyboard and screen taken together as a unit.
17 **operator**: a pre-defined word or symbol in a command line that causes a specific action to occur.

- < input device
- > output device

<

Lets look at our example of sorting words alphabetically, and assume a command called `sort` exists. (Actually, there is such a standard command, so these examples should work on any Linux system). Now create for yourself a disk file containing several words, one per line, and save the file as **/tmp/words**. To make this realistic, enter at least ten words, all different and make sure they not not in alphabetical order.

Now enter the following command:

```
sort < /tmp/words
```

and the sorted list of words should appear on the screen. What we have done is to tell the `sort` program that it's input does not come from standard in, but from a disk file called **/tmp/words**. The "<" symbol can be viewed like an arrow showing that data goes from the file to the sort program. (Note that typing `/tmp/words > sort` is a common beginner mistake but doesn't work as the first word in a command must be the name of the command - /tmp/words is not a command.)

>

Now enter the following command:

```
sort < /tmp/words > /tmp/words2
```

You will see that nothing appears on the screen. But, if you look in the **/tmp** directory, there will be a new file there called **words2**. Display it on the screen using `less` and you will see it contains the sorted list of words. Again the ">" symbol can be viewed like an arrow that sends data from the `sort` program to a disk file (don't send it to **/tmp/words** – this will overwrite the existing file!)

The "<" and ">" symbols *redirect* input and output from standard in and standard out to user-defined places, and they are called *operators* because they cause a redirect *operation* to take place.

>>

You can do more with redirection. The example above sent output to **/tmp/words2**, and created this file if it didn't exist. If it did exist, it was replaced with the new output. This may not be what you want though. If you wanted to send the output of a program to a log file for example, you definitely would not want to overwrite the existing file. You would want to *append* the output to the end of the existing file. This is done with the `>>` operator. Running the command

```
sort < /tmp/words >> /tmp/words2
```

will cause a second copy of the same list to be appended to the end of the first.

<<

This operator allows you to send multiple lines of input to a command until a certain line of text is reached. Recall how the `wall` command is used interactively – you input up to 20 lines of text followed by `Enter` until `Ctrl-D` is pressed, then `wall` stops accepting input and sends the text received out as a message.

`<<` operates similarly, except that you specify a line of text to be used instead of being limited to only a `Ctrl-D` keypress. You can actually use `wall` like this:

```
wall <<EndOfMessage
```

The text "EndOfMessage" is called a *limit string*, and the subsequent input is called a *here document* – perhaps becomes it conveys the idea of "continue reading input till you reach this line here".

In this example, `wall` will continue reading lines from standard input until you enter the exact text "EndOfMessage" on one line, then it will send the complete message. Note that the line must consist of only the text after <<, and it must be an exact match, not a partial match. Entering " EndOfMessages" will not work, as it is different from "EndOfMessage" - there is an extra leading space and an extra trailing "s".

To include spaces in the limit string, you must quote the limit string.

Files are often used as here documents, and the limit string might well be in the file, but with leading tab characters because of indentation. This will obviously not work. You can ignore leading tabs (but not leading spaces) by using the `<<-` operator like this:

```
wall <<-EndOfMessage
```

1> and 2>

What about standard error? How do we redirect that? By putting a "2" in front of an output redirection operator.

```
sort < /tmp/words > /tmp/words2 2>/tmp/words.err
```

will get input from file words, send output to file **words2** and send error messages (if any) to file **words.err**.

You can also say `1>` to redirect standard output, but as the system assumes "1" if you don't specifically say so, you will seldom see this used on it's own. Where the "1" is used however, is to send output and error output to the same place, which involves adding a "&" symbol.

>&

Adding a "&" to the ">" symbol as in "x >& y" causes output stream x to be mixed in with and sent to wherever output stream y is going. In all the examples below, ">&" is used to make the explanations simple, but ">>&" also works.

```
sort < /tmp/words 1> /tmp/words2 2>&1
```

Wow. this is quite a command. Let's pull it apart piece by piece:

The first piece is `sort < /tmp/words` – this is obvious

The next piece (`1> /tmp/words2`) redirects standard output. There is nothing unusual here, except the "1" to explicitly specify standard output is new. This is not normally done because 1 is the default.

The next part to look at is the last section `2>&1`. This is an instruction to redirect standard error, indicated by the "2>". And where does standard error get sent to? Well, as usual it goes to the place indicated by what follows the ">" sign. That is "&1" which literally means "wherever standard output is being sent"[18]. For this command, that is a file called **/tmp/words2.**

In this case, the contents of **/tmp/words2** will look exactly like they would have on screen, with error mixed in with normal output (in other words, there is no magic procedure that takes all the errors and attaches them at the bottom of normal output).

This command accomplishes exactly the same thing:

```
sort < /tmp/words 2> /tmp/words2 1>&2
```

What happened here is that normal output (`1`) gets mixed with error output (`>&2`), and both are then sent to **/tmp/words2** (`2> /tmp/words2`).

When the system figures out what you are telling it to do, it reads the redirection instructions starting on the right and working backwards, in order, towards the beginning. (The correct way to state this is *redirection is specified from right to left*).

If you try this

```
sort < /tmp/words 2>&1 1> /tmp/words2
```

you will probably not get what you want. This command will send normal output (`1`) to **/tmp/words2** (`> /tmpwords2`), then send error output (`2`) to standard output (`>&1`). Redirection is first established for the entire command, then used that way. The system is not smart enough to extract all the redirection definitions and apply them uniformly no matter in what order they are specified – humans can do this naturally without thinking about it, but computers - not being able to think- can't.

|

So far we have learned how to redirect input and output, which is incredibly useful. Redirection is normally used to indicate the start and end points of an entire command.

18 This is not complex, and there is nothing to figure out and understand here. &1 is defined as "wherever standard output is being sent".

Lets extend our sorting example further. Assume our /**tmp/words** file contains 10,000 words, and we want to sort alphabetically all words that contain the letter a. We could contact the author of `sort` and ask him to add this new function. If he's any good as a programmer, he'd tell us to go and jump in a lake – a function like that is so specific to one user that it isn't worthwhile writing it at all – it would also annoy the other thousands of users of sort that *don't* want this function.

A much better solution is to write a new program that *filters* input, looking for a pattern we specify, and then outputting it only if it meets our pattern. There is a command that does this, it is called `grep`, which looks at an input stream line by line and outputs only those lines that meet the pattern we specify. Our list of words was entered one word per line, so this suits our needs exactly. What we now need to do is take our input (redirected if necessary), pass it through `grep`, then take the output from `grep` and pass it through `sort` then end up with the output from `sort` (redirected if necessary). This looks very similar to redirection, but redirection doesn't help us here, because passing the output from `grep` to the input of `sort` is in the middle of the procedure we want to accomplish, not at the end point. What we need is piping, indicated with the "|" symbol. This is the command that does the job:

```
grep a < /tmp/words | sort > /tmp/words2
```

This command line executes `grep`, taking input from **/tmp/words**, and sending that output (via the "|" operator) to `sort`, which send the sorted list to **/tmp/words**.

You can see how this fits with the Unix design goal of "small programs that do one thing well" - instead of one large user-specific program that probably won't be useful somewhere else, we have two extremely useful programs (`grep` and `sort`) which we combined to use the way *we* wanted to use them.

Students sometimes get confused about the difference between redirection and pipes, and when to use them. In the example above ,is the connection between `grep` and `sort` a pipe or a redirect? Here are some simple rules that should clear it up:

- redirection indicates where input comes from, or where output goes to. So, you can only redirection to or from a file or other device.
- pipes connect two programs, never two files or a file and a program.
- You will normally find input redirection after the first command (it can't be before it as the file name is not a valid command).
- You will normally find output redirection after the last command on the command line.

Many examples exist of how to use this piping feature. Here are a few typical ones:

Last Tuesday was a very problematic day for the email system, many user's mail got discarded by accident. The administrator wants to know which users were affected. He knows the information is in the mail system's log files, but these files are huge, and searching them would take all day. He decides instead to do what people do best (think up solutions) and let the computer do what it does best (search huge amounts of data looking for tiny small details). He knows that the mail system logs each entry one per line, and dropped mail is indicated by the word "DROPPED", so he starts with the mail logs, runs `grep` on them looking for the word "DROPPED", then runs `grep` *again* on that output (using a pipe) looking for last Tuesday's date and finally pipes that output through another useful program called `cut` which extracts pieces out of lines – in this case he specifies the piece that contains the email user name.

A poet is tired of thinking of words that rhyme to use in his poems. Being a modern poet, he owns a computer running Linux. He also has two useful things at hand: The dictionary file from his spell-checking program which contains 80,000 words in plain text, one word per line, and a program called `reverse` that accepts words and reverses the order of the letters (the first letter becomes the last letter). So, he redirects the dictionary file through the `reverse` program, pipes that through `sort`, then pipes that output through `sort` again. Now he has a sorted list of rhyming words. (OK, this only gives rhyming words where the end of the word is spelled the same and ignores similar sounds that are spelled differently, but this is a Linux user's manual, not a book on writing poetry...)

tee

We are not finished yet with redirection and pipes. What do you do if you want to send the information in a pipe to *two* places? Take the example above where we used `grep` to extract words that contained the letter a and then sorted it. The output from `grep` is an intermediate result, and normally we are not interested in it, only in the final result. But what if we do want to see this intermediate result?

With what you know so far you would have to issue two commands like this:

```
grep a < /tmp/words > /tmp/words.tmp
sort < /tmp/words.tmp > /tmp/words2
```

Which will leave the intermediate result in **/tmp/words.tmp**. But there is a command that makes this easier, called `tee`.

`tee` takes input from standard input (which can be redirected) and sends it to standard output (which can be redirected) and also to a file whose name is listed on the command line.

To replace the two commands above with one command using `tee`:

```
grep a < /tmp/words | tee /tmp/words.tmp | sort > /tmp/words2
```

This is essentially the same as the example in the pipe section with an extra command piped in between `grep` and `sort`.

`tee` has an `-a` option: if the specified file already exists, `tee` appends to that file. Without `-a`, any existing file is overwritten.

Manipulating text files

Most of the commands in this section follow the “small program that does one thing well” philosophy. They don't appear to be very useful on their own (this observation is quite accurate), but they are designed to be *combined* with other programs (using redirection and piping) to do incredibly useful things.

Regular expressions

A regular expression (or “regexp”) is a search pattern – you use it to find text inside larger strings. When we used grep above to search for the letter “a”, the “a” was a regular expression of just one character. The simplest use of regular expression is looking for a text string. To find the string “myword” in a file called “myfile”, you use:

```
grep myword myfile
```

Regexps allow you to build complex search strings that look similar to glob expressions. But there are significant differences. Regexps look for text strings, globs look for file names. Regexps use the [,], * and ? characters but the syntax is different to that for globbing. Don't get them confused.

Searching for single characters

A regular expression is built up character by character. Letters and digits usually evaluate to themselves, like “myword” in the previous example. This evaluates to the six characters **m**, **y**, **w**, **o**, **r** and **d** in that order.

A basic rule with regular expressions is that everything between square brackets [] evaluates to a single character. [abc] is an **a**, **b** or a **c**, not the three characters “abc”. You would have to omit the square brackets to get that.

A range of possible characters is indicated with the first and last characters separated by a dash. Examples: [a-z], [A-Z], [0-9]. You can also combine ranges like this: [a-zA-Z0-9] which evaluates to any alphanumeric character. There are some predefined shortcuts for common ranges. The names are self-explanatory, and they are:

`[:alpha:]`
: all letters, i.e. a-z and A-Z

`[:digit:]`
: any decimal digit, i.e. 0-9

`[:alnum:]`
: all alphanumeric characters, i.e. a-z, A-Z and 0-9

`[:lower:]`
: any lower-case letter, i.e. a-z

`[:upper:]`
: any upper-case letter, i.e. A-Z

`[:print:]`
: all printable characters, including space

`[:graph:]`
: all printable characters, excluding space

`[:punct:]`
: all punctuation characters

`[:space:]`
: all whitespace including space, tab and vertical tab

`[:xdigit:]`
: any hexadecimal digit, i.e. 0-9, a-f, and A-F

`[:cntrl:]`
: non-printing control characters (tab, newline, newpage, etc.)

The square brackets in the shortcuts above are part of the full shortcut name. You use it like either character in the usual [ab] form. This means that `[a-zA-Z]` is equivalent to `[[:alpha:]]` when used.

The expressions we have looked at so far specify the characters to be included. If you start the expression inside square brackets with a carat (`^`), it means to exclude those characters. `[^123]` will match anything *except* 1, 2 or 3.

It doesn't look like we can search for the], `^` or – characters inside square brackets, because these mean something else. Actually there is a way, these are the rules to do it:

- to include a literal], make it the first item in the list. Regexps do not allow empty [] expressions, so it is understood that []] means to look for a "]"
- to include a literal `^`, place it anywhere except the beginning. The "except" meaning applies only if it is the first character. `[abc^]` searches for an a, b, c or a `^`.
- to include a literal -, place it last in the list. Nothing then follows it, so it can't be a range specifier. `[abc-]` matches an a, b, c or a -.

Repeating characters

Regexps include a convenient set of shortcuts for dealing with repeated characters.

There are some special characters that allow searching for characters that may repeat. You place them after the repeating character in the regexp.

?
: The preceding item is optional and matched at most once.

*
: The preceding item will be matched zero or more times.

+
: The preceding item will be matched one or more times.

{n}
: The preceding item is matched exactly n times.

{n,}
: The preceding item is matched n or more times.

{n,m}
: The preceding item is matched at least n times, but not more than m times.

Time for some examples:

`abc?`
: means the last c is optional. There are only two possible matches: ab and abc.

`[0-9][0-9][a-z]*`
: will find any string of two digits followed by any amount of lower case letters, or no letters.

`[0-9][0-9][a-z]+`
: will find any string of two digits followed by any number of lower case letters, but there must be at least one letter.

`[ ]{2,}`
: will find all sequences of two or more spaces.

So far all the examples we have looked at will find single characters that repeat. Next we look at finding complete words.

Searching for a string

You will seldom do a search for a single character. Most searches involve more than one character, with complete words being the most common. this is easy to do, just write them down in sequence one after another, or *concatenate* them. To find the character "a", do this:

```
grep a
```

To find the word "apple" do this:

```
grep apple
```

To find any three letters followed by any three digits do this:

```
[a-zA-Z]{3}[0-9]{3}
```

Of course, you already figured this out. So why mention it? Because to find repeated words (more specifically, sequences of two or more characters), you have to use brackets. To find the word "the" repeated more than once[19], you might think `the+` will do the trick. It won't. It will find "theeee" though. What you really want is the following:

```
(the)+
```

The brackets work the same way they do in arithmetic – they set *precedence*[20]. The "+" now indicates it should look for one or more repeats of the whole expression inside the brackets. You can do this with any sequence of characters.

Find this or find that

How do you go about the following: find either "smith" or "jones" or"brown"? Easy – use the "|" character.

```
smith|jones|brown
```

The expressions on either side of the "|" can be any valid regexp of any length.

Regexp precedence

When evaluating a complex regexp, there are rules that the system follows. The expression is evaluated in the following order:

1. Repetition with ?, *, + and {}
2. Concatenation – strings of characters
3. Alternates either side of a "|"

The example with repeated "the" above should now make sense. And it should be obvious why brackets in `(smith)|(jones)|(brown)` are unnecessary.

More shortcuts

To round off this discussion of regexps, there are a few more shortcut characters:

A period (.) matches any single character, so `sh.rt` finds both "shirt" and "short"

`\w` evaluates to `[[:alnum:]]`

`\W` evaluates to `[^[:alnum:]]`

`^` outside of square brackets matches the start of a line. So, the regexp `^[a]` means a line that starts with an a.

`$` outside of square brackets matches the end of a line. So, the regexp `[z]$` means a line that ends with z.

`\<` outside of square brackets means matches the start of a word, so `\<book` finds "bookmark" but not "checkbook"

19 Some writers (like me) seem to do this a lot. I often type things like "Joe kicked the the ball".

20 **precedence**: the order in which something will be executed. Precedence can be changes by putting the thing to be done first in brackets.

`\>` outside of square brackets matches the end of a word, so `book\>` finds “checkbook” but not “bookmark”

`\b` outside of square brackets matches either edge of a word, i.e. either the start or the end.

`\B` outside of square brackets is the opposite of `\b`, i.e. anything except the start or edge of a word.

A complex example

I once had to conduct a search in a document for all valid MS-DOS file names. The resulting regexp was horrific to see but it did the job. I will go through this in detail so you can see how it works. I need to go through this in detail because regexp like this are more common than you might think.

A full MS-DOS file-name looks like this: `C:\ABC\DEF\12345678.123`

To build the regexp we have to break the full file name up into it's several parts:

```
C:   \   ABC\   ABC\   12345678   .123
```

- an optional drive letter followed by a colon
- an optional backslash for an absolute path
- an optional directory name followed by a backslash. The directory name can be up between 1 and 11 characters long.
- further directories
- a mandatory file name of between 1 and eight characters
- an optional dot followed by an extension of between 1 and three characters.

The optional drive letter and colon is specified like this:

```
([a-zA-Z]:)?
```

The optional backslash is specified like this:

```
\?
```

An optional directory name, followed by a backslash is specified like so:

```
[a-zA-Z0-9]{1,11}\
```

But the file spec can contain zero or more directories, so we have to search for this pattern repeated with *. As a regexp, this becomes:

```
([a-zA-Z0-9]{1,11}\)*
```

The mandatory file name itself is the easiest part of the whole exercise:

```
[a-zA-Z0-9]{1,8}
```

And finally the optional dot and extension is:

```
([.][a-zA-Z0-9]{1,3})?
```

Now we string the whole thing together to get the final result:

```
([a-zA-Z]:)?\?([a-zA-Z0-9]{1,11}\)*[a-zA-Z0-9]{1,8}([.][a-zA-Z0-9]{1,3})?
```

I told you it was horrific.

For simplicity sake, I have taken file and directory names as consisting of only letters and digits. In real life you have to account for all the other valid MS-DOS file name characters as well like -, _ and ~.

Gotchas

We're not done yet. There are still some things to catch you out.

Regexps fall into two camps- the *basic* and *extended* types. The difference comes down to the handling of special characters such as ? and *. With basic regexps, these special characters normally lose their special meaning, so to get the special meaning you have to place a backslash before them like this: \?, *

With extended regexps, the special characters retain their special meaning, which makes things easier for the poor guy writing the regexp. Except that now, to search for a ? character you backslash it like this: \?

Oh no. Now we have the exact same syntax doing two completely different things. Basic mode – use the backslash to get the special meaning. Extended -use the backslash to get the regular meaning. You will have to check the documentation for the software you are using to see which mode it uses.

Finally, the last thing that can catch you out is the sort order of letters. In all the examples above I used `[a-z]` to mean all letters between a and z. This works, but only if your system is using the English alphabet and the sort order is strictly according to the ASCII code. This is not always the case - the Linux system understands something called a *locale* – which means where in the world you are, and what the local conventions are for displaying currency, dates, the time and the alphabet. In Europe for example, many languages use characters like è and å. English doesn't have these and usually puts them after z. But they really belong next to the regular e and a.

ASCII sorts letters this way: AB...YZab...yz. In a human language, the following makes *much* more sense: AaåBb...Eeè YyZz. What this all means at the end of the day is that [a-z] or even [a-zA-Z] might not give you what you really want, as they ignore the locale. Instead, use the [[:alpha:]] form to specify any letter – this form always takes the locale into account, including the odd letters with symbols above and below them.