UNIX 2
Enhancing your UNIX skills

Workbook

Edition 2
August 2000

UNIX 2

Enhancing your UNIX skills

Course Code: 1159

Associated Workbooks:

UNIX 2: Practical Exercise - Doc ref:3059

UNIX 2: Solution - Doc ref:3060

Edition2, August 2000

EUCS Document Number: F.5-WB-3133-8/2000

Chapter 1

Deeper UNIX

UNIX 1 - REVISION

- login and logout
- `passwd`
- files and directories
- absolute and relative pathnames
- `ls`, `cp`, `mv`, `rm`, `cat`, `mkdir`, `rmdir`, `pwd`
- `more` and `ue` (MicroEMACS)
- file and directory permissions
- printing with `lpr`, `enscript`, `lpq` and `lprm`

You should be familiar with these commands and concepts, either from the EUCS UNIX1 course, or else from your previous UNIX experience.

telnet and logout

to login and logout respectively from a UNIX computer.

passwd

to change your account password. Please change your training account password- remember the tips given in UNIX 1 about secure passwords.

files, directories and pathnames

disk files and directories can be manipulated using `ls`, `cp`, `mv`, `rm`, `cat`, `mkdir`, `rmdir` and `pwd`. Remember that pathnames can be specified relative to the root (/) directory (absolute path) or to the current (.) directory (relative path)

more and ue

files may be viewed using a pager, such as `more`. Editing may be done in the MicroEMACS text editor (`ue`), or any other editor of your choice.

permissions

files and directories have permissions to determine who may access or change them. You should understand how to display and change permissions, and what different permissions mean for both files and directories.

lpr, enscript, lpq and lprm

print jobs may be sent using `lpr`. Don't send Postscript files to text printers! Text files can be converted to Postscript using `enscript`. Use `lpq` to view the queued print jobs and `lprm` to remove them.

UNIX 1 - REVISION

- UNIX shells; bash
- shell shortcuts
  - filename completion
  - command history
  - command-line editing
  - wildcards * and ?
- redirection and piping using < > >> and |
- foreground and background processes; `kill`
- the UNIX environment
- ftp

---

shells

The shell interprets between you the user and the computer (strictly the UNIX kernel). There are different shells available; EUCS recommends the bash shell.

filename completion (<tab>)

bash can attempt to complete a half-typed file, directory or command name.

history mechanism

previous commands may be recalled and edited using the arrow keys or MicroEMACS key sequences.

** and ?*

wildcards match sets of similarly-named files or directories.

redirection

< to redirect stdin (standard input; normally your keyboard)

> to redirect stdout, deleting the previous contents. (normally your screen.)

>> to redirect stdout, appending to previous contents.

| to redirect the output of one process into the input of another.

foreground and background processes

processes may be run in the foreground or background, and can be interrupted using the `kill` command.

UNIX environment

variables can be set at the command line, or from startup files, and added to the environment using `export`.

ftp

files may be swapped between remote computers ftp. Either give a valid username and password, or give your email address for anonymous ftp.

UNIX PHILOSOPHY

- Simple tools combined to build more complex ones
  - Each program should do one job well
  - Building blocks
  - A complete programming language
- Everything is a file
  - nearly

UNIX has its roots in the late sixties. The first recognisable version was written in 1969. It was the first operating system which was independent of the computer hardware which it ran on - it was written in the C programming language.

Complexity through combination

One of the principles underlying its design was "the more complex something is, the more likely it is to be faulty." Many UNIX programs are written to perform a simple task cleanly and efficiently. These "building block" tools are combined in different ways to build complex, powerful utilities without having to duplicate what has already been written.

Many UNIX programs are *filters*; they read from a standard stream of input and write to a standard stream of output. Such commands can be connected by pipes, which attach the output of one command to the input of another. Where the underlying hardware allows, all programs which are part of a pipeline will run simultaneously. This is unlike some other systems where the first program is run to completion, then its output is passed to the second, and so on.

For instance

```
bash$ ls -l | grep ^d | wc -l
```

returns the number of subdirectories in the current directory. It does this by listing all files, selecting those which are directories, and counting them.

Everything is a file

Another original aim was to unify the way that users accessed all system resources, from files and disks to printers and networks. The idea was that once a user learnt how to access one type of resource, her skills would be transferable to others.

UNIX was only partially successful in this aim. Dennis Ritchie, one of the authors, had a second try with the Plan 9 operating system.

WHAT IS THIS MACHINE ACTUALLY DOING?

- Running programs
  - tools and utilities
  - you can create your own
- The **shell** interprets between you and the machine
  - shortcuts to make your life easier
- You have one login shell
  - EUCS recommends `bash`

Like any other computer system, UNIX runs computer programs: encoded sequences of instructions which make sense to the machine. Your user interface is usually a specialised program called a *shell*.

The Shell

After you log in, a UNIX machine will start a shell for you. The shell displays a *prompt* which might look like this:

```
bash$
```

and waits for you to type instructions. Once you press RETURN to input your commands, the shell interprets what you typed and runs the program that you intended. You can use its facilities to make it easier to work with UNIX.

Default shell

There are a number of shells available but only one can be set as your *default shell* - the one started when you log in. You may have chosen this the first time you logged in or your system administrator may have set it for you.

There are a variety of shells available, such as `sh`, `csh`, and `tcsh`. Computing Services recommends the `bash` shell for interactive use as it is powerful and freely available. In this course we will focus on `bash`.

Note for the deranged

There is no law forcing you to use a sensible default shell. You can use any program at all. For instance, you could use the `ls` program. If you do this, every time you log in you will be presented with a listing of your files and immediately logged out. Using a reasonable shell is normally better for your productivity.

In the past people have used a text editor such as `emacs` as their shell, as that could provide a window system on text-based terminals.

PRACTICAL EXERCISES

Which default shell?

The `finger` command displays information about a user. This could be yourself, or somebody else.

```
bash$ finger stella
Login name: stella                      In real life: Stella Artois
Directory: /home/stella                 Shell: /usr/local/GNU/bin/bash
Last login Thu Dec 31 23:59 on pts/666 from pint.ed.ac.uk
```

In this case the user `stella` is using the `bash` shell. You can tell because the `Shell:` field ends in the word `bash`. `/usr/local/GNU/bin/bash` is the *full path* to the `bash` command. More on this later.

Find out your own default shell

- Use the `finger` command to check what your own default shell is.

Fingering from a distance

The `finger` command will work for users on remote machines too. However the administrator of that machine may have told it not to respond for security reasons.

```
bash$ finger rob@waverley.ed.ac.uk
[waverley.ed.ac.uk]
Login       Name               TTY         Idle    When    Where
rob      Rob McCron           pts/27         2d  Mon 14:44  leo.ucs.ed.ac.uk
```

Finger a friend

- If you know somebody with an account on another UNIX machine, try to `finger` them and see what happens.

WHAT IS A UNIX PROGRAM?

- Programs can be stored in the filesystem
  - like any other data
  - they have an absolute pathname
  - use the `type` command to find them
- Some are built into the shell

The shell is a program in the same way that `ls` or `mkdir` is a program. It is simply a more sophisticated program.

In UNIX, programs are stored in the filesystem exactly the same way as data files. Normally only the system administrator can move or edit these files. For instance, here are some common commands with their locations on the Computing Services machines:

```
cat             /usr/bin/cat
MicroEMACS      /usr/local/bin/ue
xclock          /usr/bin/X11/xclock
Netscape        /usr/local/public/bin/netscape
```

The expanded form is known as the *full path* to a command.

Finding a command

The command `type` will tell you where a particular command resides.

```
bash$ type rm
rm is /usr/bin/rm
```

Shell builtins

A few simple commands are not to be found by themselves anywhere in the UNIX filesystem. These are contained within the shell program, and it handles those directly.

```
bash$ type cd
cd is a shell builtin
```

PRACTICAL EXERCISES

Locate a favourite command

- ❑ Use the `type` command to determine the location of `type` and `cat`.
- ❑ Use the `type` command to determine the location of some other commands you use regularly.

Some commands exist in multiple versions, in different places within the file system.

- ❑ Try `type` with the `-a` option.

```
bash$ type -a ls
```

Examine a binary

Many commands are stored in *binary* - a raw form of program which the computer can execute but which is not easily readable by humans.

The `xod` program displays the content of a binary file in a readable fashion. Although you won't be able to follow how the program executes, you can see embedded text and get a feel for the size of a UNIX program.

- ❑ Choose a program which resides in the file system and have a look at it using the `xod` command. You will probably have to use a pager or the output will scroll off your screen too fast to read.

```
bash$ xod /bin/ls | more
```

HOW THE SHELL INTERPRETS INPUT

- One command plus extra words
  - shell does not recognise argument types
- Scans your input for metacharacters
  - makes substitutions where necessary
  - then executes command

At its most basic level, the shell views any input as one command plus perhaps some extra words, or *arguments*, to pass to that command.

```
bash$ ls -l -F reports
```

The shell views this as an instruction to run the `ls` program and to pass it the arguments `-l`, `-F` and `reports`. The shell has no concept of what those arguments represent. It cannot tell the first two are intended as options, and the third as a directory to be listed. It passes them directly to the `ls` command.

Shell metacharacters

A number of characters have special meanings to the shell. These are known as *shell metacharacters* and they change the way the shell interprets the line.

```
bash$ ls ‘cat whichfile‘ | sort > results; echo "done"
```

The backquote, pipe, greater than, semicolon and double quote characters are all metacharacters and instruct the shell to interpret the words around them in different ways.

In order to make this all work, the shell scans your input and interprets the metacharacters. Only once it has applied their effects does it run the command.

Geek note

One of the less desirable consequences of UNIX’s evolutionary process is that the way options are presented to commands is only a convention. `ls` can accept two options as "`-l -a`" or "`-la`". Other commands are more picky and require one form or the other. If in doubt, check the manual page.

WHAT CAN I DO WITH METACHARACTERS?

- Wildcards * ? []
- Home directory ~
- Dereference variables $
- Comment #
- List of alternatives { }
- Input/output redirection < > |
- Quotes for other metacharacters '...' "..." \
- Command separators ; `...` &

*Wildcards * ? []*

Recall that ? can substitute for any one character, and * for any number of characters. Thus `coo*` would match the files `cook` and `cookie` but `coo?` would only match the file `cook`.

You can also specify a range of characters using square brackets. For instance `[a-z]` matches any lower case alphabetic character, `[A-Za-z0-9]` matches any letter or number. So `book[3-7]` would match the files `book4`, `book5`, `book6` and `book7`, but not `book1` or `book9`. More powerful matching than this is possible using regular expressions (see *Power UNIX*).

Home directory ~

The tilde character (~) refers to a user's home directory. For the user `stella`, ~ refers to `/home/stella` and `~beck` refers to `beck`'s home directory. The shell looks up the home directory and makes the substitution before passing the new word on. If it cannot find such a home directory it passes on the ~ character unchanged.

Dereferencing variables $

You are already familiar with setting and dereferencing (reading the contents of) shell variables. The dollar sign instructs the shell to substitute the contents of the shell variable being referenced.

```
bash$ NAME="Peter Parker"
bash$ echo $NAME
```

Comment #

The shell ignores any input after a hash symbol (#). This is not particularly useful when typing at the command line, but becomes very important when writing shell scripts.

Alternatives {}

`bash` and `csh` provide a way to specify a list of arguments in a command line, by enclosing them within curly brackets and separating with commas.

```
bash$ ls -l /tmp/{xx,xxx,xxxx}
-rw-rw-r--   1 issarch  eucsup     28150 Oct 22 15:42 /tmp/xx
-rw-------   1 tony     ug            7 Nov  2 15:45 /tmp/xxx
-rw-------   1 tony     ug         4894 Oct 22 15:48 /tmp/xxxx
```

Redirection of input and output < >

You should already be familiar with the symbols < and > to redirect input to and output from a command, and the pipe symbol | which connects the output of one command to the input of another. These operations are carried out by the shell.

```
bash$ update_database < infile | sort > outfile
```

Here the shell runs the `update_database`command with input read from `infile`. The results of that command are piped to `sort` and the resulting output from `sort` is written to the file `outfile`.

Quoting \ ' "

You may wish to use these metacharacters in your commands without their particular effect. For instance, you might want to display an asterisk symbol without having it expanded to match filenames.

```
bash$ echo This is a one star hotel \*
```

A backslash quotes only the next character.

```
bash$ echo 'This is a one star hotel *'
```

Single quotes can be used to display any metacharacters except a single quote itself.

```
bash$ echo "This is a one star hotel *"
```

Double quotes will protect any metacharacters other than $, backquote (`), \ and double quotes themselves. These exceptions are interpreted by the shell in the normal way.

Command separators ; & `

The semicolon `;` can be used to join a sequence of commands together.

```
bash$ clear; sleep 5; xwd
```

This is a common use. Here the screen is cleared, the computer waits for five seconds, then takes a *window dump* (a picture) of the window selected. The delay allows us to be sure that our command line is not visible.

The ampersand & specifies a command should be run in the background. We cover this concept later in this workbook.

Backquotes `...` are not like other quotes. See over the page.

BACKQUOTES

- backquotes ‘...‘ mark an enclosed UNIX command
  - contents are executed and the result substituted for the quotes
  - this is known as **command substitution**
  - $(...) is an alternative syntax
- useful when setting variables
- backquotes are not protected by weak quoting
- strong and weak quotes can be placed inside backquotes
  - as can other metacharacters

The use of a pair of backquotes ‘...‘ indicates to the shell that the text between them should be viewed as a separate command. That command is executed first and its output is substituted for the quoted segment of your command line. This is called *command substitution*.

```
bash$ echo today is ‘date‘
today is Wed Nov 25 15:00:59 GMT 1998
```

Backquotes may be used to store the results of a command in an environment variable:

```
bash$ TODAY=‘date‘; echo "today is $TODAY"
today is Wed Nov 25 15:00:59 GMT 1998
```

Backquotes may be used inside weak quotes ("), but will be interpreted literally inside strong quotes (’). So:

```
bash$ echo "today is ‘date‘"
today is Wed Nov 25 15:00:59 GMT 1998
```

but

```
bash$ echo ’today is ‘date‘’
today is ‘date‘
```

The enclosed command need not be simple. It can be any UNIX command, including wildcards, quotes and pipes. The following example would list all .txt files in the current directory whose names appear in `filelist`.

```
bash$ ls ‘cat filelist* | grep ’.txt’‘
```

Note that the single quotes do not escape the command substitution, because they are *inside* the backquotes. Don’t worry if you don’t understand the example entirely; the `grep` command is described in Power UNIX later.

HOW THE SHELL FINDS COMMANDS

- You can type commands without their full path
  - the shell finds them for you
- The shell needs to be told where to look
  - the PATH environment variable
  - you can customise this for your own purposes
- Including . in PATH is convenient but dangerous
  - allows any program in the current directory to be executed
  - potential security risk
  - safest to place . last

UNIX systems can be configured in many different ways. The shell does not magically "know" where to look for commands in the filesystem. It has to be told using the PATH environment variable.

Here we look at a simple PATH:

```
bash$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/GNU/bin:/usr/ucb
```

The PATH is a list of directories separated by colons (:). Note that most of the directories have the name `bin`. This stands for "binary". Although not all commands are necessarily stored in binary - a directory named `bin` is simply a reminder that its contents are commands.

When you type any command which is not a shell builtin, the shell explores the directories named in your PATH, in order from left to right, looking for a command. For instance, if you had the PATH above and typed `gcc`, the shell would look in `/usr/local/bin`and `/usr/bin` before finding the program you want as `/usr/local/GNU/bin/gcc`

Adding to your PATH

You can add new directories to your PATH either at the beginning or the end:

```
bash$ PATH=/home/stella/bin:$PATH
bash$ PATH=$PATH:/home/stella/bin
```

Note for the Paranoid

Some people like to include . in their PATH, so that programs in the current working directory can be run without typing the full pathname. Remember that some directories, such as `/tmp`, may have spoof programs left in them with the same name as common UNIX commands but which do damaging actions. If you must include . in your PATH, do so at the end so that it will be searched last.

PRACTICAL EXERCISES

Examine your PATH

- Use the `echo` command to look at your own PATH.
- (if on `holyrood`) Celebrity actor Al Pacino has written a short script to aid the course today. He has stored it in the directory `/usr/local/xtras` Add this to your PATH. You should now be able to run the `alpacino` command.

WRITING YOUR OWN COMMANDS

- UNIX tools are easily combined for advanced functionality
- Aliases are simple text substitutions
  - use for shortcuts and to save typing
  - or to help you get used to UNIX
  - or to always include certain options to a command

```
bash$ alias mail="pine"
```

```
bash$ alias ls='ls -F'
```

- Aliases can be removed

```
bash$ unalias mail
```

One consequence of the simplicity and modularity of most UNIX commands is that it is fairly easy to create your own tools from two or three standard commands. Aliases are the simplest way to help you do this. The bash shell also allows you to write *shell scripts* and *shell functions*. Unfortunately this course doesn't.

Aliases

These are simple text substitutions. They are useful for giving commands names that you can remember more easily or for creating various other shortcuts.

For instance, if you had recently switched to the `pine` mailer but found yourself typing `mail` by force of habit, you could define this alias:

```
bash$ alias mail="pine"
```

For the rest of your current shell, whenever you type `mail`, the shell will interpret that as if you had typed `pine`. Note that this matching only takes place from the beginning of your input - later arguments containing the word "mail" will not be changed to say "pine".

You could also use aliases to define permanent options for favourite commands.

```
bash$ alias ls='ls -F'
```

Now whenever you use `ls` the shell will add the `-F` option, appending `/` to directory names, `*` to executable files and `@` to symbolic links.

You can make aliases permanent by including them in your shell startup files (see page 21). You can also get rid of them with the `unalias` command.

SHELL SCRIPTS

- Shell scripts store a sequence of commands
  - executed in sequence
  - use for more sophisticated tasks
  - can be easily shared with other users
  - stored in the filesystem
- Good practice to specify the command interpreter
  - Bourne shell for portability
    `#!/bin/sh`

Shell scripts are text files which contain a sequence of UNIX commands to be executed. Unlike aliases, they are stored in the filesystem so they can be run by different users, and even on different UNIX systems.

Here is a typical short shell script:

```
#!/bin/sh
#  dec2hex, converts three decimal numbers to hex
# Syntax: dec2hex num num num
hexnums=‘echo "16o$1p$2p$3p" | dc‘
echo $hexnums
```

This is a utility which converts a list of three decimal numbers into hexadecimal, useful when experimenting with RGB colours. Don’t worry if you don’t follow how it works.

Points to note

The three lines which start with a # are comments. The shell ignores every line which begins with a #.

The only exception to this rule is the construction which appears on the first line. If the very first two characters of a file are `#!`, UNIX recognises the first line as an instruction to execute the script using the command interpreter specified by the remainder of that line. In this case it is `/bin/sh`, the Bourne shell, the most basic shell supplied with UNIX. It can be any program at all - remember a shell is simply a program. Often the program `perl` is specified in this way, to interpret scripts written in the Perl language.

Even though we advocate the shell `bash` for interactive use, we would suggest you write your shell scripts for `/bin/sh`. Every UNIX system is guaranteed to include the Bourne shell, but it may not support `bash`. Writing scripts for the Bourne shell guarantees they will run on other UNIX machines.

RUNNING A SHELL SCRIPT

- Shell scripts can be run in three different ways
- Specifying full path to script

  ```
  bash$ /home/stella/bin/yourscript
  ```

  - perhaps using a relative pathname

    ```
    bash$ ./yourscript
    ```

  - or using PATH
- Starting a subshell with the script as an argument

  ```
  bash$ sh yourscript
  ```

- Sourcing the script

  ```
  bash$ . yourscript
  ```

It is likely that you have run a number of shell scripts without being aware of it. To the casual user they are no different to binary programs.

Say you have a script named `datacheck`. Provided you have read and execute permission for this script, you can run it with its full pathname. If the script resides in the current directory, you can use that as a shortcut:

```
bash$ ./datacheck
```

You can also specify explicitly which shell to use to run the script. Normally this will be the Bourne shell. You will need read permission to use this method.

```
bash$ sh datacheck
```

A *subshell* is created when one shell starts another. Both of these methods run the script in a subshell.

Sourcing

If you want changes to your shell environment (see page 23) to persist, you can run the script using the current shell. This is called *sourcing* the script and signified by a dot. You need read permission.

```
bash$ . datacheck
```

Note from the old-timers

If you write more than one shell script which you will use regularly, it is worth creating a `bin` directory in your home directory and collecting your scripts there. You can then add that directory to your PATH and run your scripts by name.

BASH SHELL STARTUP FILES

- Scripts which set up your working environment
  - bash login shells use `.bash_profile`
  - bash non-login shells use `.bashrc`
  - simplest to have minimal `.bash_profile` sourcing `.bashrc`

The shell is designed to be customised in a variety of ways. It would be inconvenient to make your changes by hand every time you log in, so there is a mechanism for making these permanent - *shell startup files*.

These are shell scripts which are sourced. They record a number of commands to be executed as if typed at the prompt. They are stored as dot files (hidden files) in your home directory so you need to give `ls` the `-a` option to include them in a listing.

When you first login and start bash, it reads your `.bash_profile`. If you remain logged in and start additional shells, they will read your `.bashrc`. So you can specify different startup arrangements for login shells and other shells.

In practice most people find it easiest to have a fairly small `.bash_profile` and have it source their `.bashrc`. They can then make additions to their `.bashrc` in the knowledge that these will be picked up by any `bash`.

Note for shell connoisseurs

Different shells put their startup files in different places such as `.profile` or `.cshrc`. If you wish to use a different shell, consult the manual page to check exactly which startup files it reads.

PRACTICAL EXERCISES

Examine your dot files

- ❑ Take a look at the hidden files in your home directory.

```
bash$ ls -a
```

- ❑ Examine your shell startup files.

THE ENVIRONMENT

- Shell variables are local to your current shell
  - not inherited by child processes
- Your shell **environment** is passed to child processes
  - shell variables can be exported to the environment

    ```
    bash$ ALTEREGO="Green Goblin"
    bash$ export ALTEREGO
    ```
  - and shell functions too

    ```
    bash$ export -f sortdbase
    ```

You are already familiar with shell variables and how to store and retrieve values from these.

You can see all your shell variables using the `set` command. You are likely to have quite a few so you may wish to use a pager.

```
bash$ set | more
```

However if you run a command, the child process thus started does not have access to all your shell variables. It has access to a subset of these called the *environment*. You can view your shell environment with the command `env`.

```
bash$ env | more
```

Adding to the environment

If you wish to make a shell variable available to programs you run, you will have to add that variable to the environment. You do this with the `export` command.

```
bash$ export NICKNAME
```

The variable NICKNAME is now part of the environment ("exported" to child processes).

ENVIRONMENT PASSING

- Parent process passes environment to child process
  - not shared - the child gets its own copy
  - changes are never passed back to the parent

More on child processes

When a child process is started from a UNIX shell, the parent passes its environment to the child.

It is important to understand that the child receives a *copy* of its parent's environment, which it can then modify to suit itself. They do not share any data. And when the child process terminates, **it does not pass its environment back to the parent**.

So if you start a subshell (a child process), alter some shell variables and then exit the subshell, those changes will not persist upon your return to the original shell.

Relative note

You can view the passing of the environment as analogous to the passing of knowledge in human families. Parents teach their children many things, then time passes and the children become wise in their own right. But even when children know best, they find they can't teach their parents anything.

PRACTICAL EXERCISES

Setting and exporting a shell variable

- ❑ Create a new shell variable. Call it MOVIE and set it to the title of your favourite film.
- ❑ List your shell variables using `set` and check that MOVIE shows up.
- ❑ Display your environment using `env`. Is MOVIE part of the environment?
- ❑ Start a new subshell by hand.

  ```
  bash$ bash
  ```

- ❑ Use `echo` to view the contents of MOVIE. Is it available to the subshell?
- ❑ Exit the subshell and return to your original shell.

  ```
  bash$ exit
  ```

- ❑ Is the MOVIE you set still visible to the original shell?
- ❑ Make MOVIE part of the environment using `export`.
- ❑ Start a new subshell once more and look to see if MOVIE is available.
- ❑ Set the contents of MOVIE to something different - say, the title of a film you hate.
- ❑ Display your environment. Is MOVIE part of your environment?
- ❑ Exit your subshell.
- ❑ Look at the contents of MOVIE now. What has happened?

RUNNING MULTIPLE PROGRAMS

- ✷ You can run several programs at once under UNIX
  - and you may be sharing a machine with many other users

UNIX provides *multitasking* facilities to users - it can do many tasks at once. This means you could have your computer get on with calculating invoices and scheduling timetables while you read email or compose your quarterly report.

Furthermore UNIX machines are designed to support many users at once, each of whom may be running multiple commands at once. So there is a great degree of multitasking going on.

An individual user can run multiple programs at once. For each interactive shell, all but one program will need to be in the *background.*

Pedantic note

Many computers running UNIX have only one CPU (central processing unit). This means, strictly, they only do one thing at a time. They create the illusion of doing many at once by dividing any task into very small pieces and alternating which task they pay attention to. So with two tasks A and B, the computer will do a little of A, then a little of B, then a little of A, then a little of B, and so on until the tasks are finished.

Superhero note

This is not unlike how Superman can appear to be in two places at once by repeatedly nipping between them faster than a speeding bullet.

PROCESSES AND RUNNING COMMANDS

- A process is a program being executed
  - you can run several at once
  - when the program finishes, the process terminates
  - processes can be viewed with the ps command

```
bash$ ps -f
  UID   PID  PPID  C    STIME TTY      TIME CMD
stella 19348 19282  0   Apr 06 pts/66   0:02 -bash
```

A UNIX process is a program being executed. Processes can start other processes. We call the original the parent process and the new one a child process.

When you start a program on a UNIX machine, it is run as a new process. The program you specified, whether it was a shell script or binary program, executes as this new process. Once the program completes, whether by reaching the end successfully or stopping prematurely with an error, the new process terminates.

All this is mostly transparent to UNIX users. It becomes significant when you have to understand how shell variables are handled.

Listing your processes

```
bash$ ps
   PID TTY      TIME CMD
 18569 pts/302  0:35 bash
 19444 pts/302  0:02 ue
```

The ps command lists the processes you are running. PID stands for process ID; you can use this number to refer to a particular process. TTY is your terminal ID; a unique number denoting your connection to the machine. TIME refers to the amount of CPU time in minutes and seconds that your process has consumed.

Listing other processes

The ps command can take options to examine a single process by PID, or a particular user's processes, or all processes attached to a particular terminal.

```
bash$ ps -fp 18569
bash$ ps -fu stella
bash$ ps -ft pts/302
```

PRACTICAL EXERCISES

Examine your processes

- Use `ps` to take a look at which processes you are running. The `-f` option generates a full listing.

```
bash$ ps -f
     UID   PID  PPID  C    STIME TTY      TIME CMD
  stella 19348 19282  0   Apr 06 pts/66   0:02 -bash
```

You are already familiar with the PID, TTY and TIME fields. The other fields here are:

UID	The user ID which owns the process
PPID	The PID of the parent of this process
C	Processor utilization for scheduling - ignore this
STIME	The starting time of the process
CMD	The name of the command being executed

Snoop on other people

- Look at who is logged in using the `who` command.

```
bash$ who | more
```

- Pick some users and examine their processes. Pick somebody else's terminal and examine the processes attached to it.
- Have a look at all processes running on the system.

```
bash$ ps -ef
```

FOREGROUND AND BACKGROUND

- Pressing RETURN starts a shell job
  - single process or sequence/pipe of processes
- Foreground
  - UNIX executes your job while you wait
  - you get a new prompt once the job is finished
- Background
  - UNIX executes your job while you get on with other work
  - you get a new prompt immediately
  - the shell will inform you when the background job finishes

Jobs

When you press RETURN to end a line of command input, you start a *job*. This could be a single process, or a sequence of processes joined with metacharacters.

Foreground

When you start a job normally, you are executing it in the *foreground*. This means the shell will follow your instructions and it will not offer you the shell prompt again until your command has terminated. Because you are not offered another prompt, you can never have more than one program running in the foreground.

Background

```
bash$ sort names > sorted &
```

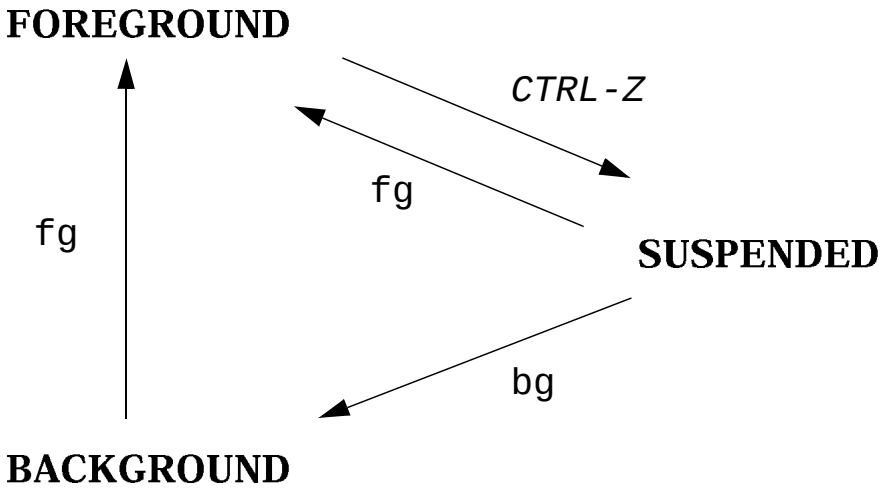
An ampersand (&) appended to a command is a request for that job to be run in the *background.* The shell will begin the job, but will then immediately offer you another command prompt so you may start another job. You may start as many jobs in the background as you want, within the bounds of common sense and your system administrator's generosity.

The shell will notify you when a background job has terminated. If you have the shell variable `notify` (must be lower case) set to any value, it will inform you immediately. Otherwise the shell will wait until it outputs a prompt. So if you are in the middle of a long editing session, you will not be informed until you quit from the editor.

Note for pensioners

The term "job" has its roots in computing history, when a single machine filling an entire room used to perform simple tasks for an entire company. Tasks for the computer would be submitted on punched cards which would be fed in by the computing staff. These were known as "batch jobs" because they were submitted in batches.

JOB AND PROCESS CONTROL



The shell offers you a mechanism to control the jobs and processes you start. The diagram above illustrates this. Except where noted below, processes can be manipulated in a similar way to jobs.

Suspension

A job can be running in the foreground or background, or it can have terminated. There is one more state it can be in - *suspended*. The computer stops work on the job, although this can be resumed at the point it was halted. Thus many hours of high-powered processing need not be lost if a job needs to be temporarily held back.

Viewing current jobs

You can view the jobs you are currently running with the `jobs` command.

```
bash$ jobs
[1]   Running                 ( sleep 600; echo Wake up! ) &
[3]+  Stopped (user)          netscape
[4]-  Terminated              xpostit &
```

The number in square brackets indicates the *job number*. You use this when referring to the job. The next column reports on the *status* of that job. Last on the line is the *description* of the job itself. Notice that background jobs are indicated by an ampersand.

The job number is sometimes followed by a single character. A plus sign (`+`) indicates the current job, and a minus sign (`-`) indicates the previous job.

In this case job 4 has just terminated, and job 3 has been suspended by the user. Job 1 is still happily running in the background

`jobs -l` will print the process ID as well as the job number.

```
bash$ jobs -l
.[1]+   21642 Running                 sleep 60 &
```

Referring to a job

You can refer to a job explicitly with its job number. Prefix the number with a `%` to indicate this is a job number. There are also some shortcuts.

`%`, `%%`, `%+` or even no job number at all can be used to refer to the current job.

`%-` can be used to refer to the previous job.

Foreground -> Suspended

To suspend a job running in the foreground, type CTRL-Z. The job will be suspended and you will be offered a new prompt for more commands.

Suspended -> Foreground

To bring a job which has been suspended into the foreground, use the `fg` command.

```
bash$ fg %3
```

This brings job 3 into the foreground. You will not be offered a new prompt until job 3 terminates.

Suspended -> Background

The command `bg` moves a job into the background.

```
bash$ bg
```

This moves the current job into the background. Note that no job number need be specified when referring to the current job.

Background -> Foreground

The `fg` command can also be used to bring a background job into the foreground.

```
bash$ fg %2
```

This brings job 2 into the foreground.

Controlling processes

You can manipulate processes directly in this way also. Use process IDs and omit the `%` sign.

```
bash$ bg 12456
```

This puts process 12456 into the background.

Tech note

A shell job could be one UNIX process, or a sequence of processes, or a number of processes connected by a pipe.

KILLING OFF JOBS AND PROCESSES

- You can terminate jobs/processes by hand with `kill`
  - sends a signal to ask job/process to shut down
  - various levels of force
  - A `kill -KILL` should terminate anything but may not be clean
- `kill` can operate on a job number
  - use `jobs` command to list current jobs
- `kill` can also operate on a PID
  - use `ps` command to list current processes
- You can only `kill` your own jobs/processes!

If a program you started is taking an unreasonable length of time, or seems to be misbehaving, you may want to terminate it. UNIX provides you with the `kill` command for this purpose. `kill` sends a signal to the job you specify, instructing it to stop work.

```
bash$ kill %4
```

This instructs job number 4 to cease. The job can close any open files and send any relevant messages in order to terminate in a "clean" fashion. Thus it may not die immediately. The signal being sent is -TERM for "terminate".

```
bash$ kill -HUP %4
```

Should the job not respond to a standard kill, you can ask a little more forcefully using the -HUP option. The job should still perform the necessary housekeeping to terminate cleanly. This depends on the programmer doing their job properly. The abbreviation comes from "hang up".

```
bash$ kill -KILL
```

If a particularly stubborn job is not responding to a kill -HUP, your final recourse is the -KILL option (also known as kill -9). This should terminate any process but may not do it cleanly, with files left in unsuitable states and no error messages issued. Thus it should only be used in a last resort.

Killing processes

You can send these signals directly to processes also. In this case you need to specify the PID and omit the % job identifier.

```
bash$ kill -HUP 18569
```

This sends a HUP to process 18569.

Pacifist note

A range of other signals can be used to communicate with processes in less destructive ways. `kill -l` will list others.

SUMMARY OF DEEPER UNIX

- The shell calls commands; it doesn't understand arguments
  - but arguments are parsed for unquoted metacharacters
- PATH is searched in order for executable commands
- Aliases and scripts simplify your favourite commands
- The shell maintains your environment.
- The environment is passed to subshells
  - but not back again!
- Multiple processes are identified by process ids
- Jobs can be switched between foreground and background
  - `kill` sends a signal to a job or process

Chapter 2

Power UNIX

REGULAR EXPRESSIONS

- Regular Expressions - describe text in a general way
- Usually normal alphanumeric strings + special metacharacters
  - these are not the same as shell metacharacters
- Available to many UNIX commands
- Trust us - this stuff is useful !

Regular expressions are a way of describing strings of text. They are used in many UNIX utilities to select, replace and discard text. Regular expressions are themselves simply text strings which usually include special characters to impart extra meaning. These so-called metacharacters seem to resemble the shell metacharacters we met in Deeper UNIX, however there are subtle but important differences!

UNIX editors, search utilities, programming languages, news readers, pagers and web search engines all use regular expressions. Understanding regular expressions can help a great deal in handling all types of data such as mail messages, data files, log files etc...

This book will introduce simple regular expressions and will look at several UNIX utilities which make use of them.

Jargon

Regular expressions are often referred to as patterns or regexps. Pattern matching refers to searching a file or data stream for a regular expression

MicroEmacs and Regular Expressions

Regular expression matching can be enabled in MicroEmacs by adding

```
add-global-mode magic
```

to its startup file `.emacsrc`

Geek Note

Regular expressions were a sexy topic with 1940's mathematicians. Stephen Kleene formalised the work of neuro-physiologists' model of the nervous system using algebraic regular sets. The notation he used to describe these regular sets was named Regular Expressions.

Thankfully, we needn't concern ourselves with the mathematical basis of this!

REGEXPS - LITERALS AND ANCHORS

- grep - searches for lines containing a regexp
  ```
  grep regexp file
  ```
- Literal strings are simple regexps
  ```
  grep celtic results
  ```
- First special characters - anchors
  - ^ = start of line
    ```
    grep '^celtic' results
    ```
  - $ = end of line
- `grep 'hearts$' results`
- regexps are case sensitive
- Always enclose regexps in single quotes on command line

---

grep

`grep` searches for the regular expression in the named file (more later !).

Literal Strings

Regexps which contain no special characters are referred to as literal strings.The command line

```
bash$ grep Celtic results
```

would print out each line in the file which contained the string `Celtic.`

Anchors - ^ and $

^ (called a caret) as the first character of a regular expression matches the start of a line. So

```
bash$ grep '^Celtic' results
```

would printout any line from the file `results` which started with the string `Celtic`

$ (called a dollar) as the last character of a regexp matches the end of a line. So, similar to the example above,

```
bash$ grep 'Hearts$' results
```

would show all lines which ended with `Hearts`.

These metacharacters anchor the literal string to the start or end of the line.

Case Sensitive

As with most things in the UNIX world, regular expressions are case sensitive.

Quoting

Special characters used in regular expressions are usually meaningful to the shell as well, so always enclose regexps in single quotes on the command line.

PRACTICAL EXERCISES

Checking the pools

The file `results` contains some football results from the Scottish Premier League. Teams which played at home appear first on the line whilst their opponents, who played away from home, appear second.

- Use `grep` with a suitable regular expression to select:

* all matches played by Hibernian

* all home matches played by Aberdeen

* all away matches played by Dunfermline

* all home matches played by Hibernian against Kilmarnock (Hint: you can join two greps together using a pipe).

REGEXPS - CHARACTER CLASSES

- . will match any single printable character
- [ ] to match a **single** character in the specified list
  - [xyz] to match a single x, y or z
  - [a-z] to match a single lower case character
  - [a-zA-Z] to match any single alphabetic character
- Use one character class for **each** character position.
  - [AB][0-6] to match a standard paper size, eg A4
- ^ to match a character class
  - [^a-z] would match any character which wasn't a lower case alphabetic

---

Character classes allow you to match a single character from a specified list

Dot

The dot character is a special character class which matches any single character

[and]

More usefully a list of characters can be specified using square brackets. This will match any **single** occurrence of one of the specified characters. A range may be specified using a hyphen and multiple ranges can be specified in the one character class. Indeed ranges and literal character classes can be combined, for example ...

[a-z] would match any one lower case alphabetic character

[a-zA-Z] would match any one alphabetic character

[a-zA-Z,!] would match a comma, an exclamation mark or any one alphabetic character

^

If the first character of a character class is a ^ (caret) it will match any character not in the specified list. So:

[^0-9] would match any character which wasn't a number.

Be careful not to confuse this with using a caret to match the start of a line.

PRACTICAL EXERCISES

- Use grep to select matches from the results file in which more than 10 goals were scored. Hint: think of how to describe a two digit number.
- Find those games in which the home team's name began with the letters A, H or from M to R inclusive.
- Modify the regexp you used above to select all fixtures except those in which the home team's name began with the letters A, H or from M to R inclusive. Hint: there is a clumsy way to do it and a neat way to do it!

REGEXPS - THE * QUANTIFIER AND \

- * for zero or more matches
  - universally recognised by regular expression utilities.
- Any metacharacter can be protected by \

```
bash$ grep '\$[1-9][0-9\.]*' foreign_price.list
```

  - a literal \ therefore must be written \\
- There is a whole lot more to regexps!

Quantifiers

Quantifiers specify how often you want a particular pattern to be matched. The pattern might be a single literal character or a character class. Quantifiers come after the pattern. Unfortunately the only quantifier recognised by all regular expression utilities is the *.

*

The asterisk (often called a "star" in regexp speak) matches any number, including none, of the preceding pattern. This is subtly different to the use of the * wildcard in shell pattern matching. The following two commands produce equivalent output:

```
bash$ ls [A-Z]*
bash$ ls | grep '[A-Z].*'
```

They both select filenames starting with a capital letter followed by any printable character. Note that the regular expression contains a dot (.) before the * quantifier. Without this dot, only filenames composed completely of capital letters would be matched.

*Escaping metacharacters with *

If you want a metacharacter in a regexp to have its literal meaning rather than its special meaning, then prefix it with \. This regexp selects prices in dollars. Note that both the currency symbol ($) and the decimal point (.) are protected:

```
bash$ grep '\$[1-9][0-9\.]*' foreign_price.list
```

The great beyond

The metacharacters described in this chapter will work with any regexp utility. However, many UNIX commands as well as scripting languages like perl will recognise even more special symbols allowing very powerful and precise pattern matching to be performed.

MORE ON GREP

- Searches files (or standard input) for regexp
  `grep` *`flags regexp file1 file2 ...`*
- Often used as a filter
  `command | grep pattern`
- Common flags
  - -v to match lines not containing regexp
    `grep -v regexp file`
  - -i for case independent matching
    `grep -i regexp file`
  - -l to list the file containing the regexp
    `grep -l regexp *`

As discussed, `grep` is a search utility which prints out any lines within a named file which contain the given pattern. If no file is specified on the command line `grep` will read its standard input (usually the output from the previous command joined by a pipe).

There are a number of `grep` commands available (eg `grep, egrep, fgrep, gnugrep`); some are general purpose search utilities whilst others serve a specific task. `egrep` is the most powerful member of the grep family as it is always available and implements full pattern matching, including metacharacters not covered in this course.

Filters

`grep` can be used to filter the output of a data stream (pipe). See later how `grep` and many other UNIX commands can be used in this way.

Flags

-v matches lines not containing the expression

```
grep -v Smithers names
```

-i specifies a case independent match

```
grep -i smithers names
```

-l lists the filenames which contain the pattern

```
grep -l smithers *
```

Flags can be joined together so

```
grep -i -l smithers *
```

lists all files in the current directory which contain the string "smithers" regardless of case.

PRACTICAL EXERCISES

- ❑ You (should) have already searched the results file for games in which one side scored 10 or more goals. How could you select only those games in which neither side scored more than nine goals?
- ❑ Which files contain the string "start"? Remember the word could be at the start of a sentence.

SED

- `sed` is a stream editor
- writes its result to stdout
- Syntax ....

  `sed` *`options address,address function args file ..`*
- Only the function is mandatory, eg

  `command | sed 1d > file`

sed

`sed` is a stream editor. This means it reads its standard input or a named file, applies the specified editing commands and writes the result to the standard output. `sed` is useful for modifying text non-interactively, such as in shell scripts, as part of a pipeline of commands or when the same edit needs to be applied to a number of files.

Each instruction takes the form of an address and a function. sed reads its input one line at a time and applies the function to the line if it matches the given address. If the address does not match the line is written to the standard output unaltered. Options may be given to `sed` to modify its behaviour and flags used to modify a function's action.

Options and addresses are not mandatory but `sed` must always be supplied with a function. When an address is omitted the whole input is selected. If no file is specified sed reads its standard input and so acts as a filter.

Saving sed results to a file

When `sed` acts on an input file it leaves its contents unaltered and writes the result of the edit to standard output. Be careful not to save the result of the edit to the input file as shown in the first example - this always leaves an empty file due to the way which the shell handles file redirection. If the result of the edit needs to be written to the input file then redirect the sed output to a temporary file and then rename it, or use the hold command as show.

```
sed 1d file > file                          BAD
sed 1d file > file.tmp; mv file.tmp file    GOOD
sed 1d file | hold file                     BEST
```

The hold command is not available on every UNIX system.

SED ADDRESSES

- Address can be numeric

```
sed 1d file
sed $d file
```

- or regexps enclosed in //

```
sed '/betty boo/d' chart
```

- Two addresses separated by commas choose a range

```
sed 1,10d file
sed '/^start/,/^end/d' chart
```

- Numeric and regexp addresses can be used together

```
sed '/^ignore/,$d' chart
```

`sed` addresses determine which lines the editing instructions will be applied to. addresses can be either numeric or regular expression.

Numeric addresses match the line number in the input stream, and so

```
bash$ sed 2d file
```

would write the contents of file to the standard output with the second line removed (d is the sed **delete** function).

$ is the special numeric address which matches the last line in the file.

A regular expression address will match all lines which contain that regular expression. Regular expression addresses are enclosed with slashes, so

```
bash$ sed '/splodge/d' chart
```

would write the contents of the file chart to the standard output with any lines containing the string "splodge" removed.

Ranges

Two addresses separated by a comma denote a range where the function will be applied to all lines between and including those matched. The function is applied when a line matches the first address and is executed on all lines until the second address is matched, so

```
bash$ sed '10,20d' chart
```

would omit between lines 10 and 20 inclusive in the output, and

```
bash$ sed '/^hello/,^bye/d' testfile
```

would omit any block of text from a line beginning with the string "hello" to a line beginning with the string "end" inclusively. Note that whilst numeric ranges can only match at most once (i.e. there is only one line 7 within a file), regexp ranges may match once, several times or not at all.

SED FUNCTIONS

- common functions -
  - d to delete lines
  - p to print lines
- common options
  - -e cmd
  - -f file
  - -n = don't print, always used with p function
    `sed -n '10,20p' file`

Functions

Functions specify the action which `sed` should perform. Common functions are

d delete lines

p print lines

Options

`sed` options modify the way sed behaves. Common options are

-e Introduces an editing instruction on the command line. If there is only one instruction it can be omitted, if there are several they should be preceded by a '-e'. For instance, to delete lines 1-10 and Rangers' fixtures from results:

```
bash$ sed -e '1,10d' -e '/Rangers/d' results
```

-f Specifies a file containing a list of editing instructions.

-n By default `sed` prints out each line that it reads. The -n option turns this off so that lines are only printed when specified (i.e. with the p function)

Examples

```
bash$ sed -n '1,5p' file
```

will print the first 5 lines of the file to standard output.

```
bash$ sed -n '/^hello/,/^bye/p'
```

will print any blocks of text from a line starting with the string 'hello' to a line starting with the string 'bye'.

PRACTICAL EXERCISES

Deleting with the stream editor

- In the `menu` file each section is marked with a heading. Use `sed` with the appropriate regexp addresses to delete the drinks section.

SED - SUBSTITUTION

- sed substitute function is s///
  ```
  sed 's/regexp/replacement/flags'
  ```
- the replacement string is not a regexp!
- flags -
  - g - substitute every occurrence on a line
    ```
    sed 's/Smith/Jones/g' letter
    ```

Substitute function

The substitute function changes the specified text to the given replacement. As with previous functions it can be preceded with an address match. All the text matched by the regexp is replaced, so be careful when using * as a quantifier. By default only the first substitution in a given line is preformed. The g flag can be used to specify all occurrences in that line of the regexp should be replaced.

Examples

```
bash$ sed 's/chips/sauted potatos/g' menu
```

Change all occurrences of "chips" to "sauted potatos".

```
bash$ sed '1,10s/[0-9][0-9]*/TBC/g' menu
```

Changes any number to the string "TBC" within the first ten lines of the file menu. Note that `[0-9][0-9]*` has been used to describe a number - a single digit followed by any number of digits

PRACTICAL EXERCISES

Wordplay with sed

- An Edinburgh Council directive has stated that all words starting with the letter 'c' should be replaced by the word "censored" in menus. Use `sed` to implement this directive on the menu file. (Hint: remember the first letter of a word could be upper or lower case.)
- Oh no! Someone has hacked into the UNIX2 training accounts and edited the `results` file! They have changed all Dunfermline's matches so they appear to have been played by Rangers, and all Ranger's matches so they appear to have been played by Dunfermline. Using `sed`, correct the file to its original state. (Hint: the -e option to `sed` will allow both changes to be made on the same line.)

AWK

- awk is a pattern matching programming language
- simple use is to select fields
  - recipe is

    ```
    awk '{ print $1 }'
    ```
- -Fchar to specify field separator

  ```
  awk -F: '{ print $1 "lives at" $2 }' bills
  ```
- awk can use regular expressions

  ```
  awk -F, '/^[Tt]oast/ { print $1 "costs" $NF }' menu
  ```

---

awk

awk is a fairly complex pattern matching programming language. It has largely been superceded by perl but can still be useful for performing simple tasks on the command line.

Column selection recipe

awk is commonly used to select fields from an input stream. awk, like many UNIX utilities, acts as a filter. awk has a print command to write out text and holds columns in the input stream in simple variables ; $1 contains the first field, $2 the second etc... The last column can also be referred to as $NF. The field separator is usually white space (i.e. space or tabs) but this can be altered using the -F flag.

Examples

```
bash$ who | awk '{ print $1 }'
```

prints out the first column from the output of the who command, i.e. a list of all users logged into the system.

```
bash$ awk -F: '{ print $1 "lives at" $2 }' bills
```

sets the field separator to a colon to print names and addresses from bills.

Regular expressions

awk incorporates support for regular expressions. If a regular expression is specified, awk will only operate on lines which contain a string matching that regexp. In this way, awk can select on both rows and columns. The regular expression is enclosed in //, and precedes the awk action.

```
bash$ who | awk '/erey[0-9]*/ {print $1 is on UNIX2}'
```

For more complicated regexps, use a pipe from grep:

```
bash$ who | grep -v 'hacker' | awk '{print $1 is OK}'
```

PRACTICAL EXERCISES

Pattern Matching

- Use `grep` to select all items from the menu file which cost 10.99 and use `awk` to print out only the names of these dishes (hint: the comma is the field separator).
- How could the same result be achieved using a single awk command?

FINDING FILES

- `find` searches through a directory tree for files or directories which match its expression

  ```
  find directory expression action
  ```

- Common expressions
  - -name 'shellpattern
  - -mtime N
  - -type c
  - -size N

  ```
  find src -name '*.pl' -print
  ```

find

`find` is our friend. It is a very powerful command for searching through a directory structure for files which match a given criteria. It is also a fairly complex command. Simplified, its syntax is

```
find directory expression action
```

where `directory` is the directory name where `find` will search down from, `expression` is the criteria for file selection and `action` is the event to be performed on a file selection.

Expressions

When expressions specify a number N, +N means more than N, N means exactly N and -N means less than N.

Common expressions are

`-name 'shellpattern'` to select files whose names match the shell pattern, the pattern should be enclosed in single quotes to prevent its expansion

`-mtime N` to select files modified in the given time period, eg -7 would select file modified within the past seven days.

`-type c` to select files of the given type, where c can be f for file, d for directory or l for link

`-size N` to select files of a given size. The number N can be followed by a "c" to describe a number of characters, so

```
bash$ find src -size +1000000c -print
```

would display the names of all files underneath the src directory whose size is greater than 1000000 characters.

FIND ACTIONS

- common action is to print, via `-print`
- Instead of printing the matched file, find can execute a command

```
find . -type f -exec grep -l "Spain" {} \;
```

- {} is a placeholder for each file matched by find.
- \; ends the command to execute

Find actions

The most common action performed on selected files is simply to print out their name using the `-print` action, so ...

```
bash$ find src -name '*.c' -print
```

would display all files under the src directory whose names ended in ".c" (i.e. c source files).

-exec

The `-exec` action can be used to execute a given command on file selection. An escaped semicolon must be used to end the action and {} is replaced by the name of the current file.

A very useful recipe employing `-exec` is

```
bash$ find dir -exec grep -l 'regexp' {} \;
```

which prints the names of files underneath directory `dir` which contain a pattern matching `regexp`

More find

Find provides many more features than are covered here, interested students can browse the find manual page for more options.

PRACTICAL EXERCISES

- Use `find` with a `grep` action to print the name of all files under the addresses tree which contain people called Stavros.

SORT

- sort, unsurprisingly, sorts files !
  - By default it sorts on the first column alphabetically
  - Can choose the column (numbered from 0)
    `sort +2 -3 file`
  - would sort on the 3rd column, ignoring the 4th onwards.
- -n option specifies a numeric sort
- -tchar specifies an alternate column separator
  `sort -t: -n +2 bills`
- sort is also a filter
  `who | sort`

---

sort

The `sort` command unsurprisingly sorts files! If no file is specified it will sort its standard input. It sorts the lines of text on the named column and if no column is specified it uses the first column. Columns are mysteriously numbered from 0, so the 2nd column is actually numbered as 1. By default `sort` collates alphabetically, a numeric sort is specified using the `-n` flag. The column separator is white space though this can be changed via the -t flag, which takes an argument of the new column separator. If no input file is specified `sort` will read its standard input.

Examples

```
bash$ sort +2 file1 > file2
```

alphabetically sorted file1 on the third column placing the results in file2

```
bash$ sort -t: -n +2 bills > bills.sorted
```

would numerically sort the colon separated file `bills` on the third column into the file bills.sorted.

Other options

-r reverses the order of the sort

-f case insensitive sort

-d ignore non-alphanumeric characters

-M sort by month (Jan before Feb before Mar.....)

PRACTICAL EXERCISES

- Use `grep` to select only the dishes (i.e. to discard the headings) from the menu file and sort to collate the menu by price.

FILTERS

- `grep`, `sed`, `awk` and `sort` are examples of filters.
- filters take input from stdin and send output to stdout
- several filters may be joined by pipes

```
command1 datafile | command2 | command3 .......
```

- Only the first command needs an input file, eg:

```
grep '^Celtic' results | sed '/Thistle$/d'
awk '{ print $2 $NF }' menu | sort -n +1
```

Filters

Many of the commands met in this chapter have been described as "filters". A filter is any program which by default reads from standard input and writes to standard output. This means that filters can be joined end-to-end in a pipe | . As the data passes along the pipe it is modified at each stage, and the filtered output is delivered at the end. The ability to join command output using pipes is a very powerful feature of UNIX, allowing smaller commands to be used as the building blocks for larger tasks.

When several filters are joined together in a pipe, only the first command needs to have an input file specified. The others take their input from the previous filter. Compare the good and bad examples below.

Good example

Suppose you wished to search a file `cats` for lines containing the pattern 'Tom', substitute 'Tom' with 'Top', and print the 3rd and last columns of each such line in alphabetical order on the last column. You could use `grep`, `sed`, `awk` and `sort` in turn, saving the intermediate results to temporary files at each stage. However, each of these commands is a filter which uses stdin and stdout by default, so the problem can be solved using a four-stage pipe, all on one line:

```
bash$ grep 'Tom' cats | sed 's/Tom/Top/' |
                awk '{print $3 $NF}' | sort +1
```

Bad example

A common error made by people starting to use more complicated pipes is to repeat the input file, in this case **cats**, at every stage of the pipe like follows:

```
bash$ grep 'Tom' cats | sed 's/Tom/Top/' cats |
          awk '{print $3 $NF}' cats | sort +1 cats
```

This can cause unpredictable results!

PRACTICAL EXERCISES

The Celtic manager is concerned that his defence underplays during pressure fixtures against big-name opposition at home. Using the `results` file and a single command line, sort Celtic's home opponents according to the number of goals Celtic gave away and advise him whether his worries are well-founded.

Hint: Firstly find Celtic's home games; the number of goals Celtic gave away will then be in the fourth column, and their opponents' names in the last column. Finally, pipe into a suitable `sort` command.

Still using a single command line, repeat the exercise, this time excluding Hibernian from your analysis, and referring to Aberdeen as "Dons"

SUMMARY OF POWER UNIX

- Regular expressions
  - `. ^ $ [] [^] *`
- `grep [-i] [-v] [-l] regexp file`
- `sed`
  - addresses (numeric, regexp) and actions (p,d,s)
- `awk [-Fchar] '/regexp/ { action }'`
- `find dir -name shellpattern -print`
  - `find dir condition -exec command {} \;`
- `sort [-n] [-r] [+col] [-col] file`
- filters

Chapter 3

UNIX toolkit

A SWISS ARMY KNIFE OF COMMANDS

- Many commands available to users
  - Basic set available on all UNIX systems
  - Most systems make additional commands available
- How to learn about new commands
  - Other users
  - PATH environment variable
  - On-line help systems
- man command
  - `man -k` *`keyword`*
  - `man -s` *`number command`*
    `man -s 2 chmod`

The UNIX operating system has evolved over many years and includes many useful commands and utility programs. Most system administrators will also install additional programs for their users.

This book explores some of the most popular commands which are commonly available.

Finding new commands

Most UNIX users will have a few commands they use regularly. One excellent way to discover new commands is by talking to other experienced users.

The machine "holyrood" at Edinburgh University supports a subject-based help command which can respond to keywords. You can use this to discover new commands you may be interested in. For instance:

```
bash$ help printing
bash$ help compilers
```

The web-based tutorial unixhelp contains lots of information too, at:

```
http://unixhelp.ed.ac.uk/
```

Another possibility is to explore the directories listed in your PATH environment variable. You should see many commands you do not recognise and you can consult their manual pages to find out what they do.

More on man

`man -k` takes a keyword as argument, and lists commands associated with that keyword for which a man page exists. Useful when you don't know the exact command. Unfortunately you often get no matches or too many to deal with!

Some man pages have more than one section; the section number can be specified using man -s. Compare the result of `man chmod` or `man -s 1 chmod` with `man -s 2 chmod`

EXAMINING AN UNKNOWN FILE

- `file` command
  - Attempts to determine the type of a file
  - Follows symbolic links
- Tries to be clever with text files
  - Attempts to identify a programming language
  - Not foolproof

You may occasionally come across a UNIX file of unknown type. Perhaps this was a file you created some time ago or a file obtained from another user. It is not a good idea to `cat` such files to the screen as they may contain control characters which could interfere with your terminal settings.

The `file` command can be used to examine an unknown file.

```
bash$ file newfile
```

If `newfile` is plain text, the first 512 bytes are examined to see if it appears to be a known programming language. If not plain text, the file may include a *magic number* which describes its type. See the manual page for more details.

Options

`-h`	Do not follow symbolic links
`-f names`	The file `names` contains a list of filenames to be examined

Sceptic Note

`file` is not guaranteed to be accurate when examining text files. It can mistake, say, a VRML file for a program written in C.

PRACTICAL EXERCISES

*Using the **file** command*

- ❑ The directory `types` contains a few interesting files. Take a look at them and try to work out what they are.
- ❑ Try using wildcards with the `file` command.

MORE ON FILES

- `ls` has many options
  - `-d` show directory name not its content
  - `-R` recursively list subdirectories
  - `-t` sort by time stamp
  - `-r` reverse order of sort
  - `-u` use last access, not last modification (for `-l` or `-t`)
  - `-q` show non-printable characters as ?
  - `-b` show non-printable characters in octal form
- Options can be combined
  - `ls -lart`

The humble `ls` command allows you to alter its behaviour in many ways. You should already be familiar with the `-a` (list dot files also) and `-l` (list in long format) options. Others are equally useful. Here are a few more.

Handling directories

By default, if given a directory as an argument, `ls` will list the directory's contents rather than the directory itself. `-d` will prevent this behaviour. The `-R` option will list the contents of every subdirectory encountered, until it has listed the entire file tree below the target.

Sorting

Normally `ls` sorts its output by alphabetical order. The `-t` option tells it to sort by modification time, listing the most recently modified files first. You can reverse this order by combining it with the `-r` option. Notice that upper and lower case options mean different things.

The time stamp which `ls` consults to do this sorting is by default the file's last modification time. However it can be instructed to use the time of last access to the file with the `-u` option. Note that `-u` will have no visible effect unless combined with a long listing (`-l`) or a sort (`-t`).

Combining options

`ls` is a good example of how command options can work together to produce powerful behaviour. For instance:

```
bash$ ls -lart
```

This means - list all files, in long format, and show the most recently modified files last.

PRACTICAL EXERCISES

Explore your home directory

- ❑ Try listing the contents of your home directory in different ways.

  Which options to `ls` work best for you?

- ❑ If you like, set up an alias so that these options are automatically given each time.

  ```
  bash$ alias ls='ls -F'
  ```

  You can make this permanent by editing it into your shell startup files.

Odd characters in filenames

Occasionally you will come across a file which somehow has some control characters implanted in its filename. Normally `ls` does not display these. This can cause a great deal of frustration.

- ❑ The directory `oddfile` contains a text file which has such a character in its filename. Use `ls` to list the directory without any options.
- ❑ Now try to read the file using `cat`.

  Surprised? There is a character in the filename which `ls` does not display by default.

- ❑ Try an option which will display non-printable characters.
- ❑ Now devise a way to read the file. (Hint: there are several.)

CHANGING FILE TIMESTAMPS

- `touch` command
  - creates new empty files
  - or updates time of an existing file

For each file in a UNIX filesystem, two times are recorded - the time the file was last modified, and the time it was last accessed. These can be changed. You might wish to do this if your system regularly backs up any files which have been recently changed.

```
bash$ touch tiger
```

This will update both modification and access times. If `tiger` does not exist it will be created as a new empty file. The `-c` option can be used to prevent that.

```
bash$ touch -m tiger
bash$ touch -a tiger
```

These update only the modification or access time respectively.

Geek note

One further timestamp is attached to each file, or more strictly, the *i-node* which the operating system uses to keep track of the file. This records the last time the filename was changed or its permissions altered, or certain other operations performed. Users cannot alter this "i-node change time".

Geek note 2

An alternative way to create a new empty file is to use the redirect metacharacter:

```
bash$ > tiger
```

If the file `tiger` already exists its contents will be wiped. You might wish to do this when working with temporary files.

PRACTICAL EXERCISES

Be touchy

- ❑ Touch a few files. Try updating modification and access times independently.
- ❑ Create a new, empty file using `touch`.
- ❑ Create a new, empty file using the output redirection metacharacter
- ❑ Try touching a file you don't own, like `/etc/passwd`

DIFF

- `diff` displays differences between two text files

```
diff menu menu.new
3c3
< fried squid                    10.99
---
> fried squid                    11.99
```

diff

The `diff` command displays the differences per line between two text files. Lines which differ from the first file are tagged with a "<", lines from the second file with a ">". `diff` displays the line numbers of the differing lines separated by an identifier, before displaying the text. Two numbers separated by a comma denote lines between those line numbers. The identifiers are

d	the named lines exists in first file but not in the second
a	the named lines exists in the second file but not in the first file
c	the named lines exist in both files but differ

Example

```
bash$ diff numbers1 numbers2
7a8,10
> eight
> nine
> ten
```

shows that after line 7 in file numbers1 the lines between lines 8 and 10 in file numbers2 do not exist. The text is then shown preceded by a > for those lines in file numbers2.

Whodunnit

The files `who-list1` and `who-list2` contain the output of the who command before and after an infamous hacking attempt on our systems. Use `diff` to spot who logged out and who logged in.

CMP

- cmp performs a byte by byte comparison of two files
- will work with non-text files
  - No output means files are the same
  - -1 option shows byte number and differing bytes in octal

```
cmp -l data1 data2
      24   1   2
```

cmp

Whilst `diff` only allows comparison of text files the `cmp` command allows comparison of non-text (binary) files. If the two files are identical it will produce no output. If the files differ it will print the character and line position at which the difference occurs. `cmp` can be coaxed to give a bit more information with the -1 flag, on which it prints the byte position of the difference and the octal value of the differing characters, so ...

```
bash$ cmp -l data1 data2
    24   1   2
```

shows that files `data1` and `data2` differ on character 24, and that position in `data1` there is a byte of octal value 1 (a CTRL-A) whilst `data2` contains a byte of octal value 2 (a CTRL-B).

Geek Note

Other useful utilities for dealing with binary data are the octal dump commands `od` and `xod`, and the manual page for the ASCII character set (available via `man ascii`).

DISK USAGE

- `du` - summarise disk usage
  - displays in 512-byte blocks
  - `-s` display only total of named directory(ies)
  - `-a` print each file
  - `-k` report in kilobytes rather than blocks (not all UNIXs)

Where has all your disk space gone?

The `du` command displays the number of disk blocks (512 bytes each) used by the current directory. You can also give arguments - directories you wish to survey instead of the current directory.

```
bash$ ls -F
apple   cathy   old/
bash$ du
4       ./old
10      .
```

The `-s` option will stop `du` listing each subdirectory as it counts them. The `-a` option will list each file it reads.

```
bash$ du -s
10      .
bash$ du -a
2       ./cathy
2       ./apple
2       ./old/alfred
4       ./old
10      .
```

Notice that four entire disk blocks are taken up by the directory `./old` itself (not filespace taken up by its contents). These store information that the system uses to manage the directory.

`du -as` is commonly used to list files but not directories.

Pernickity Note

The `du`, `ls -l` and `quota -v` commands do not always agree about disk usage and file sizes. Files are not always stored as simple blocks of data on disk, and scattered storage may be differently. Some inconsistency can be overlooked.

PRACTICAL EXERCISES

Check your disk usage

- How many disk blocks does your home directory occupy?

  So far you have always been looking at files which you own. But what about files you do not have permission to read?

  ```
  bash$ du /var/adm
  ```

  It appears that you have no problem checking that directory. But if you use the `-r` flag, `du` will report subdirectories it cannot access.

  ```
  bash$ du -r /var/adm
  ```

  So is the figure for disk usage correct in this case?

- Try combining `du` and `sort`.

  ```
  bash$ du -sk * | sort -n
  ```

  In what circumstances could this be useful?

WORD COUNTS

- wc - Word count
  - Counts lines, words and characters

The `wc` command is a simple yet extremely useful utility when working with text files. Its default behaviour is to display the number of lines, words and characters in a text file.

```
bash$ wc story
     258    2999   16919 story
```

The file `story` has 258 lines, 2999 words and 16919 characters.

It is also possible to count each item individually

```
bash$ wc -l story
     258 story
bash$ wc -w story
    2999 story
bash$ wc -c story
   16919 story
```

Geek note

Technically `wc -c` counts the number of bytes in a file, not the number of characters. However this will almost always amount to the same thing. This is also the value given as part of the output from `ls -l`.

Nosey note

You can count how many users are logged in to your machine by combining `who` and `wc`:

```
bash$ who | wc -l
```

`head` AND `tail`

- `head`
  - displays first *n* lines of a file
- `tail`
  - slightly more sophisticated
  - `-`*n* displays last *n* lines of a file
  - +*n* displays from line *n* to end of file
  - `-f` can follow a file being written to

Displaying the beginning of a file

The `head` command displays the first 10 lines of a file by default. You can also give `head` a numerical argument to specify how many lines you want displayed.

```
bash$ head -40 story
```

Displaying the end of a file

The `tail` command does much the same but you may specify a starting point relative to the beginning or the end of the file.

```
bash$ tail -10 story
```

displays the final 10 lines of the file. With a 158 line file,

```
bash$ tail +149 story
```

gives the same output.

If `tail` is used relative to the end of the file, its output is stored in a buffer of limited size. This means you cannot reliably use it this way for very large files.

Watching a file

At some point you may wish to monitor what is being written to a text file; perhaps logs from some application.

```
bash$ tail -f logfile
```

With this option, `tail` does not terminate upon reaching the end of `logfile`. It monitors the file for any additions, and displays these as they happen. Use CTRL-C to interrupt it when you are finished.

Geek note

`tail` can also accept an offset measured in bytes or disk blocks rather than lines of text.

CHECKING SPELLING

- Spellchecking program - `spell`
  - Draws its vocabulary from `/usr/share/lib/dict/words`
  - Displays all unrecognised words
  - Uses US spelling by default, `-b` option.

UNIX has a built-in spelling checker. When run on a text file, it generates a list of unrecognised words. Proper nouns and abbreviations are included in its standard dictionary. By default `spell` uses American, not British spelling, and so needs the `-b` option if your document is intended for a UK readership.

```
bash$ spell -b story
Gomez
color
thier
jessie
crimeinal
Morticia
```

All of these words occur in the text and are unrecognised. Some are genuine spelling errors, others simply unfamiliar names or informal language. One is an American spelling.

Geek note

The dictionary does not include literal copies of every word `spell` can recognise. Instead it stores base words like "whisper" and `spell` creates their derivatives (e.g. "whispers", "whispered") by applying rules of grammar. You can observe this at work by using the `-v` flag.

Note for sophistical linguists

`spell` has a basic vocabulary but can be relied upon to be present in every UNIX installation. A more powerful tool named `ispell` may be available.

PRACTICAL EXERCISES

Clever text handling

- Look at the text file `moonstone`. Check it for spelling errors. Does it include any words which would be spelt differently in the United States?
- Now use the word count program to find out how many lines it has.

  At this point you can examine the first or last few lines using the appropriate command.
- Try to combine the `head` and `tail` commands to examine the section between lines 40 and 50.

  (Hint: you will have to use a pipe)
- Can you achieve the same result by using the head and tail commands the other way round?

DATE AND TIME

- `date` command
  - gives date and time
- output can be customised
  - use `date '+`*`format`*`'` eg.

```
bash$ date '+%A %b %e'
Friday Aug 15

bash$ date '+%l.%M%p %Z'
3.06PM BST
```

Telling the time

The `date` command will display the date and time. Its default format looks like this:

```
bash$ date
Fri Aug 15 15:56:38 BST 1997
```

However `date` can take an argument which indicates a preferred format for its output. This must begin with a `+` and can contain arbitrary alphanumeric characters, spaces etc. It can also contain conversion specifications which begin which a `%` symbol and print the date or time in different ways. Here are a few:

`%A`	full weekday name
`%a`	abbreviated weekday name
`%b`	abbreviated month name
`%e`	day of the month (1-31)
`%H`	hour in 24-hour clock
`%j`	day number of year (1-366)
`%M`	minutes (00-59)
`%n`	take a new line
`%S`	seconds (00-59)
`%y`	two-digit year
`%Y`	four-digit year

Consult the manual page for `strftime` to see a full list of conversion specifications.

PRACTICAL EXERCISES

Customising time and date display

- Try to display the date and time in these formats:

```
26 Aug
```

```
Tuesday 26 Aug 2000
```

```
This is day number 238 in the year
```

`16:59` *(using only one conversion specification)*

```
Tue Aug 26
17.02
```

SCHEDULING JOBS

- `at` command
  - can schedule commands to be executed at a future time
  - system administrator can restrict access to this facility

You can leave instructions for the system to execute commands at a specific time, even if you are not logged in. This uses the `at` command and the sequence of commands to be executed is known as a job.

```
bash$ at 0700 Dec 25
at> echo "Wake up - it's Christmas."
at> /usr/local/bin/openstocking
at> (press CTRL-D to indicate you are finished)
```

Your commands will be executed at the specified time and you will receive email informing you of their result.

Specifying time and date

These are all valid formats for specifying the time your commands will be executed. Consult the manual page for others.

```
bash$ at 1230 tomorrow
bash$ at 5pm Friday
bash$ at now + 1 day
```

Other options

If you have a shell script containing the commands you want executed, you can simply point the at command at that file with the `-f` option.

```
bash$ at -f myscript now + 2 minutes
```

You can also list the jobs you have already scheduled.

```
bash$ at -l
956921400.a     Fri Apr 28 12:30:00 2000
```

and delete a job from the schedule.

```
bash$ at -r 956921400.a
```

PAUSE FOR A MOMENT

- `sleep`
  - Pause for a number of seconds
  - commonly used with other commands
  - or in shell scripts

There are times, particularly during shell programming, when you may wish the command interpreter to pause for some time before executing further commands. The `sleep` command takes a single argument, a number of seconds to pause for. It is often used when joining commands together in sequence.

```
bash$ clear; sleep 5; xwd
```

Here the current window is cleared, the system pauses for 5 seconds to allow for any delays in clearing the window, then a snapshot of the screen is taken using the X window dump command.

`sleep` is most commonly seen in shell scripts.

```
#!/bin/sh
#
# scaryscript - don't run this

echo "I will get you sacked in ten seconds."
sleep 10
cat secret_job_app | mail current.boss@loyalty.com
```

HOW TO TIME A TASK

- `time` command
  - How long does a task take to carry out?
  - Does not display time of day
- Returns three times in seconds
  - Elapsed real time between start and finish of job
  - User CPU time
  - System CPU time

A programmer who is interested in making their code more efficient will find the `time` command useful. It measures three different quantities: the real time the job takes to execute, and how long user and system processes have occupied the CPU for.

The CPU is the central processing unit which does the actual work in a computer. If you are not sure whether you need to know CPU timings then you probably don't.

```
bash$ time bigcommand
real        7.4
user        0.4
sys         0.5
```

Geek note

User time represents the actual time the CPU spends doing computation on your behalf. System time represents other necessary work done to process your command (disk access, memory referencing etc.)

In this example, the system time plus user time is significantly less than the real time taken. This is normal and happens because UNIX machines multitask; everyone shares the CPU's time.

CALENDAR

- `cal` command
  - prints a simple text calendar by month or year
    ```
    bash$ cal 1999
    ```
    ```
    bash$ cal 5 3000
    ```
  - displays current month if no arguments
    ```
    bash$ cal
    ```
  - years must be given in four digits - so we have a y10k problem!

UNIX will display a basic calendar in text format when requested using the `cal` command.

```
bash$ cal 1 1999
   January 1999
 S  M Tu  W Th  F  S
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Be sure to give the year in its full four-digit form. `cal 99` will work perfectly well but will give you a calendar for the year 99AD rather than the year 1999.

PRACTICAL EXERCISES

Days like these

- ❑ Is the year 2000 a leap year? 2100? 3000?
- ❑ What day does Christmas fall on this year? What about in seven years?
- ❑ What day of the week were you born on? What day of the week will you be (were you) 70 on?
- ❑ Spot anything strange about September 1752?

WATCHING FOR NEW EMAIL

- How to tell when new mail arrives
  - Use `biff`

    ```
    bash$ /usr/ucb/biff y
    ```
- Other options under X windows
  - `xmailwatcher`
  - `xbiff`

If you are waiting for an important piece of email you may want to know the moment it arrives. UNIX provides tools to alert you on the delivery of new mail.

`biff` is a tool which writes a message to your command line whenever new mail arrives. This can look messy. You turn biff on and off with the arguments y or n.

The `biff` tool was part of an older version of the UNIX operating system, so it is kept in a different binary directory, `/usr/ucb`. You can either add this to your PATH or give the full pathname of the command.

```
bash$ /usr/ucb/biff y
```

This will turn on mail reporting.

```
bash$ /usr/ucb/biff n
```

This will turn off mail reporting.

Fascinating historical note

Much of the development of the UNIX operating system was done at the University of California at Berkeley - hence the UCB in `/usr/ucb`. Legend has it that the campus dog was called Biff and that he used to bark at the postman.

WHO IS LOGGED IN?

- How to tell who is logged in
  - who command
  - best used with a pager

Who is logged on?

The who command displays a list of all users currently logged on to the same machine as you.

```
bash$ who
mjb        pts/353      Aug 14 10:13    (kojak.ucs.ed.ac.uk)
paddy      pts/572      Aug 19 15:16    (prism.ph.ed.ac.uk)
johnf      pts/300      Aug 12 11:04    (marble.epcc.ed.ac.uk)
erds08     pts/351      Aug 14 10:20    (canopus.ucs.ed.ac.uk)
jack       pts/500      Aug 19 11:17    (audsec.ucs.ed.ac.uk)
elspjrm    pts/521      Aug 19 14:43    (gfs0-033.publab.ed.ac.uk)
ntdesk     pts/423      Aug 18 15:26    (don.emwac.ed.ac.uk)
```

The user name is given first, followed by a code for the line that user has been allocated by the UNIX system. Next comes the date and time that user logged in, then the name of the machine they logged in from.

PRACTICAL EXERCISES

Mail notification

- ❑ Set up notification of new mail on your current session.
- ❑ Send yourself some mail to check it works.

Who is logged on?

- ❑ Look to see who is logged on to your machine at the moment.
- ❑ Find out which terminal you are logged into.
- ❑ How would you count how many users are logged onto your system at any particular time?

SIMPLE ARITHMETIC

- `bc`
  - Useful as a simple calculator
  - Scales up to be a simple programming language

```
bash$ bc
2+7
9
7/6
1
scale=3
7/6
1.166
a=7
b=2
a^b
49
```

Command line arithmetic

There are X-windows based calculators available for UNIX, but most calculations can be done much faster from the command line. `bc` is a versatile calculator which takes a sequence of instructions terminated by a CTRL-D.

Some instructions

`23+5`	addition
`23-5`	subtraction
`23*5`	multiplication
`23/5`	division
`23%5`	remainder
`23^5`	power
`y=7`	assign value to a variable
`scale=`*x*	calculate accurate to *x* digits after the decimal point

Further functionality

`bc` supports a good deal more functionality than mentioned here. In particular the `-l` option can be given to gain access to a maths library with sine, logarithms and other functions.

Geek note

`bc` does not actually do any calculation itself; it is a preprocessor for the `dc` command, which is a calculator with a less friendly interface.

PRACTICAL EXERCISES

Using the command-line calculator

- ❑ OK Einstein, what is.....

  2+2+3=?

  two to the power of four divided by two?

  two to the power of (four divided by two)?

  63 divided by 8 to four decimal places?

  The remainder when dividing 767546 by 33674?

  five to the power of five to the power of five?

REDIRECTING STANDARD ERROR

- Standard error can be redirected like standard output or standard input
  - Useful when making logs of sessions
  - Redirect with 2>
  - Use the syntax 2>&1 to combine with standard output
- /dev/null will throw away output
  - Standard output and standard error can be redirected to /dev/null

---

Revision

Recall that standard input and output can be redirected using the symbols < and > respectively.

```
bash$ command < inputfile > outputfile
```

Standard error

There is one further commonly used stream: *standard error*. UNIX programs normally use this stream to report any problems.

By default standard error is set to the same place as standard output - the screen. However if you redirect standard output, you will continue to receive error messages on your screen. You can send them somewhere else with the syntax 2>.

```
bash$ command > outputfile 2> errorfile
```

One commonly seen construction is 2>&1. This means "redirect standard error to the same place as standard output". You might use this while debugging a program, so you could examine its output and errors together.

```
bash$ command > outputfile 2>&1
```

/dev/null

/dev/null is a dummy file which accepts any data sent to it, but does not store it or display it on the screen. This can be used when a program produces verbose standard or error output which is not needed.

Futility note

It is possible to read standard input from /dev/null This will always work, but the input will contain no data.

INTERCEPTING PIPES

- Examine the data in the middle of a pipeline
  - `tee` command
  - writes standard input to both standard output and a file

    ```
    ... blah | blah | tee blahfile | blah | blah | ...
    ```
- Commonly used at end of pipe to monitor output
  - write output to both a file and screen

The `tee` command takes a filename as argument. It copies its standard input to both that file and to its standard output. This provides a method of looking at the state of your data at any point along a pipeline.

`tee` might be useful if you are running a job which will take some time to execute. You can direct its output to an appropriate file but also have the output sent to the screen so you can watch for problems.

```
bash myprogram | tee results
```

Using the `-a` option will append output to the file rather than overwrite it.

The `-i` option instructs `tee` to ignore interrupts.

CUSTOMISING YOUR TERMINAL

- `stty` command
  - A number of settings can be changed

Using `stty`

You can display your current terminal settings with the `-a` option:

```
bash$ stty -a
```

This shows the terminal options you have available and their current settings. You can change these settings using `stty`. For instance, modern Sun keyboards have both Del and Backspace keys. They give the sequences CTRL-? and CTRL-H respectively. Here are the commands to set Backspace to delete one letter at a time (`erase`) and Del to erase a whole word at a time (`werase`)

```
bash$ stty erase ^H
bash$ stty werase ^?
```

Some other options

`intr`	interrupt character (normally CTRL-C)
`eof`	end of file character (normally CTRL-D)
`susp`	suspend character (normally CTRL-Z)

TAR

- `tar` - originally for tape archiving
- Now commonly used to package files together
- Recipes ...
  - to create a tar file
    `tar cvf dir.tar dir`
  - to list a tar files contents
    `tar tvf dir.tar`
  - to extract a tar files contents
    `tar xvf dir.tar`

tar

`tar` was originally written for tape archiving but it is most commonly used these days for packaging files together (rather like the PC zip command). Much of the UNIX software held throughout the Internet will be contained in tar files.

Recipes

The `tar` command takes many arguments, the most common ways of using it are shown below ...

```
bash$ tar cvf distribution.tar src bin license/*
```

will create a tar file called `distribution.tar`which contains the directory trees `src` and `bin`, as well as the contents of the license directory (but not the actual license directory itself). The `c` flag above tells `tar` to create a tar file, the `v` flag instructs `tar` to be verbose and the `f` flag takes an argument to specify the name of the output file.

```
bash$ tar tvf blogs.tar
```

lists the contents of the tar file called `blogs.tar`. Here the `t` flag instructs `tar` to show us the file's table of contents

```
bash$ tar xvf flumps.tar
```

extracts the contents of the `flumps.tar` tar file. The `x` flag instructs `tar` to extract.

Miser Note

tar files do not save disk space; in fact they use slightly more than the original directory tree. However, a tar file, once created, may be compressed using the standard UNIX `compress` command, or on some systems the GNU utility `gzip`. Depending on the type of data, 50-90% space saving may be achieved.

PRACTICAL EXERCISES

Tar very much

- create a new directory in your home directory, and `cd` into it. Only extract tar files from this exercise into this directory, otherwise your files may become muddled.
- create a tar file containing the practicals and examples directory trees. Use the `find` command, if you can't remember where these are.
- list the files contained in the tar file. Compare them with a recursive directory listing of the practicals and examples directory trees.
- extract the tar file into your new directory. Use the `ls` and `du` commands to compare the sizes of the tar file and the extracted directory trees. How much space has the tar file saved/wasted compared to the original directories?

SUMMARY OF UNIX TOOLKIT

- `man [-k]` and `help`
- `file`, `ls [-dRtruqb]`, `touch`, `diff`, `cmp`
- `du [-k] [-a|-s]`
- `wc {-c|-w|-l}`, `head [{+|-}n]` and `tail [-f|{+|-}n]`
- `spell [-b]`
- `date`, `cal`, `at [-l|-r]`, `time`
- `sleep`
- `biff {-y|-n}`
- `who`
- `stty`
- redirection: 2>, 2>&1 and `tee`
- `tar`

Feedback

Please let us know if there is anything in this document which you didn't find clear, or which we have omitted, so that we may improve future editions. Please send your comments (on a photocopy of this page if you like) to:

G.Chetty
Information Services Section
Computing Services,
Edinburgh University,
Main Library,
George Square,
Edinburgh EH8 9LJ

or by email to: G.Chetty@ed.ac.uk

Document: **UNIX 2: Enhancing your UNIX skills**

Edition: **2.0 August 2000**

Your name and address:

Comments: