# UNIX Administration Course



### Version 1.0


mapleson@gamers.org
Tel: (+44) (0)1772 893297
Fax: (+44) (0)1772 892913
WWW: http://www.futuretech.vuurwerk.nl/


### Detailed Notes for Day 1 (Part 2)

**UNIX Fundamentals: Files and the File System.**

At the lowest level, from a command-line point of view, just about everything in a UNIX environment is treated as a file - even hardware entities, eg. printers, disks and DAT drives. Such items might be described as 'devices' or with other terms, but at the lowest level they are visible to the admin and user as files somewhere in the UNIX file system (under /dev in the case of hardware devices). Though this structure may seem a little odd at first, it means that system commands can use a common processing and communication interface no matter what type of file they're dealing with, eg. text, pipes, data redirection, etc. (these concepts are explained in more detail later).

The UNIX file system can be regarded as a top-down tree of files and directories, starting with the top-most 'root' directory. A directory can be visualised as a filing cabinet, other directories as folders within the cabinet, and individual files as the pieces of paper within folders. It's a useful analogy if one isn't familiar with file system concepts, but somewhat inaccurate since a directory in a computer file system can contain files on their own as well as other directories, ie. most office filing cabinets don't have loose pieces of paper outside of folders.

UNIX file systems can also have 'hidden' files and directories. In DOS, a hidden file is just a file with a special attribute set so that 'dir' and other commands do not show the file; by contrast, a hidden file in UNIX is any file which begins with a dot '.' (period) character, ie. the hidden status is a result of an aspect of the file's name, not an attribute that is bolted onto the file's general existence. Further, whether or not a user can access a hidden file or look inside a hidden directory has nothing to do with the fact that the file or directory is hidden from normal view (a hidden file in DOS cannot be written to). Access permissions are a separate aspect of the fundamental nature of a UNIX file and are dealt with later.

The 'ls' command lists files and directories in the current directory, or some other part of the file system by specifying a 'path' name. For example:

```
ls /
```

will show the contents of the root directory, which may typically contain the following:

```
CDROM    dev        home     mapleson   proc       stand    usr
bin      dumpster   lib      nsmail     root.home  tmp      var
debug    etc        lib32    opt        sbin       unix
```

**Figure 1. A typical root directory shown by 'ls'.**

Almost every UNIX system has its own unique root directory and file system, stored on a disk

within the machine. The exception is a machine with no internal disk, running off a remote server in some way; such systems are described as 'diskless nodes' and are very rare in modern UNIX environments, though still used if a diskless node is an appropriate solution.

Some of the items in Fig 1. are files, while others are directories. If one uses an option '-F' with the ls command, special characters are shown after the names for extra clarity:

```
/   - directory

*   - executable file

@   - link to another file or directory
      elsewhere in the file system
```

Thus, using 'ls -F' gives this more useful output:

```
CDROM/   dev/        home/    mapleson/   proc/       stand/   usr/
bin/     dumpster/   lib/     nsmail/     root.home   tmp/     var/
debug/   etc/        lib32/   opt/        sbin/       unix*
```

**Figure 2. The root directory shown by 'ls -F /'.**

Fig 2 shows that most of the items are in fact other directories. Only two items are ordinary files: 'unix' and 'root.home'. 'unix' is the main UNIX kernel file and is often several megabytes in size for today's modern UNIX systems - this is partly because the kernel must often include support for 64bit as well as older 32bit system components. 'root.home' is merely a file created when the root user accesses the WWW using Netscape, ie. an application-specific file.

Important directories in the root directory:

```
/bin        - many as-standard system commands
              are here (links to /usr/bin)

/dev        - device files for keyboard, disks, printers, etc.

/etc        - system configuration files

/home       - user accounts are here (NFS mounted)

/lib        - library files used by executable programs

/sbin       - user applications and other commands

/tmp        - temporary directory (anyone can create
              files here). This directory is normally
              erased on bootup

/usr        - Various product-specific directories, system
              resource directories, locations of online help
              (/usr/share), header files of application
              development (usr/include), further system
              configuration files relating to low-level
              hardware which are rarely touched even by an
              administrator (eg. /usr/cpu and /usr/gfx).

/var        - X Windows files (/var/X11), system services
              files (eg. software licenses in /var/flexlm),
              various application related files (/var/netscape,
              /var/dmedia), system administration files and data
```

```
                (/var/adm, /var/spool) and a second temporary
                directory (/var/tmp) which is not normally erased
                on bootup (an administrator can alter the
                behaviour of both /tmp and /var/tmp).

/mapleson   - (non-standard) my home account is here, NFS-
              mounted from the admin Indy called Milamber.
```

**Figure 3. Important directories in the root directory.**

Comparisons with other UNIX variants such as HP-UX, SunOS and Solaris can be found in the many FAQ (Frequently Asked Questions) files available via the Internet [1].

Browsing around the UNIX file system can be enlightening but also a little overwhelming at first. However, an admin never has to be concerned with most parts of the file structure; low-level system directories such as /var/cpu are managed automatically by various system tasks and programs. Rarely, if ever, does an admin even have to look in such directories, never mind alter their contents (the latter is probably an unwise thing to do).

From the point of view of a novice admin, the most important directory is /etc. It is this directory which contains the key system configuration files and it is these files which are most often changed when an admin wishes to alter system behaviour or properties. In fact, an admin can get to grips with how a UNIX system works very quickly, simply by learning all about the following files to begin with:

```
/etc/sys_id   - the name of the system (may include full domain)

/etc/hosts    - summary of full host names (standard file,
                added to by the administrator)

/etc/fstab    - list of file systems to mount on bootup

/etc/passwd   - password file, contains user account information

/etc/group    - group file, contains details of all user groups
```

**Figure 4. Key files for the novice administrator.**

Note that an admin also has a personal account, ie. an ordinary user account, which should be used for any task not related to system administration. More precisely, an admin should only be logged in as root when it is strictly necessary, mainly to avoid unintended actions, eg. accidental use of the ’rm’ command.

## A Note on the ’man’ Command.

The manual pages and other online information for the files shown in Fig 4 all list references to other related files, eg. the man page for ’fstab’ lists ’mount’ and ’xfs’ in its ’SEE ALSO’ section, as well as an entry called ’filesystems’ which is a general overview document about UNIX file systems of all types, including those used by CDROMs and floppy disks. Modern UNIX releases contain a large number of useful general reference pages such as ’filesystems’. Since one may not know what is available, the ’k’ and ’f’ options can be used with the man command to offer suggestions, eg. ’man -f file’ gives this output (the -f option shows all man page titles for entries that begin with the word ’file’):

```
ferror, feof, clearerr,
```

```
  fileno (3S)                 stream status inquiries
file (1)                      determine file type
file (3Tcl)                   Manipulate file names and attributes
File::Compare (3)             Compare files or filehandles
File::Copy (3)                Copy files or filehandles
File::DosGlob (3)             DOS like globbing and then some
File::Path (3)                create or remove a series
                              of directories
File::stat (3)                by-name interface to Perl's built-in
                              stat() functions
filebuf (3C++)                buffer for file I/O.
FileCache (3)                 keep more files open than
                              the system permits
fileevent (3Tk)               Execute a script when a file
                              becomes readable or writable
FileHandle (3)                supply object methods for filehandles
filename_to_devname (2)       determine the device name
                              for the device file
filename_to_drivername (2)    determine the device name
                              for the device file
fileparse (3)                 split a pathname into pieces
files (7P)                    local files name service
                              parser library
FilesystemManager (1M)        view and manage filesystems
filesystems: cdfs, dos,
fat, EFS, hfs, mac,
iso9660, cd-rom, kfs,
nfs, XFS, rockridge (4)       IRIX filesystem types
filetype (5)                  K-AShare's filetype
                              specification file
filetype, fileopen,
filealtopen, wstype (1)       determine filetype of specified
                              file or files
routeprint, fileconvert (1)   convert file to printer or
                              to specified filetype
```

**Figure 5. Output from 'man -f file'.**

'man -k file' gives a much longer output since the '-k' option runs a search on every man page title containing the word 'file'. So a point to note: judicious use of the man command along with other online information is an effective way to learn how any UNIX system works and how to make changes to system behaviour. All man pages for commands give examples of their use, a summary of possible options, syntax, further references, a list of any known bugs with appropriate workarounds, etc.

The next most important directory is probably /var since this is where the configuration files for many system services are often housed, such as the Domain Name Service (/var/named) and Network Information Service (/var/yp). However, small networks usually do not need these services which are aimed more at larger networks. They can be useful though, for example in aiding Internet access.

Overall, a typical UNIX file system will have over several thousand files. It is possible for an admin to manage a system without ever knowing what the majority of the system's files are for. In fact, this is a preferable way of managing a system. When a problem arises, it is more important to know *where* to find relevant information on how to solve the problem, rather than try to learn the solution to every possible problem in the first instance (which is impossible).

I once asked an experienced SGI administrator (the first person to ever use the massive Cray T3D supercomputer at the Edinburgh Parallel Computing Centre) what the most important thing in his

daily working life was. He said it was a small yellow note book in which he had written where to find information about various topics. The book was an index on where to find facts, not a collection of facts in itself.

Hidden files were described earlier. The '-a' option can be used with the ls command to show hidden files:

```
ls -a /
```

gives:

```
./                  .sgihelprc          lib/
../                 .ssh/               lib32/
.Acroread.License   .varupdate          mapleson/
.Sgiresources       .weblink            nsmail/
.cshrc              .wshttymode         opt/
.desktop-yoda/      .zmailrc            proc/
.ebtpriv/           CDROM/              sbin/
.expertInsight      bin/                stand/
.insightrc          debug/              swap/
.jotrc*             dev/                tmp/
.login              dumpster/           unix*
.netscape/          etc/                usr/
.profile            floppy/             var/
.rhosts             home/
```

**Figure 6. Hidden files shown with 'ls -a /'.**

For most users, important hidden files would be those which configure their basic working environment when they login:

```
.cshrc
.login
.profile
```

Other hidden files and directories refer to application-specific resources such as Netscape, or GUI-related resources such as the .desktop-sysname directory (where 'sysname' is the name of the host).

Although the behaviour of the ls command can be altered with the 'alias' command so that it shows hidden files by default, the raw behaviour of ls can be accessed by using an absolute directory path to the command:

```
/bin/ls
```

Using the absolute path to any file in this way allows one to ignore any aliases which may have been defined, as well as the normal behaviour of the shell to search the user's defined path for the first instance of a command. This is a useful technique when performing actions as root since it ensures that the wrong command is not executing by mistake.

### Network File System (NFS)

An important feature of UNIX is the ability to access a particular directory on one machine from another machine. This service is called the 'Network File System' (NFS) and the procedure itself is called 'mounting'.

For example, on the machines in Ve24, the directory /home is completely empty - no files are in it whatsoever (except for a README file which is explained below). When one of the Indys is turned on, it 'mounts' the /home directory from the server 'on top' of the /home directory of the local machine. Anyone looking in the /home directory actually sees the contents of /home on the server.

The 'mount' command is used to mount a directory on a file system belonging to a remote host onto some directory on the local host's filesystem. The remote host must 'export' a directory in order for other hosts to locally mount it. The /etc/exports file contains a list of directories to be exported.

For example, the following shows how the /home directory on one of the Ve24 Indys (akira) is mounted off the server, yet appears to an ordinary user to be just another part of akira's overall file system (NB: the '#' indicates these actions are being performed as root; an ordinary user would not be able to use the mount command in this way):

```
AKIRA 1# mount | grep YODA
YODA:/var/www on /var/www type nfs (vers=3,rw,soft,intr,bg,dev=c0001)
YODA:/var/mail on /var/mail type nfs (vers=3,rw,dev=c0002)
YODA:/home on /home type nfs (vers=3,rw,soft,intr,bg,dev=c0003)
AKIRA 1# ls /home
dist/    projects/  pub/     staff/     students/  tmp/    yoda/
AKIRA 2# umount /home
AKIRA 1# mount | grep YODA
YODA:/var/www on /var/www type nfs (vers=3,rw,soft,intr,bg,dev=c0001)
YODA:/var/mail on /var/mail type nfs (vers=3,rw,dev=c0002)
AKIRA 3# ls /home
README
AKIRA 4# mount /home
AKIRA 5# ls /home
dist/    projects/   pub/     staff/      students/   tmp/     yoda/
AKIRA 6# ls /
CDROM/   dev/        home/    mapleson/   proc/       stand/   usr/
bin/     dumpster/   lib/     nsmail/     root.home   tmp/     var/
debug/   etc/        lib32/   opt/        sbin/       unix*
```

**Figure 7. Manipulating an NFS-mounted file system with 'mount'.**

Each Indy has a README file in its local /home, containing:

```
The /home filesystem from Yoda is not mounted for some reason.
Please contact me immediately!

Ian Mapleson, Senior Technician.

3297 (internal)
mapleson@gamers.org
```

After /home is remounted in Fig 7, the ls command no longer shows the README file as being present in /home, ie. when /home is mounted from the server, the local contents of /home are completely hidden and inaccessible.

When accessing files, a user never has to worry about the fact that the files in a directory which has been mounted from a remote system actually reside on a physically separate disk, or even a different UNIX system from a different vendor. Thus, NFS gives a seamless transparent way to merge different files systems from different machines into one larger structure. At the department where I studied years ago [2], their UNIX system included Hewlett Packard machines running HP-UX, Sun machines running SunOS, SGIs running IRIX, DEC machines running Digital UNIX,

PCs running an X-Windows emulator called Windows Exceed, and some Linux PCs. All the machines had access to a single large file structure so that any user could theoretically use any system in any part of the building (except where deliberately prevented from doing so via local system file alterations).

Another example is my home directory /mapleson - this directory is mounted from the admin Indy (Technicians' office Ve48) which has my own extra external disk locally mounted. As far as the server is concerned, my home account just happens to reside in /mapleson instead of /home/staff/mapleson. There is a link to /mapleson from /home/staff/mapleson which allows other staff and students to access my directory without having to ever be aware that my home account files do not physically reside on the server.

Every user has a 'home directory'. This is where all the files owned by that user are stored. By default, a new account would only include basic files such as .login, .cshrc and .profile. Admin customisation might add a trash 'dumpster' directory, user's WWW site directory for public access, email directory, perhaps an introductory README file, a default GUI layout, etc.

**UNIX Fundamentals: Processes and process IDs.**

As explained in the UNIX history, a UNIX OS can run many programs, or processes, at the same time. From the moment a UNIX system is turned on, this process is initiated. By the time a system is fully booted so that users can login and use the system, many processes will be running at once. Each process has its own unique identification number, or process ID. An administrator can use these ID numbers to control which processes are running in a very direct manner.

For example, if a user has run a program in the background and forgotten to close it down before logging off (perhaps the user's process is using up too much CPU time) then the admin can shutdown the process using the kill command. Ordinary users can also use the kill command, but only on processes they own.

Similarly, if a user's display appears frozen due to a problem with some application (eg. Netscape) then the user can logon to a different system, login to the original system using rlogin, and then use the kill command to shutdown the process at fault either by using the specific process ID concerned, or by using a general command such as killall, eg.:

```
killall netscape
```

This will shutdown all currently running Netscape processes, so using specific ID numbers is often attempted first.

Most users only encounter the specifics of processes and how they work when they enter the world of application development, especially the lower-level aspects of inter-process communication (pipes and sockets). Users may often run programs containing bugs, perhaps leaving processes which won't close on their own. Thus, kill can be used to terminate such unwanted processes.

The way in which UNIX manages processes and the resources they use is extremely tight, ie. it is very rare for a UNIX system to completely fall over just because one particular process has caused an error. 3rd-party applications like Netscape are usually the most common causes of process errors. Most UNIX vendors vigorously test their own system software to ensure they are, as far as can be ascertained, error-free. One reason why alot of work goes into ensuring programs are bug free is that bugs in software are a common means by which hackers try to gain root (admin) access

to a system: by forcing a particular error condition, a hacker may be able to exploit a bug in an application.

For an administrator, most daily work concerning processes is about ensuring that system resources are not being overloaded for some reason, eg. a user running a program which is forking itself repeatedly, slowing down a system to a crawl.

In the case of the SGI system I run, staff have access to the SGI server, so I must ensure that staff do not carelessly run processes which hog CPU time. Various means are available by which an administrator can restrict the degree to which any particular process can utilise system resources, the most important being a process priority level (see the man pages for 'nice' and 'renice').

The most common process-related command used by admins and users is 'ps', which displays the current list of processes. Various options are available to determine which processes are displayed and in what output format, but perhaps the most commonly used form is this:

```
ps -ef
```

which shows just about everything about every process, though other commands exist which can give more detail, eg. the current CPU usage for each process (osview). Note that other UNIX OSs (eg. SunOS) require slightly different options, eg. 'ps -aux' - this is an example of the kind of difference which users might notice between System V and BSD derived UNIX variants.

**The Pipe.**

An important aspect of processes is inter-process communication. From an every day point of view, this involves the concept of pipes. A pipe, as the name suggests, acts as a communication link between two processes, allowing the output of one processes to be used as the input for another. The pipe symbol is a vertical bar '|'.

One can use the pipe to chain multiple commands together, eg.:

```
cat *.txt | grep pattern | sort | lp
```

The above command sequence dumps the contents of all the files in the current directory ending in .txt, but instead of the output being sent to the 'standard output' (ie. the screen), it is instead used as the input for the grep operation which scans each incoming line for any occurence of the word 'pattern' (grep's output will only be those lines which do contain that word, if any). The output from grep is then sorted by the sort program on a line-by-line basis for each file found by cat (in alphanumeric order). Finally, the output from sort is sent to the printer using lp.

The use of pipes in this way provides an extremely effective way of combining many commands together to form more powerful and flexible operations. By contrast, such an ability does not exist in DOS, or in NT.

Processes are explained further in a later lecture, but have been introduced now since certain process-related concepts are relevant when discussing the UNIX 'shell'.

**UNIX Fundamentals: The Shell Command Interface.**

A shell is a command-line interface to a UNIX OS, written in C, using a syntax that is very like the C language. One can enter simple commands (shell commands, system commands, user-defined commands, etc.), but also more complex sequences of commands, including expressions and even entire programs written in a scripting language called 'shell script' which is based on C and known as 'sh' (sh is the lowest level shell; rarely used by ordinary users, it is often used by admins and system scripts). Note that 'command' and 'program' are used synonymously here.

Shells are not in any way like the PC DOS environment; shells are very powerful and offer users and admins a direct communication link to the core OS, though ordinary users will find there is a vast range of commands and programs which they cannot use since they are not the root user.

Modern GUI environments are popular and useful, but some tasks are difficult or impossible to do with an iconic interface, or at the very least are simply slower to perform. Shell commands can be chained together (the output of one command acts as the input for another), or placed into an executable file like a program, except there is no need for a compiler and no object file - shell 'scripts' are widely used by admins for system administration and for performing common tasks such as locating and removing unwanted files. Combined with the facility for full-scale remote administration, shells are very flexible and efficient. For example, I have a single shell script 'command' which simultaneously reboots all the SGI Indys in Ve24. These shortcuts are useful because they minimise keystrokes and mistakes. An admin who issues lengthy and complex command lines repeatedly will find these shortcuts a handy and necessary time-saving feature.

Shells and shell scripts can also use variables, just as a C program can, though the syntax is slightly different. The equivalent of if/then statements can also be used, as can case statements, loop structures, etc. Novice administrators will probably not have to use if/then or other more advanced scripting features at first, and perhaps not even after several years. It is certainly true that any administrator who already knows the C programming language will find it very easy to learn shell script programming, and also the other scripting languages which exist on UNIX systems such as perl (Practical Extraction and Report Language), awk (pattern scanning and processing language) and sed (text stream editor).

perl is a text-processing language, designed for processing text files, extracting useful data, producing reports and results, etc. perl is a very powerful tool for system management, especially combined with other scripting languages. However, perl is perhaps less easy to learn for a novice; the perl man page says, "The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal)." I have personally never had to write a perl program as yet, or a program using awk or sed. This is perhaps a good example if any were required of how largely automated modern UNIX systems are. Note that the perl man page serves as the entire online guide to the perl language and is thus quite large.

An indication of the fact that perl and similar languages can be used to perform complex processing operations can be seen by examining the humourous closing comment in the perl man page:

```
"Perl actually stands for Pathologically Eclectic
 Rubbish Lister, but don't tell anyone I said that."
```

Much of any modern UNIX OS actually operates using shell scripts, many of which use awk, sed and perl as well as ordinary shell commands and system commands. These scripts can look quite complicated, but in general they need not be of any concern to the admin; they are often quite old (ie. written years ago), well understood and bug-free.

Although UNIX is essentially a text-based command-driven system, it is perfectly possible for most

users to do the majority or even all of their work on modern UNIX systems using just the GUI interface. UNIX variants such as IRIX include advanced GUIs which combine the best of both worlds. It's common for a new user to begin with the GUI and only discover the power of the text interface later. This probably happens because most new users are already familiar with other GUI-based systems (eg. Win95) and initially dismiss the shell interface because of prior experience of an operating system such as DOS, ie. they perceive a UNIX shell to be just some weird form of DOS. Shells are not DOS, ie.:

- DOS is an operating system. Win3.1 is built on top of DOS, as is Win95, etc.
- UNIX is an operating system. Shells are a powerful text command interface to UNIX and not the OS itself. A UNIX OS uses shell techniques in many aspects of its operation.

Shells are thus nothing like DOS; they are closely related to UNIX in that the very first version of UNIX included a shell interface, and both are written in C. When a UNIX system is turned on, a shell is used very early in the boot sequence to control what happens and execute actions.

Because of the way UNIX works and how shells are used, much of UNIX's inner workings are hidden, especially at the hardware level. This is good for the user who only sees what she or he wants and needs to see of the file structure. An ordinary user focuses on their home directory and certain useful parts of the file system such as /var/tmp and /usr/share, while an admin will also be interested in other directories which contain system files, device files, etc. such as /etc, /var/adm and /dev.

The most commonly used shells are:

```
bsh         - Bourne Shell; standard/job control
            - command programming language

ksh         - modern alternative to bsh, but still restricted

csh         - Berkeley's C Shell; a better bsh
            - with many additional features

tcsh        - an enhanced version of csh
```

**Figure 8. The various available shells.**

These offer differing degrees of command access/history/recall/editing and support for shell script programming, plus other features such as command aliasing (new names for user-defined sequences of one or more commands). There is also rsh which is essentially a restricted version of the standard command interpreter sh; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. Shells such as csh and tcsh execute the file /etc/cshrc before reading the user's own .cshrc, .login and perhaps .tcshrc file if that exists.

Shells use the concept of a 'path' to determine how to find commands to execute. The 'shell path variable', which is initially defined in the user's .cshrc or .tcshrc file, consists of a list of directories, which may be added to by the user. When a command is entered, the shell environment searches each directory listed in the path for the command. The first instance of a file which matches the command is executed, or an error is given if no such executable command is found. This feature allows multiple versions of the same command to exist in different locations (eg. different releases of a commercial application). The user can change the path variable so that particular commands will run a file from a desired directory.

Try:

```
echo $PATH
```

The list of directories is given.

WARNING: the dot '.' character at the end of a path definition means 'current directory'; it is dangerous to include this in the root user's path definition (this is because a root user could run an ordinary user's program(s) by mistake). Even an ordinary user should think twice about including a period at the end of their path definition. For example, suppose a file called 'la' was present in /tmp and was set so that it could be run by any user. Entering 'la' instead of 'ls' by mistake whilst in /tmp would fail to find 'la' in any normal system directory, but a period in the path definition would result in the shell finding la in /tmp and executing it; thus, if the la file contained malicious commands (eg. '/bin/rm -rf $HOME/mail'), then loss of data could occur.

Typical commands used in a shell include (most useful commands listed first):

```
cd              - change directory
ls              - list contents of directory
rm              - delete a file (no undelete!)
mv              - move a file
cat             - dump contents of a file
more            - display file contents in paged format
find            - search file system for files/directories
grep            - scan file(s) using pattern matching
man             - read/search a man page (try 'man man')
mkdir           - create a directory
rmdir           - remove directory ('rm -r' has the same effect)
pwd             - print current absolute working directory
cmp             - show differences between two files
lp              - print a file
df              - show disk usage
du              - show space used by directories/files
mail            - send an email message to another user
passwd          - change password (yppasswd for systems with NIS)
```

**Figure 9. The commands used most often by any user.**

Editors:

```
vi              - ancient editor. Rarely used (arcane),
                  but occasionally useful, especially
                  for remote administration.

xedit
jot
nedit           - GUI editors (jot is old, nedit is
                - newer, xedit is very simple).
```

**Figure 10. Editor commands.**

Most of these are not built-in shell commands. Enter 'man csh' or 'man tcsh' to see which commands are part of the shell and hence which are other system programs, eg. 'which' is a shell command, but 'grep' is not; 'cd' is a shell command, but 'ls' is not.

vi is an ancient editor developed in the very early days of UNIX when GUI-based displays did not exist. It is not used much today, but many admins swear by it - this is only really because they

know it so well after years of experience. The vi editor can have its uses though, eg. for remote administration: if you happen to be using a Wintel PC in an Internet cafe and decide to access a remote UNIX system via telnet, the vi editor will probably be the only editor which you can use to edit files on the remote system.

Jot has some useful features, especially for programmers (macros, "Electric C Mode"), but is old and contains an annoying colour map bug; this doesn't affect the way jot works, but does sometimes scramble on-screen colours within the jot window. SGI recommends nedit be used instead.

xedit is a very simple text editor. It has an extremely primitive file selection interface, but has a rather nice search/replace mechanism.

nedit is a newer GUI editor with more modern features.

jot is specific to SGI systems, while vi, xedit and nedit exist on any UNIX variant (if not by default, then they can be downloaded in source code or executable format from relevant anonymous ftp sites).

Creating a new shell:

```
sh, csh, tcsh, bsh, ksh - use man pages to see differences
```

I have configured the SGI machines in Ve24 to use tcsh by default due to the numerous extra useful features in tcsh, including file name completion (TAB), command-line editing, alias support, file listing in the middle of a typed command (CTRL-D), command recall/reuse, and many others (the man page lists 36 main extras compared to csh).

Further commands:

```
which          - show location of a command based
                 on current path definition
chown          - change owner ID of a file
chgrp          - change group ID of a file
chmod          - change file access permissions
who            - show who is on the local system
rusers         - show all users on local network
sleep          - pause for a number of seconds
sort           - sort data into a particular order
spell          - run a spell-check on a file
split          - split a file into a number of pieces
strings        - show printable text strings in a file
cut            - cut out selected fields of
                 each line of a file
tr             - substitute/delete characters from
                 a text stream or file
wc             - count the number of words in a file
whoami         - show user ID
write          - send message to another user
wall           - broadcast to all users on local system
talk           - request 1:1 communication link
                 with another user
to_dos         - convert text file to DOS format
                 (add CTRL-M and CTRL-Z)
to_unix        - convert text file to UNIX format
                 (opposite of to_dos)
```

```
su              - adopt the identity of another user
                  (password usually required)
```

**Figure 11. The next most commonly used commands.**

Of the commands shown in Fig 11, only 'which' is a built-in shell command.

Any GUI program can also be executed via a text command (the GUI program is just a high-level interface to the main program), eg. 'fm' for the iconic file manager/viewer, 'apanel' for the Audio Panel, 'printers' for the Printer Manager, 'iconbook' for Icon Catalog, 'mouse' for customise mouse settings, etc. However, not all text commands will have a GUI equivalent - this is especially true of many system administration commands. Other categories are shown in Figs 12 to 17 below.

```
fx              - repartition a disk, plus other functions
mkfs            - make a file system on a disk
mount           - mount a file system (NFS)
ln              - create a link to a file or directory
tar             - create/extract an archive file
gzip            - compress a file (gunzip)
compress        - compress a file (uncompress).
                  Different format from gzip.
pack            - a further compression method (eg. used
                  with man pages and release notes)
head            - show the first few lines in a file
tail            - show the last few lines in a file
```

**Figure 12. File system manipulation commands.**

The tar command is another example where slight differences between UNIX variants exist with respect to default settings. However, command options can always be used to resolve such differences.

```
hinv            - show hardware inventory (SGI specific)
uname           - show OS version
gfxinfo         - show graphics hardware information (SGI-specific)
sysinfo         - print system ID (SGI-specific)
gmemusage       - show current memory usage
ps              - display a snapshot of running process information
top             - constantly updated process list (GUI: gr_top)
kill            - shutdown a process
killall         - shutdown a group of processes
osview          - system resource usage (GUI: gr_osview)
startconsole    - system console, a kind of system monitoring
                  xterm which applications will echo messages into
```

**Figure 13. System Information and Process Management Commands.**

```
inst            - install software (text-based)
swmgr           - GUI interface to inst (the preferred
                  method; easier to use)
versions        - show installed software
```

**Figure 14. Software Management Commands.**

```
cc,
```

```
CC,
gcc             - compile program (further commands
                  may exist for other languages)
make            - run program compilation script
xmkmf           - Use imake on an Imakefile to create
                  vendor-specific make file
lint            - check a C program for errors/bugs
cvd             - CASE tool, visual debugger for C
                  programs (SGI specific)
```

**Figure 15. Application Development Commands.**

```
relnotes        - software release notes (GUI: grelnotes)
man             - manual pages (GUI: xman)
insight         - online books
infosearch      - searchable interface to the above
                  three (IRIX 6.5 and later)
```

**Figure 16. Online Information Commands (all available from the 'Toolchest')**

```
telnet          - open communication link
ftp             - file transfer
ping            - send test packets
traceroute      - display traced route to remote host
nslookup        - translate domain name into IP address
finger          - probe remote host for user information
```

**Figure 17. Remote Access Commands.**

This is not a complete list! And do not be intimidated by the apparent plethora of commands. An admin won't use most of them at first. Many commands are common to any UNIX variant, while those that aren't (eg. hinv) probably have equivalent commands on other UNIX platforms.

Shells can be displayed in different types of window, eg. winterm, xterm. xterms comply with the X11 standard and offer a wider range of features. xterms can be displayed on remote displays, as can any X-based application (this includes just about every program one ever uses). Security note: the remote system must give permission or be configured to allow remote display (xhost command).

If one is accessing a UNIX system via an older text-only terminal (eg. VT100) then the shell operates in 'terminal' mode, where the particular characteristics of the terminal in use determine how the shell communicates with the terminal (details of all known terminals are stored in the /usr/lib/terminfo directory). Shells shown in visual windows (xterms, winterms, etc.) operate a form of terminal emulation that can be made to exactly mimic a basic text-only terminal if required.

Tip: if one ever decides to NFS-mount /usr/lib to save space (thus normally erasing the contents of /usr/lib on the local disk), it is wise to at least leave behind the terminfo directory on the local disk's /usr/lib; thus, should one ever need to logon to the system when /usr/lib is not mounted, terminal communication will still operate normally.

The lack of a fundamental built-in shell environment in WindowsNT is one of the most common criticisms made by IT managers who use NT. It's also why many high-level companies such as movie studios do not use NT, eg. no genuine remote administration makes it hard to manage

clusters of several dozen systems all at once, partly because different systems may be widely dispersed in physical location but mainly because remote administration makes many tasks considerably easier and more convenient.

**Regular Expressions and Metacharacters.**

Shell commands can employ regular expressions and metacharacters which can act as a means for referencing large numbers of files or directories, or other useful shortcuts. Regular expressions are made up of a combination of alphanumeric characters and a series of punctuation characters that have special meaning to the shell. These punctuation characters are called metacharacters when they are used for their special meanings with shell commands.

The most common metacharacter is the wildcard '*', used to reference multiple files and/or directories, eg.:

Dump the contents of all files in the current directory to the display:

```
cat *
```

Remove all object files in the current directory:

```
rm *.o
```

Search all files ending in .txt for the word 'Alex':

```
grep Alex *.txt
```

Print all files beginning with 'March' and ending in '.txt':

```
lp March*.txt
```

Print all files beginning with 'May':

```
lp May*
```

Note that it is not necessary to use 'May*.*' - this is because the dot is just another character that can be a valid part of any UNIX file name at any position, ie. a UNIX file name may include multiple dots. For example, the Blender shareware animation program archive file is called:

```
blender1.56_SGI_6.2_ogl.tar.gz
```

By contrast, DOS has a fixed file name format where the dot is a rigid aspect of any file name. UNIX file names do not have to contain a dot character, and can even contain spaces (though such names can confuse the shell unless one encloses the entire name in quotes "").

Other useful metacharacters relate to executing previously entered commands, perhaps with modification, eg. the '!' is used to recall a previous command, as in:

```
!!           - Repeat previous command
!grep        - Repeat the last command which began with 'grep'
```

For example, an administrator might send 20 test packets to a remote site to see if the remote system is active:

```
ping -c 20 www.sgi.com
```

Following a short break, the administrator may wish to run the same command again, which can be done by entering '!!'. Minutes later, after entering other commands, the admin might want to run the last ping test once more, which is easily possible by entering '!ping'. If no other command had since been entered beginning with 'p', then even just '!p' would work.

The '^' character can be used to modify the previous command, eg. suppose I entered:

```
grwp 'some lengthy search string or whatever' *
```

grep has been spelled incorrectly here, so an error is given ('gwrp: Command not found'). Instead of typing the whole line again, I could enter:

```
^w^e
```

The shell searches the previous command for the first appearance of 'w', replaces that letter with 'e', displays the newly formed command as a means of confirmation and then executes the command. Note: the '^' operator can only search for the first occurrence of the character or string to be changed, ie. in the above example, the word 'whatever' is not changed to 'ehatever'. The parameter to search for, and the pattern to replace any targets found, can be any standard regular expression, ie. a valid sequence of ASCII characters. In the above example, entering '^grwp^grep^' would have had the same effect, though is unnecessarily verbose.

Note that characters such as '!' and '^' operate entirely within the shell, ie. they are not 'memorised' as discrete commands. Thus, within a tcsh, using the Up-Arrow key to recall the previous command after the '^w^e' command sequence does not show any trace of the '^w^e' action. Only the corrected, executed command is shown.

Another commonly used character is the '&' symbol, normally employed to control whether or not a process executed from with a shell is run in the foreground or background. As explained in the UNIX history, UNIX can run many processes at once. Processes employ a parental relationship whereby a process which creates a new process (eg. a shell running a program) is said to be creating a child process. The act of creating a new process is called forking. When running a program from within a shell, the prompt may not come back after the command is entered - this means the new process is running in 'foreground', ie. the shell process is suspended until such time as the forked process terminates. In order to run the process in background, which will allow the shell process to carry on as before and still be used, the '&' symbol must be included at the end of the command.

For example, the 'xman' command normally runs in the foreground: enter 'xman' in a shell and the prompt does not return; close the xman program, or type CTRL-C in the shell window, and the shell prompt returns. This effectively means the xman program is 'tied' to the process which forked it, in this case the shell. If one closes the shell completely (eg. using the top-left GUI button, or a kill command from a different shell) then the xman window vanishes too.

However, if one enters:

```
xman &
```

then the xman program is run in the 'background', ie. the shell prompt returns immediately (note the space is optional, ie. 'xman&' is also valid). This means the xman session is now independent of the process which forked it (the shell) and will still exist even if the shell is closed.

Many programs run in the background by default, eg. swmgr (install system software). The 'fg' command can be used to bring any process into the foreground using the unqiue process ID number which every process has. With no arguments, fg will attempt to bring to the foreground the most recent process which was run in the background. Thus, after entering 'xman&', the 'fg' command on its will make the shell prompt vanish, as if the '&' symbol had never been used.

A process currently running in the foreground can be deliberately 'suspended' using the CTRL-Z sequence. Try running xman in the foreground within a shell and then typing CTRL-Z - the phrase 'suspended' is displayed and the prompt returns, showing that the xman process has been temporarily halted. It still exists, but is frozen. Try using the xman program at this point: notice that the menus cannot be accessed and the window overlay/underlay actions are not dealt with anymore.

Now go back to the shell and enter 'fg' - the xman program is brought back into the foreground and begins running once more. As a final example, try CTRL-Z once more, but this time enter 'bg'. Now the xman process is pushed fully into the background. Thus, if one intends to run a program in the background but forgets to include the '&' symbol, then one can use CTRL-Z followed by 'bg' to place the process in the background.

Note: it is worth mentioning at this point an example of how I once observed Linux to be operating incorrectly. This example, seen in 1997, probably wouldn't happen today, but at the time I was very surprised. Using a csh shell on a PC running Linux, I ran the xedit editor in the background using:

```
xedit&
```

Moments later, I had cause to shutdown the relevant shell, but the xedit session terminated as well, which should not have happened since the xedit process was supposed to be running in background. Exactly why this happened I do not know - presumably there was a bug in the way Linux handled process forking which I am sure has now been fixed. However, in terms of how UNIX is supposed to work, it's a bug which should not have been present.

Actually, since many shells such as tcsh allow one to recall previous commands using the arrow keys, and to edit such commands using Alt/CTRL key combinations and other keys, the need to use metacharacter such as '!' and '^' is lessened. However, they're useful to know in case one encounters a different type of shell, perhaps as a result of a telnet session to a remote site where one may not have any choice over which type of shell is used.

**Standard Input (stdin), Standard Output (stdout), Standard Error (stderr).**

As stated earlier, everything in UNIX is basically treated as a file. This even applies to the concept of where output from a program goes to, and where the input to a program comes from. The relevant files, or text data streams, are called stdin and stdout (standard 'in', standard 'out'). Thus, whenever a command produces a visible output in a shell, what that command is actually doing is sending its output to the file handle known as stdout. In the case of the user typing commands in a shell, stdout is defined to be the display which the user sees.

Similarly, the input to a command comes from stdin which, by default, is the keyboard. This is why, if you enter some commands on their own, they will appear to do nothing at first, when in fact they are simply waiting for input from the stdin stream, ie. the keyboard. Enter 'cat' on its own and see what happens; nothing at first, but then enter any text sequence - what you enter is echoed back to the screen, just as it would be if cat was dumping the contents of a file to the screen.

This stdin input stream can be temporarily redefined so that a command takes its input from somewhere other than the keyboard. This is known as 'redirection'. Similarly, the stdout stream can be redirected so that the output goes somewhere other than the display. The '<' and '>' symbols are used for data redirection. For example:

```
ps -ef > file
```

This runs the ps command, but sends the output into a file. That file could then be examined with cat, more, or loaded into an editor such as nedit or jot.

Try:

```
cat > file
```

You can then enter anything you like until such time as some kind of termination signal is sent, either CTRL-D which acts to end the text stream, or CTRL-C which stops the cat process. Type 'hello', press Enter, then press CTRL-D. Enter 'cat file' to see the file's contents.

A slightly different form of output redirection is '>>' which appends a data stream to the end of an existing file, rather than completely overwriting its current contents. Enter:

```
cat >> file
```

and type 'there!' followed by Enter and then CTRL-D. Now enter 'cat file' and you will see:

```
% cat file
hello
there!
```

By contrast, try the above again but with the second operation also using the single '>' operator. This time, the files contents will only be 'there!'. And note that the following has the same effect as 'cat file' (why?):

```
cat < file
```

Anyone familiar with C++ programming will recognise this syntax as being similar to the way C++ programs display output.

Input and output redirection is used extensively by system shell scripts. Users and administrators can use these operators as a quick and convenient way for managing program input and output. For example, the output from a find command could be redirected into a file for later examination. I often use 'cat > whatever' as a quick and easy way to create a short file without using an editor.

Error messages from programs and commands are also often sent to a different output stream called stderr - by default, stderr is also the relevant display window, or the Console Window if one exists on-screen.

The numeric file handles associated with these three text streams are:

```
0     - stdin
1     - stdout
2     - stderr
```

These numbers can be placed before the < and > operators to select a particular stream to deal with.

Examples of this are given in the notes on shell script programming (Day 2).

The '&&' combination allows one to chain commands together so that each command is only executed if the preceding command was successful, eg.:

```
run_my_prog_which_takes_hours > results && lp results
```

In this example, some arbitrary program is executed which is expected to take a long time. The program's output is redirected into a file called results. If and only if the program terminates successfully will the results file be sent to the default printer by the lp program. Note: any error encountered by the program will also have the error message stored in the results file.

One common use of the && sequence is for on-the-spot backups:

```
cd /home && tar cv . && eject
```

This sequence changes directory to the /home area, archives the contents of /home to DAT and ejects the DAT tape once the archive process has completed. Note that the eject command without any arguments will search for a default removable media device, so this example assumes there is only one such device, a DAT drive, attached to the system. Otherwise, one could use 'eject /dev/tape' to be more specific.

The semicolon can also be used to chain commands together, but in a manner which does not require each command to be successful in order for the next command to be executed, eg. one could run two successive find commands, searching for different types of file, like this (try executing this command in the directory /mapleson/public_html/sgi):

```
find . -name "*.gz" -print; find . -name "*.mpg" -print
```

The output given is:

```
./origin/techreport/compcon97_dv.pdf.gz
./origin/techreport/origin_chap7.pdf.gz
./origin/techreport/origin_chap6.pdf.gz
./origin/techreport/origin_chap5.pdf.gz
./origin/techreport/origin_chap4.pdf.gz
./origin/techreport/origin_chap3.pdf.gz
./origin/techreport/origin_chap2.pdf.gz
./origin/techreport/origin_chap1.5.pdf.gz
./origin/techreport/origin_chap1.0.pdf.gz
./origin/techreport/compcon_paper.pdf.gz
./origin/techreport/origin_techrep.pdf.tar.gz
./origin/techreport/origin_chap1-7TOC.pdf.gz
./pchall/pchal.ps.gz
./o2/phase/phase6.mpg
./o2/phase/phase7.mpg
./o2/phase/phase4.mpg
./o2/phase/phase5.mpg
./o2/phase/phase2.mpg
./o2/phase/phase3.mpg
./o2/phase/phase1.mpg
./o2/phase/phase8.mpg
./o2/phase/phase9.mpg
```

If one changes the first find command so that it will give an error, the second find command still

executes anyway:

```
% find /tmp/gurps -name "*.gz" -print ; find . -name "*.mpg" -print
cannot stat /tmp/gurps
No such file or directory
./o2/phase/phase6.mpg
./o2/phase/phase7.mpg
./o2/phase/phase4.mpg
./o2/phase/phase5.mpg
./o2/phase/phase2.mpg
./o2/phase/phase3.mpg
./o2/phase/phase1.mpg
./o2/phase/phase8.mpg
./o2/phase/phase9.mpg
```

However, if one changes the ; to && and runs the sequence again, this time the second find command will not execute because the first find command produced an error:

```
% find /tmp/gurps -name "*.gz" -print && find . -name "*.mpg" -print
cannot stat /tmp/gurps
No such file or directory
```

As a final example, enter the following:

```
find /usr -name "*.htm*" -print & find /usr -name "*.rgb" -print &
```

This command runs two separate find processes, both in the background at the same time. Unlike the previous examples, the output from each command is displayed first from one, then from the other, and back again in a non-deterministic manner, as and when matching files are located by each process. This is clear evidence that both processes are running at the same time. To shut down the processes, either use 'killall find' or enter 'fg' followed by the use of CTRL-C twice (or one could use kill with the appropriate process IDs, identifiable using 'ps -ef | grep find').

When writing shell script files, the ; symbol is most useful when one can identify commands which do not depend on each other. This symbol, and the other symbols described here, are heavily used in the numerous shell script files which manage many aspects of any modern UNIX OS.

Note: if non-dependent commands are present in a script file or program, this immediately allows one to imagine the idea of a multi-threaded OS, ie. an OS which can run many processes in parallel across multiple processors. A typical example use of such a feature would be batch processing scripts for image processing of medical data, or scripts that manage database systems, financial accounts, etc.

---

References:


1. HP-UX/SUN Interoperability Cookbook, Version 1.0, Copyright 1994 Hewlett-Packard Co.:

   `http://www.hp-partners.com/ptc_public/techsup/SunInterop/`

   comp.sys.hp.hpux FAQ, Copyright 1995 by Colin Wynd:

   `http://hpux.csc.liv.ac.uk/hppd/FAQ/`

2. Department of Computer Science and Electrical Engineering, Heriot Watt University,

Riccarton Campus, Edinburgh, Scotland:

`http://www.cee.hw.ac.uk/`