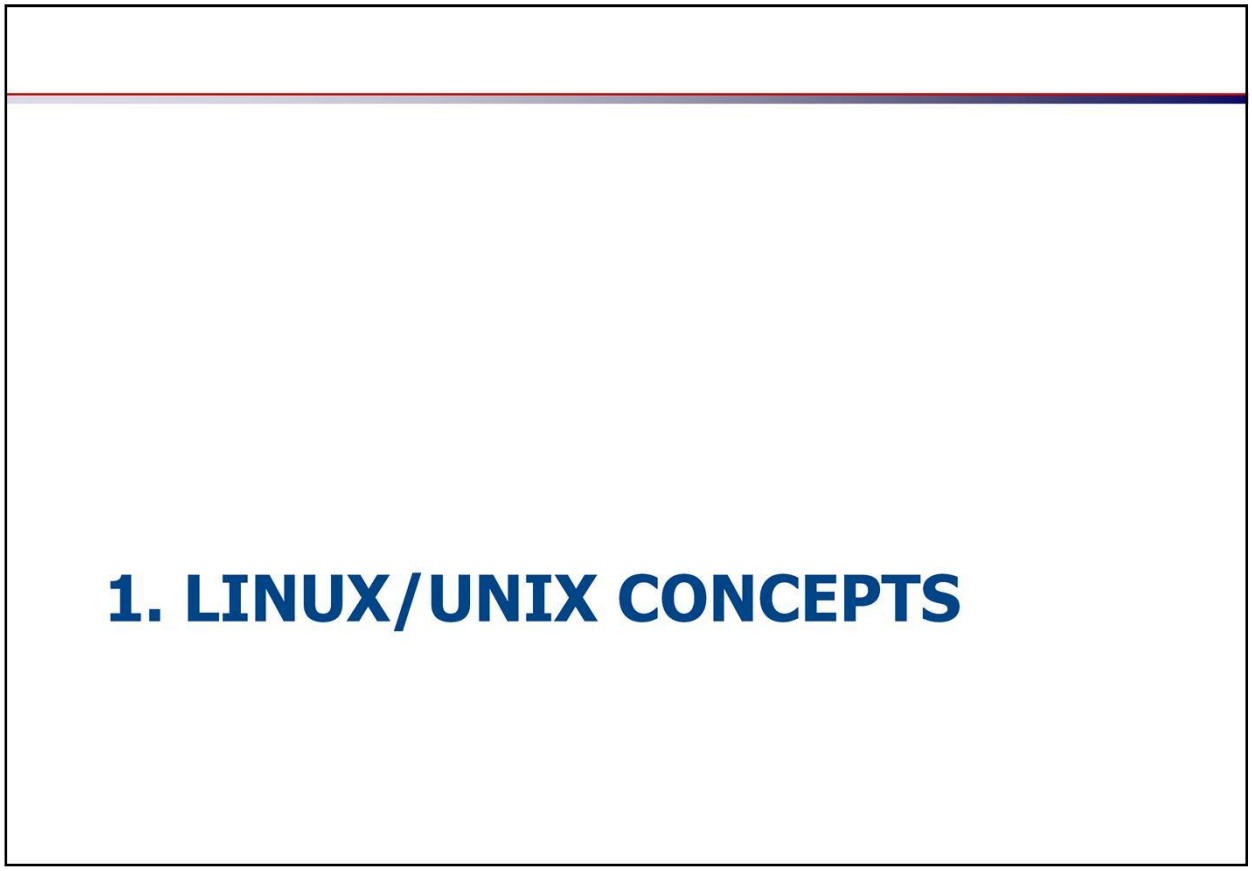
Unix as a Platform

Objectives

At the end of this course you will be able to

- Describe the architecture and basic philosophy of the Unix/Linux family of operating systems
- Appreciate the importance of Unix/Linux as a server side platform
- Construct and manage a hierarchical structure of files and directories
- Protect files and directories using standard Unix protection mechanisms
- Manipulate text files using standard Unix utilities and editors
- Utilize I/O redirection and pipes to perform more complex operations
- Interrogate and manipulate processes
- Use the online manual for reference and help
- Customize the interactive environment using shell variables and configuration files
- Automate common tasks by building shell programs

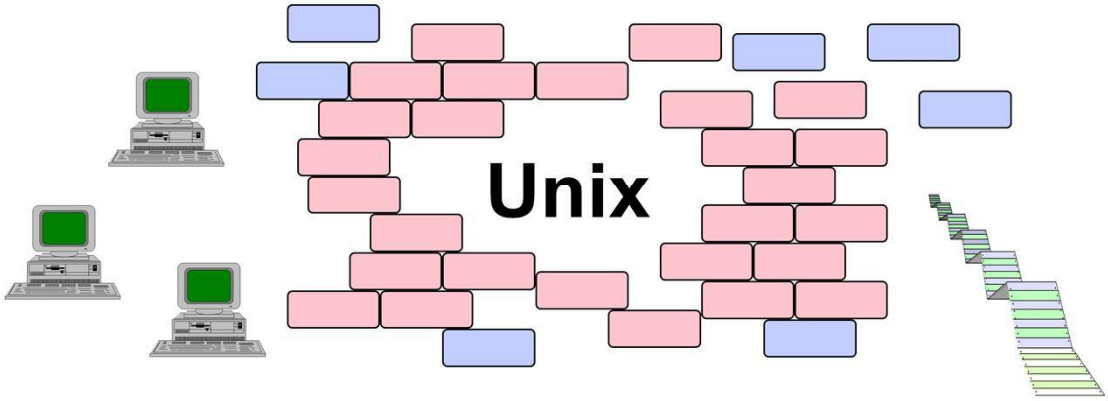

1. LINUX/UNIX CONCEPTS

Linux/Unix Concepts

- What is Unix?
- Major Features
- Potted History
- Free Software Foundation
- Linux and GNU
- Commercial Linux
- Files & Processes
- Login Files & Processes
- Unix Structure
- Unix Kernel Responsibilities
- Linux Systems Today

What is Unix?

- **Multi-user, multi-process, multi-access operating system**



- **Allows users to run programs, manage their own files and use devices**

Unix™ is a multi-user, multi-process, multi-access operating system. This means that it can support multiple simultaneous users, each executing multiple programs.

Unix provides an operating environment for users to run programs, manage files, accessing devices, communicate with each other and co-ordinate their activities.

Unix is commonly used in networking environments, allowing data and resources to be shared amongst the connected machines.

Major Features

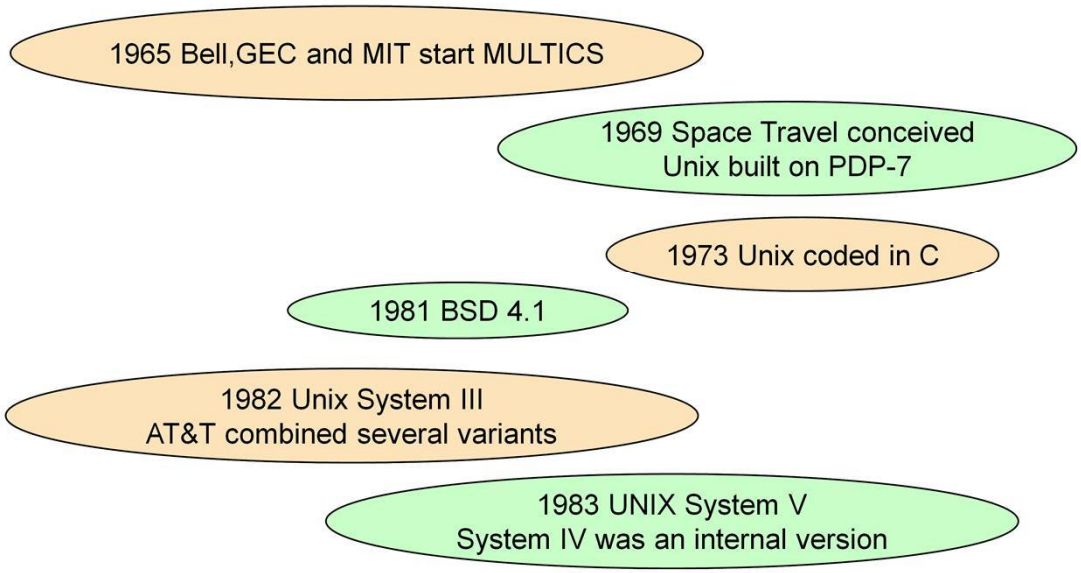
- Simple, powerful, user interface
- Complex commands are made from simple ones
- Hierarchical file system
- Consistent file format, the byte stream
- Simple, consistent, peripheral interface
- Hides machine architecture from user

Unix provides hundreds of commands each designed to do one thing well. Through a Unix shell (command line interpreter) collections of such commands are combined to perform complex tasks.

In Unix, files on disk, devices and the input and output of running programs are considered files. All physical devices have filenames, and behave as ordinary files.

The fundamental component of information in Unix is the byte stream. It allows files, devices and even programs to be used interchangeably as the source or destination of data; and thus allows the underlying machine architecture to be hidden from the user.

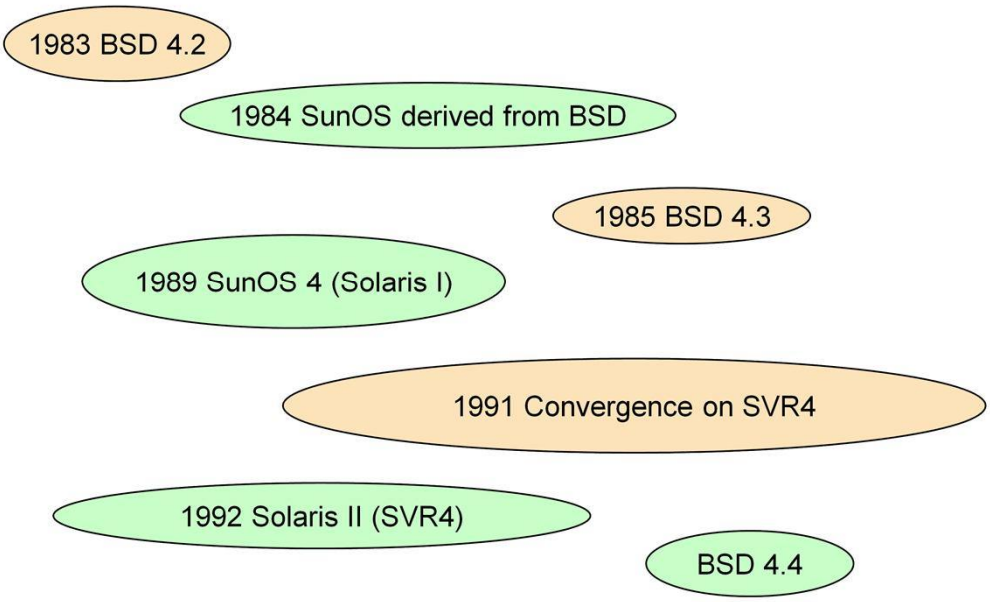
Potted History



Unix started life as the support environment for a Space Traveller game developed at Bell Laboratories. It was originally coded in PDP assembler, some parts were developed in B (an interpreted language influenced by BCPL) and later the entire system was re-coded in C. C was developed by Dennis Ritchie for Unix to overcome the performance limitations of B. Unix was named by Brian Kernighan as a pun on an earlier system called 'Multics'.

Once Unix had become a stable product within Bell Laboratories, consideration was given to selling it. However, due to a Consent Decree Bell had signed with the US Federal government in 1956, it was forbidden to market computer products. Instead, Bell laboratories (specifically, the Unix Systems Group of AT&T) gave the system to Universities for educational purposes. Research at the University of California at Berkeley lead to the development of a variant of the Unix system. Most commercial flavours of Unix are based upon AT&T system V Unix, or BSD (Berkeley Software Distribution) 4.x Unix.

Potted History



SunOS™ is Sun Microsystems flavour of Unix. It is based largely upon BSD Unix, with NFS and NIS network extensions to allow files to be shared and managed around a network.

In the interest of standardisation, Sun Microsystems, together with a number of other major workstation manufacturers, have converged on SVR4 Unix. System V Release 4 incorporates many of the features of earlier System V versions of Unix, in addition to many other facilities. Sun's SVR4 product is Solaris II, and all preceding SunOS products are now referred to as Solaris I.

Commercial Unix – the Unix Wars

- **A.T.&T. took stake in Sun**
  - to help fund future development
  - System V Release 4
- **Other companies formed Open Software Foundation**
  - IBM, Hewlett Packard, DEC, etc.
  - to build equivalent system
- **Emergence of Windows NT as potential rival forced a settlement of the feud**

As Sun developed as a force in the Unix world, AT&T acquired a stake in the company with the aim of funding future development of the platform. Other companies such as SCO joined in a project to develop a single "unified" version of Unix that incorporated the best of AT&T's system V, Sun's BSD based SunOS and SCO's Xenix (which they had acquired from Microsoft...)

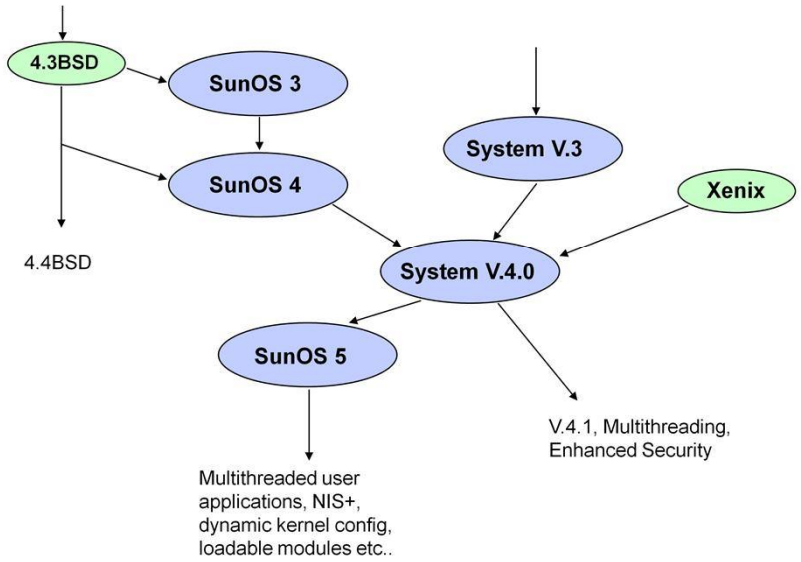
Other companies who were beginning to see the commercial value of Unix systems became concerned at this development, fearing that prohibitive licensing costs would prevent them from taking advantage of the new moves to Unix. They formed an alliance known as the Open Software Foundation and set out to develop a "rival" system.

In reaction to this, Sun and AT&T formed Unix International and the "Unix Wars" began.

Much vitriol was exchanged between the two camps, before a realisation that the emergence of Windows NT from Microsoft actually presented a real threat to both camps. An uneasy truce developed, which then moved towards "peace" as the disparate companies attempted to unite against the "common enemy".

Out of this period, the most notable developments to appear in the Unix software world were System V Release 4, the result of the original project within Unix International, and Motif, a GUI toolkit based on work done by the Open Software Foundation.

Solaris



Solaris is Sun Microsystems' implementation of Unix. It has developed from the System V Release 4 (SVR4) version of UNIX. SVR4 was a joint project carried out largely by Sun Microsystems and AT&T as a way of unifying the different versions of UNIX that were being used throughout the industry.

The aim was to define a common code base from which "proprietary" versions of UNIX could develop, sharing a well defined set of functionality.

The first result of the project was SVR4.0, which Sun took as their baseline to develop SunOS 5.0, the operating system component of what was to become Solaris 2. Because of the need to support multiprocessor hardware, Sun made a number of changes to the kernel architecture at SVR4.0, adding support for multi-threading and multiprocessor systems amongst other things. A number of additional user level facilities were also added, such as the network naming service NIS+ and the OpenWindows GUI.

Sun has developed subsequent versions of Solaris 2.x from this baseline, not from any subsequent developments in SVR4. In addition, Sun removed certain features of SVR4 from Solaris 2, such as the System V file system, Xenix file system support and the SVR4 boot file system.

The current version is known as Solaris 10 - a highly sophisticated scalable operating system that runs on a range of hardware, both Intel and SPARC based, from small desktop and even laptop systems to large, mainframe class platforms that support up to 100 CPUs.

The Free Software Foundation

- **Founded by Richard Stallman**
  - original Emacs developer
- **Aim was to build a complete Unix like system**
  - GNU (Gnu's Not Unix)
  - to run Unix software but not "be" Unix
- **Developed innovative licensing strategy**
  - software available free of charge
  - source code to be made available
  - redistribution without cost allowed
  - users/developers free to fix bugs and make improvements

In 1983, a developer called Richard Stallman posted a message to the Usenet newsgroup net.unix-wizards, announcing that he was going to build a Unix compatible operating system that would be distributed free of charge. The system was to be called GNU (for GNU's Not Unix) and would be able to run applications that ran on the traditional versions of Unix.

Stallman was known primarily for his work in developing the original Emacs text editor, but since then has been the prime mover in the world of "free" software. He cites "freedom" as having four components

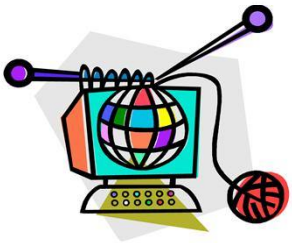
> the freedom to use software (i.e. software should be available without cost)
>
> the freedom to examine how software works (i.e. source code should be available)
>
> the freedom to redistribute software (either as a package or part of a larger system)
>
> the freedom to improve software (by fixing bugs or including enhancements)

To this end software developed by the GNU project is distributed under a rather different licensing agreement to other software, sometimes known as the GPL (GNU Public License). Variants on this include the LGPL (Lesser GPL). GNU software even includes a "copyleft" statement – the antithesis of the copyright!!

The first software to appear from the project was a C compiler, known as gcc, which is very highly regarded, to the extent that it is used extensively even today. A Shell, and copies of the more commonly used Unix command line utilities followed. These products could be implemented/ported onto all of the commercial Unix systems and indeed many were used in preference to the built-in software on these systems.

Minix

- **Written by Andrew Tannenbaum**
  - accompanied textbook
  - "Operating System Design and Implementation"
- **Small "Unix-like" kernel**
  - distributed with source code
  - teaching aid for university course
  - based on Intel x86 architecture



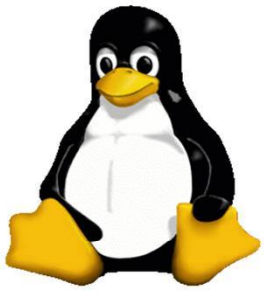
- **Back to Unix "basics"**
  - emphasis on simplicity
  - nothing spurious included
  - single floppy disk...

In the mid 1980s, Professor Andrew Tannenbaum published a textbook on operating system design and implementation, using for illustration a simple Unix-like kernel clone that he had written. The book contained the entire source code for this OS, which he called Minix, and the book also included a floppy disk containing the code, so that interested readers could build the system for themselves. Minix was designed to operate on the Intel x8 architecture, so it was able to take advantage of the growing numbers of personal computers that were appearing.

Minix was designed to illustrate the best principles of Unix as it had been at the beginning of its life. Simplicity was the key to its design, nothing was included unless it was definitely needed. It could be argued that Minix was a reaction to the perceived "bloating" of Unix as it was exploited as a commercial product.

Linus Thorvalds

- **Student at University of Helsinki**
- **Began a project to implement Unix like system for Intel386**
  - inspired by Minix
  - more functionality
- **Initial announcement in October 1991**
  - Usenet newsgroup comp.os.minix
- **Source made available for those interested**
  - GNU utilities added to form complete system
- **Thorvalds now responsible for maintaining Linux kernel**



Inspired by Minix, a student at the University of Helsinki called Linus Thorvalds began work on a project to implement a Unix system for the Intel 386 system. His aim was to improve on Minix, which had been developed primarily as a teaching aid, and provide a system that could be used for serious work, particularly in conjunction with the work of the FSF.

His original announcement in 1991 generated a large amount of interest, and developers soon began contributing to the project by adding device drivers, and also including utilities from the GNU family to complement the kernel and produce a more complete system.

Even today, Thorvalds remains "In charge" of all Linux kernel work, although there are many more talented developers who contribute to the project at that level. Having worked for some years at the Silicon Valley startup company Transmeta, he has recently taken leave of absence to work for the Open Source Development Laboratories to supervise Linux kernel work full time.

GNU/Linux – A New Force

- **Linux kernel + GNU Utilities**
  - gave almost complete system
  - equivalent to commercial Unix
- **Graphics support provided by X Window System**
  - also free software
- **Initially available only on Intel**
  - ported to others e.g. SPARC, HP
- **Packaged systems known as "distributions"**
  - Slackware
  - Debian
  - etc...

The combination of GNU utilities and the Linux kernel proved a potent force. Demand grew very quickly in the research/hobbyist arena – Intel based PCs were now appearing in many people's homes, giving enthusiasts the possibility of using or developing software for Linux outside the office environment.

More and more people contributed to the projects, and the system grew.

Although originally intended for the Intel platform, it was not long before Linux was ported to others, including the SPARC platform of Sun, PA-RISC of HP and Alpha of DEC. These were never so popular, however, and the main emphasis remained on the Intel based PC.

While the initial Linux systems had been, like early Unix systems, command line oriented, it was not long before support for Graphics – including GUIs – was added via the X Window System. This software had always been distributed free of charge, although there were some issues with the Motif GUI from the Open Software Foundation that led to alternatives being developed.

Distribution of Linux systems was coordinated through a few Internet related points, with popular early distributions being provided by Slackware and Debian (usually for the cost of the media). More "distributions" developed as the software grew in popularity.

Commercial Linux

- **Linux is free software**
  - possible to charge for "services"
  - support, documentation, etc.
- **Red Hat**
  - main player in US
- **SuSE**
  - Now owned by Novell
- **IBM**

Although extremely popular with hobbyists and "hackers", Linux had little or no acceptance in the commercial world. Companies were reluctant to utilise "amateur" software in place of more tried and tested commercial solutions, despite the obvious cost benefits.

In order to try and counteract this, a number of companies were set up to distribute Linux systems. The advantages of this from a commercial perspective were geared to the fact that customers had an obvious point of contact in the event of problems. (Original users of Linux dispute the value of this, saying that any problems with Linux had the whole developer community available for support via the Internet.)

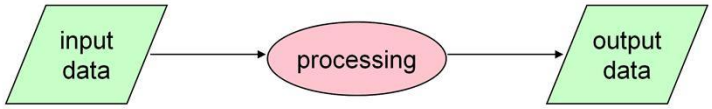
Since Linux is distributed under the terms of the GNU licenses, it is not possible for such companies to charge for the software. However they could make money by charging for services such as packaging, documentation, support and possibly additional software developed by themselves. Nevertheless Linux distributions packaged and sold in this way are generally still much cheaper than their commercial equivalents (Unix based or Windows).

An early player in this area was Caldera, formed by ex Novell staff initially to market a close of MS-DOS called DR-DOS. They added Linux products to their portfolio shortly afterwards. Caldera was never really a hit with users, and has since been taken over by SCO. RedHat and SuSE are today's best known commercial suppliers of Linux – RedHat is best known in the US and has by far the major share of the market there. SuSE began as a German company and had a much greater presence in Europe. Now owned by Novell, SuSE provides a range of packaged Linux solutions similar to those of RedHat.

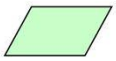
Other major manufacturers now support Linux openly today. Of these, perhaps the biggest is IBM for whom Linux is seen as a major strategic product – they have ported it to almost all of their systems. Others such as HP and Dell offer Linux.

Unix/Linux Key Concepts

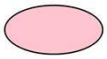
- **Computer programs process data**



- **There are only two entities in Unix**



files provide input and receive output



processes manipulate data to produce new data

The business of computer programs is to process data. To read data from some source, apply some computation, and generate result data. This model is supported directly by Unix.

In Unix there are only two entities: files and processes. Files represent the data being read or written, processes are the active entities reading, processing and writing data.

Files & Processes

- **Everything is a process or a file**
- **Files are passive entities**
  - streams of bytes stored on disks
  - interfaces to devices
  - the input and output streams of running programs
- **Processes are active entities**
  - instances of running programs
  - instructions for the CPU
- **Every process starts life as a file**
  - Unix commands are stored as program files on the disk

In Unix, everything is a process or a file. No other entities exist, not even disks, printers, terminals or networks---in Unix all of these things appear as files. The idea of using file names to represent devices saves introducing another concept. When a devices file is read or written, Unix ensures that the interactions are propagated to the particular device which the file represents.

Files are passive entities, unable in themselves to do anything. Processes are active entities, in some sense they have life. Consider a human as a process, and a suitcase as data. The suitcase cannot move itself, because it does not have life. A human must be applied to the suitcase in order for it to move. Likewise a process is applied to a file in order for it to be processed.

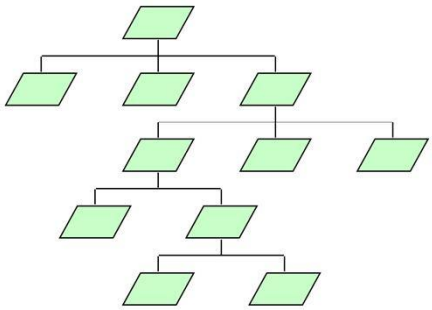
Unix is unusual compared with many operating systems in that process creation is relatively inexpensive. As a consequence, each command is executed as a single process. This differs from other systems which often run commands as procedure or function calls within a central command process.

In Unix, each invocation of a command gives rise to a new process. The process is created from the command's program file, executes and then dies. It is not unusual for one command to be executed by one or more users simultaneously, giving rise to many independent processes (instances) each performing the same task.

Note that the instructions as to what a process should do are stored in a corresponding program file on disk. When a command is issued, the text of its program file is used to make the process. The CPU interprets each instruction within the process to carry out the work.

Organisation of Files

- **Files are organised as an inverted tree**



The Unix File System consists of a single root directory which contains files.

Files may be data, programs devices or directories.

This logical file system may be composed of many physical devices and networks.

Both process and file entities in Unix are organised as trees. The tree used to hold files is called the Unix File System (UFS). The tree used to hold processes is simply called the process tree.

The Unix File System is organised as an inverted tree; the root is at the top, and branches and leaves in the form of directories and files grow down. A directory is a special file which can hold other files. Since these files may themselves be directories, a tree structure is formed.

Unix systems contain only one logical file system. The file system may span multiple partitions and disks, cross networks and exist in multiple physical forms. However, the illusion maintained by the operating system, is that the file system is one, coherent, tree. As a user moves around the file system, Unix ensures that the physical joins between disks or networks remain hidden.

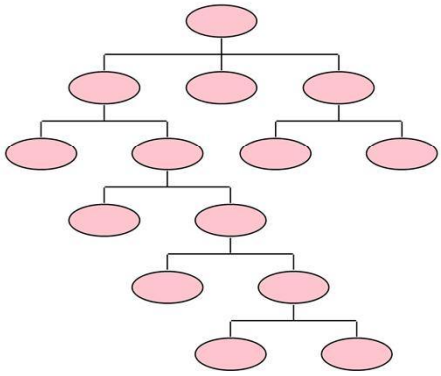
Since there is only one file system, all user files, programs and devices exist in the same name space. By convention, programs and devices exist in their own sub-directories, and users exist in their own sub-directories.

Organisation of Processes

- **Processes are organised into a process tree**

All processes have a parent process (save the first) and may have child processes.

Each command is executed as a new process, and is the child of the process which invoked it.



Unix does not discriminate between the execution of system related programs and user programs. Both give rise to processes.

The first process in a Unix system is called `init` and it runs as part of the procedure of making the machine ready for users. `init` creates child processes which set-up the machine and ultimately prompt the user to login. Once a user has logged in, a new child process is created to enable the user to enter commands. This process is called a shell.

The shell prompts the user to enter a command. For each command that the user enters, the shell spawns a new processes. The shell is therefore the parent of these processes. The shell is itself the child of `init`, since this gave rise to its creation when the user logged in. Many Unix commands also give rise to child processes, which may in turn give rise to new generations of processes. Thus a tree of processes is seen to have been spawned, starting from the great grand parent of all processes, `init`.

Login

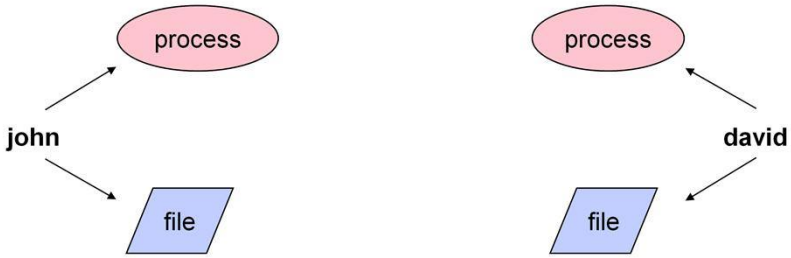
- **Multi-user operating systems require users to login**
  - validate username
  - associate with 'account'
- **The login process establishes the user's**
  - initial process (shell)
  - initial directory (home directory)
  - Identity (User and Group)
- **Users view Unix through**
  - interactive shells
  - custom menus
  - windowing environments
  - applications

In order to use a Unix machine users must login. In this process the system determines who the user is (the username), verifies this information (by requesting a password), and then associates with the user their file and process resources. Specifically, the user is associated with a sub-directory of the file system (their home directory) and an initial process (usually an interactive shell).

Humans tend to use names to distinguish similar objects, machines tend to prefer numbers. During the login process the system associates a UID (user identity number corresponding to the username) and GID (group identity number corresponding to the user's default group) with the user. This is subsequently used to label all files and processes created by the user.

What are Users?

- **Users are the owners of files and processes**



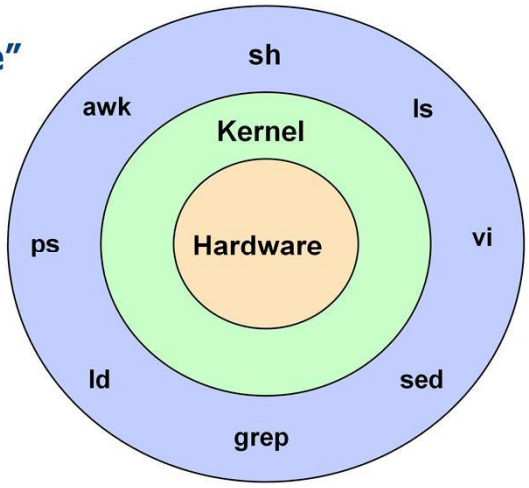
- **In Unix everything carries a UID and GID**

Users are not first class entities within Unix, only files and processes can claim this status. Users are simply attributes of files and processes. In Unix, every file and every processes must be owned by someone and exist in a group.

When users first login their initial process (the shell) and file (their home directory) belongs to them. Every subsequent file or process created by a user is stamped with the users identity (UID) and the users default group identity (GID). The UID is usually unique and has a one-to-one mapping with the users username; the GID is shared by users working together. The GID provides a means by which users can gain joint access to shared files and commands. In addition to the default GID, every user has up to 16 additional group ids, which are used for access permissions only (not for file ownership).

The Structure of Unix

- **Unix is "glue"**



System calls are function calls made by processes wishing to access the kernel.

The system calls used by the file system include **`open()`**, **`close()`**, **`creat()`**, **`read()`** and **`write()`**

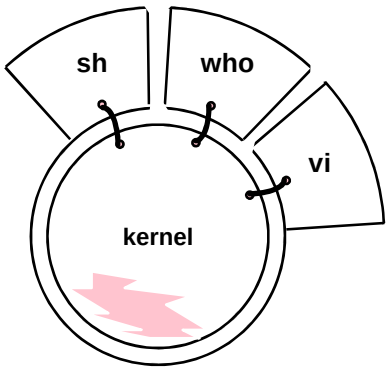
- **Applications exploit kernel resources through system calls**

Unix is structured in a highly modular fashion. The programs which users run exist as independent entities and exploit the resources provided by the kernel. The kernel provides the fabric or glue to hold the system together.

The Unix kernel also abstracts specific configuration and type information pertaining to devices and presents such devices through the file interface. This allows many of the operations which are defined for normal (plain) files, to be applied to device files. For example, changing access permissions with `chmod`.

Kernel Responsibilities

- **Process Management**
  - coordinating access to processor(s)
  - supported by kernel scheduling policy
- **Memory Management**
  - virtual memory (RAM + swap disk)
  - memory protection and virtual address
- **File System Management**
  - maintaining logical hierarchical file structu
  - joining disks, partitions and networks
- **Device Management**
  - logical file interface to all devices
  - supporting block and character devices



Process management concerns scheduling the many concurrent Unix processes onto the available processors. With support from the scheduling daemon the kernel allocates times slices to each process and switches between them. Provided the system is not overloaded, the user should not be aware of this.

Memory management is concerned with providing enough memory for the running processes. This usually entails taking a partition of disk (known as the swap), and paging (or swapping) out processes when they are not able to run. For example, when they are waiting on a slow I/O event.

File system management concerns maintaining the tree file structure. The kernel ensures that multiple partitions, perhaps on multiple disks, and network files, appear as a single hierarchy on the local machine.

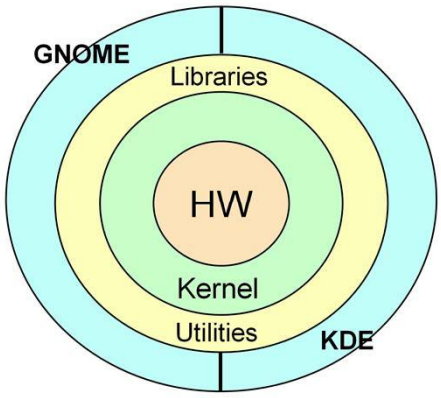
Unix provides a file interface to all devices. This is supported by the device management part of the kernel. In general, device file names are mapped to device drivers in the kernel which actually talk to the hardware.

Blocking & Character Devices

Devices are classified as being either block or character oriented. For example, a terminal is usually considered to be a character special device, whilst a disk is a block special device. The distinction concerns whether the kernel should buffer data before sending it to the device. Disks provide both character and block interfaces. The character interface is used when formatting and partitioning the disk; the block interface is used to access the disk when it is connected to the file system.

Linux Systems Today

- **Kernel versions numbered separately**
  - most recent are 2.6.x
- **Two main GUI environments**
  - KDE
  - GNOME
- **Many packaged systems**
  - databases
  - web server
  - browser
  - development environment
  - Office/productivity tools



Linux distributions today are structured in much the same way as traditional Unix systems. The main characteristic is the sheer amount of software that is available. Not all of this can be classed as of commercial quality, but a surprising amount is very robust and powerful and used in many real-world areas.

Typically in a distribution you will find at least one commercial quality database management system – the most common are MySQL and PostgreSQL. The most widely used Web server on the Internet is Apache, a fundamental component of all Linux distributions. Web browsers such as Firefox are available. Today most Linux systems are shipped with office tools such as OpenOffice (a Microsoft Office equivalent), mail/calendar manager such as Evolution (which provides the same functionality as Microsoft Outlook), and others.

2. WORKING WITH UNIX/LINUX

Working with Unix/Linux

- Logging In
- Basic Commands
- Keyboard Control
- File System Commands
- Process Commands
- On-Line Manual
- Unix Command Format
- The Shell
- Shell Wildcards
- Shell I/O Redirection
- Changing the Password
- Logging Out

Logging In

- **Enter a username and password**
  - to a Solaris / Linux workstation using the graphical dialog
  - to telnet or other terminal client when remotely connecting

```
login: fred
password:
Last login: Sat Jan 23 13:03:58 on console

Welcome to ACME Perfect Unix

$
```

To access a Unix machine a user must first login. On some machines a login prompt is displayed directly on the monitor or terminal; on others a login prompt appears in a window on the monitor.

At the login prompt type your username and (when prompted) the password. If successful, various information is displayed and the shell issues a prompt waiting for input. The above information consists of the time in which the user last logged in (this is useful to check that no one else has been using your account), the"message-of-the-day" (maintained by the administrator), a message confirming the receipt of mail, and finally the shell prompt.

At this point the user is at the top of both their process and file trees. The user is in their home directory, the directory which contains files and sub-directories which they own. In addition the user is interacting with their login shell. This process is owned by the user and will invoke other commands on the user's behalf.

Linux Login and Session Management

- **Login is often remote**
  - via telnet or ssh (e.g. putty)
- **Login may occur through graphical UI**
  - configured at system installation
  - allows user to select GUI environment
- **Command line startup available**
  - login as normal
  - use `startx` command
- **X Session management supported**
  - remember apps from previous session

During system installation, the user can select whether to login through a graphical form or in the traditional text oriented way. The graphical form is the most flexible and user-friendly, offering various options as to which GUI environment should be used. It also permits the user to shut down or reboot the system without logging in.

Logging in at the graphical form will immediately start the requested GUI environment (as a default it will normally use the environment that the user started on last login). From the command line, the GUI environment can be started using a command like

```
startx
```

Based on configuration files that we will see later in this chapter, the command starts the basic X Window system and a number of the associated applications.

The main KDE and GNOME environments both understand *session management* as defined as part of the X Window System. This provides for applications saving their state when they shut down, so that an environment can be recreated next time as close as possible to the same state. When we consider that the X Window Manager (whichever one is being used) can also remember its state, we can see that it is possible to recreate the same environment each time a user logs in or starts the GUI.

Linux Graphical User Interfaces

- **Many different "look and feel" options**
  - Motif
  - "NextStep"
  - Windows-like

- **Most users use integrated GUI environment**
  - KDE
  - GNOME

Although the original versions of Linux provided the traditional command line oriented user interface, manyusers today access Linux through a graphical user interface. The increasing availability of device drivers for the main graphics cards in use today make it feasible to provide a GUI on most PC systems.

The graphics capabilities of Linux are based on the X Window System, a long established framework for graphics that forms the basis for most Unix based GUIs.

X operates at a very basic level, with details of the look and feel of a user interface provided through layers of libraries and applications generally known as "toolkits".

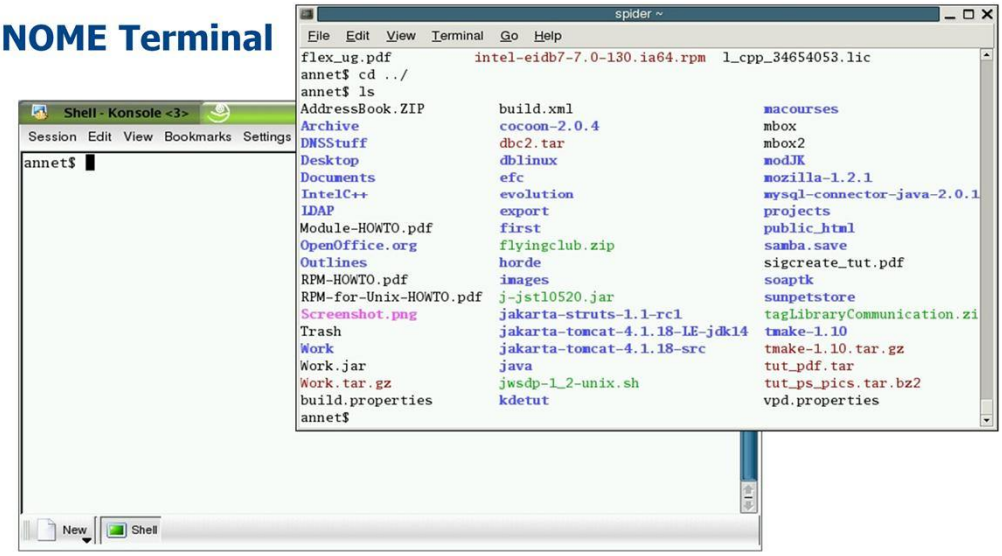
In the commercial Unix world, most GUIs are based on the Motif toolkit that was developed by the Open Software Foundation – this has more recently been developed into the Common Desktop Environment or CDE.

Motif was actually a commercially licensed product, however a free equivalent product (known as LessTif **J**) has been produced and offers compatible behaviour on Linux. Other GUI environments available include one that mimics the NextStep environment that was part of the NeXT range of workstations – this is known as AfterStep. Also available is an environment that presents a look and feel compatible with that of Windows 2000.

However, most users use one of the two integrated GUI environments developed specifically for Linux. These are KDE which began as a project in Europe, and GNOME, which was part of the GNU project. Both of these offer a complete graphical user environment consisting of applications, configurability and application development frameworks.

Command Line Tools

- Konsole
- GNOME Terminal



The command line remains the most powerful and flexible mechanism to exploit the features of Unix and its standard command set. Even when using a GUI such as KDO or Gnome, many users will still employ a terminal emulator program such as Konsole or Gnome Terminal (or the older and less sophisticated Xterm) as their main interface to the system.

When accessing a Unix/Linux system remotely, it is normal to use the command line, usually through a program like ssh (often wrapped in a tool like putty). This course concentrates on the command line interfaces to Linux and Unix as these are the most widely used mechanisms when Unix/Linux is deployed primarily as a server.

Basic Commands

Getting simple information

```
$ date
Wed Jan 27 11:11:33 GMT 1993
$ pwd
/home/fred
$ who am i
sparc2!fred ttyp3 Jan 27 09:28 (:0.0)
$ ls
report errors updates letter
$ who
fred     ttyp3    Jan 27 09:28
prjlg    ttyp4    Jan 26 15:26
roger    ttyp5    Jan 26 17:15
$
```

Users issue commands by simply typing their name into the shell (along with any arguments). The shell will then invoke the corresponding program, giving rise to a process which will carry out the required task.

In the above, `date` displays the current time and date, `pwd` (print working directory) shows your current location within the file system, `who am i` tells you who you logged in as (you may have several usernames), `ls` lists the files in the current directory and `who` tells you who is logged in to the machine.

Within Unix there are hundreds of simple commands such as these. In isolation the commands are quite limited, however Unix allows them to be executed in sequence. Complex commands are composed of chains of simple ones, where often the output of one commands forms the input to another. Because each command is little more than a building block, their output tends to be terse. In the above their are no headers or footers; for example, to identify the columns of output produced by the who command. This is because the output may form the input of another command, in which case such extra data would be problematic.

Keyboard Control

Essential keyboard control characters

^C	interrupt command
^Z	suspend command
^D	end of file
^S	suspend output
^Q	continue output
^H (^?) [DELETE]	delete last character
^W	delete last word
^U	delete line

Unix provides the usual set of keyboard control sequences, plus a few extra. To cause any of the above to occur, hold the control key down (sometimes marked CTRL) and then depress the required character.

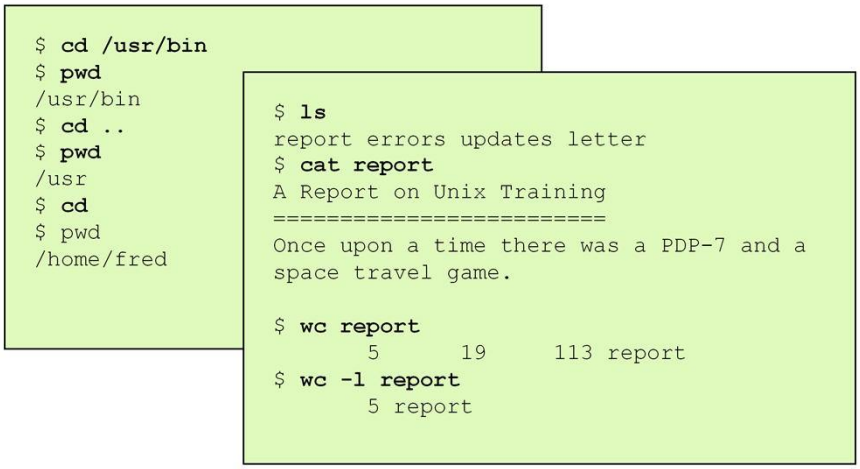
The above definitions are for the “usual” mapping and may vary from one user to another. Differences occur because of hardware setup and because of the software terminal settings. The `stty` command is used to establish specific key settings; for example,

```
stty erase <BACKSPACE>
```

makes the backspace key the erase character.

File System Commands

Moving around the file system and viewing files



Unix provides a variety of commands for dealing with the file system. The `cd` command (as adopted by DOS) moves the shell to a specified directory within the logical filesystem. If no arguments are given, then the command locates the shell at the user's home directory. To determine the directory currently occupied by the shell, use the `pwd` (print working directory) command.

The notion of the shell being in a directory deserves some consideration. The shell is a running process. When a user asks the shell to invoke a command, the command is executed with respect to the shell's current directory. That is, the shell process passes to the new command's process its current location. The location is subsequently used by the command whenever it needs to access local files. For example, the `ls` command shows the files in the current directory, by default, because it learns of the current directory from its parent process, the shell.

The `cat` command is used to view the contents of a file. In the absence of any other information, `cat` expects to find the file in the current directory, as established by the shell. `cat` is an abbreviation for concatenate, meaning to append. It appends the specified file or files to the end of the terminal file. The terminal file is a special device file associated with the physical terminal device (or workstation window) which the user is using.

`wc` counts the number of words, lines and characters in the specified files. Using the optional '`-l`' argument, `wc` only displays the number of lines in the file.

Process Commands

- Almost all commands give rise to new processes

```
$ more report
A Report on Unix Training
=========================
Once upon a time there was a PDP-7 and
a space travel game.

$ cd
$ pwd
/home/fred

$ ps -f
UID   PID PPID C STIME    TTY   TIME CMD
ed    139  101 1 09:42:04 pts/3 0:08 bash
ed    507  139 1 09:51:02 pts/4 0:00 ps -f

$
```

more is invoked as a new process by the shell

cd and pwd are commands to the shell itself and do not give rise to new processes

Almost all commands issued by users give rise to the creation of a new process. The shell itself has minimal build-in functionality. Its purpose is simply to find the requested program file, invoke it, wait for the process to terminate, and then prompt the user for the next command.

cd and pwd are the exceptions to this rule and are built into the shell. The notions of the current directory and of changing the current directory concern the shell itself. Therefore, these commands need to act on the shell directly, not on some child process.

To determine what processes are currently running use the ps command. Using the -u uid (your userid, as listed by the id command) option it displays all processes owned by the user; using the -ef options, it displays all processes on the system. In order to see the size of all programs, use ps -ely.

Documentation

On line manual pages available

- may not be comprehensive
- some systems may include HTML formatted pages

Info pages

- based on emacs documentation
- menu based documentation browser
- intended to complement or even replace man pages

HOWTOs

- tutorial documents covering all aspects of system usage, administration and development
- /usr/share/doc/howto/en
- HTML format available

Linux is supplied with a substantial range of online documentation.

Like most Unix systems, there is the online reference manual accessed through the man command. However some pages note that they may not be complete or up to date. This is because there was an intention to move away from this form of online documentation towards a more browser based approach developed from the Info system, which was part of the emacs online documentation.

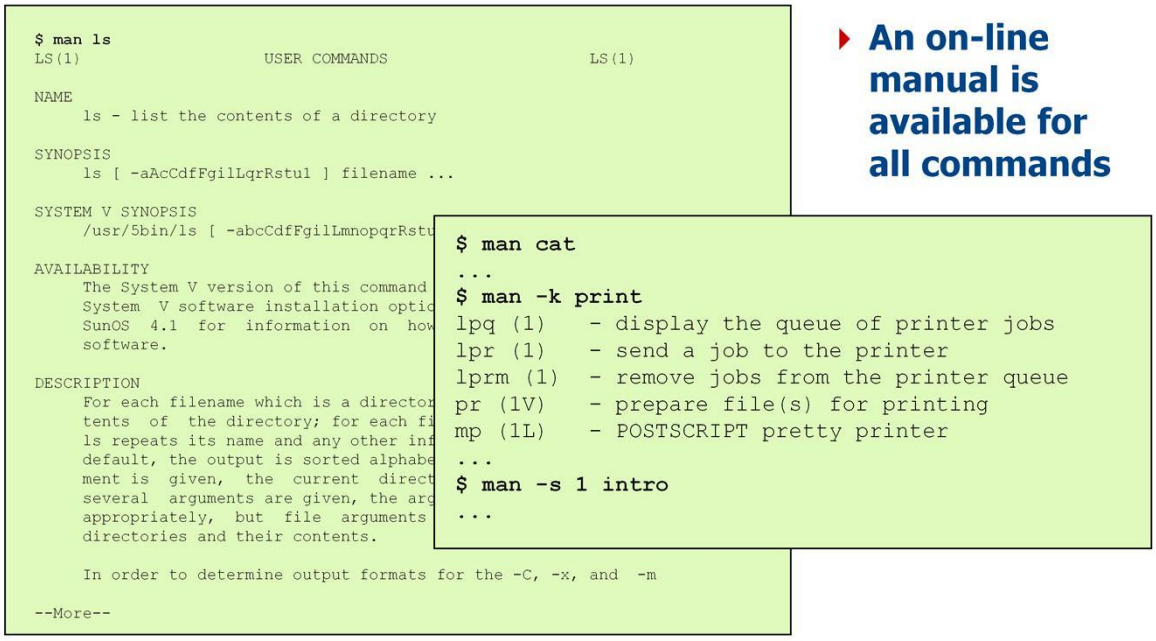
In addition, a number of more detailed, tutorial style documents are available with the system. These are known as "HOWTO" documents, mainly because they describe "how to" do something. The HOWTO documents are available in the directory

/usr/share/doc/howto/en

(en for "English" – other languages are usually available). The HOWTO documents are now usually presented in html format, so can be viewed conveniently in a web browser. There is information here on everything from basic usage topics such as using the CD Player application, to developing device drivers for obscure hardware, to configuring Voice over IP services.

Indeed, the directory /usr/share/doc will normally contain a substantial additional amount of documentation in the form of papers and even online books.

On-Line Manual



Most Unix systems are setup with an on-line manual. On current systems this may take several forms, and be accessible through window-based hypertext applications. However, a command line version of the manual should also be available.

The manual pages are stored in the file system within the directory `/usr/share/man`. Each command has its own manual entry and may be retrieved using the `man` command. Manual pages are automatically paginated through the more command. It is possible to search a manual entry by using the search commands within more.

Although `man` provides a means of determining how a command works, it is not very useful if a user is unsure of a command's name. To search the manual pages for all commands which may be suitable for a specified task, perform a keyword search. This is achieved using the `-k` option. Note that keyword searches only work if the system administrator has properly configured the manual pages using `catman`, and where appropriate has set up the `MANPATH` variable.

The manual pages are in fact divided into a series of volumes. Each volume may be directly accessed by supplying a numerical argument from `1` to `8`. To determine the nature of each volume, examine the intro entry which exists in each of them.

Entering Commands

Commands follow a standard format

```
command [-options] [files ...]
```

Syntax

- command name, possible options, possible files
- parts of command are separated by white space
- entire command is CASE SENSITIVE

Philosophy

- each command is simple and designed to do one thing well
- by default commands tend not to generate headers or trailers
- most commands are terse, and take single character options

Unix commands follow a relatively standard format. They consist of the command name, a series of single character options preceded with a dash, and an optional set of files on which the command is to operate.

Unfortunately, due to the somewhat uncontrolled evolution of Unix, this standard is not always adhered to. Some commands do not require a leading dash to indicate an argument string, others require words to identify each option rather than the usual single character. Of perhaps more concern, is the inconsistency which exists between commands. For example, a -l option may mean long format to one command and quite the opposite to another.

The various parts of a command are separated by white space, that is, any number spaces, tabs or (where it is apparent to the command) new lines. In general, commands, options and file names are lower case in Unix. The only time that commands or file names exist in upper case, is when it is necessary to emphasise them. For example, some directories may contain a README file, drawing the users attention before they interfere with the directory's contents.

Unix is case sensitive. A lower case name is distinct from an upper case name, or indeed a mixed name. Command options are also case sensitive. In fact, where upper case options are available, they often do the opposite to their lower case counterparts.

Unix Command Format

- **Listing the files in the /etc directory in long format**

```
ls -la /etc
```

- **Looking at the contents of three files**

```
cat report update letter
```

- **Adding line numbers to the displayed files**

```
cat -n report update letter
```

The above three examples demonstrate the general format of Unix commands.

In the first example two options are supplied to the `ls` command. It is clear to `ls` that `-la` are options because of the leading dash. Without the dash, ls would attempt to list the contents of a directory called `la`.

The second example shows the use of the `cat` command with three filename arguments, but no options. `cat`, as its name suggests, will concatenate each file in turn and write the composite output to the default output file---the users terminal. Most Unix commands are able to operate on as many file arguments as are supplied. The commands simply apply themselves to each file in turn.

In the last example an option is supplied to `cat` requesting that line numbers be generated for the composite output file.

The Shell

- **Login shell is invoked as the user logs in**
  - interactive command line interpreter
  - invokes commands for the user
  - dies when the user logs out
- **There are a family of shells available on Unix**
  - bourne shell sh $
  - korn shell ksh $
  - C shell csh %
  - Bourne Again Shell bash $
- **Shells can help users with commands**
  - generate lists of filenames
  - redirect command input and output
  - construct command pipelines

Commands which a user submits to Unix are interpreted by the shell. The shell is the process which reads the characters typed in from the keyboard, and eventually invokes the corresponding program. Commands (except for a few minor exceptions) exists as program files somewhere within the logical file system. Part of the job of the shell is to find the location of a requested command, which it does by searching a local variable called its PATH.

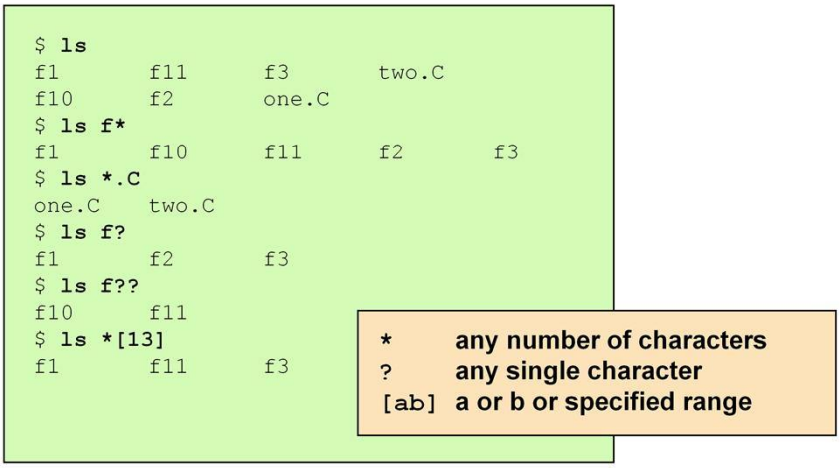
There are a number of shells on Unix systems. Since the shell is simply a program, many have been written for Unix as it has evolved. The most important shells are the Bourne shell (the original shell for Unix developed by Steven Bourne), the C shell (the California shell developed at UCB) and the Korn shell (developed by David Korn). The Korn shell is currently the most popular shell.

In addition to invoking commands, the shell can assist users by automatically generating file name argument lists. Using special wildcard symbols, the shell may be requested to complete a command by generating all file names which match the specified wildcard.

The shell also has considerable influence over the default input and output streams used by a child process. By default, processes read input form the keyboard and write output to the display. However, since these are simply files in Unix, the shell is able to change the files which these correspond to. In fact, the shell can even arrange that the output of command is fed directly into the input of another.

Shell Wildcards

Wildcards may be used to generate filenames



The shell provides a series of wildcard characters which users may use to ask the shell to automatically complete or generate lists of filenames. In the above examples, * is used to match any number of any character. The `ls f*` command matches all file names which begin with f followed by any number of any character. ? matches any single character, and [ ] matches a range of specified characters.

Note that the wildcard mechanism implemented in Unix is different from that used by other operating systems, and in particular from that used by DOS. In DOS individual utilities expand wildcards and therefore interpret them in their own individual ways. In Unix, by contrast, the shell expands the wildcards before passing the generated arguments to commands. Therefore, in Unix individual commands do not have to understand wildcards or implement the supporting code. More importantly, unlike systems such as DOS, the semantics of wildcard symbols in Unix is completely consistent.

Shell I/O Redirection

The output of a command can be redirected

- to a file
- to another command

```
$ ls > new
$ cat new
f1       f11      f3       two.C
f10      f2       one.C    new
$ cat f1 f2 f3 > all
$ ls | wc -l
       9
$ ls /etc | more
...
$ echo hello > afile
$ cat afile
hello
```

>	redirect output
<	redirect input
|	pipeline commands

The output from ls pipe wc -l is 9 because of the eight original files, plus the extra file all created from the cat command.

The shell may be used to redirect input and output for commands. By default the input for a command (if not taken from a supplied file) is read form the keyboard (the standard input). Similarly, the output for a command is written to the display (or standard output) by default. The shell understands a number of special characters which direct it to change the files which constitute the standard input or output of a command.

The > indicates that the output should be written to the specified file, the < indicates that the input should be read from a specified file, and the | symbol indicates that the output of the left command should be directed to the input of the right command.

In the first example above the output of `ls` is stored in a file called `new`. This is useful if the output needs to be saved or used in another context. In the next example three files are concatenated together by `cat`, and then appended (one after the other and in sequence) onto the standard output. Due to the shell's redirection, the standard output is the file called *`all`*, not the display.

In the next example a command pipeline is established. The output of the `ls` command is the input of the `wc` command; with the `-l` option this outputs the number of files in the current directory. This command sequence is indicative of the power of Unix. Individual commands are not in themselves very powerful. However, by arranging them in pipelines complex tasks can be undertaken. Unix is a powerful toolkit, in which commands interoperate with each other.

Changing the Password

- **Use the passwd command**
  - can only change your own password!

```
$ passwd
Old password:
New password:
Retype new password
$
```

The passwd command allows users to change their passwords. Passwords are the first line of defence against unauthorised access to the machine. They should be chosen carefully and kept secret. Good passwords contain a mixture of alphabetics and numerics. In Unix, they may be of mixed case and contain punctuation characters.

Various security enhancement products exist for Unix. For example, kerberos provides an added security layer in a large networked environment

Logging out

- **Logout with `^D`**
  - end of input stream ...

```
$ ^D

Solaris 10

login:
```

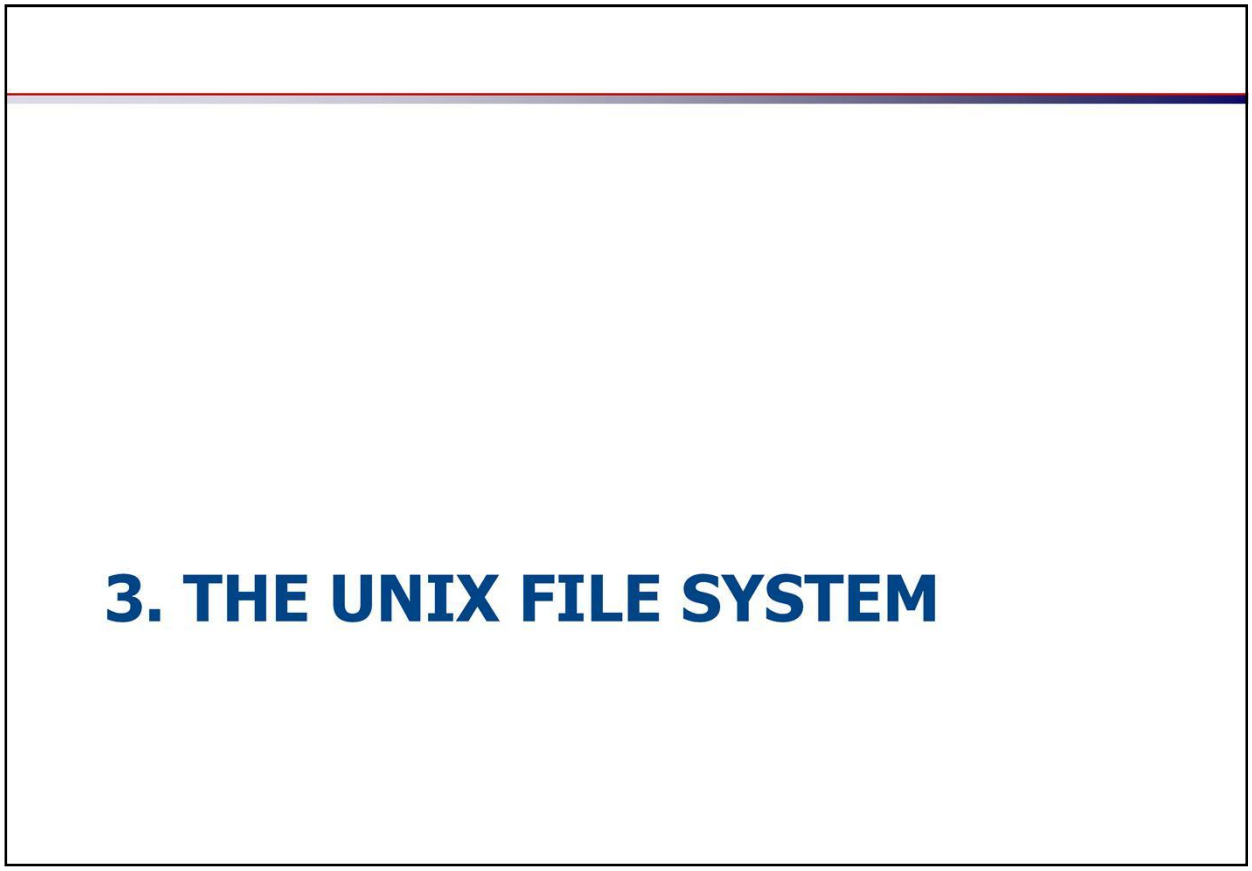
- **Some shells require `logout` or `exit` command**
  - configured via a shell variable, like `set -o ignoreeof`

In Unix all devices are considered to be files. When the shell is executing it reads its input data from the terminal file. The input data (usually commands) is read until there is no more, i.e., until the end-of-file character is detected. To generate this character from the terminal, type ^D. Note that both the csh and ksh ignore this character if the ignoreeof variable is set. They require logout or exit to terminate.

Unix running on workstations, or even on PCs, should never be terminated by powering down the machine. In line with most big operating systems, Unix must be formally shutdown (by the administrator) before the hardware can be switched off. Switching off the machine prematurely may cause damage to the file system structure. Moreover, it disconnects any other users also logged into the machine.

Exercise

- **Refer to the "Working with Unix" exercise in the Exercises booklet.**

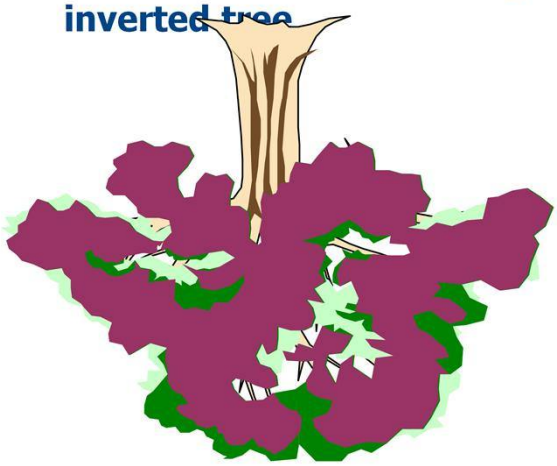

3. THE UNIX FILE SYSTEM

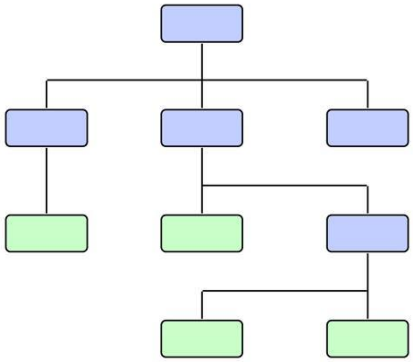
The Unix File System

- Hierarchical Structure
- Directory Paths
- Traversing the File System
- Examining Directory Contents
- Using Shell Wildcards
- Building the File System
- Copying Files
- Copying with Wildcards
- Moving Files
- Deleting Files
- Linking Files

The Unix File System

- An organised way of storing files
- The structure of the file system can be thought of as an inverted tree



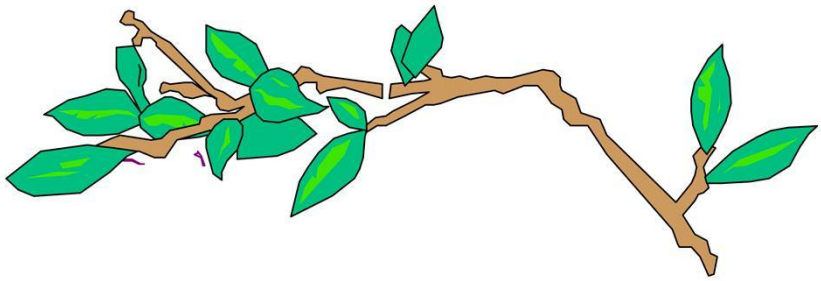


Unix employs a tree structure to store files. Starting from an initial top-level directory (the root directory) sub-directories successively organise information into categories, and then subcategories. There are no limits on the depth to which the tree structure can grow.

Unix differs from other hierarchical file stores (such as those provided in DOS and VMS) in that there is only one tree. The single tree structure hides multiple disks, partitions and even the network when NFS (the Network File System) is employed.

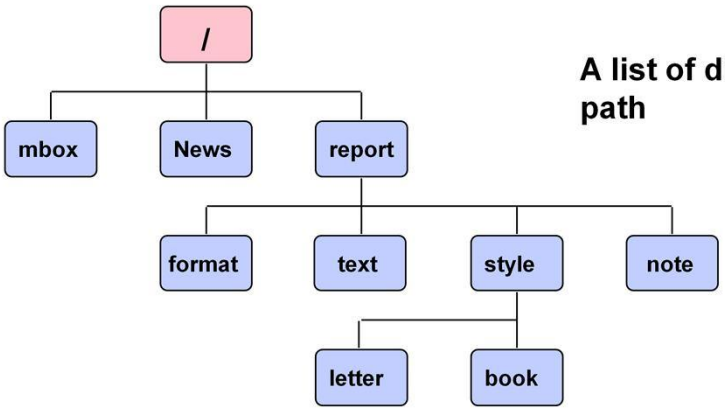
Hierarchical Structure

- **Directories are files which hold information on other files**
- **Directories can be viewed as branches and files as leaves**
- **Since directories are just other files, they can also be stored inside directories**



The Unix file system is organised into a hierarchical tree structure in which directories are branches and files leaves. The purpose of directories is to group together related files. However, since files may themselves be directories, it follows that directories may contain sub-directories.

Directory Paths



A list of directories forms a path

- **Absolute path names start from root**
  - `/report/style/book`
- **Relative path names start from the current directory**
  - `style/book`

Path names describe routes through the file system. A relative path name is a route from the current working directory; an absolute path name is the route from the top of the file system.

Absolute path names begin with a leading slash and are unique. Relatively path are not unique since they depend on the directory in which the path is specified.

Traversing the File System

- **Every directory contains two special directory files**

  ".." parent directory

  "." current directory

```
$ cd ../style

$ pwd
/report/style

$ cd ..
$ pwd
/report
```

```
$ cd /report/style

$ pwd
/report/style

$ cd /report/text
$ pwd
/report/text

$ cd .
$ pwd
/report/text
```

The directories "." and ".." are convenient names for the current and parent directories. ".." allows path names to traverse back up through the tree.

Use the cd command to 'change directory' and the pwd command to print the current working directory. Note that cd without any arguments takes the user to their home directory.

Examining Directory Contents

- **The contents of a directory can be listed using:**
  - **ls [-aAcCdfFgilLqrRstu1] filename ...**

```
$ pwd
/report

$ ls
format   note     style
text

$ ls -F
format*  note     style/
text/

$ ls -aF
./       format*  style/
../      note     text/

$ ls ..
mbox     News     report

$ ls /report/style
book letter
```

```
$ ls -l
-rwx------ 1 greg dude 100 Jan 21 20:01 format
-rw------- 1 greg dude 873 Jan 21 19:59 note
drwx------ 2 greg dude 512 Jan 21 19:58 style
drwx------ 2 greg dude 512 Jan 21 19:58 text

$ ls /
mbox News report

$ ls -l style
-rw------- 1 greg dude 22  Jan 21 20:59 book
-rw------- 1 greg dude 230 Jan 21 20:04 letter

$ ls -ld style
drwx------ 2 greg dude 512 Jan 21 19:58 style
```

The `ls` command is used to display the contents of the specified directory. It takes a variety of options that affect which files are displayed and the way their information is formatted.

`-l`	long listing
`-a`	all files, including those beginning with '.'
`-g`	used with `-l` for group ownership
`-d`	the directory file not its contents
`-F`	show file type

Using Shell Wildcards

- **Recall shell wildcards for filename expansion**

```
$ ls p*
pint    plastered  pub

$ ls /etc/*/m*
/etc/adm/messages
/etc/dp/modem
/etc/openwin/modules
```

- **Note the shell expands wildcards**

```
$ echo hello world
hello world

$ echo p*
pint plastered pub
```

The Unix shell provides wildcard expansion to generate filenames for commands. To list all filenames that being with p (as above), then "*" is used to tell the shell to generate the filenames automatically. The shell searches the specified directory to find the files.

*	matches zero or more characters
?	matches exactly one character
[ABC]	matches either A or B or C
[A-Za-z]	matches any single letter

Note that wildcard expansion is different in Unix than for DOS. In DOS each individual utility interprets * and may associate a different meaning to the symbol. This is not possible in Unix, since the shell interprets the *, generates an argument list, and then calls the specified command. In the above, echo simple writes to the display its list of arguments; the list of arguments beginning with p were generated by the shell prior to invoking echo.

Building the File System

`mkdir` creates directories, `rmdir` removes them

```
$ pwd
/report

$ ls -F
format*   note  style/  text/

$ mkdir biblio
$ ls -F
Biblio/  note  text/
format*  style/
```

```
$ rmdir biblio
$ ls -F
format*  note  style/  text/

$ rmdir style
rmdir: style: Directory not empty
```

mkdir [-p] dir1 [dir2 ...]

The `mkdir` command is used to create new directories, and `rmdir` to remove directories. In keeping with most Unix commands, the commands may be supplied as many filename arguments as is required. In the following

```
mkdir one two three four five six /tmp/seven
```

six directories are created within the current directory, and a seventh is created beneath `/tmp`. Notice, however, that a minimum of one directory must be supplied to the command.

Using the `-p` option, `mkdir` is able to create missing parent directories as needed

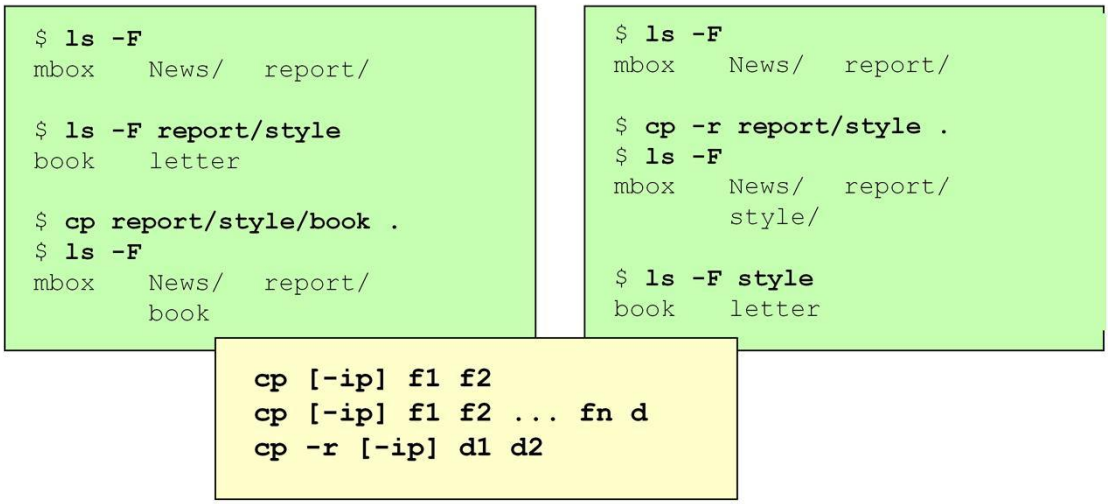
```
mkdir -p first/second/third
```

will create the missing parent directories first and second if they do not already exist.

Note that it is not possible to remove a directory with `rmdir` if it contains other files. To remove the directory, first remove all the files and sub-directories which it contains. The powerful (and dangerous) `rm -r` command is useful for this.

Copying Files

cp copies files and directories around the file system



`cp` is used to copy files and directories around the file system. Note that copy means duplicating the bytes on disk representing the contents of the files being copied.

`cp` is used with two arguments when copying from one file to another and with many arguments when copying a collection of files into a directory. In the case of the latter, the directory must exist and be the last argument. `cp` may also be used to copy the contents of one directory to another. In this case the `-r` (recursive) option must be supplied. When copying directories, if the target (d2) exists, then the source (d1) is created within it. A file f1 within d1 may now also be accessed as d2/d1/f1. If, however, the target does not exist, then it is created and the actual contents of d1 are copied into it. Therefore, a file f1 within d1, may now also be accessed as d2/f1.

By default, the `cp` command overwrites any files which already exist with the target name. The `-i` (interactive) option may be used in order to get `cp` to prompt prior to overwriting any existing files.

To preserve a file's modification time and permission bits, use the `-p` option. If it is also necessary to preserve the file's ownership, then the `cpio` command should be used. Some versions of `cp` support this behaviour directly, such as GNU `cp` with the -a option.

Copying with Wildcards

- **Wildcards may be used when copying collection of files**

```
$ cp *.c /home/george/src
```

- **Wildcards cannot be used for names which don't exist**

```
$ ls
chapter1.txt      chapter2.txt
$ cp *.txt *.bak
```

Wildcards may be used with the cp command to generate a list of filename arguments to be copied. However, it is important to realise that the cp command is not involved in the expansion of the wildcard symbols. The wildcards are replaced by the shell with any matching filenames, prior to invoking the command.

Since the shell generates the filename lists, based upon the files which currently exist in the directory, it is impossible for the shell to generate names which do not already exist. Therefore, the semantics of the DOS copy command do not apply to the Unix cp command.

In the example given above, the user attempts to backup two chapters of a book, by copying them to new names with the extension *.bak*. In DOS this would be successful, in the older Unix systems it would be catastrophic. The shell would replace **.txt* with the two files and **.bak* with nothing. It would then invoke the cp command as follows

```
cp chapter1.txt chapter2.txt
```

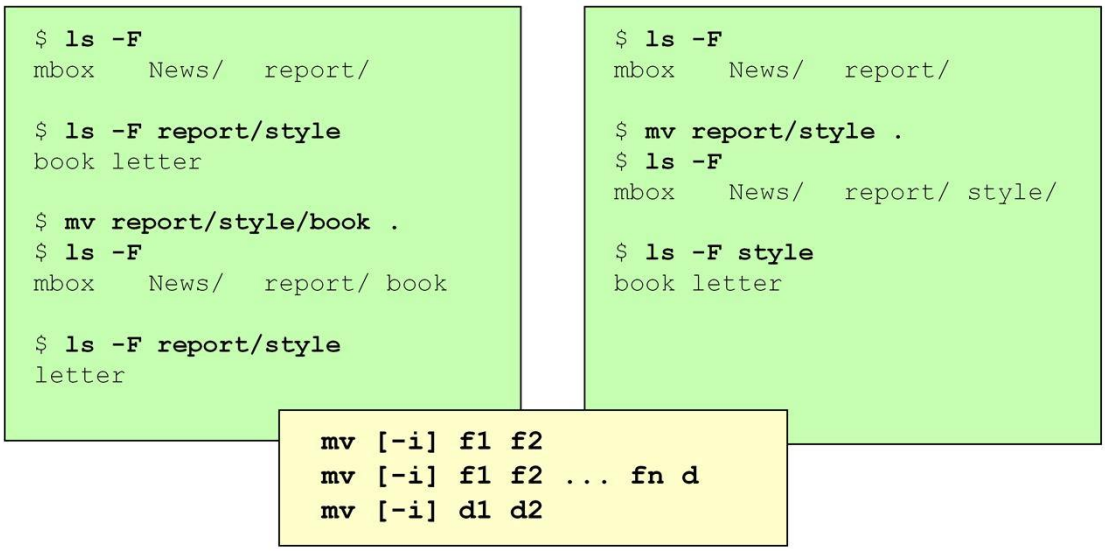
which is most unfortunate since the purpose of the activity was to backup not destroy the files!

Unix is not wrong in its behaviour, it is in fact entirely consistent. The semantics of the DOS copy command are, however, somewhat strange. In DOS, the * in the first argument is used as a wildcard and the * in the second as a place holder indicating where to substitute the first part of the filename, and what to tack on the end.

Fortunately, recent shells pass the * to the application if they are unable to perform an expansion and this gives rise to a cp error. The problem would be solved in Unix by copying the files to a backup sub-directory.

Moving Files

Files and sub-directories can be moved



`mv` is used to rename files and directories. It does not cause the contents of the file to be physically moved, only the file's name is changed in its directory.

The new name may be a path to another directory, so `mv` can in fact move a file or directory from one place to another.

Note that there is no need for a recursive option when moving a directory since files contained within the directory don't care what it is called. More specifically, the contents of a directory file are the files stored within it, and `mv` does not effect file contents. The `-i` option may be used if there is a danger of overwriting existing files.

Deleting Files

- `rm` deletes files and directory structures

```
$ ls -F
mbox     News/    report/  book
$ rm book
$ ls -F
mbox     News/    report/
```

```
$ ls -F
mbox     News/    report/  style/
$ rm -r style
$ ls -F
mbox     News/    report/
```

- `rm -i` requests confirmation before deleting files

```
$ rm -i book
rm: remove book? n

$ ls -F
mbox     News/    report/  book
```

```
$ rm -i book
rm: remove book? y

$ ls -F
mbox     News/    report/
```

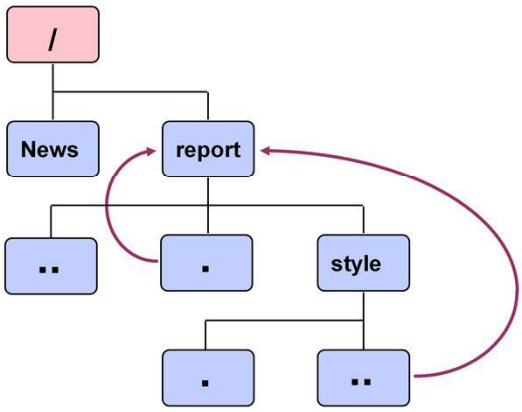
The `rm` command deletes files and directories. Beware that in Unix a deleted file is lost forever. There is no mechanism to allow a file to be un-deleted since the disk space associated with the file may immediately be re-used by some other process. To recover a deleted file, the administrator must be asked to restore it from a system backup. It is unlikely that the restored file will contain recent changes made to the file.

The `rm` command also has `-i` and `-r` options similar to those of `cp`. `-i` allows interactive use of the command so that a user may stop the command from accidentally deleting a file. The `-r` option is necessary if directory structures must be deleted.

There are several other useful options with `rm`. The `-f` (force) option instructs `rm` to remove files without asking. It would ask if the `-i` option has been set, or if the permissions on the file to be removed are not read/write. In the case of the latter, `rm` will override the permission if the user owns the file.

Linking Files

Unix directories have multiple names



All directories have at least two names, their name in the parent directory and '.' in themselves.

With each sub-directory, a new name is created for the parent, '..'

It is also possible to create multiple names for files

The Unix file system is held together through links. Each file is identified by a link name, or file name as it is usually called. Every file in the file system must have a name (a link), however it is possible (and often necessary) that some files have multiple links.

The above example shows the multiple names associated with directories. All directories have at least two names, their name in the parent and '.' in themselves. Should they have sub-directories, then a new name is generated for them in each sub-directory, '..' The '.' and '..' directory links are created automatically when a new subdirectory is made. They are used as a short-hand notation to refer to the current or parent directories.

Linking Files

- **Commands have different names to indicate behaviour**

```
$ ls -li /usr/ucb/vi
12334 -rwxr-xr-x  7 root   bin    204800 Jul 23    1992 /usr/ucb/vi
```

- **Use the `ln` command to create links**

```
$ echo hello > afile
$ ls -l afile
-rw-r--r--  1 fred   dude        6 Jan 14 23:16 afile
$ ln afile newName
$ ls -l afile newName
-rw-r--r--  2 fred   dude        6 Jan 14 23:16 afile
-rw-r--r--  2 fred   dude        6 Jan 14 23:16 newName
$ cat newName
hello
```

ln [-s] f1 f2

It is sometimes useful for normal files to have multiple names. For example, the standard Unix editor, `vi`, has seven names. The different names for `vi` include `ed`, `view` and `vipw`. The purpose of the multiple names is to give the illusion that there are many programs providing `vi` like editing facilities rather than simply one. When invoked, `vi` checks to see what it was called and changes its behaviour accordingly. This saves users having to remember large combinations of options.

The `ln` command enables users to create their own links. In the above example, the names *`newFile`* and *`afile`* are linked to the same file. The fact that *`afile`* existed first is not relevant, both are equal. Changes made to the file through the name *`afile`* would be the same if made through the file name *`newFile`*. The two names refer to exactly the same area on disk.

A useful application of links is to give the illusion that a file exists in multiple directories. This is achieved by specifying path names with the *`ln`* command. In the event that the directories physically exist on different partitions, the `-s` option (for symbolic) must be used to establish the link.

Note that links are destroyed by using the `rm` command. When the number of links referring to a file is zero, the actual file contents are removed. In fact, the `rm` command is implemented using the `unlink`() system call.

Exercise

- **Refer to the "The Unix File System" exercise in the Exercises booklet.**

4. FILTERS AND WORKING WITH FILES

Filters and Working With Files

- Filters
- Displaying Files
- Displaying Parts of Files
- Searching Files
- Quoting
- Sorting Files
- Comparing Files
- Command Pipelines
- Cutting & Pasting Files
- Showing Non-Printable Characters
- Line Printing Files

Unix Files

- **Stored within the Unix file system**
- **Streams of characters without structure**
- **Contain anything: data, text, executable code**
- **Directories are files containing references to the files stored inside them**
- **Devices are logical files accessed through the filesystem**

Files are stored within the Unix file system and contain streams of bytes without structure. That is, notions of records, fields and padding do not mean anything to Unix. Such high-level structuring is layered onto the system by application programs.

Files may contain data, text or executable code. Directories are special files which contain references to the files stored within them, specifically they contain filenames and file i-node numbers (the system's name for the file).

Since devices are accessed as files within Unix, they too have filenames. Usually (though not necessarily) device files reside in the */dev* directory. Device files can be opened, read, written and closed just like ordinary files. However, transactions with a device file are mapped by the Unix kernel to the physical device. A long listing of a device file shows (in place of size) the index in the kernel of the device driver software controlling the physical device.

File Names

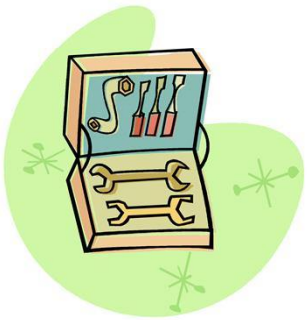
- Files are referenced by filenames, relative to the directory in which they are stored
- File names may contain most characters (even spaces)
- Maximum filename length is 256 characters
- By convention most filenames are lower case

Most Unix systems allow up to 256 characters to be used in a filename. The name may consist of almost any character (including spaces and non-printable characters). However, a slash / may not be used as it separates the components of a path name.

By convention, filenames are only in capitals when it is necessary to draw attention to them (README, for example). Most files and indeed virtually all commands use low-case characters.

Filters

- **Standard pattern of Unix commands:**
  - Accept 0 or more command line arguments
  - Arguments are treated as filenames, process the files
  - Write output to STDOUT
  - If no command line arguments, read from STDIN
- **Allows commands to be used standalone, or in pipeline commands**



One of Unix's great strengths is the ability to combine individual utilities together into larger, more complex commands. The reason this is possible is largely down to a standard structure of commands and how they handle their I/O. Such commands are known as filters.

The commands described in this chapter meet these rules, and thus can act as building blocks for building customised, highly flexible commands. They can also, of course, act within shell scripts.

Displaying Files

- `cat` is used to display files

```
$ cat file1
This is the only line in file1

$ cat file1 file2 file3
This is the only line in file1
This is the only line in file2
This is the only line in file3
```

- `more` may be used to paginate files

```
$ more long_file
This is the first line
This is the second line
         .
         .
         .
This is the twenty third line
--More--(33%)
```

`cat` is used to display the contents of a file. `cat` abbreviates concatenate, meaning 'append'. By default, `cat` is used to append files in the file system to the end of the device file representing the display.

Note that when supplied with multiple file arguments, `cat` applies itself to each, one after the other. The files are thereby concatenated one after the other on the display (standard output). This behaviour is the norm for most Unix commands.

`more` is useful when the file to be displayed is greater than a single screen. Using `more`, the file can be viewed page by page. To see the next page of output type <space>, to see the next line type <return>. `more` shows in the bottom left corner the percentage of the files which has been seen. Type "h" within more to discover the other functions it can perform.

Displaying Parts of Files

head displays the first lines of a file

```
$ head userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csp0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23

$ head -4 userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
```

```
$ cat userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csp0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
prjdpm   ttypa    Jan 20 (blobby)
simon    ttypb    Jan 21 (csgi32)
janet    ttyq1    Jan 22
```

The head command enables the first n lines of a file to be concatenated onto the standard output, usually the display. In the absence of the option selecting the number of lines, 10 lines are output.

head is particularly useful when the user is only interested in the first few lines of a large file. It is also useful in command pipelines, selecting the top ten (or so) lines from the output of the preceding command. For example, selecting only the largest ten files in a directory listing.

Displaying Parts of Files

`tail` displays the last lines of a file

```
$ tail -3 userlist
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23

$ tail +6 userlist
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csp0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
```

```
$ cat userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csp0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
```

```
tail +|-number [-f] [files ...]
```

The `tail` command displays the last n lines in a file, the last 10 lines by default. Like `head`, it is useful when dealing with large files since it allows only the portion of the file of interest to be displayed.

`tail` is more powerful than head because it is able to output the end of the file relative to the start or the end. Using `-n` `tail` operates relative to the end, using `+n` it operates relative to the start.

tail takes another useful argument which enables it to monitor files as they grow. Using `-f` `tail` outputs the end of the file and then waits. As the file grows, due to other processes writing to it, `tail` displays the new lines. This is particularly useful if it is necessary to watch log files as they grow.

Searching Files

grep searches files for strings

```
$ cat userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp3    Jan 22 (csc0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
```

```
$ grep csc userlist
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
james    ttyp3    Jan 22 (csc0a)
```

```
grep [-cilnvw] <RE> [files ...]
```

Regular Expressions (RE) are string templates

`grep` stands for global regular expression print. It is used to search the specified file (or files, or standard input) for all lines which contain the specified regular expression (pattern) and then to print them.

`grep` is a member of a family of commands to search files. `egrep` is extended `grep` and understands a bigger template language, `fgrep` is fixed `grep` and does not understand regular expressions at all.

Searching Files

grep options

- `-c` display count of matching lines
- `-i` case insensitive
- `-l` list names of files containing matching lines
- `-n` precede each line by its line number
- `-v` only display lines that do not match
- `-w` search for the expression as a word

```
$ cat userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
```

```
$ grep -c csc userlist
3

$ grep -n csc userlist
3:csc5gf  ttyp1  Jan 19 (csgi05)
4:csc6eq  ttyp2  Jan 22 (csgi47)
7:james   ttyp5  Jan 22 (csc0a)

$ grep -v tty userlist
gb      console Jan 23
```

grep takes a number of options to modify its behaviour. Options make it case insensitive, count matching lines and precede lines by their line number.

The -l option is useful when searching several files (perhaps hundreds) for a file which contains some known entry. Using -l grep will display the filename in which the match is found.

```
grep -l string *
```

The -v option is useful to negate the search pattern. Using this option all lines which do not contain the pattern are printed. Sometimes it is easier to use a regular expression to describe what is not wanted, rather than what is wanted.

The -w option is useful for looking for regular expressions that describe complete words; especially useful if the expression may be at the beginning, end or middle of the line. Note that \< ... \> is more generally used to define word boundaries.

Searching Files

Regular expressions are templates for strings

`.`	match any character
`*`	match zero or more occurrences of previous character
`^`	match beginning of line
`$`	match end of line
`[abc]`	match any one of a, b or c
`[a-z]`	match any character in range

```
$ grep '^csc' userlist
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47

$ grep -v ')$' userlist
gb       console  Jan 23
james    ttyp4    Jan 23
gb       ttyp8    Jan 23

$ grep ro.er userlist
roger    ttyp0    Jan 23 (csgi55)
robert   ttyp6    Jan 21 (dosh$1)
```

```
$ cat userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc0a)
robert   ttyp6    Jan 21 (csp0a)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
```

Regular expressions provide templates for the strings being searched. Rather than looking for the string anywhere on the line, a regular expression allows context information to be included also. For example, looking for all lines which begin or end with a specified string.

Note that the * in a regular expression is quite different (and more flexible) than that used by the shell for filename wildcards. In the shell * means any number of any character, whilst in regular expressions it means any number of the preceding character.

The single quotes placed around the regular expressions in the above examples, insulate the enclosed characters from the shell.

Quoting

- **Quoting allows characters to be hidden from the shell**

```
$ echo hello     world
hello world
$ echo "hello     world"
hello     world
$ echo p*
pub  pint plastered
$ echo "p*"
p*
```

"..."	quotes spaces & wildcards
'...'	also quotes $, ^, "
\.	quotes next character

- **'\' also used as quote character in regular expressions**

The shell provides a number of mechanisms to quote or escape special characters. The quoting mechanisms insulate the characters from the shell, and allow them to be passed directly to the program being invoked.

For example, the * character is used by the shell and by programs which understand regular expressions. Since the shell reads the command line first, it will carry out file name expansion, even though the * may actually have been intended as a regular expression for the program being invoked. Quoting gives the user the opportunity to tell the shell to leave the special characters alone.

There are a variety of special characters used by the shell. *, ? and [] are the standard wildcards and may be escaped using "" double-quotes. $ is used in variable substitution and ^ in the C-shells command line history mechanism. These must be escaped using the single-quotes ', which may also be used for wildcards.

The backslash \ escapes only the following character but is as powerful as single-quotes. In addition, for the C-shell it escapes ! The backslash may also be passed to programs interpreting regular expressions as a quote. For example, allowing `grep` to interpret . as a literal character.

Searching Files

```
$ grep '(csc.*)' userlist
james    ttyp5    Jan 22 (csc$a)

$ grep 'tty[qts]' userlist
janet    ttyq1    Jan 22
prj1sb   ttyq2    Jan 23
prj3js   ttyt5    Jan 23 (csparc)

$ grep 'tty.[0-2]' userlist
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
janet    ttyq1    Jan 22
prj1sb   ttyq2    Jan 23

$ grep '\$' userlist
james    ttyp5    Jan 22 (csc$a)
robert   ttyp6    Jan 21 (dosh$1)
```

```
$ cat userlist
gb       console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc$a)
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)
gb       ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
prjdpm   ttypa    Jan 20 (blobby)
simon    ttypb    Jan 21 (csgi32)
janet    ttyq1    Jan 22
prj1sb   ttyq2    Jan 23
prj3js   ttyt5    Jan 23 (csparc)
```

The first example displays all lines inside the file *userlist* that contain (csc followed by any number of any characters followed by).

The next example displays all lines that contain tty followed by either q or t or s.

The third example displays all lines which contain tty followed by any single character, followed by a character in the range 0 to 2. Note that the range must be a positive range through the character set.

Quoting from grep

Just as it is sometimes necessary to insulate characters from the shell, it may be necessary to hide characters from applications invoked from the shell. grep uses the \ to escape characters. In the last example, $ is escaped from both the shell and grep so that all lines containing a literal $ character are printed.

Sorting Files

`sort` organises files into alpha-numeric order

```
$ cat userlist
dpm      console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc$a)
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)
dpm      ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
```

```
$ sort userlist
admis    ttyp7    Jan 23 (suna)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
dpm      console  Jan 23
dpm      ttyp8    Jan 23
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc$a)
prj1gf   ttyp9    Jan 22 (blobby)
robert   ttyp6    Jan 21 (dosh$1)
roger    ttyp0    Jan 23 (csgi55)
sta4rf   ttyp3    Jan 23 (csun02)
```

sort	**-n**	sort on numeric value
	-r	reverse the sort
	+#	skip # fields to start of sort key

`sort` organises the lines in the input files into alphanumeric order. sort divides each line into fields (by default, separated by spaces or tabs) and sorts each field at a time.

`sort` provides options to change the field separator, to allow the sort to start at an arbitrary field (or sub-field) and to interpret fields numerically. The last is important since 19 is less than 2 according to the character values in the ASCII table, but clearly greater than 2 numerically.

Sorting Files

```
$ cat userlist
dpm      console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc$a)
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)
dpm      ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
prjdpm   ttypa    Jan 20 (blobby)
simon    ttypb    Jan 21 (csgi32)
janet    ttyq1    Jan 22
prj1sb   ttyq2    Jan 23
prj3js   ttyt5    Jan 23 (csparc)
```

```
$ sort +3n userlist
csc5gf   ttyp1    Jan 19 (csgi05)
prjdpm   ttypa    Jan 20 (blobby)
robert   ttyp6    Jan 21 (dosh$1)
simon    ttypb    Jan 21 (csgi32)
csc6eq   ttyp2    Jan 22 (csgi47)
james    ttyp5    Jan 22 (csc$a)
janet    ttyq1    Jan 22
prj1gf   ttyp9    Jan 22 (blobby)
admis    ttyp7    Jan 23 (suna)
dpm      console  Jan 23
dpm      ttyp8    Jan 23
james    ttyp4    Jan 23
prj1sb   ttyq2    Jan 23
prj3js   ttyt5    Jan 23 (csparc)
roger    ttyp0    Jan 23 (csgi55)
sta4rf   ttyp3    Jan 23 (csun02)
```

In this example a numerical `sort` is applied to the fourth field of the input file (+3 specifies the number of fields to skip).

In keeping with most Unix (filter) commands, `sort` works on any number of input files. If filenames are not specified, `sort` reads its input from the keyboard (or rather, from standard input where ever that happens to be).

Command Pipelines

- **Printing line 27 from a file**

```
head -27 afile | tail -1
```

- **Listing the largest ten files in the current directory**

```
ls -l | sort +3rn | head
```

- **List only the sub-directories in the current directory**

```
ls -l | grep '^d'
```

The philosophy of Unix is to:

write programs that do one thing and do it well

write programs to work together

write programs to handle character streams

The above examples demonstrate the utility of this philosophy. By using the pipe mechanism provided by the shell, the output of the left command is passed as the input to the one on its right.

The details of pipelining will be properly explained in a later module. It is enabled by using the pipe symbol | and by not supplying a filename to the right hand commands. When these commands realise that an input file has not been specified, they take there input from the standard input. The standard input is the keyboard if the program is executed directly, or the output of the preceding command if it is executed in a pipeline.

The first example shows that lateral thinking which must sometimes be applied in order to find an elegant solution. Having determined the first 27 lines, `tail` displays the last of these. Therefore, line 27 of *`afile`* is output.

In the next example `ls` generates a long listing of the files in the current directory. The fourth field in the listing is the size of the file in bytes. The output of `ls` is sorted on the fourth field in reverse, numerical order. This is passed into head, which displays only the first ten lines of the input.

The last example uses `grep` to display only those entries generated by `ls` which begin with *`d`*. These are directories.

Comparing Files

`diff` displays the differences between two files

```
$ cat story1
once upon a time there were
three bears that worked in
the City

$ cat story2
once upon a time there were
four bears that worked in
the City
```

```
$ diff story1 story2
2c1
< three bears that worked in
---
> four bears that worked in

$ diff -e story1 story2
2c
four bears that worked in
.
```

`diff` displays only the differences between two files. Lines in the first file which differ from the second are prefixed with "<", lines in the second file which differ from the first are prefixed with ">".

A useful option for `diff` is `-e`. This causes `diff` to generate output which describes exactly how *`file2`* can be generated from *`file1`*. The output is in the 'ed' editor language, and may be applied to `ed` automatically within a shell program (script). The technique is used by revision control systems to save having to store multiple copies of the same data on disk.

Other related commands include `cmp`, `comm` and `diff3`.

Cutting Fields

- `cut` removes selected fields from each line of the file

```
$ cat userlist
dpm      console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc$a)
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)
```

```
$ cut -d" " -f1 userlist
dpm
roger
csc5gf
csc6eq
sta4rf
james
james
robert
admis
```

cut selects fields from lines or collections of characters from lines, excluding everything else. In the above example, cut selects field one from file *userlist*, where fields are delimited by a single space. By default the field delimiter is a tab.

The opposite function is provided by paste, which joins the lines of distinct files into single lines of one file. For example, given two input files as follows

```
$ cat file1
dave
pete
jane
$ cat file 2
1000
2000
3000
```

pasting the files together writes a new file to the standard output (the display) in which the files are joined column by column.

```
$ paste file1 file2
dave 1000
pete 2000
jane 3000
```

Duplicate Lines

- **`uniq` removes duplicate adjacent lines from files**

  `uniq -c` count number of duplicate line

  `uniq -d` only print repeated lines

  `uniq -u` only print non-repeated lines

```
$ cat userlist
james    ttyp4    Jan 23
robert   ttyp6    Jan 21 (dosh$1)
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)
```

```
$ uniq userlist
james    ttyp4    Jan 23
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)

$ uniq -c userlist
1 james  ttyp4    Jan 23
2 robert ttyp6    Jan 21 (dosh$1)
1 admis  ttyp7    Jan 23 (suna)

$ uniq -u userlist
james    ttyp4    Jan 23
admis    ttyp7    Jan 23 (suna)
```

`uniq` removes duplicate lines from files. Duplicate entries may appear when two or more files are merged (see `cat`) or when a file is updated by multiple processes. In order for `uniq` to work, the duplicate lines must be adjacent in the file. This is usually achieved by first sorting the file, and then by piping the output of `sort` into `uniq`.

```
sort filename | uniq
```

`uniq` does not have a sort operation built in since this contradicts the general philosophy in Unix of reuse. The shell provides glue which allows the output of one command to become the input of another, and thereby allows utilities to use the functionality of others.

Counting Words

wc counts lines, words and characters

- wc -l just report lines
- wc -w just report words
- wc -c just report characters

```
$ cat userlist
dpm      console  Jan 23
roger    ttyp0    Jan 23 (csgi55)
csc5gf   ttyp1    Jan 19 (csgi05)
csc6eq   ttyp2    Jan 22 (csgi47)
sta4rf   ttyp3    Jan 23 (csun02)
james    ttyp4    Jan 23
james    ttyp5    Jan 22 (csc$a)
robert   ttyp6    Jan 21 (dosh$1)
admis    ttyp7    Jan 23 (suna)
dpm      ttyp8    Jan 23
prj1gf   ttyp9    Jan 22 (blobby)
```

```
$ wc userlist
   17   19   556  userlist

$ wc -l userlist
   17   userlist

$ wc -w userlist
   91   userlist

$ wc -c userlist
   556  userlist
```

wc counts the number of words, characters and lines in a file. Any number of filenames may be specified and in the event of no filename, the standard input (output from preceding command or keyboard) is used.

Using command pipelines the number of files in a directory can easily be determined. If the output of ls is made the input of wc, then the number of lines is *almost* the number of files in the directory,

```
ls -l | wc -l
```

This turns out to be almost the right answer because ls takes the rather unusual behaviour or outputting a total in addition to one line per file. Consequently the count is out by one. The correct solution is determined by using

```
ls | wc -l
```

This works because ls outputs one filename per line whenever the output is redirected or piped. (ls cheats somewhat by only formatting its output if the actual stdout is a terminal device.)

wc may of course be applied to the output of any command to determine the number of lines, words or characters generated. To determine the number of users on the machine, for example,

```
who | wc -l
```

Transliterating Files - the tr command

`tr` translates characters from input to output

- Default input is STDIN
- Default output is STDOUT

```
$ tr 'aeiou' 'AEIOU'
Hello
HEllO
```

Many possibilities with arguments

- character classes

```
$ tr [:lower:] [:upper:]
Hello
HELLO
```

- white space manipulation

```
$ tr ' ' '\012'
hello world
hello
world
```

The tr commands takes two "strings" as its arguments. The first is a set of characters to match in the input, the second is a set of characters to which any matched strings are translated in the output (ie first char in first string translated to first char in second string, etc).

There are many ways of conveniently specifying groups or classes of characters in both input and output. We can also use octal representations of ASCII character codes.

Printing Files

- `lpr` sends files to a printer

```
$ lpr -Plw story1
```

- `lpq` queries the state of the print queue

```
$ lpq -Plw
lw is ready and printing
Rank     Owner    Job      Files       Total size
active   george   555      userlist    385 bytes
1st      george   556      story1      456 bytes
```

- `lprm` dequeues jobs from the print queue

```
$ lprm -Plw 556
suna2: dfA556sparc2 dequeued
suna2: dfA556sparc2 dequeued
```

`lpr`, `lpq` and `lprm` send, query and delete print jobs. `lpr` copies each of the files specified to the printer queue, from which they will eventually be printed. `lpr` takes any number of filenames arguments, and reads standard input if none are given. The `-P` option identifies a different destination printer.

`lpq` lists the jobs waiting to be printed, including a print job number. The number must be used with `lprm` in order to remove the print job.

`lpr` may also be used at the end of a pipeline of commands, thus allowing the output of any command to be directly printed.

```
ls -l | lpr
```

Searching for Files

- **`find` recursively searches any part of the file system**
  - it can print the file name found
  - it can search for files based on several file attributes
  - it can apply any Unix command to the files found

```
$ find / -name ls -print
/usr/bin/ls
/usr/lib/ls
/usr/5bin/ls
```

```
find <dir> -pred [-pred ...]
```

Searching for Files

- `find` understand a variety of predicates

```
-name <filename pattern>
-user <user name>
-group <group name>
-size [+|-] <size in blocks>
-perm [-] <octal number>
-atime [+|-] <days>
-mtime [+|-] <days>
-ctime [+|-] <days>
-type [d|f|l]
-inum <i-node number>
-print
-ls
```

- **Question predicates select files, action predicates operate on the files**

The `name` predicate selects files which match the file name pattern. The pattern may be a single string, a wild card similar to the wild cards used by the shell. Note, however, that since `find` has to interpret the wild cards, they must be quoted.

`user` selects files with owned by the specified user, and group selects files in the specified group. `size` selects files greater than (+), less than (-) or exactly equal to the specified size in 512 byte blocks. `perm` selects files based on their permission bits. Files exactly matching the specified octal pattern or files containing one of the specified permission bits (-) are selected.

`atime`, `mtime` and `ctime` relate to the files access, modification and creation (i-node) time stamps. The time is given as greater than (+), less than (-) or exactly equal to the specified days.

`type` is true for files of type d (directory), f (plain files) or l (symbolic links). `inum` is true if for files associated with the specified i-node number.

Action Predicates

`print` and `ls` are action predicates. The do not select files, they simply pass the files through to the next predicate. However, as a side effect an action is performed. `print` displays the file names, `ls` produces long listings of files.

Searching for files

- **list all files owned by root**

```
find / -user root -ls
```

- **print all files older than 3 months, greater than 100K**

```
find / -atime +90 -size +200 -print
```

- **all C programs from the current directory**

```
find . -name "*.c" -print
```

- **all SUID root programs and e-mail administrator**

```
find . -user root -perm -4000 -print \
| mail root@pluto
```

If `find` searches from a relative path name it generates relative names for the files which it finds. If it searches from an absolute path name then absolute names are generated.

This is relevant when find is being used to generate file names for archiving commands (`cpio`). Relative archives may be restored anywhere, whereas files in an absolute archives have to be restored into the same location from which they were copied.

Manipulating Files

- `find` can execute any Unix command on selected files

```
find . -name core -exec rm {} \; -print
```

- All core files in or below the current directory are deleted
  - {} represents the file being processed
  - ; terminates the Unix command
  - \ escapes the ; from the shell

- An alternative predicate prompts for confirmation

```
find . -name core -ok rm {} \; -print
```

The exec predicate makes any Unix command programmably recursive.

Exercise

- **Refer to the "Filters and Working With Files" exercise in the Exercises booklet.**


5. FILE ACCESS CONTROL

File Access Control

- File Access Rights
- Applying Access Rights
- Displaying Permissions
- Set ID Permission Bits
- Sticky Permission Bits
- Using Directory Permissions
- chmod & Setting Permissions
- Changing UIDs and GIDs

File Access Rights

- Prevent unauthorised access
- Prevent accidental damage
- Based on relationship between processes and files
- Every process is owned by someone and belongs to some group
- Every file is owned by someone and belongs to some group

Since Unix is a multi-user operating system it is important that the resources (and in particular files) allocated to one user are not unexpectedly affected by another.

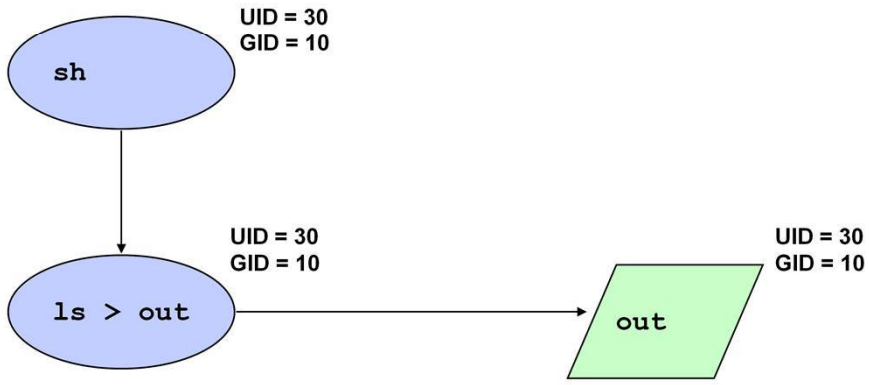
Unix employs a two layered model for security: authentication and authorisation. Authentication refers to the process by which the system determines that a user is who they claim to be. This is achieved in Unix using passwords. Authorisation concerns the rights of a user to access files around the system. This is controlled in Unix using file access permissions.

The permission model provided in Unix is based around the notion of ownership. Every process is owned by someone and every file belongs to someone. The various relationships between files and processes ownership enable appropriate access permissions to be applied.

Note that in Unix all processes start out as files (program files in the filesystem), and that devices are accessed through device files. As a consequence, the permission model for plain files applies equally to programs and devices.

Applying Access Rights

- **When you login your shell is owned by you and is stamped with your UID (User ID) and GID (Group ID)**
- **Almost every process you spawn and every file you create is stamped with your UID and GID**



When a user first logs into the machine, they are requested to supply a username and password. The verification that the password is correct is the process of authentication. If successful a login shell is invoked on behalf of the user. The shell is stamped with the users personal UID and shared GID.

The UID (User Identity) is generally unique to the user, and has a one to one mapping with the user's username. The GID is shared amongst other users involved in the same activities. As shall be seen, users sharing the same GID may be able to access files which users outside of the group cannot.

Every process spawned by the user, inherits the UID and GID established in their shell. Every file created by the user is also marked with their UID and GID. In effect, everything that the user does during their session is marked with their identity.

Applying Access Rights

- **Access rights are based on the relationship between processes and files**


```
if UID (process) = UID (file)
    apply users access rights
else if GID (process) = GID (file)
    apply group access rights
else
    apply others access rights
```

Almost all activities on Unix involve the interaction of at least one process and one file. For example, to display a simple message the echo process would need to access the user's terminal file.

Access permissions are applied when processes access files. Depending on the relationship between a process and a file, one of three sets of access permissions are applied. If both file and process are owned by the same person then the user (or owner) permissions are applied; otherwise, if both the file and process belong to the same group then the group permissions are applied; otherwise the other permissions are applied.

An important thing to notice with these permissions is that only one category can ever apply. It is possible that others may be able to access the file while the owner cannot! (However, the owner can change the permissions to grant themselves access.)

Super User

There are only two kinds of user in Unix, normal users and the superuser. The superuser has the username *root*, and unlike all other users is not bound by the permission mechanism. Therefore, the relationships outlined above do not apply. If the UID of the process is 0 (that is, root), then access to the file is granted.

Clearly the root user is dangerous. Users who find themselves requiring the privilege of root should take extreme care. Unix is unforgiving of errors.

Where network file systems are involved, a local root user does not have any special rights on remotely mounted files.

Permissions

- **Every file carries permissions for its owner (user), group and others (everyone else)**

- **Access rights for files**

r	*read*	read contents
w	*write*	update and implies delete
x	*execute*	attempt to run program
s	*set ID*	change the UID or GID of process

- **Access rights for directories**

r	*list*	list contents
w	*write*	create and delete *any* files
x	*search*	search directory (cd)
t	*sticky*	control write access to directories

Having determined the relationship between a process and the file it wishes to access, a set of permission bits are applied.

If the file is a plain file (non directory) then the process may be granted read, write, execute and/or set ID access. Read means the user may examine its contents (i.e., `cat` *`file`*), write means that the user may change the files contents (but not necessarily delete the file), and execute means that the user may attempt to run the file as a program. The file will crash if it does not contain program text. The set ID permission causes the new process to assume the identity (UID and/or GID) of the program file rather than the parent process.

If the file is a directory, then the process may list the contents of the directory, create or delete files within the directory and/or search the directory. Note that x is overloaded to mean search not execute in this context. Also note that the ability to write to a directory allows the process to overwrite or delete any of the files within the directory; even those for which the process does not have direct access. To modify this dangerous behaviour, the sticky bit on directories requires processes to have appropriate access to files within the directory, in addition to write access on the directory itself.

Displaying Permissions

- **Use the `ls -l` command to display access rights**

```
$ ls -l reports
drwxrw-r-- 2 greg   dude     512 Jan 21 19:58 aDirectory
-r--r--r-- 1 greg   dude     873 Jan 21 19:59 aFile
```

type	user	group	others
d	rwx	rw-	r--
-	r--	r--	r--

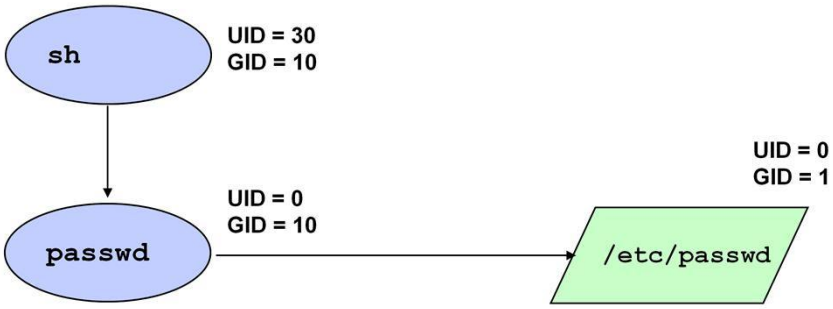
Use the `ls` command with the `-l` option to display the permission bits associated with a file.

The permission bits are organised in three sets of three bits, following the file type field. The sets identify the access rights for the owner (user), members of the group and all others.

Set ID Permission Bits

- **Upon execution UID and/or GID are taken from the program, not from the parent process**

```
$ ls -l /etc/passwd /usr/bin/passwd
-rw-r--r--   1 root   sys   931   Sep 17  13:19 /etc/passwd
-rwsr-xr-x   5 root   sys   32768 Jul 23   1992 /usr/bin/passwd
```



In most situations, when a user invokes a new command the resultant process inherits the UID and GID from the parent. In this way, the UID and GID established when the user first logged in, is passed on to all child processes. Everything the user does therefore carries the user's identity.

This mechanism is sometimes insufficient. Consider the situation of a user wishing to change their password. Passwords are physically stored in an encrypted form within the `/etc/passwd` file. Although users are able to read the contents of the file, they do not have permission to write to it. If write permission was granted, then users would be able to add new users and set the passwords of others to null. This would entirely undermine Unix security.

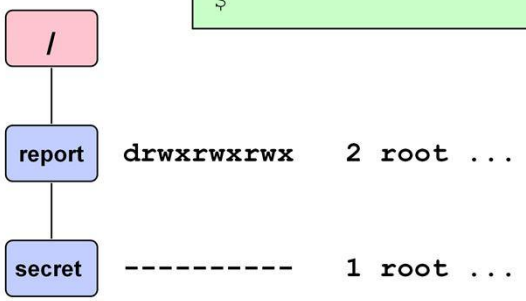
The problem is that the `passwd` program needs to change the `/etc/passwd` file when a user updates their password. When the program is executed by a user it should inherit the UID and GID from the user's shell. However, if this was the case, the process would not gain write access to the `passwd` file. Only root has write access to this file.

The solution is the set ID bit. The long listing of the `passwd` command above (`/usr/bin/passwd`) shows an s in place of the user's execute bit. This is the set UID bit (in place of the group's execute bit it is called the set GID bit). The permission tells the system to run the `passwd` command with the UID of the owner of the command, not with the UID of the parent process. In effect, the `passwd` command runs as root, even though it is invoked by a normal user.

Sticky Permission Bits

- **Files may be overwritten or deleted if the directory is writable**

```
$ whoami
fred
$ rm /report/secret
rm: override protection 0 for /report/secret? y
$ ls /report
$
```



It is sometimes necessary for directories to be writable by many or sometimes all users. For example, /tmp is a general purpose directory which any process may use to store temporary files.

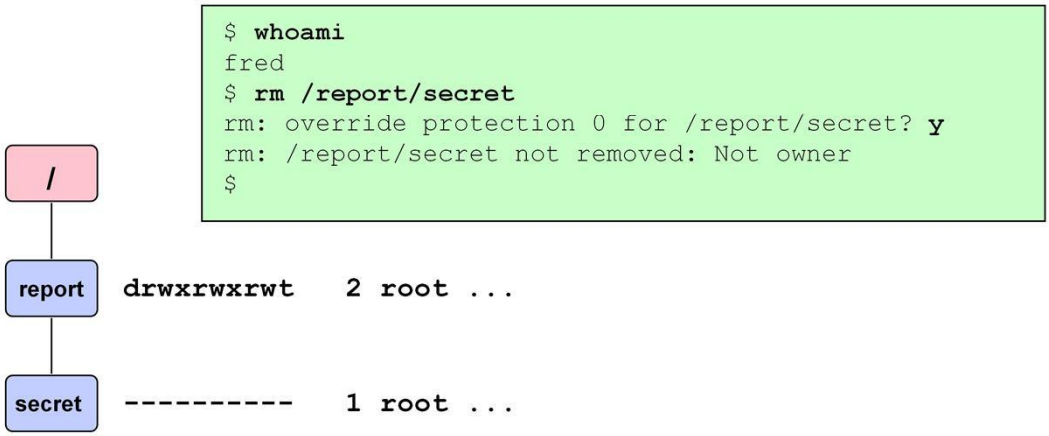
To achieve this, write permission is asserted for the user, group and other permission bits on the directory. A problem, however, is that write permission to a directory means that a user may create or delete any file in the directory, even if they do not have explicit access to the file itself.

In the example above, the file called secret which is owned by root with no access rights for anyone, is deleted by fred.

The reason that this can occur is that write access on a directory allows users to change the directory file itself. The directory is, in every respect, a file. However, the contents of the directory file are the names (and i-node references) of each of the files stored within it. By removing the name (recall that a name is just a link, and that rm simply unlinks file names) from the directory, the file within the directory is removed. The actual permissions on the file itself are irrelevant.

Sticky Permission Bits

- **Using the sticky bit user must have both write access to the directory and the file to overwrite or delete it**



To resolve the problems created by giving write access to directory files, the sticky bit may be used.

The sticky bit appears as at the end of the sequence of permissions. Although it sits over the execute bit for others, it actually applies to all of the write permissions in the sequence. The sticky bit is a write modifier. It changes the meaning of write such that users can only create files in a directory, or remove files from a directory, if they have appropriate access to the files in the directory, in addition to the directory itself. In other words, in order to delete a file, a user must be able to write to the directory, and to the file contained within it.

Note that the owner of the directory is still able to delete files which they do not have explicit write access to. This is because the owner of the directory has the ability to remove the sticky bit.

Setting Permissions

- `chmod` is used to change permissions

```
chmod [-R] <permission mode> file [files ...]
```

- The -R option enables recursion through a directory
- The permission mode is set using
  - a symbolic notation
  - a numeric notation

File permissions may be changed by the owner of the file or (of course) by the superuser.

The `chmod` command (change permission mode) allows the permissions to be changed on single files or directory structures. When specifying new permissions, either a symbolic or a numeric notation may be used.

Symbolic Notation

- **The symbolic notation allows file permissions to be changed relative to the current permission**

```
chmod  u  +  r  files ...
       g  -  w
       o  =  x
       a     s
             t
```

```
$ ls -l aFile
-rw------- 1 greg     873 Jan 21 19:59 aFile

$ chmod go+r aFile
$ ls -l aFile
-rw-r--r-- 1 greg     873 Jan 21 19:59 aFile

$ chmod g-r aFile
$ ls -l aFile
-rw----r-- 1 greg     873 Jan 21 19:59 aFile

$ chmod ug+rx,o-w aFile
```

In the symbolic notation codes are used to indicate who the permission should apply to, how the permission should be applied, and what permission is being changed.

The permission is applied to u (user), g (group) o (others), a (all of them), or any combination thereof. Permissions may be added, subtracted or assigned:

+	add new permission to the existing bits
-	remove new permission from the existing bits
=	override existing permission with new permissions

The last example above demonstrates how several symbolic expressions have be combined using a comma.

Numeric Notation

- **Treats the permissions as a bit pattern**
- **Each group of 3 bits is considered an octal value**

s s t	r w x	r - x	r - -	
4 2 1	4 2 1	4 2 1	4 2 1	
0+0+0	4+2+1	4+0+1	4+0+0	= 754

```
$ ls -l aFile
-rw------- 1 greg dude   873 Jan 21 19:59 aFile

$ chmod 777 aFile
$ ls -l aFile
-rwxrwxrwx 1 greg dude   873 Jan 21 19:59 aFile

$ chmod 456 aFile
$ ls -l aFile
-r--r-xrw- 1 greg dude   873 Jan 21 19:59 aFile
```

The numeric notation is based on the idea that permission bits are either on or off, and that they therefore have a binary representation. Since the bits are clustered into three sets of three bits, a three digit octal number representing the permissions is easily determined.

Note that the numeric notation does not allow the permission bits to be changed relative to their current value. However, this is possible in the symbolic notation using the + and - operators.

An extra leading octal digit may be used with the numeric notation. This provides for set UID (4) set GID (2) and sticky(1). Setting a file's permissions to 7777 would enable everything, `rwsrwsrwt`.

Note, a lower case `s` or `t` implies that the `x` permission is also enabled. If `x` is absent, then the `S` and `T` appear as capital letters.

Default Permissions

umask defines the default permission bits

```
$ umask
22
$ mkdir DIR
$ touch FILE
$ ls -ld DIR FILE
drwxr-xr-x  2 greg  dude         512 ...
-rw-r--r--  1 greg  dude           0 ...
```

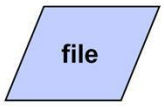
r w x	r w x	r w x		
4 2 1	4 2 1	4 2 1		
0+0+0	0+2+0	0+2+0	022	umask
4+2+1	4+0+1	4+0+1	755	default for directories
0+0+1	0+0+1	0+0+1		
4+2+0	4+0+0	4+0+0	644	default for files

When files and directories are first created, they receive a default set of permission bits. The default settings are based on a value maintained by the shell (and all processes) called the umask. The umask masks out those permissions which should not be asserted by default. It is therefore the opposite to the permission actually desired.

When a directory is created the maximum permission of 777 is masked by the value in the umask. By default this is 022, giving rise to files directories permissions of 755 (i.e., AND NOT umask with 777). In the case of plain files, it is inappropriate that they should be created with the execute permission set by default (not many files are programs). Consequently, 111 is also masked when they are created.

Changing Ownership

- **UID and GID of a file may be changed**



`chown`
`chgrp`

- **UID and GID of a process may be changed**

`login, su`
`newgrp`

If a process is unable to access a file, changing the files permission mode is not the only way to enable access. Permissions are based on the relationship of a file and a process. The relationship may be changed (and thus a different set of permissions will apply) by changing the UID or GID of the process or the file.

`chown` changes the ownership (UID) of the file, `chgrp` changes the group (GID) ownership of the file.

`login` causes the current user to logout and login as the new user, automatically attracting their UID and GID. `su` (switch user) allows one user to become another, leaving the original shell waiting. Once the user terminates the new shell, they return to the waiting parent shell.

`newgrp` allows the current shell to move into another group by changing its GID. To successfully execute this command you must be a member of the group you intend to move into (or provide a password if the group has been defined with one). The consequence of being in a new group is that access may now be granted to files based on the file's group permission bits.

Changing UIDs and GIDs

Changing file ownership and file group ownership

```
chown fred file
chown -R fred dir
chgrp staff file1 file2
chgrp -R staff dir1 dir2
chown fred:staff newfile
```

Changing the user's current group

- you must be a member of the new group

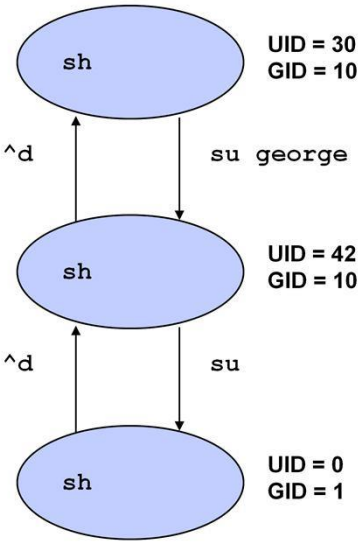
```
newgrp infotech
```

chown and chgrp require the new user or group, and the files which the command should be applied to. The operations may be applied recursively to a directory structure using the -R option.

In traditional Unix systems, users were permitted to change the ownership of files which they own. Therefore, one could give a file away but not take it back. However, due to various security problems associated with ownership, the chown command is no longer in the user's domain. Only the superuser can execute this command.

Switching UIDs and GIDs

- `su` enables one user to "become" another




```
$ id
uid=30(greg) gid=10(dude)
$ su george
Password: ***
$ id
uid=42(george) gid=10(dude)
$ su
Password: ***
# id
uid=0(root) gid=1(daemon)
# ^d
$ ^d
$ ^d

Solaris

login:
```

`su` allows one user to switch to another. By providing the user's username and password, `su` creates a new shell with the new user's UID and GID. Any commands subsequently executed are done so with the identity of the new user.

In the example above, the `id` command is used to determine the current UID and GID of the shell. By using `su`, *fred* is able to become *george*, and then *george* is able to become *root*. As each of the shells are terminated, control returns to the waiting parent shell. fred is logged out when he terminates his login shell.

Note, if `su` is used without a username argument then root is the default. su is often used with a dash argument

```
su - george
```

The dash tells the new shell to configure itself appropriately for the new user, and to relocate itself in the user's home directory. Without the dash, little more than the UID and GID are changed.

Exercise

- Refer to the "Permissions" exercise in the Exercises booklet.

6. EDITING FILES

Editing Files

- A Review of Unix Editors
- ed Line Editor
- ed Commands
- vi Screen Editor
- vi Operating Modes
- Setting the vi Environment
- Defining Macros & Abbreviations

Unix Editors

Unix editors exhibit different characteristics

`ed` interactive, buffered, line oriented

`vi` interactive, buffered, screen oriented

`sed` non-interactive, non-buffered, stream oriented
(discussed in later section)

Unix provides a variety of editors. Lots have been with the system for many years, others are recent editions. The three editors presented in this chapter are likely to be available on all versions of Unix.

`ed` is the simplest editor. It is interactive, buffered and line oriented. Files being edited are copied into memory, and edits are applied interactively to the copy. When complete, the changes may be written back (or discarded) to the original file. `ed`'s line orientation means that lines are edited one at a time.

`sed` is a streamed version of `ed`. Rather than copying the file into memory, `sed` successively reads lines from the input file, edits them, and then writes the output to the standard output. `sed` is useful within pipelines as a programmable filter.

`vi` is the primary full screen editor on Unix systems. Like `ed` it works on a copy of the file in memory, but unlike `ed` it allows the user to view the file a page at a time. `vi` (pronounced vee-eye) is an extremely powerful and sophisticated editor. As such, the learning curve tends to be rather steep.

Editors have always been near the heart of the Unix cult. Users either love or hate them. Another popular editor, which is often used in place of `vi`, is `emacs`. However, since this is not shipped with the Unix operating system, it is not always available.

ed Line Editor

- **ed is an interactive editor**
- **ed is available on all Unixes whatever their state**
  - it is the original Unix editor
  - minimal systems always make ed available
- **ed will work on any kind of terminal**
  - ed does not requires special screen handling capabilities
  - it even works on teletype paper terminals
- **ed can be used in batch mode**
  - it can be used inside shell programs to edit potentially hundreds of files automatically

ed has the distinction of being the oldest editor shipped with Unix. Although other editors are used in preference for day-to-day editing needs, ed is still useful in a number of situations.

When Unix systems are in their maintenance mode (in SunOS this is referred to as miniroot), the number of commands available is dramatically reduced. Since ed has the minimal system requirements (in terms of memory and terminal capabilities) it is often available when other editors are not.

ed is also useful within shell scripts. These are small programs containing sequences of shell and Unix commands. By embedding ed within such scripts, it is possible to automate the editing of a number of files. The editing could also be performed at night, if execution of the script is delayed until this time.

Using ed

```
ed [-p prompt] [filename]
```

```
$ ed edtext
?edtext: No such file or directory
a
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
.
w
100
q
$
```

- **Files are copied into memory for editing**

ed is not the most friendly of programs. When it is invoked the only dialogue into which it enters is to inform the user of errors. It writes a ? whenever something is wrong.

ed does not even use a prompt by default. The -p is useful if a prompt is required when using the program.

When ed starts it reads the contents of the file being edited into memory, displays the number of bytes read, and then waits. If the file does not exist, then a message is displayed to that effect (?<file> on Solaris).

ed has two modes of operation. The command mode allows users to supply commands to move around the file, perform copies and substitutions, and to write the edits onto disk. Some commands move the user into input mode. In this mode everything which is typed goes directly into the file, until a . is typed at the beginning of a line.

In the above example the session starts off with ed indicating that the file does not exist. Currently, the session is in command mode. By typing the command a, the session moves to input mode. All the characters subsequently typed go into the buffer. When complete, the full-stop (period) . returns the session to the command mode. The buffer is then written to the file (by default, the one used when ed was started), and the session terminated.

ed Commands

Commands have a simple and regular structure

```
[addr[,addr]]<character command> [parameters]

1                       i               insert
5,20                    a               append
/fred/          c               change
g/jane/                 d               delete
/john/,/jack/ p                 print
                        w file  write to file
                        r file  read from file
                        s/RE/RS/g       substitute RE for RS
                        t addr  transfer to addr
                        q               quit
                        Q               really quit
```

ed commands conform to a regular structure. They are used in the command mode of the editor to perform edits on part or all of the file.

Some commands cause the session to change to input mode (i, a, c), others causes lines to be deleted or printed (d,p), and others cause the data to be written to or read from a file.

Each command applies to the current line unless an address is specified. The current line is initially the first line in the file, but this will change to the most recent line edited, as the session progresses. To set the current line, simply type in a line number.

By using an address, commands may be applied to specific lines or (where appropriate) to ranges of lines. A specific line is indicated by using its line number or supplying a regular expression (RE). In the latter, the editor will skip forward until a line containing the RE is found. A range of addresses can also be supplied, ranging positively through the file. Ranges are defined between two line numbers or two REs, and are used inclusively. As a convenience,. refers to the current line, and $ to the last.

Sometimes it is useful to search a file for all lines which contain an RE and apply an edit. This is achieved with the g qualifier to the RE address. For example, to print all lines containing a specified RE

```
g/RE/p
```

In fact, this activity was so useful that a grep command (g/re/p) was written as a general utility.

There are two quit commands. q quits only if all the changes have been written to a file; Q forces the quit and discards unwritten changes.

Entering Text

```
$ cat edtext
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
```

```
$ ed -p '>' edtext
100
>3i
john    J Jack          p5
.
>1,$p
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
john    J Jack          p5
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
>
```

In the above example, the text on the left is the original file, and the text on the right shows the edited file.

In this example `ed` is invoked on a file called *`edtext`*, and a `>` prompt is specified. Once started, `ed` informs the user that the input file contains 100 bytes.

`3i` inserts the following text before line 3. `1,$p` prints all lines from 1 to the end-of-file, thus verifying that the edit was successful. However, no changes have yet been written to the original file.

Changing and Deleting Text

```
$ cat edtext
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
```

```
$ ed -p '>' edtext
100
>/greg/c
john    J Jack          p5
.
>1,$p
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
>
```

```
$ ed -p '>' edtext
100
>/Fletcher/,/Ball/d
>1,$p
neilb   Neil Bowers     p8
>
```

The file is searched for the first line containing the RE `greg`. This is then changed (replaced) by the following text.

In the second example all lines between the REs *Fletcher* and *Ball*, inclusive, are deleted. This is verified by `1,$p`. Note that the original file is still intact because the changes have not yet been written.

Reading Text

```
$ cat edtext
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
```

```
$ cat text
*** nearly time for tea ***

$ ed -p '>' edtext
100
>/george/r text
>1,$p
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
*** nearly time for tea ***
neilb   Neil Bowers     p8
>
```

The file is searched for a line containing the RE *dpm*. Once found the contents of *edtext* are read again, and appended after this line.

Writing Text

```
$ cat edtext
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
```

```
$ ed -p '>' edtext
100
>3,$w temp
>Q

$ cat temp
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
```

Lines 3 to the end-of-file are written to a file called *temp*. The original file, *edtext*, is left unchanged.

Text Transfer and Substitution

```
$ ed -p '>' edtext
100
>2,3t$
>1,$p
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
greg    G Fletcher      p4
greg    G Fletcher      p6
>
```

```
$ ed -p '>' edtext
100
>1,$s/G/Greg
>1,$p
shaun   Shaun Fletcher  p2
greg    Greg Fletcher      p4
greg    Greg Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
>
```

Note that both of the above boxes contain examples. In the left box all of the lines from 2 to 3 inclusive, and transferred (copied) to the end of the file.

The right had box shows the use of the substitute command. From line 1 to the end-of-file, the first *G* on each line is replaced with *Greg*. Note that further Gs on each line are left unaltered. To make the substitute command apply itself to all instances of G on each line, the g (global) qualifier must be used

```
1,$s/G/Greg/g
```

vi Screen Editor

- `vi` is an interactive editor
- `vi` is the most widely used Unix editor
- `vi` is based on `ex` (similar to `ed`)

Using vi

vi [-r] [filename]...

```
$ vi vitext
```

```
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
~
~
~
~
~
~
"vitext" 5 lines, 100 characters
```

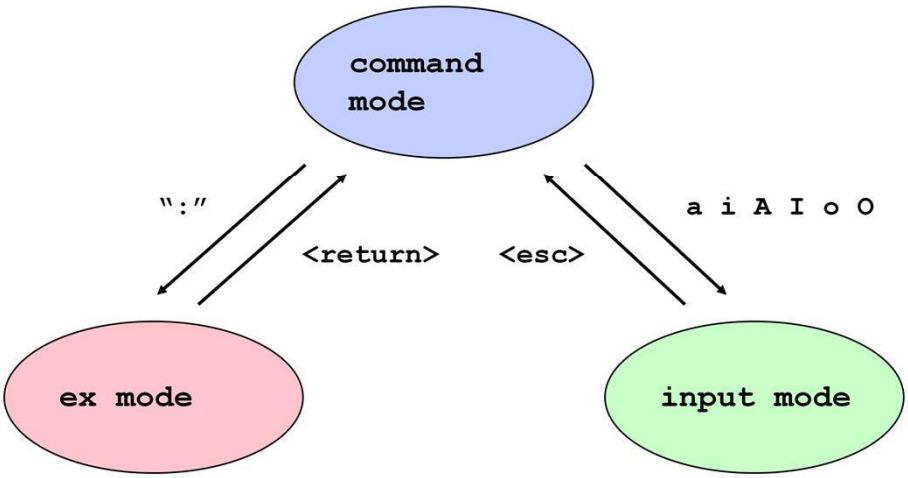
- **Files are loaded into a buffer**

`vi` may optionally be invoked with a list of file names being those files to be edited. When many files are edited together, `vi` allows named paste buffers to be shared between the files.

The `-r` option allows files to be recovered in the event that the system crashes during an edit session. The mechanism works because `vi` maintains a journal file on the local machine from which the original file can be recovered. Note that using AFS and NFS users may login to virtually any machine to access their files. However, vi recovery journals are usually kept on the machine where the `vi` crashed. Therefore, login to the original machine to recover lost files.

vi Modes

- vi has three modes



In command mode virtually every character on the keyboard now defines a command. Some commands move around the document some command copy or delete text. Several commands change mode and move into the input mode. In this mode the characters have their literal meanings and go into the file. To leave input mode type escape.

The ex mode supports the ed and sed commands previously discussed. Use the ex mode to apply global operations, to insert other files and to write and exit the editor.

Command Mode

- **General command format**

```
[count]command[arg]
```

- `count` **repeatedly applies the command**
- `arg` **affects the operation of the command**
- `vi` **is case sensitive**

Learning `vi` can be difficult because there are hundreds of codes mapped onto keystrokes which must be remembered. However, the overall structure is quite simple.

When in command mode each character is a command. Upper case is distinct from lower case and from control. Some commands require further parameters, such as `d`. This command means delete, but further indication must be given of what is to be deleted. `dw` deletes a word, `d$` deletes to end of line, `dd` deletes the current line, and so on. Commands may also be prefixed with a count which (in most cases) indicates the number of times the command should be performed. So, for example, `5dd` will delete five lines from the current down.

Moving Around the Text Buffer

h	cursor left	w	word forward	^	start of line
j	cursor down	b	word backward	$	end of line
k	cursor up ^u	page up		G	go to line
l	cursor right	^d	page down		

```
Once upon a time there were three
bears, who lived in the deepest
darkest wood.  One day, their
weetabix was stolen
```

- **Movement commands can be modified**
  - `10h` **cursor left 10 characters**
  - `4w` **forward 4 words**
  - `8G` **go to line 8**
  - `G` **go to last line**

Arrow keys can also be used on most terminals.

Adding and Replacing Text

i	insert before cursor	r	replace character
I	insert at start of line	R	overwrite rest of line
a	insert after cursor	o	open line below
A	insert at end of line	O	open line above

```
shaun   Shaun Fletcher  p2
greg    G Fletcher      p4
greg    G Fletcher      p6
george  G Ball          p7
neilb   Neil Bowers     p8
~
~
~
~
~
~
"vitext" 5 lines, 100 characters
```

Deleting Text

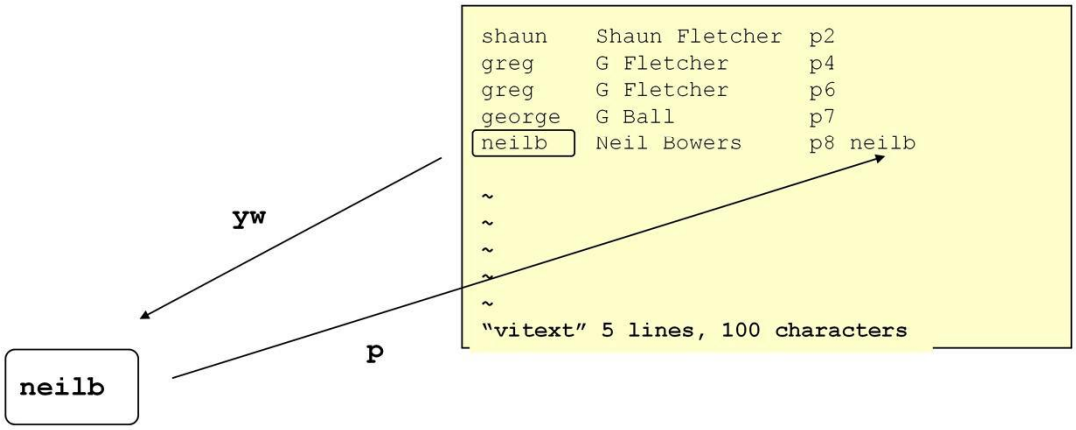
`x`	delete current character
`X`	delete character before cursor
`d`	delete text
`D`	delete the rest of the line

- **delete commands can be modified**

`3x`	delete next 3 characters
`d1G`	delete current line and all lines above it
`5dd`	delete next 5 lines
`d/fred`	delete until an occurrence of "fred"

Cut and Paste

y	yank text into buffer	"ay	yank to named buffer
p	paste buffer before	"ap	paste from named buffer
P	paste buffer after		



Text may be copied or moved from one place to another using the paste buffers. There are nine default buffers used implicitly with commands like yw (yank a word), dw (cut a word), and p (paste whatever is in the top buffer).

The default buffers are essentially stacked such that a new yank or copy places the text into buffer 1. A subsequent yank or cut moves the contents of buffer 1 to buffer 2, and then places the new text into buffer 1. This behaviour continues through to buffer 9, after which the data is lost. To access buffers other than 1 the address command should be used, as in "4p to paste the contents of the buffer 4. Any letter can in fact be used to name a buffer, so "ayy yanks the current line into buffer *a*, whilst "ap pastes it down again.

Search Commands

/	search forwards for RE	.	repeat last change
?	search back for RE	u	undo last change
n	repeat search	N	reverse search

```
shaun    Shaun Fletcher  p2
greg     G Fletcher      p4
greg     G Fletcher      p6
george   G Ball          p7
neilb    Neil Bowers     p8
~
~
~
~
~
/p6$
```

Search for the next line that ends in *p6*. This is, of course, a regular expression forward search through the file.

Note that searching forward or backward through the file will wrap when the bottom or top of the file is found.

ex Mode

- ex mode allows you to enter ex (ed) commands

`1,10d`	delete lines from 1 to 10
`/george/w file`	write all lines until george to file
`r story1`	read story1 at current position

The example illustrates how to do a global search and replace within `vi`. The address range `1,$` indicates the range of lines for the substitution, and the `g` on the end indicates that all matches on each line should be substituted. Naturally, '*this*' in the substitute may be a regular expression.

vi Environment

Vi is highly customizable

- use `set` from ex mode

```
~
~
:set all
noautoindent            nonumber        noslowopen
autoprint               nonovice        nosourceany
noautowrite             nooptimize      tabstop=8
directory=/usr/tmp      prompt          wrapmargin=0
[Hit return to continue]
```

set all	show all set options available
set autoindent	indent each line automatically
set number	number each line
set showmode	show vi mode
set tabstop=n	set the tab width
set wrapmargin=5	automatic text wrap

Use showmode to get vi to display the current mode of operation. In the bottom right corner of the screen a message reminds the user of the current mode.

Initialising vi

- **When vi is invoked it reads from an initialisation file**
  - defines macros and abbreviations
  - defines the environment

```
$ cd
$ cat .exrc
set autoindent
set showmode
map z O/*^[jo*/^[j
abb mroe more
```

When `vi` is invoked it reads the `.exrc` file from the user's home directory. This defines the initial environment, mappings and abbreviations used by `vi`.

In fact, vi looks in the current directory before looking in the home. If a file called `.exrc` exists locally, then this is used instead of the main file in the home. This enables `vi` to be customised depending on the directory the user is currently working from.

Also note that a shell environment variable called `EXINIT` may be used to initialise `vi`. Environment variables are covered later in the course.

Leaving vi

From the command mode

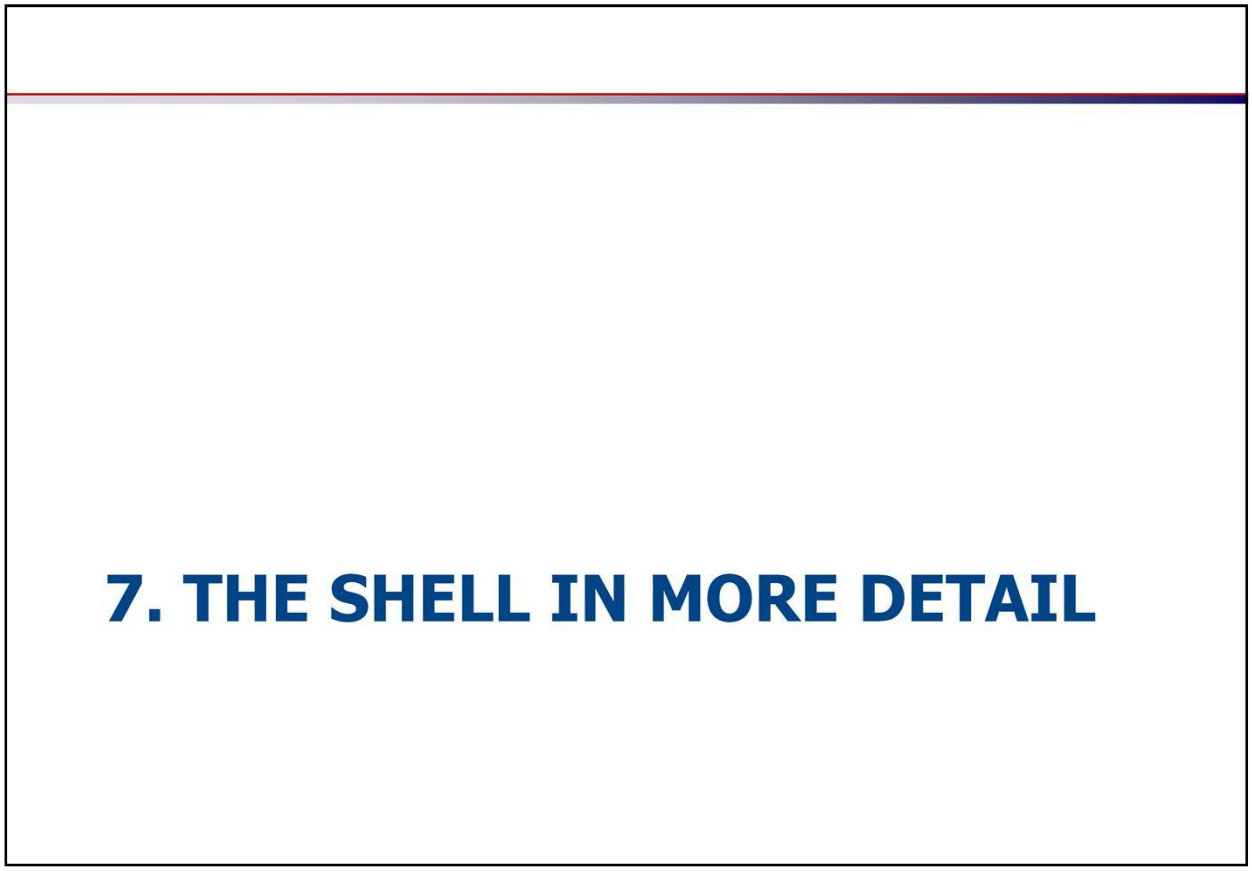
`ZZ` write file then quit

From the ex mode

`:wq` write file then quit
`:x` write file then quit (same as `wq`)
`:q!` force quit without save
`:q` quit

Exercise

- **Refer to the "Editors" exercise in the Exercises booklet.**

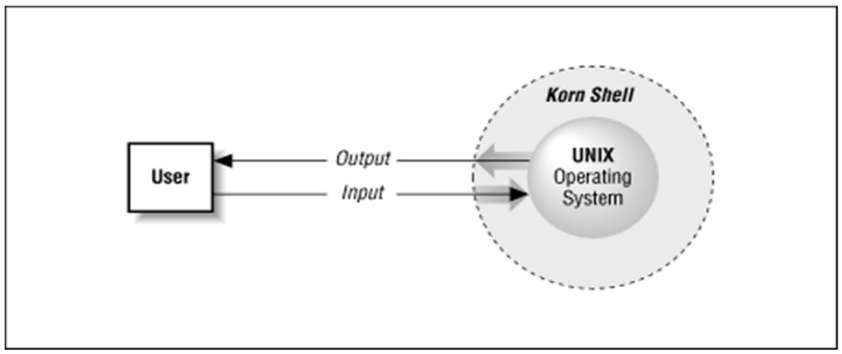

7. THE SHELL IN MORE DETAIL

The Shell In More Detail

- What Is a Shell?
- Features of the Korn Shell
- Interactive Shell Use
- Filenames and Wildcards
- Standard I/O
- I/O Redirection
- Pipelines
- Controlling Jobs
- Quoting
- Process Lifecycle
- Shell Variables
- Aliases

Review: What Is A Shell?

- **The shell is a layer around the UNIX operating system**
- **The shell can be any program that:**
  - Takes input from the user
  - Translates it into instructions that the operating system can understand
  - Conveys the operating system's output back to the user



The shell's job, then, is to translate the user's command lines into operating system instructions. For example, consider this command line:

```
sort -n phonelist > phonelist.sorted
```

This means, "Sort lines in the file phonelist in numerical order, and put the result in the file phonelist.sorted." Here's what the shell does with this command:

1. Breaks up the line into the pieces `sort`, `-n`, `phonelist`, `>`, and `phonelist.sorted`. These pieces are called words.

2. Determines the purpose of the words: `sort` is a command, `-n` and `phonelist` are arguments, and `>` and `phonelist.sorted`, taken together, are I/O instructions.

3. Sets up the I/O according to `> phonelist.sorted` (output to the file `phonelist.sorted`) and some standard, implicit instructions.

4. Finds the command `sort` in a file and runs it with the option `-n` (numerical order) and the argument phonelist (input filename).

Of course, each of these steps really involves several substeps, each of which includes a particular instruction to the underlying operating system.

Remember that the shell itself is not UNIX-just the user interface to it. UNIX is one of the first operating systems to make the user interface independent of the operating system.

Features Of The Korn Shell

- **The Bourne shell is the "standard" shell, however ...**
- **The Korn shell is becoming increasingly popular and includes the best features of the C shell as well as several advantages of its own such as:**
- **Command-line editing modes**
- **Job control**
  - Giving the ability to stop, start, and pause any number of commands at the same time
- **New options and variables for customization**
- **Expanded programming including:**
  - Function definition
  - More control structures
  - Built-in regular expressions and integer arithmetic
  - Advanced I/O control
  - And more ...

Although the Bourne shell is still known as the "standard" shell, the Korn shell is becoming increasingly popular and is destined to replace it. In addition to its Bourne shell compatibility, it includes the best features of the C shell as well as several advantages of its own. It also runs more efficiently than any previous shell.

The Korn shell's command-line editing modes are the features that tend to attract people to it at first. With command-line editing, it's much easier to go back and fix mistakes than it is with the C shell's history mechanism-and the Bourne shell doesn't let you do this at all.

The other major Korn shell feature that is intended mostly for interactive users is job control, which gives the ability to stop, start, and pause any number of commands at the same time. This feature was borrowed almost verbatim from the C shell.

The rest of the Korn shell's important advantages are mainly meant for shell customisers and programmers. It has many new options and variables for customization, and its programming features have been significantly expanded to include function definition, more control structures, built-in regular expressions and integer arithmetic, advanced I/O control, and more.

Interactive Shell Use

- **You engage in a login session that begins when you log in and ends when you exit or press `CTRL-D`**
- **Command lines are lines of text ending in `RETURN` that you type in to your terminal or workstation.**
  - By default, the shell prompts you for each command with a `$`
- **Shell command lines consist of one or more words**
  - Separated on a command line by blanks or TABs
  - The first word on the line is the *command*
  - The rest (if any) are *arguments* (also called *parameters*) to the command
  - Arguments are often names of files, but not necessarily
  - An *option* gives the command specific information on what it is supposed to do

```
lp -d printer1 -h afile
```

When you use the shell interactively, you engage in a login session that begins when you log in and ends when you exit or press CTRL-D. You can set up your shell so that it doesn't accept CTRL-D, i.e., it requires you to type exit to end your session.

During a login session, you type command lines in to the shell; these are lines of text ending in RETURN that you type in to your terminal or workstation.

By default, the shell prompts you for each command with a dollar sign, though this can be changed.

Commands, Arguments, and Options

Shell command lines consist of one or more words, which are separated on a command line by blanks or TABs. The first word on the line is the command. The rest (if any) are arguments (also called parameters) to the command, which are names of things on which the command will act

An option is a special type of argument that gives the command specific information on what it is supposed to do. Options usually consist of a dash followed by a letter - this is a convention rather than a hard-and-fast rule

Sometimes options take their own arguments. For example:

```
lp -d printer1 -h afile
```

has two options and one argument. The first option is `-d printer1`, which means "Send the output to the printer (destination) called `printer1`". The second option suppresses header printing, and the argument is the name of the file to be printed

Filenames & Wildcards

```
$ ls
five    one     three
four    six     two

$ f*
five  four
$ echo f*
five four

$ ???
one  six  two
$ [!t]*
five  four  one   six
$ [ft]*
five   four   three  two

$ g*
g*: No such file or directory
$ echo g*
g*
```

`?`	Any single character
`*`	Any string of characters
`[set]`	Any character in *set*
`[!set]`	Any character *not* in *set*

Sometimes you need to run a command on more than one file at a time. The most common example of such a command is `ls`, which lists information about files. In its simplest form, without options or arguments, it lists the names of all files in the working directory except special hidden files, whose names begin with a dot (.).

Filenames are so important in UNIX that the shell provides a built-in way to specify the pattern of a set of filenames without having to know all of the names themselves. You can use special characters, called wildcards, in filenames to turn them into patterns.

The ? wildcard matches any single character, so that if your directory contains the files `program.c`, `program.log`, and `program.o`, then the expression `program.?` matches `program.c` and `program.o` but not `program.log`.

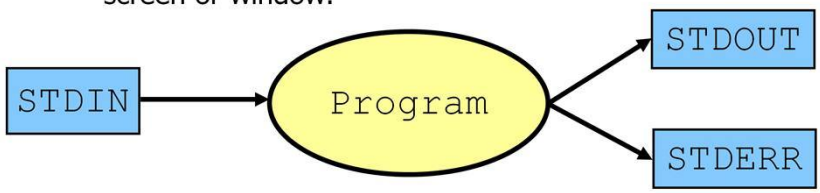
The asterisk (*) is more powerful and far more widely-used; it matches any string of characters. The expression `program.*` will match all three files in the previous paragraph. Notice that * can stand for nothing: both `*ed` and `*e*` match `ed`. Also notice that the last example shows what the shell does if it can't match anything: it just leaves the string with the wildcard untouched.

The remaining wildcard is the set construct. A set is a list of characters (e.g., `abc`), an inclusive range (e.g., `a-z`), or some combination of the two. If you want the dash character to be part of a list, just list it first or last.

The process of matching expressions containing wildcards to filenames is called wildcard expansion. This is just one of several steps the shell takes when reading and processing a command line.

Standard I/O

- **Each UNIX program has:**
  - A single way of accepting input called *standard input* (`STDIN`)
  - *A* single way of producing output called *standard output* (`STDOUT`)
  - And a single way of producing error messages called *standard error* (`STDERR`)
- **All shells handle standard I/O in basically the same way**
- **Each program that you invoke has all three standard I/O channels set to your terminal or workstation**
  - Standard input is your keyboard, and standard output and error are your screen or window.



By convention, each UNIX program has a single way of accepting input called standard input, a single way of producing output called standard output, and a single way of producing error messages called standard error output, usually shortened to standard error. Of course, a program can have other input and output sources as well, as we shall see.

Standard I/O was the first scheme of its kind that was designed specifically for interactive users at terminals, rather than the older batch style of use that usually involves decks of punch-cards. Since the UNIX shell provides the user interface, it should come as no surprise that standard I/O was designed to fit in very neatly with the shell.

All shells handle standard I/O in basically the same way. Each program that you invoke has all three standard I/O channels set to your terminal or workstation, so that standard input is your keyboard, and standard output and error are your screen or window. For example, the `mail` utility prints messages to you on the standard output, and when you use it to send messages to other users, it accepts your input on the standard input. This means that you view messages on your screen and type new ones in on your keyboard.

When necessary, you can redirect input and output to come from or go to a file instead, or you can also hook up programs into a pipeline, in which the standard output of one program feeds directly into the standard input of another.

This makes it possible to use UNIX utilities as building blocks for bigger programs. Many UNIX utility programs are meant to be used in this way: they each perform a specific type of filtering operation on input text.

I/O Redirection

- **Redirection changes:**
  - Where the data passed to `STDIN` comes from
  - Where the data from `STDOUT` and `STDERR` goes to
- **Redirected input reads from a specified file instead of the keyboard**

```
$ cat < one
```

- **Redirected output send data to the specified file instead of the terminal**
  - Redirecting `STDOUT` and `STDERR` to the same file requires special handling

```
$ date > now
$ ls one two madeup >list 2>errors
$ ls one two madeup >list2 2>&1
```

The shell lets you redirect standard input so that it comes from a file. The notation `command < filename` does this. The shell sets things up so that command takes standard input from a file instead of from a terminal.

For example, if you have a file called fred that contains some text, then:

```
$ cat < fred
```

will print fred's contents out onto your terminal, and:

```
$ sort < fred
```

will sort the lines in the fred file and print the result on your terminal (this is as if the utilities don't take filename arguments).

Similarly, `command > filename` causes the command's standard output to be redirected to the named file.

The classic example of this is:

```
$ date > now
```

The `date` command prints the current date and time on the standard output; the above command saves it in a file called now.

Input and output redirectors can be combined. For example: the `cp` command is normally used to copy files; if for some reason it didn't exist or was broken, you could use cat in this way:

```
$ cat < file1 > file2
```

This would be similar to:

```
$ cp file1 file2
```

More on I/O Redirection

- **Normal redirection with > causes destination file to be overwritten**
  - Even if command does not execute
- **Use >> to append to existing file**
  - Still creates file if it does not already exist

```
$ date > now
$ ls one two madeup >> now
$ cat now
Tue Jul  3 13:47:19 BST 2007
one
two
madeup
```

Normal redirection of output causes the destination file to be emptied. The shell does this before even attempting tp execute the command, so even if nothing happens, our file will still lose its contents.

To prevent this, use the >> sequence, this appends output to the existing file. If the file does not exist it will still be created however.

Redirecting Input - Here Documents

- **Use to redirect STDIN from command itself**
  - Very useful in shell scripts
- **Use <<**

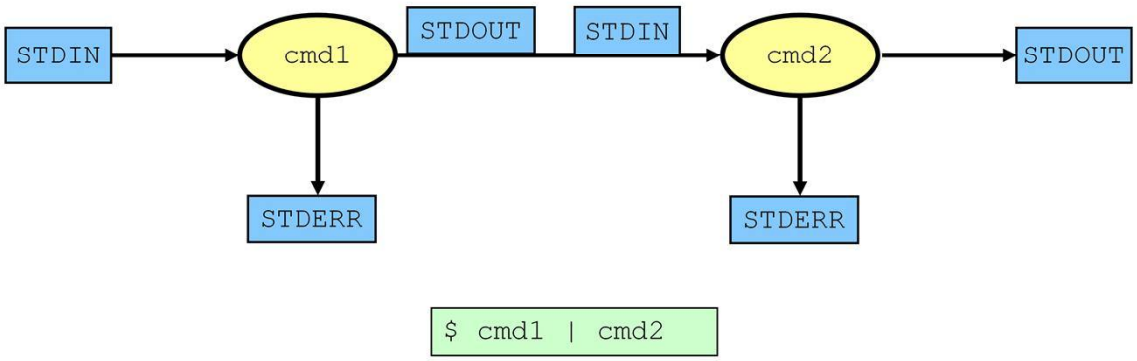
```
$ mail somebody@somewhere.com << END
This is a test message.
It can have multiple lines if
necessary.
It ends at the line beginning END
END
```

This becomes STDIN for the command

Here documents may look strange in interactive shell usage, but are extremely powerful when constructing scripts.

Pipelines

- **Pipelines send the `STDOUT` from one process to the `STDIN` of another**
  - `STDERR` is still sent to the terminal, unless otherwise redirected



It is also possible to redirect the output of a command into the standard input of another command instead of a file. The construct that does this is called the pipe, notated as |

A command line that includes two or more commands connected with pipes is called a pipeline.

Pipes are very often used with the more command, which works just like cat except that it prints its output screen by screen.

Pipelines can get very complex and they can also be combined with other I/O directors. To see a sorted listing of the file filea a screen at a time, type:

```
$ sort < filea | more
```

Here's a more complicated example. The file `/etc/passwd` stores information about users' accounts on a UNIX system. Each line in the file contains a user's login name, user ID number, a password marker, home directory, login shell, and other info. The first field of each line is the login name; fields are separated by colons (:), for example:

```
joe:x:284:93:Joe Bloggs:/home/joe:/usr/bin/ksh
```

To get a sorted listing of all users on the system, type:

```
$ cut -d: -f1 < /etc/passwd | sort
```

The cut command extracts the first field (`-f1`), where fields are separated by colons (`-d:`), from the input. The entire pipeline will print a list that looks like this:

```
bob
frank
...
```

Background Jobs

- **UNIX is a multi-tasking operating system**
  - It can many things at once
- **The shell allows you to control many programs at the same time**
  - Jobs can be put into the background and be left to run there
- **Follow the command with the `&` character**
- **List all jobs in the shell using the `jobs` command**

```
$ ls -lR / >files.txt &
[1]     564
$ ps
   PID TTY      TIME CMD
   528 pts/1    0:01 ksh
   564 pts/1    0:01 ls
   565 pts/1    0:00 ps
$ jobs
[1] +  Running    ls -lR / >files.txt &
```

UNIX permits multiple things to run at the same time, which means running more than one program at the same time. You do this when you invoke a pipeline; you can also do it by logging on to a UNIX system as many times simultaneously as you wish. The shell also lets you run more than one command at a time during a single login session. Normally, when you type a command and hit RETURN, the shell will let the command have control of your terminal until it is done; you can't type in further commands until the first one is done.

If you want to run a command that does not require user input and you want to do other things while the command is running, put an ampersand (&) after the command. This is called running the command in the background, and a command that runs in this way is called a background job. In contrast, a job run the normal way is called a foreground job. When you start a background job, you get your shell prompt back immediately, enabling you to enter other commands.

The most obvious use for background jobs is programs that take a long time to run, or for graphical applications, such a starting a web browser from the command line.

Right after you type the command, you will see a line like this:

```
[1] 564
```

followed by your shell prompt, meaning that you can enter other commands. Those numbers give you ways of referring to your background job. The number in the [] is the job number, and is specific to this instance of the shell. The second number is the PID (process ID) and is unique across the who of the system on which it is running.

Moving Jobs To The Background Or Foreground

- **The Korn shell provides built-in commands to manage jobs:**
  - `jobs` list the jobs started in the shell
  - `bg %n` will move a job to the background
  - `fg %n` will move a job into the foreground
  - `stop %n` will stop a background job – a stopped job can be restarted
  - `^Z` will stop a foreground job
- **For each of these commands, the `n` is the number in the square brackets from the `jobs` command**

```
$ ls -lR / >files.txt &
[1]     567
$ jobs
[1] +  Running           ls -lR / >files.txt &
$ stop %1
$ jobs
[1] + Stopped (SIGSTOP)  ls -lR / >files.txt &
```

Jobs may be moved between the foreground and background, or temporarily stopped, and then started again at a later time.

The jobs command lists all of the jobs that are currently controlled by the shell, including those that have been stopped. For all of the commands which manipulate these jobs, the number is square brackets is used, preceded by the % character.

`bg %n` will put job number n into the background.

`fg %n` will bring job number n into the foreground.

`stop %n` will stop job number n in the background.

Typing ^Z will stop a job that is running in the foreground.

To kill a given process, use the `kill` command, with either the job number or PID for the process, e.g., for the command running above, it could be killed using:

```
$ kill %1
```

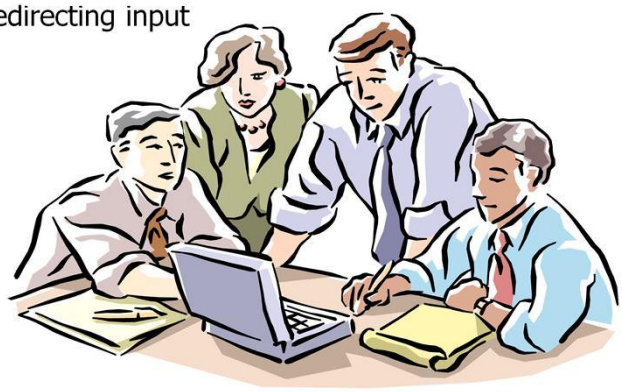
but ONLY from within the shell from which it was started, or:

```
$ kill 567
```

from any shell you are running on the system.

How Shell Interprets Commands

- **The shell reads and interprets what you type on the command line**
- **The order in which it processes your input is:**
  1. Reads the line
  2. Parses your input
  3. Processes directives, such as redirecting input
  4. Runs the command
  5. Provides a new shell prompt



After the shell has provided the user with a prompt, they can then type in a command, for example:

```
$ ls –l *.txt > output.txt
```

The first thing that the shell does is to parse the text that has been entered.

It finds tokens that mean something to the shell.

In this example, `ls` is the name of the command that will be executed, the > character means that the shell has to manage the redirection of output to a file called `output.txt`, and the * indicates that the shell has to expand the wildcard into all matching filenames, and pass those names to the command.

The first task will the redirecting the output, followed by the wildcard expansion, finally executing the `ls` command.

The shell will then have to wait until the command has finished executing before giving the user a new prompt to type in the next command.

Creating New Processes

- **Typing a command spawns a child process**
  - A child process is forked from the parent
  - The child gets its `pid` and `ppid`
  - The parent receives the value 0 if all is ok
- **The parent process then waits until the child process ends**
- **The child process inherits the parents attributes**
- **The child process runs**
- **When the child process dies:**
  - Its exit code is sent to its parent, waking the parent process up
  - NO other information is passed back

A new process is started by typing a command at the prompt

A `fork` system call is made which creates the new process. This spawns a child process, which inherits it's parents global variables. The parent is sent a return value from the `fork` – it should be `0` if the fork was successful. The child process receives it's PID from the fork.

The parent then receives a `WAIT` signal, and will do nothing else until the child process dies.

The child process executes, and when it has finished, send it's an `EXIT` signal is sent to the parent process notifying it. No other information is passed back to the parent process, so, if, for example, the child process changes it's `PATH` variable, this change is lost when the child process dies.

The parent process will then continue normally.

Shell Variables

- **A shell variable:**
  - Has a String value
  - Can be interpreted as a number
  - Can be initialised by the system
  - Can be set by you
  - By convention has a name that is in uppercase
  - Can be global
  - Can be local

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables.

Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell.

Local shell variables are used only by that shell and are not passed on to other processes.

A child process cannot pass a variable back to its parent process.

The shell and some text processing programs will allow meta-characters, or wild cards, and replace them with pattern matches.

Common Shell Variables

- **PATH**
  - Paths to be searched for commands, e.g. `/usr/bin:/usr/local/bin`
- **PWD**
  - Present working directory
- **EDITOR**
  - The path to your default editor, e.g. `/usr/bin/vi`
- **GROUP**
  - Your login group, e.g. `staff`
- **IFS**
  - Internal field separators, usually any white space (defaults to tab, space and <newline>)
- **PS1**
  - The primary prompt string, Bourne and Korn shells only (defaults to `$`)
- **PS2**
  - The secondary prompt string, Bourne and Korn shells only (defaults to `>`)
- **SHELL**
  - The login shell you're using, e.g. `/usr/bin/csh`
- **TERM**
  - Your terminal type, e.g. `xterm`
- **HOME**
  - Path to your home directory, e.g. `/export/home/joe`

Many environment variables will be set automatically when you login.

You can modify them or define others with entries in your startup files or at anytime within the shell.

Some variables you might want to change are `PATH` and `DISPLAY`.

The `PATH` variable specifies the directories to be automatically searched for the command you specify.

You set a global environment variable with a command similar to the following for the Bourne and Korn shells:

```
$ NAME=value; export NAME
```

You can list your global environmental variables with the env command. You unset them with the `unset` (Bourne shell) commands.

To set a local shell variable use the set command with the syntax below for the Bourne and Korn shells:

```
$ name=value
```

The current value of the variable is accessed via the `$name`, or `${name}`, notation.

e.g.

```
$ echo $LOGNAME
frank
```

Local and Global Shell Variables

- **Recall most commands require new process to be created**
  - Child process inherits data from parent
  - Shell variables are part of the shell's data
- **Local shell variables not seen in child processes**
- **Use export command to place variables in global "environment"**

```
$ MYVAR=hello
$ export MYVAR
```

When we set a variable, it is by default local to the shell. This means that it will not be visible to commands executed by the shell in sub-processes. There are some variables that are important to executed commands, for example the vi editor needs access to the TERM variable to know how it should drive the full-screen operations.

To make a shell variable global, use the export command. After this, it is visible in all sub-processes.

Note that the subprocess sees a copy of the global variable, if it changes the value, these changes are reflected in processes started from the subprocess, but will not be reflected back into the parent process.

Examining Variables

Notice how the set command displays both local and global variables.

Aliases

- **Aliases allow you to create alternative commands**
- **For example:**
  - If you have a habit of typing something incorrectly
  - To prevent the use of dangerous commands, such as `rm -r`
- **To create an alias**

```
$ alias alias_name=string

For example:
$ alias l="ls -l"
```

- **To list the currently set aliases**

```
$ alias
cat=/usr/bin/cat
r='fc -e -'
..
```

- **To remove an alias**

```
$ unalias alias_name

For example:
$ unalias l
```

Aliases can be used to create shortcuts to commands that you regularly use

This is done quite simply as follows:

```
$ alias me="/usr/ucb/whoami"
```

Aliases can also be used to make sure that when you type in a command, any dangerous options are made a little more safe.

For example

```
$ alias rm="rm -i"
```

Now the `rm` command will always prompt you, even if you use the `-r` option

Using the `alias` command by itself will list the currently set aliases

To remove an alias, use the `unalias` command. If you merely set the alias to do nothing, it will still exist:

```
$ alias me=""
```

Exercise

- Refer to the "The Shell" exercise in the Exercises booklet.

8. BASIC SHELL PROGRAMMING

Basic Shell Programming

- Shell Programming
- A Simple Script
- Command Line Arguments
- Interactive Input
- Conditional Expressions
- The test Command
- The case Statement
- for Loops
- while Loops
- Numerical Expressions
- Exercises

Shell Programming

- **Shell programs are called scripts and contain collections of Unix commands**
- **Bourne shell programming is the most popular**
  - most scripts in Unix are written in the Bourne shell
  - system administration scripts are Bourne
  - consistent with ksh
- **The Bourne shell provides**
  - string variables (no numeric variables)
  - conditional statements
  - looping constructs
  - interactive input
  - all the usual glue

In addition to the shell's various roles, shell commands may also be placed into executables files to form shell programs or scripts. Anything which can be typed on the command line may be placed into a script, and vice versa.

Shell scripts act primarily as glue; binding together collections of Unix commands and utilities to perform some task. The are often used when sequences of Unix commands have to be executed together, and are particularly useful in Unix administration.

The shell offers a variety of programming constructs, including conditional statements and looping constructs. Recall that any of these constructs may also be used interactively.

A Simple Script

- **A script is a collection of Unix commands**

```
$ cat first
#!/bin/sh
ls
pwd
date

$ first
Access    sh10    sh16    sh21    sh7
Afile     sh11    sh17    sh22    sh8
Last      sh12    sh18
/home/george/shellprog
Thu Oct 07 23:20:58 BST 1994
```

- **Anything which may be typed into the shell can be put into a script**
  - and vice versa

The above simple shell script simply contains a collection of Unix commands; `ls`, `pwd` and `date`. When the script is executed the commands are executed in sequence just as if they had been typed at the command line.

The first line in the script is a comment because it begins with #. However, the sequence #! is special in that it denotes the program which should be executed to interpret the following script. Since this is a shell script, it must be interpreted by the shell program. The binary for the bourne shell is stored in `/bin/sh`.

Note that #! is interpreted by the Operating System kernel, not by the shell; it is only special when used in the first line of a file.

Executing Scripts

- **Ensure the interactive shell can find the script**
  - `PATH=/u/evad/bin ; export PATH`
  - `./script`
- **Give the file execute permission**
  - `chmod +x script`
- **Debugging options**
  - `sh -vx script`

Once created a shell script appears just like any other command. When the name is typed into the interactive shell (which may be any shell; i.e., bourne, Korn or csh), the shell (after it looks internally for the command) searches the directories in the PATH in order to find the program file. Its not just looking for the file name, however, it is actually looking for a matching file with the execute permission bit enabled. Once found, the program is forked by the interactive shell.

In order to make a shell script accessible to the interactive shell, it must be placed in a directory included in the PATH (or modify the PATH to include its directory) and have its execute permission set.

The PATH is not searched if a pathname is supplied when entering the program name. In this case, the pathname (relative or absolute) is followed directly. It is still necessary for the execute bit to be set, however.

The last way to execute a shell script is to invoke the interpreting shell directly, and pass the name of the script as an argument. This is particularly useful if extra options are to be supplied to the interpreting shell. In the example above, `-v` (verbose) and `-x` (expand) are debug options which ask the shell to print each statement before executing it, and to expand expressions.

Command Line Arguments

- **Command line arguments are $0 to $9**

```
$ cat go
#!/bin/sh
echo $1 $2 $3 $4
echo $4 $3 $2 $1

$ go one two three four five
one two three four
four three two one

$ go one two
one two
two one
```

```
$ cat args
#!/bin/sh
echo $0
echo $#
echo $*

$ args one two three
args
3
one two three
```

$0	Command name
$#	Number of arguments
$*, $@	All of the arguments

Command line arguments from are supplied to a script through a series of pre-defined variables. $0 is the script name; $1 through $9 hold the first nine arguments; $# indicates the number of arguments (excluding the command name), and $* (and $@) contain all arguments.

It is useful to pass command line arguments into scripts, just as it is useful to pass options and parameters into any Unix command.

Shifting Arguments

- `shift` moves arguments down to access others

```
$ cat next
#!/bin/sh
echo $1
shift
echo $1
shift
echo $1

$ next red green 1
red
green
1
```

$0 is left untouched

$1 is lost

$2 moves to $1, $3 to $2, ...

$9 gets 10th argument

To access command line arguments beyond the ninth, the existing set of argument stored in $1 through $9 need to be shifted. The `shift` command moves $2 to $1, discarding what was previously in $1. All the other arguments are also shifted, allowing $10 (which cannot otherwise be accessed) to be shifted into $9.

The limitations imposed by having only $1 through $9 are usually avoided by using $* (or $@) to invoke sub-commands, or by using the `for` command (described later) to process all arguments in turn.

All Positional Parameters

"$*" One argument of all parameters from $1

"$@" All arguments from $1

```
$ cat one
#!/bin/ksh
echo $#
echo "$*"
echo "$@"
./two "$*"
./two $*
./two "$@"
./two $@

$ cat two
#!/bin/ksh
echo $#
```

```
$ ./one "hello out" there
2
hello out there
hello out there
1
3
2
3
```

Both $* and $@ evaluate to all of the positional parameters. However, when quoted, their behaviour is quite different. "$*" evaluates to one value consisting on all of the positional parameters starting from $1. "$@" evaluates to all of the positional parameters, starting from $1. That is, "$@" is "$1" "$2" "$3" "$4", not just one argument. The "" around $@ don't join all the arguments together, they simply ensure that any spaces in the supplied arguments are maintained.

The above example shows the significance of the distinction between these two parameters. When script one calls script two with its arguments, the correct number of arguments is only passed when "$@" is used.

Interactive Input

- `read` reads from standard input

```
$ cat hi
#!/bin/sh
echo "hello"
echo -n "What's your name & address? "
read name address
echo goodbye $name of $address

$ hi
hello
What's your name & address? George Scotland
goodbye George of Scotland
```

- words are passed into each variable supplied
  - remaining words on input line are placed in last variable

Not all inputs to the script need to be passed as command line arguments. Indeed, a script can prompt for further input using the echo command, and then read in the input using the read command.

The `read` command looks at the input as consisting of a series of strings (tokens) separated by white space (spaces and tabs), and terminated by a new line. As each token is read, it is placed into the next variable in the list supplied to read. In the above example, 'dave' is placed into the variable `name`, and 'morgan' is placed into the variable `address`. Had only one token been supplied, then `address` would have been left undefined. Had more than two tokens been supplied, then `address` would contain the second and all remaining tokens.

Interactive Copy Command

Using interactive input to prompt for arguments

```
$ cat copy
#!/bin/sh
echo "Enter source file: "
read source
echo "Enter target file: "
read target
echo "Copying from $source to $target"
cp $source $target

$ copy
Enter source file: message
Enter target file: backup
Copying from message to backup
```

Note, no argument checking has been done

In this example a simple interactive copy program is presented. The user is first prompted for the source and target file names. Then the `cp` command is used to copy the file from `$source` to `$target`.

No argument checking has been performed in this script. If any of the filenames are wrong, then the script is relying on `cp` to raise an error. This is not a good programming style.

Conditional Expressions

Basic if statement

```
if <command>
then
   statements
fi
```

Return code of command determines condition

```
if mkdir fred
then
    cd fred
fi
```

The return code of the previous command is stored in `$?`

```
echo $?
```

The `if` construct allows for conditional execution of command. The condition on the if construct is quite different from traditionally languages, in that it is a Unix command (or sequence of commands). Basically, the condition command is executed and based on its internal exit status, the `if` construct will determine whether or not to execute the statements.

Since the condition is itself a statement, absolutely anything can be 'evaluated' to determine whether the statements in the body of the if are executed. In the above example, the `mkdir` command is executed. If it exits true (exit code 0), then the script performs the `cd`, otherwise it does not.

Regardless of the `mkdir` return code, program control flow continues at the first statement after the `if` construct.

Conditional Expressions

```
if <command>
then
    statements
else
    statements
fi
```

`then` statements are executed if command succeeds (returns zero)

`else` statements are executed if command fails (returns non zero)

Return code of command determines condition

```
if <command>
then
    if <command>
    then
        statements
    else
        statements
    fi
fi
```

```
if <command>
then
    statements
elif <command>
then
    statements
else
    statements
fi
```

There are several varieties of the `if` construct in shell scripting. First an `else` clause may be specified to list those statements to be executed in the event that the condition is false. Next, an `elif` construct is available to provide multi-if constructs. In this scenario, control is passed to the first matching `elif` condition, and then resumes after the entire `if` construct.

The test Command

- `test` evaluates general conditions

```
test -f <filename>
test -d <filename>
test -r <filename>
test -w <filename>
test -x <filename>
test -s <filename>
```

`test` is used to evaluate file conditions, and numeric and string relationships

It returns 0 if the test succeeds, non zero otherwise

- Numeric and string tests

```
test n1 -eq n2
test n1 -ne n2
test n1 -ge n2
test n1 -le n2
test n1 -lt n2
test n1 -gt n2
```

```
test s1 = s2
test s1 != s2
test s1 < s2
test s1 > s2
```

To allow general conditional expressions, such as numeric, string and file tests, a command called test is available. This is just another Unix command. However, it doesn't do anything except evaluate the condition (as specified in its command line arguments) and exit with a return code indicating whether the condition holds true.

In addition to the tests above, the condition may be negated, ANDed or ORed with other conditions (`!`, `-a`, `-o`).

Note! Never call a program `test` otherwise it may be used in place of the real `test` command. (Or conversely, the shell built-in command may be executed instead of your `test` program.)

Using the if Statement

```
$ cat check
#!/bin/sh
if test -f .profile
then
  echo ".profile found"
else
  echo ".profile not found"
fi

$ check
.profile not found
```

```
$ cat check2
#!/bin/sh
if test -f .profile
then
  echo ".profile found"
elif test -f .cshrc
then
  echo ".cshrc found"
else
  echo "nothing found"
fi

$ check2
nothing found
```

In the first example above, `test` is used to determine whether the file `.profile` exists in the current directory. If it does, then `test` exits with a return code of zero. otherwise non-zero. Based on this return code, the `if` construct decides which path to follow; causing one of the two `echo` statements to be executed.

Another name for test

- `[` is another name for test, looks more natural
- `]` requires a closing bracket
- Take care to separate arguments with spaces

```
$ cat hay
#!/bin/sh
if [ `who | wc -l` -eq 1 ]
then
        echo "You're on your own"
else
        echo "You're not alone"
fi

$ hay
You're not alone
```

- Notice use of back-quote to determine how many users are logged on

To make the `test` command look more like a traditional `if` conditional expression, the command has an alternate name. When called `[` the `test` command must be terminated with a closing `]` argument. Whilst this may be more syntactically pleasing, note that spaces must be provided after the `[`, between each part of the expression, and before `]`. This is because `[` is a command, and the other bits are arguments.

In the above example, the back quotes are used to cause the `who | wc` to be executed and replaced by their output to STDOUT. This forms the first argument to `[` and is followed by the arguments `-eq`, `1` and `]`.

An Intelligent Interactive Copy

- **Copy script now checks supplied files**

```
$ cat copy2
#!/bin/sh
echo -n "Enter source file: "
read source
echo -n "Enter target file: "
read target

if [ ! -f $source ]
then
  echo "$0: $source: no such file"
elif [ ! -r $source ]
then
  echo "$0: $source: permission denied"
elif [ -f $target ]
then
  echo "$0: $target: file already exists"
else
  echo "Copying $source to $target"
  cp $source $target
fi
```

A more intelligent copy script is presented above. Prior to copying the specified file, this verifies that the source file exists and is readable, and that the target file does not exist (though it may be an existing directory).

An Intelligent Interactive Copy

```
$ copy2
Enter source file: crud
Enter target file: new
copy2: crud: no such file

$ copy2
Enter source file: access
Enter target file: any
copy2: access: permission denied

$ ls -l access
----------  1 dave          0 Oct 20 22:29 access

$ copy2
Enter source file: afile
Enter target file: afile
copy2: afile: file already exists

$ copy2
Enter source file: afile
Enter target file: newone
Copying afile to newone
```

Case Statement

- **Provides a multi-way if statement (like switch in C)**

```
case <value> in
    pattern1)     commands
                   ;;
    pattern2)      commands
                   ;;
    .
    .
    .
esac
```

- **The patterns are shell wild cards**
  - '*' matches everything

A more efficient method (and somewhat more readable method) of forming multiple if-else constructs is to use the `case` construct. This is a multi-way if statement and is similar to the `switch` statement in C.

First <value> is evaluated and then compared with each of the patterns in turn. When a matching pattern is found, the commands following the pattern (often placed on the right) are executed. When a `;;` is found, control is passed to the first statement after the case construct — after the word `esac`. If the `;;` are omitted, then execution drops through and continues with commands associated with the other patterns. For example, once the commands associated with pattern1 are executed, then (given that the `;;` are removed) those tied to pattern2 will be executed.

The patterns may be written using shell meta-characters. Therefore, ranges and wildcards are valid, i.e., [a-z], ???, *.

Case Statement

Converting the week day number to a string

```
$ cat today
#!/bin/sh
case `date +%w` in
        1)      echo its Monday ;;
        2)      echo its Tuesday ;;
        3)      echo its Wednesday ;;
        4)      echo its Thursday ;;
        5)      echo its Friday ;;
        6)      echo its Saturday ;;
        0)      echo its time to stay in bed ;;
        *)      echo weird, day eight? ;;
esac

$ today
its Thursday
```

The result of executing the `date` command is compared against each of the patterns. The terminating `;;` ensure that only the commands tied to each pattern are executed.

for Loop

- **Iterates through a list of values**

```
$ cat colours
#!/bin/sh
for i in red green yellow blue
do
  echo $i
done

$ colours
red
green
yellow
blue
```

- **The loop terminates when the list is empty**
- **Back-quote ` allows any command to generate the list**

The shell provides a number of looping constructs, causing a program to iterate while a condition is true, or (in the case of the `for` loop) over a list of values.

In the `for` loop, the loop control variable (`i` above) iterates over a series of values (red, green, yellow and blue in the above example). With each iteration, the loop variable is assigned the next value in the list, and the statements in the body of the loop are executed. When the value list is exhausted, the loop terminates. Control is passed to the first statement after the end of the loop.

for Loop

- **Using shell wild cards to generate the for list**

```
$ cat showfiles
#!/bin/sh
for i in *
do
    echo "-->> $i <<--"
done
```

```
$ showfiles
-->> access <<--
-->> adir <<--
-->> afile <<--
-->> last <<--
-->> message <<--
-->> new <<--
-->> newname <<--
-->> newone <<--
-->> sh1 <<--
-->> sh10 <<--
-->> sh11 <<--
-->> sh12 <<--
```

The `for` loop is particularly powerful because the value list over which it iterates may be generated in numerous ways. In the above example, a shell-wild card is used to match all files in the current directory; often the back-quote is used to grab the output of an arbitrary command so that this may be iterated over.

for loop

Using shell wild cards to generate the for list

```
$ cat mmv
#!/bin/sh

echo "Enter old extension: "
read old
echo "Enter new extension: "
read new

for i in *.$old
do
  j=`echo $i|sed s/$old\$/$new/`
  echo "Renaming $i to $j..."
  mv $i $j
done
```

```
$ mmv
Enter old extension: eps
Enter new extension: ps
Renaming one.eps to one.ps...
Renaming two.eps to two.ps...
Renaming x.y.eps to x.y.ps...
```

This program (`mmv` for multiple move) renames all files matching one extension to a new extension. It also shows a more interesting way of combining shell wildcards with shell variables in order to build the `for` loop.

for Loop

- **Generating the for list from other commands**

```
$ cat mailall
#!/bin/sh
for i in `who | cut -d" " -f1 | sort |
uniq`
do
  mail $i < message
done
```

- **HERE documents are often used to form the input text rather than using a separate file**

Here documents allow input data to be embedded within the script requiring the input data. In the example below, the `mail` program is executed for each of the users logged in.

Since `mail` reads from the standard input executing it would require a message to be interactively typed at the keyboard, (or in the above case, redirected from another file). Using the 'here' document, the text can be provided directly from the script file.

Here documents begin with '<<' followed by a terminator. The next block of text supplied is taken as literal characters for the standard input, until the terminator is found on a line on its own. In the example below, the terminator chosen was EOF. Once this is found, the input for the `mail` program is complete, and further text is considered as part of the enclosing shell script.

```
#!/bin/sh
for i in `who | cut -d" " -f1 | sort | uniq`
do
       mail $i << EOF
Yo dudes,
it must be time for a beer
have a better one
dave
EOF
done
```

The while Loop

- **`while` loops iterate while a condition remains true**

```
$ cat doit
#!/bin/sh
while [ "$1" != "end" ]
do
        echo $1
        shift
done

$ doit one two three end
one
two
three
```

- **Double-quotes around $1 safeguard `test` syntax against no arguments being supplied**

`while` loops iterate while a condition remains true. `while` loops are entry condition loops, meaning that the conditional expression is evaluated before the first and each iteration. Like the `if` construct, the `while` loop uses the return code of Unix commands as its condition. It is therefore often used in conjunction with the `test` command.

The quotes around `$1` ensure that the syntax for the `test` command is not invalidated. If no arguments are supplied with this script, the `$1` evaluates to nothing, and `test` will complain about the missing argument. When placed in quotes, `$1` will at least evaluate to an empty string — this is an argument and therefore the syntax is valid.

Numerical Expressions

- **`expr` may be used to evaluate expressions**
  - shell does not understand numbers
  - each argument must be separated by spaces

```
$ cat numbers
#!/bin/sh
echo -n "enter two numbers: "
read one two

echo add: `expr $one + $two`
echo sub: `expr $one - $two`
echo mul: `expr $one \* $two`
echo div: `expr $one / $two`

$ numbers
enter two numbers: 6 2
add: 8
sub: 4
mul: 12
div: 3
```

Since the bourne shell does not understand numbers (there are just sequences of characters), another Unix command must be used for numerical expressions.

expr reads its string arguments, interprets them as numbers in numeric expressions, evaluates the expression, and returns the result as a string. * is escaped with the \ in the above example, because of the possible conflict with the shell wild-card.

while Loop

- **Iterating while expression is true**

```
$ cat counter
#!/bin/sh
count=0
while [ $count -lt 10 ]
do
        echo $count
        count=`expr $count + 1`
done
```

```
$ counter
0
1
2
3
4
5
6
7
8
9
```

- **Ensure correct spacing is used for `expr` and `test` commands**

In this example, expr is used to adjust the loop control variable count until it has a string value equivalent to the number 10.

Looping Forever

- `break` jumps out of loop, `continue` jumps to start of next iteration

```
$ cat forever
#!/bin/sh
while true
do
  echo -n "Yes? "
  read input
  case $input in
    quit)  break
           ;;
    loop)  continue
           ;;
    *) echo "sure dude"
           ;;
  esac
  echo "round we go"
done
echo "bye bye"
```

```
$ forever
Yes? ahhhhh
sure dude
round we go
Yes? quit
bye bye

$ forever
Yes? loop
Yes? ok
sure dude
round we go
Yes? quit
bye bye
```

`break` and `continue` cause premature termination or continuation of a loop. These constructs are useful but can sometimes obscure the control flow within a script.

Exercise

- **Refer to the "Basic Shell Programming" exercise in the Exercises booklet.**

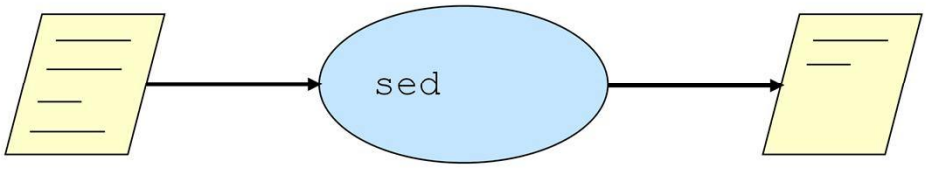

9. INTRODUCING SED AND AWK:
UNIX'S POWER TOOLS

sed and awk – Unix Power Tools

- sed – the stream based editor
- How sed works
- sed command addressing and style
- Adding, deleting, substituting and saving text
- sed scripts
- The awk family
- Patterns and actions in awk
- Working with fields
- BEGIN and END actions
- awk scripts

What Is sed?

- A non-interactive, stream-oriented editor
- Input flows through the program and is directed to standard output



- sed works on lines of text
- sed is a filter for text files

sed is a non-interactive, stream-oriented editor.

It is stream-oriented because, like many UNIX programs, input flows through the program and is directed to standard output. Input typically comes from a file but can be directed from the keyboard. Output goes to the terminal screen by default but can be captured in a file instead. sed works by interpreting a script specifying the actions to be performed.

sed offers capabilities that seem a natural extension of interactive text editing. For instance, it offers a search-and-replace facility that can be applied globally to a single file or a group of files. While you would not typically use sed to change a term that appears once in a particular file, you will find it very useful to make a series of changes across a number of files.

Using sed is similar to writing simple shell scripts. You specify a series of actions to be performed in sequence. Most of these actions could be done manually from within vi, such as replacing text, deleting lines, inserting new text, etc. The advantage of sed is that you can specify all editing instructions in one place and then execute them on a single pass through the file. You don't have to go into each file to make each change. sed can also be used effectively to edit very large files that would be slow to edit interactively.

How Does sed work?

- **sed often appears to be very complex, but its basic operation is quite simple**
- **The three basic principles of how sed works:**

1. All editing commands in a script are applied in order to each line of input
2. Commands are applied to all lines (globally) unless line addressing restricts the lines affected by editing commands
3. The original input file is unchanged. The editing commands modify a copy of original input line and the copy is sent to standard output

To use sed, you write a script that contains a series of editing actions and then you run the script on an input file. sed allows you to take what would be a hands-on procedure in an editor such as vi and transform it into a procedure that is executed from a script.

When performing edits manually, you come to trust the cause-and-effect relationship of entering an editing command and seeing the immediate result. There is usually an undo command that allows you to reverse the effect of a command and return the text file to its previous state. Once you learn an interactive text editor, you experience the feeling of making changes in a safe and controlled manner, one step at a time.

Most people new to sed will feel there is greater risk in writing a script to perform a series of edits than in making those changes manually. The fear is that by automating the task, something will happen that cannot be reversed. The object of learning sed is to understand it well enough to see that your results are predictable. In other words, you come to understand the cause-and-effect relationship between your editing script and the output that you get.

Addressing In sed

- **A sed command can specify zero, one, or two addresses**
  - Like ed
- **Addresses can be either:**
  - Line numbers (1, 2 ... $)
  - Regular expressions (/^this$/)
- **Examples of addressing in sed:**

p	Print all lines
1d	Delete the first line
/^$/d	Delete all empty lines
$d	Delete the last line
25,$d	Delete line 25 to the end of the file
1,/^$/d	Delete from the first line to the first empty line
1,3!d	Delete all lines except the first 3

An address can be a regular expression describing a pattern, a line number, or a line addressing symbol.

If no address is specified, then the command is applied to each line.

If there is only one address, the command is applied to any line matching the address.

If two comma-separated addresses are specified, the command is performed on the first line matching the first address and all succeeding lines up to and including a line matching the second address.

If an address is followed by an exclamation mark (!), the command is applied to all lines that do not match the address.

Using sed

Commands use the following structure:

```
sed [-n] '[addr[,addr]]<function>parameters' [file[s]]
```

1	/s/RE/RS/flags	Substitute
5,10	d	Delete
/this/ p		Print
/here/,/there/	r *filename*	Read file
	w filename	Write file

Command line options are also available:

-f	Allows a sed script file to be specified
-e	Precedes each edit when multiple edits are defined
-n	Suppress the default output

sed commands may be types directly from the keyboard or stored in a script file.

The edits are described in an expression, rather than being interactively supplied from an environment. Consequently, the same edit expression is applied to each line of input.

The other interesting thing to notice with sed is the relationship of the delete and print commands.

Since sed does not write changes back to the source file, the commands simply toggle as to whether or not a line should be printed.

By default, sed prints all lines. Therefore, lines have to be deleted to stop them being printed.

When the –n option is used, sed does not print any lines by default. In this case, the only lines printed are those explicitly printed with p.

Depending on the complexity of the expression, it is sometimes easier to delete lines which are not of interest, than it is to print those which are. Conversely, it is sometimes easier to print lines that to delete them.

Text Substitution

```
$ cat users
ewan    Ewan Smith      p1
mike    Mike Carew      p2
mark    M Walsh         p4
dude    D Mallon        p6
gb      George Ball     p8

$ sed 's/ p/Port-/' users
ewan    Ewan Smith      Port-1
mike    Mike Carew      Port-2
mark    M Walsh         Port-4
dude    D Mallon        Port-6
gb      George Ball     Port-8
```

```
$ sed '/M/d' users
ewan    Ewan Smith      p1
gb      George Ball     p8

$ sed -n '/M/p' users
mike    Mike Carew      p2
mark    M Walsh         p4
dude    D Mallon        p6

$ sed -n '4s/D/Dave/p' users
dude    Dave Mallon        p6
$ sed -n 's/  */:/gp' users
ewan:Ewan:Smith:p1
mike:Mike:Carew:p2
mark:M:Walsh:p4
dude:D:Mallon:p6
gb:George:Ball:p8
```

In the first sed program, on the left hand side, no address is supplied so the command applies to all lines in the file. The command substitutes a space followed by the letter p with Port-. Note that this implicitly selects only those lines with a space followed by a p. Also note that the space used to identify the p near to the end of the line and avoid any other lower case p's that may exist in the input file.

The next sed program deletes all lines which contain the regular expression M. By default, sed prints all of the other lines.

The opposite is achieved in the next example. In this, only lines which contain M are displayed. This is because the -n option suppresses printing by default, and the p command causes lines matching the regular expression to be printed.

In the next example, lines are not printed by default, the edit is applied to line 4, and due to the p modifier on the substitute command, line 4 is printed after the substitution.

In the last example, we wish to perform the same substitution many times on the same line, in this case, substitute 1 or more spaces for the : character, and so use the g option at the end of the command which searches globally along the line.

Text Deletion

```
$ cat users
ewan    Ewan Smith      p1
mike    Mike Carew      p2
mark    M Walsh         p4
dude    D Mallon        p6
gb      George Ball     p8
```

```
$ sed 4d users
ewan    Ewan Smith      p1
mike    Mike Carew      p2
mark    M Walsh         p4
gb      George Ball     p8


$ sed '/^m/d' users
ewan    Ewan Smith      p1
dude    D Mallon        p6
gb      George Ball     p8
```

In the first program line 4 is deleted. All other lines are displayed by default.

In the next example, all lines beginning with `m` are deleted, leaving the lines not matching this regular expression to be printed.

Printing Text

```
$ cat users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
gb       George Ball      p8
```

```
$ sed -n '4,5p' users
dude     D Mallon         p6
gb       George Ball      p8

$ sed -n '/^m/,/^m/s/p.//p' users
mike     Mike Carew
mark     M Walsh
```

Due to the -n option lines are no longer printed by default. However, lines 4 to 5 are printed explicitly.

The last example seems more complex than the others, but simply consists of an address and a command. The address is a range between two regular expressions, the command is a substitution of p followed by any character, for nothing.

Due to the -n option, the p qualifier (at the end of the line) has been added to the substitute command to ensure that lines involved in the edit are printed.

The sed program reads: from the first line that begins with an m until another line begins with an m (inclusive), substitute p followed by a character for nothing. Do not print lines by default, only print lines involved in the substitution.

Writing Files

```
$ cat users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
gb       George Ball      p8
```

```
$ sed -n '1,3w users2' users

$ cat users2
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
```

Do not print lines by default.

Lines in the range 1 through 3 are written to the file `users2`.

Reading Files

```
$ cat users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
gb       George Ball      p8
```

```
$ cat text
 ####    Beer time     ####

$ sed '/p4/r text' users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
 ####    Beer time     ####
dude     D Mallon         p6
gb       George Ball      p8
```

Search for all of the lines containing the regular expression `p4`, and read (after the line) the contents of the file `text`.

Using Multiple sed Commands

```
$ cat users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
gb       George Ball      p8
```

```
$ sed -n -e '/M/p' -e '/ewan/p' users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
```

Multiple commands can be applied consecutively to a line by sed. Each of the different commands, including the first one, must have the –e option placed in front of them.

In this example, we will print out a line that contains a capital M, and print out a line if it contains ewan.

NOTE: If a line matched both of these searches, it would have been printed out twice. Beware of this when performing any substitutions.

Using sed Scripts

- **Create a script containing sed commands**

```
$ cat users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
gb       George Ball      p8

$ cat commands
/M/p
/ewan/p
```

- **Then execute the commands**

```
$ sed -n -f commands users
ewan     Ewan Smith       p1
mike     Mike Carew       p2
mark     M Walsh          p4
dude     D Mallon         p6
```

You can put multiple commands into an external scrip file, that can then be passed to sed.

Using the –f option specifies the name of the script file.

When creating the script file, you do not need to put all of the command within single quotes.

By virtue of it being in an external file means that everything is already hidden from the shell.

awk, nawk & gawk

- **awk is a high-level file manipulation language**
- **Based on pattern matching and processing**
- **awk allows:**
  - Files to be accessed as records and fields
  - Automatic, typed, variables to hold data
  - Arithmetic and string operations
  - Looping and conditional statements
  - Report generation
- **awk's family includes**
  - nawk (new awk standard in SVR4)
  - gawk (gnu awk from the FSF)
- **Named after Aho, Weinberger & Kernighan**

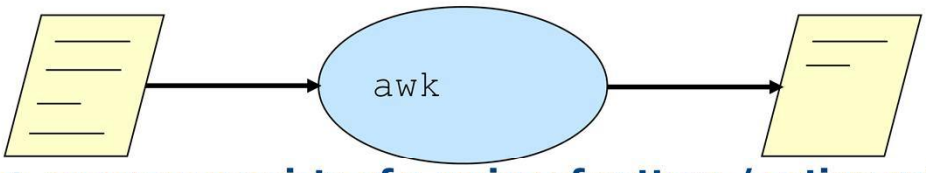
awk is a powerful scripting language which can perform many of the tasks provided by the other utility programs.

However, awk is an interpreted language and therefore the payback is performance.

There are many versions of awk, reflecting developments over many years. However the basic capabilities of all of them are the same (at least for the purposes of this discussion).

File Processing With awk

- awk programs can be used as filters



- An awk program consists of a series of pattern / action pairs
  - The patterns match (address) lines
  - The actions indicate what should be done:

```
/ pattern / { action }
/ pattern / { action }
/ pattern / { action }
```

awk reads each line of the input (a record) and divides it into parts (the fields). Each line is then compared against the patterns.

Wherever the pattern matches, the corresponding action is performed.

Lines may match several patterns, in which each of the corresponding actions will be executed (in order) for the line.

If no pattern is specified then the action is applied to all lines. If no action is specified, then the default action is to print the line.

Two special patterns may exist in the program.

The pattern BEGIN identifies an action to be performed before processing starts.

The pattern END indicates an action to be performed before processing ends.

BEGIN allows initialisation steps to be performed such as setting values in variables. END allows termination actions to be performed such as printing a summary of the processed data.

Using A Pattern

- **Using a simple pattern `awk` can behave like `grep`**

```
$ cat input
ewan     x   100    1    /home/ewan    /bin/ksh
dave     x   101    100  /home/dave    /bin/ksh
mark     x   102    100  /home/mark    /bin/sh
mike     x   103    101  /home/mike    /bin/sh
george   x   104    101  /home/george  /bin/sh
graham   x   105    100  /home/graham  /bin/ksh
john     x   106    102  /home/john    /bin/sh
paul     x   107    100  /home/paul    /bin/ksh

$ nawk '/ksh$/' input
ewan     x   100    1    /home/ewan    /bin/ksh
dave     x   101    100  /home/dave    /bin/ksh
graham   x   105    100  /home/graham  /bin/ksh
paul     x   107    100  /home/paul    /bin/ksh
```

- **NOTE: The default action is to print**

An awk script often contains regular expressions and other characters which should be hidden from the shell using single quotes.

In the above, only a pattern is provided. All lines ending with ksh (users of the Korn shell) are printed.

Addressing Fields

- **Individual fields can be addressed**
  - `$1`, `$2`, `$3`, ... identify fields one, two and three

```
$ nawk '{print $5}' input
/home/ewan
/home/dave
/home/mark
/home/mike
/home/george
/home/graham
/home/john
/home/paul

$ nawk '/ksh/ {print $3,$5,$1}' input
100 /home/ewan ewan
101 /home/dave dave
105 /home/graham graham
107 /home/paul paul
```

```
$ cat input
ewan    x   100   1    /home/ewan    /bin/ksh
dave    x   101   100  /home/dave    /bin/ksh
mark    x   102   100  /home/mark    /bin/sh
mike    x   103   101  /home/mike    /bin/sh
george  x   104   101  /home/george  /bin/sh
graham  x   105   100  /home/graham  /bin/ksh
john    x   106   102  /home/john    /bin/sh
paul    x   107   100  /home/paul    /bin/ksh
```

In the first example above only an action is supplied. This is therefore applied to all lines of the input. The action is to print the 5th field.

awk automatically divides input lines into a series of fields. Any of the fields can be accessed, for example, $50 would be the 50th field.

The number of fields stored in a variable called NF (field number).

To access the last field use $NF. Note that $0 refers to the entire line.

In general, $VAR prints the VARth (contents of variable VAR) column of the current input line.

The second example prints the 3rd, 5th and 1st fields of all lines containing the regular expression ksh.

More On Addressing Fields

- **Records and record numbers may be addressed**
  - `$0` the entire record (line)
  - `NR` the record number

```
$ nawk '{print NR " --> " $0}' input
1 --> ewan    x   100   1    /home/ewan    /bin/ksh
2 --> dave    x   101   100  /home/dave    /bin/ksh
3 --> mark    x   102   100  /home/mark    /bin/sh
4 --> mike    x   103   101  /home/mike    /bin/sh
5 --> george  x   104   101  /home/george  /bin/sh
6 --> graham  x   105   100  /home/graham  /bin/ksh
7 --> john    x   106   102  /home/john    /bin/sh
8 --> paul    x   107   100  /home/paul    /bin/ksh
```

The variable NR keeps counting across multiple files.

There is a related variable called FNR which counts line numbers relative to the current file.

Testing Fields

Both patterns & actions can examine fields

```
$ nawk '$5 ~ /home\/[me]/ {print $0}' input
ewan    x   100    1    /home/ewan    /bin/ksh
mark    x   102    100  /home/mark    /bin/sh
mike    x   103    101  /home/mike    /bin/sh

$ nawk '{if ($5 ~ /home\/[me]/) print $0}' input
ewan    x   100    1    /home/ewan    /bin/ksh
mark    x   102    100  /home/mark    /bin/sh
mike    x   103    101  /home/mike    /bin/sh

$ nawk '$5 !~ /home\/[jg]/ {if ($3 > 102) print $1,$3,$6}' input
mike 103 /bin/sh
paul 107 /bin/ksh
```

`$5 ~ /RE/`	matches RE
`$5 !~ /RE/`	does not match RE

Conditional and looping expressions may appear in awk actions.

In the last example above, the `if` statement ensures that the fields are only printed if `$3` is greater than `102`.

A conditional statement allows you to make a test before performing an action.

A conditional statement is introduced by `if` and evaluates an expression placed in parentheses. The syntax is:

```
if ( expression ) action1 [else action2]
```

Multiple Pattern / Action Pairs

- Use ; to separate multiple pattern / action pairs

```
$ nawk '/^m/ {print $1 "\t" $5} ; /^g/ {print $1 "\t" $6}' input
mark    /home/mark
mike    /home/mike
george  /bin/sh
graham  /bin/ksh

$ nawk '/^[mg]/ {print $1 "\t" $5} ; /^g/ {print $1 "\t" $6}' input
mark    /home/mark
mike    /home/mike
george  /home/george
george  /bin/sh
graham  /home/graham
graham  /bin/ksh
```

Multiple pattern / action pairs can be applied to a line.

In the example above, these are mutually exclusive, and so if a line matches the first pattern, the first action is performed, if it matches the second pattern, the second action is performed. If neither pattern matches, nothing is done.

NOTE: As shown in the second example, if more than one pattern matches a line, then all of the actions associated with the matching patterns are performed.

Using Variables

Variables are used to store & accumulate data

- Can be introduced anywhere within action
- No declaration, initialisation or explicit type necessary
- Standard arithmetic and string functions available

```
$ nawk '{tot = tot + $4; print $4 "\t" tot}' input
1       1
100     101
100     201
101     302
101     403
100     503
102     605
100     705
```

awk allows variables to be introduced anywhere.

There is no need to provide a typed declaration for them.

Also note, unlike the shell variables do not need to be prefixed with a $ to be dereferenced.

In fact, $VAR will give you the VARth column of the current input line.

Operators available include:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
^	Exponentiation
**	Exponentiation (not POSIX compliant – beware!!!)

Assignment operators (such as +=, -= etc) are available, as are relational operators (such as <, >, >= etc), boolean operators (&&, || and !) and increment and decrement operators (++ and --)

Using An END Action

- **END action is executed after processing last record**

```
$ nawk '{tot += $4} ; END {print "The total is " tot }' input
The total is 705

$ ls -l
total 34
-rw-r--r--   1 ewan     other          13 Feb  6 07:57 commands
-rw-r--r--   1 ewan     other         372 Feb  6 07:56 input
-rw-r--r--   1 ewan     other       12638 Feb  6 07:57 ls
-rw-r--r--   1 ewan     other         428 Feb  6 07:56 passwd
-rw-r--r--   1 ewan     other         135 Feb  6 07:56 users
$ ls -l | nawk '{tot += $5}; END {print tot}'
13586
```

- **BEGIN may be used for pre-actions**
  - Outputting report headings
  - Setting initial values in variables
  - Setting field separators (FS and OFS)

BEGIN and END are not pairs. They are used as distinct patterns to identify actions which should be performed before processing of the input starts, and after processing finishes.

BEGIN is useful for initialising variables, writing titles and setting up awk control variables. The FS (input field separator) is used to distinguish each field (it may also be given via a -F option); the OFS (output field separator) is written whenever a ',' appears in a print statement.

The END action allows totals and report generation activities to be performed.

Pre-Defined Variables

- **awk provides a number of pre-defined variables**

FILENAME	Name of the input file
FS	Input field separator (default tab, space)
RS	Input record separator (default newline)
OFS	Output field separator (default space)
ORS	Output record separator (default newline)
NF	Number of fields in current record
NR	Current record (line) number

```
$ ls -l | nawk 'BEGIN {OFS="<>"}; ; {print $1,$3,$9}'
total<><>
-rw-r--r--<>ewan<>commands
-rw-r--r--<>ewan<>input
-rw-r--r--<>ewan<>ls
-rw-r--r--<>ewan<>passwd
-rw-r--r--<>ewan<>users
```

There are a number of system or built-in variables defined by awk.

awk has two types of system variables.

The first type defines values whose default can be changed, such as the default field and record separators.

The second type defines values that can be used in reports or processing, such as the number of fields found in the current record, the count of the current record, and others.

These are automatically updated by awk; for example, the current record number and input file name.

There are a set of default values that affect the recognition of records and fields on input and their display on output.

The system variable FS defines the field separator. By default, its value is a single space, which tells awk that any number of spaces and / or tabs separate fields.

FS can also be set to any single character, or to a regular expression.

The output equivalent of FS is OFS, which is a space by default.

Building awk Scripts

- **Complex scripts may be stored in files**

```
$ cat go
        {
                total += $5
        }
END     {
                print "Total number of bytes are " total
        }

$ ls -l | nawk -f go
Total number of bytes are 13658
```

- **NOTE: This structure must be followed**
  - awk is not free form and some layouts are illegal

You must be careful when creating awk scripts, as the structure is not truly free form.

As a rule, when opening new braces, put the code inside on a new line by itself.

Later code examples show how, for example, looping constructs should be written inside your scripts.

Automating awk Scripts

The OS can automatically load awk

- Saves having to remember that 'go' is an awk script
- Unix has many scripting languages
- Separates the function of the script from its implementation language

```
$ cat go
#!/usr/bin/nawk -f
     {
         total += $5
     }
END  {
         print "Total number of bytes are " total
     }

$ chmod +x go

$ ls -l | ./go
Total number of bytes are 13677
```

The OS can be used to automatically load awk when an awk script is executed.

It is unreasonable to expect users to know the language in which every program is written. Therefore, Unix provides a mechanism in which the appropriate interpreting process is executed automatically.

The first line in the above program is a comment as far as awk is concerned. However, it is a directive used in the loading procedure to invoke the program /usr/bin/nawk with the -f option and the name of the script as an argument. I.e.:

```
/bin/nawk -f go
```

Exercise

- **Refer to the "Power Tools – sed and awk" exercise in the Exercises booklet.**

10. ADVANCED SHELL PROGRAMMING

Chapter 10: Advanced Shell Scripting Features

- Parameter Types
- Parameter Substitution
- An Example Of Using Parameter Substitution
- The `set` Command
- The `readonly` command
- The `unset` Command
- The `eval` Command
- The `wait` Command
- The `$!` Variable
- The `trap` Command
- Signals & Their Numbers
- Functions
- The `return` Command

Parameter Types

Parameters include

- The arguments passed to a program (the positional parameters)
- The special shell variables such as `$#` and `$?`
- Ordinary variables, also known as keyword parameters

Positional parameters cannot be assigned values directly

- However, they can be reassigned values with the `set` command
- Keyword parameters are assigned values simply by writing

```
$ variable=value
```

The format is a bit more general than that shown

- Actually, you can assign several keyword parameters at once using the format

```
$ variable=value variable=value ...
```

Parameter Substitution

- **In the simplest form, to have the value of a parameter substituted, you simply precede the parameter with a dollar sign, as in `$i` or `$9`**

- **There are however, some additional ways of performing substitution**
  - It is possible, based upon a given circumstance, to decide whether or not to substitute a parameter for a default value
  - For example:

```
$ echo ${parameter:-value}
```

  - This will echo the value of *parameter* if it's set and non-null; otherwise, it will echo *value*

An Example Of Using Parameter Substitution

- **The following two code fragments perform exactly the same task:**

```
echo Using editor ${EDITOR:-/bin/vi}
```

```
if [ -n "$EDITOR" ]
then
   echo Using editor $EDITOR
else
   echo Using editor /bin/vi
fi
```

The `set` Command

- **The shell's `set` command is a dual-purpose command:**
  - It is used both to set various shell options as well as to reassign the positional parameters `$1`, `$2`, and so forth
- **There is no way to directly assign a value to a positional parameter; for example, this will NOT work:**

```
$ 1=100
```

- **These parameters are initially set on execution of the shell program**
  - The only way they may be changed is with the `shift` or the `set` commands
- **If words are given as arguments to `set` on the command line, those words will be assigned to the positional parameters `$1`, `$2`, and so forth**
  - The previous values stored in the positional parameters will be lost forever

Using The `set` command

- **So:**

```
$ set a b c
```

  - Assigns `a` to `$1`, `b` to `$2`, and `c` to `$3`
  - `$#` also gets set to `3`

- **An Example:**

```
$ set one two three four
$ echo $1:$2:$3:$4
one:two:three:four
$ echo $#           # This should be 4
4
$ echo $*           # What does this reference now?
one two three four
$ for arg; do echo $arg; done
one two three four
$
```

The `readonly` Command

- The `readonly` command is used to specify variables whose values cannot be subsequently changed. For example:

```
$ readonly PATH HOME
```

makes the `PATH` and `HOME` variables read-only

- Subsequently attempting to assign a value to these variables causes the shell to issue an error message:

```
$ PATH=/bin:/usr/bin:.:
$ readonly PATH
$ PATH=$PATH:/users/joe/bin
sh: PATH: is read-only
$
```

- Here you see that after the variable `PATH` was made read-only, the shell printed an error message when an attempt was made to assign a value to it

Using `readonly` Variables

- **To get a list of your read-only variables, type `readonly -p` without any arguments**
  - This will give the name and value associated with any read-only variables
  - In the Bourne shell, just type `readonly` (without any arguments). The Bourne shell will only report variables that are read-only, and not their values
- **You should be aware of the fact that the read-only variable attribute is not passed down to subshells**
- **Also, after a variable has been made read-only in a shell, there is no way to "undo" it**

The `unset` Command

- **Sometimes you may want to remove the definition of a variable from your environment**
- **`unset` removes both exported and local shell variables**
- **To do so, you type unset followed by the names of the variables:**

```
$ x=100
$ echo $x
100
$ unset x          # Remove x from the environment
$ echo $x
$
```

- **You can't unset a read-only variable. Furthermore, the variables `IFS`, `MAILCHECK`, `PATH`, `PS1`, and `PS2` cannot be unset**
  - Also, some older shells do not support the `unset` command
  - Both the Bourne and Korn shells support `unset`

The `eval` Command

- **The format of the `eval` command is as follows:**

```
eval command-line
```

where `command-line` is a normal command line that you would type at the terminal

- **When you put `eval` in front of a normal command, however, the net effect is that the shell scans the command line twice before executing it**
  - For the simple case, this really has no effect:

```
$ eval echo hello
hello
$
```

  - Actually, what happens is that `eval` simply executes the command passed to it as arguments; so the shell processes the command line when passing the arguments to `eval`, and then once again when `eval` executes the command
  - The net result is that the command line is scanned twice by the shell

Problems In Interpreting The Command Line

- **Consider the following example without the use of `eval`:**

```
$ pipe="|"
$ ls $pipe wc -l
|: No such file or directory
wc: No such file or directory
-l: No such file or directory
$
```

- **The errors come from `ls`. The shell takes care of pipes and I/O redirection before variable substitution, so it never recognizes the pipe symbol inside pipe**

- **The result is that the three arguments `|`, `wc`, and `-l` are passed to `ls` as arguments**

Correct Command Line Interpretation With `eval`

- **Putting `eval` in front of the command sequence gives the desired results:**

```
$ eval ls $pipe wc -l
   16
```

- **The first time the shell scans the command line, it substitutes `|` as the value of pipe. Then `eval` causes it to rescan the line, at which point the `|` is recognized by the shell as the pipe symbol**
  - The `eval` command is frequently used in shell programs that build up command lines inside one or more variables
  - If the variables contain any characters that must be seen by the shell directly on the command line (that is, not as the result of substitution), `eval` can be useful
  - Command terminator (`;`, `|`, `&`), I/O redirection (`<`, `>`), and quote characters are among the characters that must appear directly on the command line to have any special meaning to the shell

The `wait` Command

- **If you submit a command line to the background for execution, that command line runs in a `subshell` independent of your current shell (the job is said to run asynchronously)**
- **At times, you may want to wait for the background process (also known as a child process because it's spawned from your current shell—the parent) to finish execution before proceeding.**
  - For example, you may have sent a large `sort` into the background and now want to wait for the sort to finish because you need to use the sorted data.
- **The `wait` command is for such a purpose. Its general format is:**

```
wait process-id
```

where `process-id` is the process id number of the process you want to wait for

- **If omitted, the shell waits for all child processes to complete execution. Execution of your current shell will be suspended until the process or processes finish execution**

Using The `wait` Command

- **You can try the wait command at your terminal:**

```
# Send it to the background
# Job number & process id from the shell
$ sort big-data > sorted_data &
 [1] 3423

# Do some other work
$ date
Wed Oct 2 15:05:42 EDT 2002

# Now wait for the sort to finish
$ wait 3423
$              #When sort finishes, prompt is returned
```

The $! Variable

- **If you have only one process running in the background, then wait with no argument suffices**
- **However, if you're running more than one command in the background and you want to wait on a particular one, you can take advantage of the fact that the shell stores the process id of the last command executed in the background inside the special variable $!**
- **So the command:** `wait $!`

  waits for the last process sent to the background to complete execution
  - As mentioned, if you send several commands to the background, you can save the value of this variable for later use with `wait`

```
$ prog1 &
$ pid1=$!
```

The `trap` Command

- **When you press Ctrl+C at your terminal during execution of a shell program, normally that program is immediately terminated, and your command prompt returned**
  - This may not always be desirable
  - For instance, you may end up leaving a bunch of temporary files that won't get cleaned up
- **The pressing of the `Ctrl-C` key at the terminal sends what's known as a signal to the executing program**
  - The program can specify the action that should be taken on receipt of the signal. This is done with the `trap` command, whose general format is:

```
trap commands signals
```

where `commands` is one or more commands that will be executed whenever any of the signals specified by signals is received.

Signals & Their Numbers

- **Numbers are assigned to the different types of signals, and the more commonly used ones are summarized below**
  - A more complete list is given under the `trap` command in Appendix B

Signal	Generated for
0	Exit from the shell
1	Hangup
2	Interrupt (for example, Ctrl+C key)
15	Software termination signal (sent by kill by default)

An Example Of The `trap` Command

- **The following shows how you can remove some files and then exit if someone tries to abort the program from the terminal:**

```
$ trap 'rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit' 2
```

- **From the point in the shell program that this `trap` is executed**
  - The two files `work1$$` and `dataout$$` will be automatically removed if signal number `2` is received by the program
  - If the user interrupts execution of the program after this trap is executed, you can be assured that these two files will be cleaned up
  - The exit that follows the `rm` is necessary because without it execution would continue in the program at the point that it left off when the signal was received

Functions

- **To define a function, you use the general format:**

```
name () { command; ... command; }
```

- **Where:**
  - `name` is the name of the function
  - The parentheses denote to the shell that a function is being defined,
  - And the commands enclosed between the curly braces define the body of the function

- **These commands will be executed whenever the function is executed**
  - Note that at least one whitespace character must separate the `{` from the first command, and that a semicolon must separate the last command from the closing brace if they occur on the same line

Defining Functions

- The following defines a function called `nu` that displays the number of logged-in users:

```
$ nu () { who | wc -l; }
```

- You execute a function the same way you execute an ordinary command: simply by typing its name to the shell:

```
$ nu
22
$
```

- If you have some functions that you commonly use, you can put them into your `.profile` file

Function Scope

Functions exist only in the shell in which they are defined

- They can't be passed down to subshells
- Because the function is executed in the current shell, changes made to the current directory or to variables remain after the function has completed execution

```
$ db () {
> PATH=$PATH:/uxn2/data
> PS1=DB:
> cd /uxn2/data
> }
$ db                    # Execute it
DB:
```

As you see, a function definition can continue over as many lines as necessary

- The shell displays your secondary command prompt until you close the definition with the }

Removing a Function Definition

- To remove the definition of a function from the shell, you use the unset command with the `-f` option
- This is the same command you use to remove the definition of a variable to the shell

```
$ unset -f nu
$ nu
sh: nu: not found
$
```

The `return` Command

- **If you execute an `exit` command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function**
- **If you instead want to just terminate execution of the function, you can use the `return` command, whose format is:**

```
return n
```

- **The value `n` is used as the return status of the function**
  - If omitted, the status returned is that of the last command executed. This is also what gets returned if you don't execute a return at all in your function
- **The return status is in all other ways equivalent to the exit status:**
  - You can access its value through the shell variable `$?`, and you can also `test` it in `if`, `while`, and `until` commands

Exercise

- **Refer to the "Advanced Shell Programming" exercise in the Exercises booklet.**