INTRODUCTION TO LINUX

(The most interesting parts of the text at http://www.doc.ic.ac.uk/~wjk/UnixIntro/)

ZAGREB 2011.

CONTENTS

1. About operating system 3
2. The UNIX filesystem and filesystem actions 6
3. Useful UNIX commands 13
4. Introduction to vi editor 24
5. Shells and system commands 27
Notice 32

1.1. Operating System

An operating system (OS) is a resource manager. It takes the form of a set of software routines that allow users and application programs to access system resources (e.g. the CPU, memory, disks, modems, printers network cards etc.) in a safe, efficient and abstract way.

For example, an OS ensures safe access to a printer by allowing only one application program to send data directly to the printer at any one time. An OS encourages efficient use of the CPU by suspending programs that are waiting for I/O operations to complete to make way for programs that can use the CPU more productively. An OS also provides convenient abstractions (such as files rather than disk locations) which isolate application programmers and users from the details of the underlying hardware.

The architecture of a typical operating system consists of the following:

1) The operating system kernel is in direct control of the underlying hardware. The kernel provides low-level device, memory and processor management functions (e.g. dealing with interrupts from hardware devices, sharing the processor among multiple programs, allocating memory for programs etc.)

2) Basic hardware-independent kernel services are exposed to higher-level programs through a library of system calls (e.g. services to create a file, begin execution of a program, or open a logical network connection to another computer).

3) Application programs (e.g. word processors, spreadsheets) and system utility programs (simple but useful application programs that come with the operating system, e.g. programs which find text inside a group of files) make use of system calls. Applications and system utilities are launched using a shell (a textual command line interface) or a graphical user interface that provides direct user interaction.

Operating systems (and different flavours of the same operating system) can be distinguished from one another by the system calls, system utilities and user interface they provide, as well as by the resource scheduling policies implemented by the kernel.

1.2. A Brief History of UNIX

UNIX has been a popular OS for more than two decades because of its multi-user, multi-tasking environment, stability, portability and powerful networking capabilities. What follows here is a simplified history of how UNIX has developed (to get an idea for how complicated things really are, see the web site http://www.levenez.com/unix/).

In the late 1960s, researchers from General Electric, MIT and Bell Labs launched a joint project to develop an ambitious multi-user, multi-tasking OS for mainframe computers known as MULTICS (Multiplexed Information and Computing System). MULTICS failed, but it did inspire Ken Thompson, who was a researcher at Bell Labs, to have a go at writing a simpler operating system himself. He wrote a simpler version of MULTICS on a PDP7 in assembler and called his attempt UNICS (Uniplexed Information and Computing System). Because memory and CPU power were at a premium in those days, UNICS (eventually shortened to UNIX) used short commands to minimize the space needed to store them and the time needed to decode them - hence the tradition of short UNIX commands we use today, e.g. `ls, cp, rm, mv` etc.

Ken Thompson then teamed up with Dennis Ritchie, the author of the first C compiler in 1973. They rewrote the UNIX kernel in C - this was a big step forwards in terms of the system's portability - and released the Fifth Edition of UNIX to universities in 1974. The Seventh Edition, released in 1978, marked a split in UNIX development into two main branches: SYSV (System 5) and BSD (Berkeley Software Distribution). BSD arose from the University of California at Berkeley where Ken Thompson spent a sabbatical year. Its development was continued by students at Berkeley and other research institutions. SYSV was developed by

AT&T and other commercial companies. UNIX flavours based on SYSV have traditionally been more conservative, but better supported than BSD-based flavours.

The latest incarnations of SYSV (SVR4 or System 5 Release 4) and BSD Unix are actually very similar. Some minor differences are to be found in file system structure, system utility names and options and system call libraries.

Linux is a free open source UNIX OS for PCs that was originally developed in 1991 by Linus Torvalds, a Finnish undergraduate student. Linux is neither pure SYSV or pure BSD. Instead, incorporates some features from each (e.g. SYSV-style startup files but BSD-style file system layout) and aims to conform with a set of IEEE standards called POSIX (Portable Operating System Interface). To maximise code portability, it typically supports SYSV, BSD and POSIX system calls.

The open source nature of Linux means that the source code for the Linux kernel is freely available so that anyone can add features and correct deficiencies. This approach has been very successful and what started as one person's project has now turned into a collaboration of hundreds of volunteer developers from around the globe. The open source approach has not just successfully been applied to kernel code, but also to application programs for Linux.

As Linux has become more popular, several different development streams or distributions have emerged, e.g. Redhat, Slackware, Mandrake, Debian, and Caldera. A distribution comprises a prepackaged kernel, system utilities, GUI interfaces and application programs.

1.3. Architecture of the Linux Operating System

Linux has all of the components of a typical OS:

Kernel
The Linux kernel includes device driver support for a large number of PC hardware devices (graphics cards, network cards, hard disks etc.), advanced processor and memory management features, and support for many different types of filesystems (including DOS floppies and the ISO9660 standard for CDROMs). In terms of the services that it provides to application programs and system utilities, the kernel implements most BSD and SYSV system calls, as well as the system calls described in the POSIX.1 specification.

The kernel (in raw binary form that is loaded directly into memory at system startup time) is typically found in the file /boot/vmlinuz, while the source files can usually be found in /usr/src/linux.

Shells and GUIs
Linux supports two forms of command input: through textual command line shells similar to those found on most UNIX systems (e.g. sh - the Bourne shell, bash - the Bourne again shell and csh - the C shell) and through graphical interfaces (GUIs) such as the KDE and GNOME window managers. If you are connecting remotely to a server your access will typically be through a command line shell.

System Utilities
Virtually every system utility that you would expect to find on standard implementations of UNIX (including every system utility described in the POSIX.2 specification) has been ported to Linux. This includes commands such as `ls, cp, grep, awk, sed, bc, wc, more`, and so on. These system utilities are designed to be powerful tools that do a single task extremely well (e.g. `grep` finds text inside files while `wc` counts the number of words, lines and bytes inside a file). Users can often solve problems by interconnecting these tools instead of writing a large monolithic application program.

Like other UNIX flavours, Linux's system utilities also include server programs called daemons which provide remote network and administration services (e.g. `telnetd` and `sshd` provide remote login facilities, `lpd` provides printing services, `httpd` serves web pages, `crond` runs regular system administration tasks automatically). A daemon (probably derived from the Latin word which refers to a beneficient spirit who

watches over someone, or perhaps short for "Disk And Execution MONitor") is usually spawned automatically at system startup and spends most of its time lying dormant (lurking?) waiting for some event to occur.

Application programs
Linux distributions typically come with several useful application programs as standard. Examples include the `emacs` editor, `xv` (an image viewer), `gcc` (a C compiler), `g++` (a C++ compiler), `xfig` (a drawing package), `latex` (a powerful typesetting language) and `soffice` (StarOffice, which is an MS-Office style clone that can read and write Word, Excel and PowerPoint files).

1.4. Logging into (and out of) UNIX Systems

When you connect to a UNIX computer remotely (using telnet) or when you log in locally using a text-only terminal, you will see the prompt:

```
login:
```

At this prompt, type in your usename and press the enter/return/ key. Remember that UNIX is case sensitive (i.e. Will, WILL and will are all different logins). You should then be prompted for your password:

```
login: will
password:
```

Type your password in at the prompt and press the enter/return/ key. Note that your password will not be displayed on the screen as you type it in.

If you mistype your username or password you will get an appropriate message from the computer and you will be presented with the `login:` prompt again. Otherwise you should be presented with a shell prompt which looks something like this:

```
$
```

To log out of a text-based UNIX shell, type "exit" at the shell prompt (or if that doesn't work try "logout"; if that doesn't work press ctrl-d).

1.5. Changing your password

One of the things you should do when you log in for the first time is to change your password.

The UNIX command to change your password is `passwd`:

```
$ passwd
```

The system will prompt you for your old password, then for your new password. To eliminate any possible typing errors you have made in your new password, it will ask you to reconfirm your new password.

Remember the following points when choosing your password:

(1) Avoid characters which might not appear on all keyboards, e.g. '£'.
(2) The weakest link in most computer security is user passwords so keep your password a secret, don't write it down and don't tell it to anyone else. Also avoid dictionary words or words related to your personal details (e.g. your boyfriend or girlfriend's name or your login).

(3) Make it at least 7 or 8 characters long and try to use a mix of letters, numbers and punctuation.

1.6. General format of UNIX Commands

A UNIX command line consists of the name of a UNIX command (actually the "command" is the name of a built-in shell command, a system utility or an application program) followed by its "arguments" (options and the target filenames and/or expressions). The general syntax for a UNIX command is

```
$ command -options targets
```

Here `command` can be though of as a verb, `options` as an adverb and `targets` as the direct objects of the verb. In the case that the user wishes to specify several options, these need not always be listed separately (the options can sometimes be listed altogether after a single dash).

2.1. The UNIX Filesystem

The UNIX operating system is built around the concept of a filesystem which is used to store all of the information that constitutes the long-term state of the system. This state includes the operating system kernel itself, the executable files for the commands supported by the operating system, configuration information, temporary workfiles, user data, and various special files that are used to give controlled access to system hardware and operating system functions.

Every item stored in a UNIX filesystem belongs to one of four types:

Ordinary files
Ordinary files can contain text, data, or program information. Files cannot contain other files or directories. Unlike other operating systems, UNIX filenames are not broken into a name part and an extension part (although extensions are still frequently used as a means to classify files). Instead they can contain any keyboard character except for '/' and be up to 256 characters long (note however that characters such as *,?,# and & have special meaning in most shells and should not therefore be used in filenames). Putting spaces in filenames also makes them difficult to manipulate - rather use the underscore '_'.

Directories
Directories are containers or folders that hold files, and other directories.

Devices
To provide applications with easy access to hardware devices, UNIX allows them to be used in much the same way as ordinary files. There are two types of devices in UNIX - block-oriented devices which transfer data in blocks (e.g. hard disks) and character-oriented devices that transfer data on a byte-by-byte basis (e.g. modems and dumb terminals).

Links
A link is a pointer to another file. There are two types of links - a hard link to a file is indistinguishable from the file itself. A soft link (or symbolic link) provides an indirect pointer or shortcut to a file. A soft link is implemented as a directory file entry containing a pathname.

2.2. Typical UNIX Directory Structure

The UNIX filesystem is laid out as a hierarchical tree structure which is anchored at a special top-level directory known as the root (designated by a slash '/'). Because of the tree structure, a directory can have many child directories, but only one parent directory. Fig. 2.1 illustrates this layout.

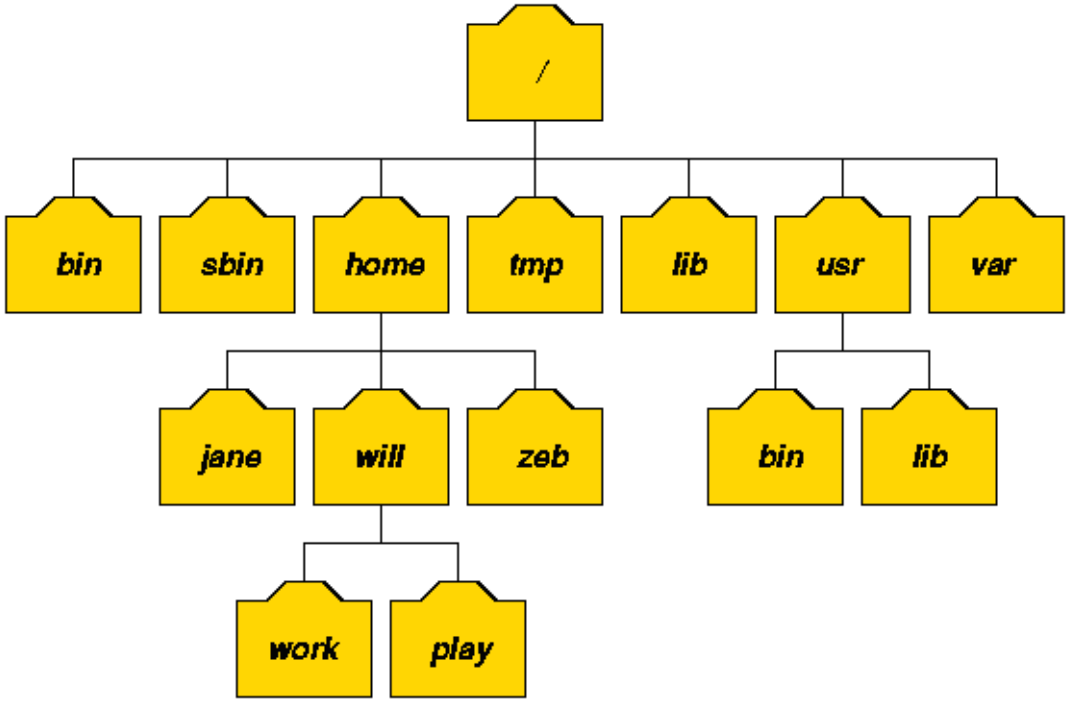


Fig. 2.1: Part of a typical UNIX filesystem tree

To specify a location in the directory hierarchy, we must specify a path through the tree. The path to a location can be defined by an absolute path from the root /, or as a relative path from the current working directory. To specify a path, each directory along the route from the source to the destination must be included in the path, with each directory in the sequence being separated by a slash. To help with the specification of relative paths, UNIX provides the shorthand "`.`" for the current directory and "`..`" for the parent directory. For example, the absolute path to the directory "`play`" is `/home/will/play`, while the relative path to this directory from "`zeb`" is `../will/play`.

Fig. 2.2 shows some typical directories you will find on UNIX systems and briefly describes their contents. Note that these although these subdirectories appear as part of a seamless logical filesystem, they do not need be present on the same hard disk device; some may even be located on a remote machine and accessed across a network.

Directory	**Typical Contents**
/	The "root" directory
/bin	Essential low-level system utilities
/usr/bin	Higher-level system utilities and application programs
/sbin	Superuser system utilities (for performing system administration tasks)
/lib	Program libraries (collections of system calls that can be included in programs by a compiler) for low-level system utilities
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home or /	User home directories containing personal file space

homes	for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files
/dev	Hardware devices
/proc	A pseudo-filesystem which is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

Fig. 2.2: Typical UNIX directories

When you log into UNIX, your current working directory is your user home directory. You can refer to your home directory at any time as "~" and the home directory of other users as "~<login>". So ~will/play is another way for user jane to specify an absolute path to the directory /homes/will/play. User will may refer to the directory as ~/play.

2.3. Directory and File Handling Commands

This section describes some of the more important directory and file handling commands.

pwd (print [current] working directory)

pwd displays the full absolute path to the your current location in the filesystem. So

```
$ pwd
/usr/bin
```

implies that /usr/bin is the current working directory.

ls (list directory)

ls lists the contents of a directory. If no target directory is given, then the contents of the current working directory are displayed. So, if the current working directory is /,

```
$ ls
bin   dev  home  mnt   share  usr  var
boot  etc  lib   proc  sbin   tmp  vol
```

Actually, ls doesn't show you all the entries in a directory - files and directories that begin with a dot (.) are hidden (this includes the directories '.' and '..' which are always present). The reason for this is that files that begin with a . usually contain important configuration information and should not be changed under normal circumstances. If you want to see all files, ls supports the -a option:

```
$ ls -a
```

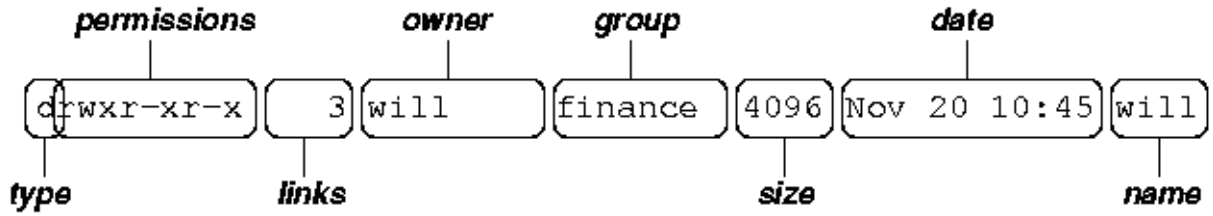
Even this listing is not that helpful - there are no hints to properties such as the size, type and ownership of files, just their names. To see more detailed information, use the -l option (long listing), which can be combined with the -a option as follows:

```
$ ls -a -l
```

(or, equivalently,)

```
$ ls -al
```

Each line of the output looks like this:



where:

(1) type is a single character which is either 'd' (directory), '-' (ordinary file), 'l' (symbolic link), 'b' (block-oriented device) or 'c' (character-oriented device).

(2) permissions is a set of characters describing access rights. There are 9 permission characters, describing 3 access types given to 3 user categories. The three access types are read ('r'), write ('w') and execute ('x'), and the three users categories are the user who owns the file, users in the group that the file belongs to and other users (the general public). An 'r', 'w' or 'x' character means the corresponding permission is present; a '-' means it is absent.

(3) links refers to the number of filesystem links pointing to the file/directory (see the discussion on hard/soft links in the next section).

(4) owner is usually the user who created the file or directory.

(5) group denotes a collection of users who are allowed to access the file according to the group access rights specified in the permissions field.

(6) size is the length of a file, or the number of bytes used by the operating system to store the list of files in a directory.

(7) date is the date when the file or directory was last modified (written to). The `-u` option display the time when the file was last accessed (read).

(8) name is the name of the file or directory.

`ls` supports more options. To find out what they are, type:

```
$ man ls
```

`man` is the online UNIX user manual, and you can use it to get help with commands and find out about what options are supported. It has quite a terse style which is often not that helpful, so some users prefer to the use the (non-standard) `info` utility if it is installed:

```
$ info ls
```

`cd` (change [current working] directory)

`$ cd` path

changes your current working directory to path (which can be an absolute or a relative path). One of the most common relative paths to use is '..' (i.e. the parent directory of the current directory).

Used without any target directory

```
$ cd
```

resets your current working directory to your home directory (useful if you get lost). If you change into a directory and you subsequently want to return to your original directory, use

```
$ cd -
```

`mkdir` (make directory)

```
$ mkdir directory
```

creates a subdirectory called directory in the current working directory. You can only create subdirectories in a directory if you have write permission on that directory.

`rmdir` (remove directory)

```
$ rmdir directory
```

removes the subdirectory directory from the current working directory. You can only remove subdirectories if they are completely empty (i.e. of all entries besides the '.' and '..' directories).

`cp` (copy)

`cp` is used to make copies of files or entire directories. To copy files, use:

```
$ cp source-file(s) destination
```

where source-file(s) and destination specify the source and destination of the copy respectively. The behaviour of `cp` depends on whether the destination is a file or a directory. If the destination is a file, only one source file is allowed and `cp` makes a new file called destination that has the same contents as the source file. If the destination is a directory, many source files can be specified, each of which will be copied into the destination directory. Section 2.6 will discuss efficient specification of source files using wildcard characters.

To copy entire directories (including their contents), use a recursive copy:

```
$ cp -rd source-directories destination-directory
```

`mv` (move/rename)

`mv` is used to rename files/directories and/or move them from one directory into another. Exactly one source and one destination must be specified:

```
$ mv source destination
```

If destination is an existing directory, the new name for source (whether it be a file or a directory) will be destination/source. If source and destination are both files, source is renamed destination. N.B.: if destination is an existing file it will be destroyed and overwritten by source (you can use the `-i` option if you would like

to be asked for confirmation before a file is overwritten in this way).

`rm` (remove/delete)

`$ rm` target-file(s)

removes the specified files. Unlike other operating systems, it is almost impossible to recover a deleted file unless you have a backup (there is no recycle bin!) so use this command with care. If you would like to be asked before files are deleted, use the `-i` option:

```
$ rm -i myfile
rm: remove 'myfile'?
```

`rm` can also be used to delete directories (along with all of their contents, including any subdirectories they contain). To do this, use the `-r` option. To avoid `rm` from asking any questions or giving errors (e.g. if the file doesn't exist) you used the `-f` (force) option. Extreme care needs to be taken when using this option - consider what would happen if a system administrator was trying to delete user `will`'s home directory and accidentally typed:

```
$ rm -rf / home/will
```

(instead of `rm -rf /home/will`).

`cat` (catenate/type)

`$ cat` target-file(s)

displays the contents of target-file(s) on the screen, one after the other. You can also use it to create files from keyboard input as follows (`>` is the output redirection operator, which will be discussed in the next chapter):

```
$ cat > hello.txt
hello world!
[ctrl-d]
$ ls hello.txt
hello.txt
$ cat hello.txt
hello world!
$
```

`more` and `less` (catenate with pause)

`$ more` target-file(s)

displays the contents of target-file(s) on the screen, pausing at the end of each screenful and asking the user to press a key (useful for long files). It also incorporates a searching facility (press '`/`' and then type a phrase that you want to look for).

You can also use `more` to break up the output of commands that produce more than one screenful of output as follows (`|` is the pipe operator, which will be discussed in the next chapter):

```
$ ls -l | more
```

`less` is just like `more`, except that has a few extra features (such as allowing users to scroll backwards and forwards through the displayed file). `less` not a standard utility, however and may not be present on all UNIX systems.

2.4. Making Hard and Soft (Symbolic) Links

Direct (hard) and indirect (soft or symbolic) links from one file or directory to another can be created using the `ln` command.

`$ ln` filename linkname

creates another directory entry for filename called linkname (i.e. linkname is a hard link). Both directory entries appear identical (and both now have a link count of 2). If either filename or linkname is modified, the change will be reflected in the other file (since they are in fact just two different directory entries pointing to the same file).

`$ ln -s` filename linkname

creates a shortcut called linkname (i.e. linkname is a soft link). The shortcut appears as an entry with a special type ('`l`'):

```
$ ln -s hello.txt bye.txt
$ ls -l bye.txt
lrwxrwxrwx  1 will finance 13 bye.txt -> hello.txt
$
```

The link count of the source file remains unaffected. Notice that the permission bits on a symbolic link are not used (always appearing as `rwxrwxrwx`). Instead the permissions on the link are determined by the permissions on the target (`hello.txt` in this case).

Note that you can create a symbolic link to a file that doesn't exist, but not a hard link. Another difference between the two is that you can create symbolic links across different physical disk devices or partitions, but hard links are restricted to the same disk partition. Finally, most current UNIX implementations do not allow hard links to point to directories.

2.5. Specifying multiple filenames

Multiple filenames can be specified using special pattern-matching characters. The rules are:

- '`?`' matches any single character in that position in the filename.
- '*' matches zero or more characters in the filename. A '*' on its own will match all files. '`*.*`' matches all files with containing a '`.`'.
- Characters enclosed in square brackets ('[' and ']') will match any filename that has one of those characters in that position.
- A list of comma separated strings enclosed in curly braces ("{" and "}") will be expanded as a Cartesian product with the surrounding characters.

For example:

`???` matches all three-character filenames.
`?ell?` matches any five-character filenames with '`ell`' in the middle.
`he*` matches any filename beginning with '`he`'.
`[m-z]*[a-l]` matches any filename that begins with a letter from '`m`' to '`z`' and ends in a letter from '`a`' to

'l'.
`{/usr,}{/bin,/lib}/file` expands to `/usr/bin/file /usr/lib/file /bin/file` and `/lib/file`.

Note that the UNIX shell performs these expansions (including any filename matching) on a command's arguments before the command is executed.

2.6. Quotes

As we have seen certain special characters (e.g. '*', '-','{' etc.) are interpreted in a special way by the shell. In order to pass arguments that use these characters to commands directly (i.e. without filename expansion etc.), we need to use special quoting characters. There are three levels of quoting that you can try:

- Try insert a '\' in front of the special character.
- Use double quotes (") around arguments to prevent most expansions.
- Use single forward quotes (') around arguments to prevent all expansions.

There is a fourth type of quoting in UNIX. Single backward quotes (`) are used to pass the output of some command as an input argument to another. For example:

```
$ hostname
rose
$ echo this machine is called `hostname`
this machine is called rose
```

3.1. File and Directory Permissions

Permission	File	Directory
read	User can look at the contents of the file	User can list the files in the directory
write	User can modify the contents of the file	User can create new files and remove existing files in the directory
execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

Fig 3.1: Interpretation of permissions for files and directories

As we have seen in the previous chapter, every file or directory on a UNIX system has three types of permissions, describing what operations can be performed on it by various categories of users. The permissions are read (`r`), write (`w`) and execute (`x`), and the three categories of users are user/owner (`u`), group (`g`) and others (`o`). Because files and directories are different entities, the interpretation of the permissions assigned to each differs slightly, as shown in Fig 3.1.

File and directory permissions can only be modified by their owners, or by the superuser (`root`), by using the `chmod` system utility.

`chmod` (change [file or directory] mode)

`$ chmod` options files

`chmod` accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits (octal is like decimal except that the digit range is 0 to 7 instead of 0 to 9). Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions onto their corresponding octal digits is as follows:

---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

For example the command:

```
$ chmod 600 private.txt
```

sets the permissions on `private.txt` to `rw-------` (i.e. only the owner can read and write to the file).

Permissions may be specified symbolically, using the symbols `u` (user), `g` (group), `o` (other), `a` (all), `r` (read), `w` (write), `x` (execute), `+` (add permission), `-` (take away permission) and `=` (assign permission). For example, the command:

```
$ chmod ug=rw,o-rw,a-x *.txt
```

sets the permissions on all files ending in `*.txt` to `rw-rw----` (i.e. the owner and users in the file's group can read and write to the file, while the general public do not have any sort of access).

`chmod` also supports a `-R` option which can be used to recursively modify file permissions, e.g.

```
$ chmod -R go+r play
```

will grant group and other read rights to the directory `play` and all of the files and directories within `play.`

`chgrp` (change group)

`$ chgrp` group files

can be used to change the group that a file or directory belongs to. It also supports a `-R` option.

3.2. Inspecting File Content

Besides `cat` there are several other useful utilities for investigating the contents of files:

`file` filename(s)

`file` analyzes a file's contents for you and reports a high-level description of what type of file it appears to be:

```
$ file myprog.c letter.txt webpage.html
myprog.c:      C program text
letter.txt:    English text
webpage.html:  HTML document text
```

file can identify a wide range of files but sometimes gets understandably confused (e.g. when trying to automatically detect the difference between C++ and Java code).

`head, tail` filename

`head` and `tail` display the first and last few lines in a file respectively. You can specify the number of lines as an option, e.g.

```
$ tail -20 messages.txt
$ head -5 messages.txt
```

`tail` includes a useful `-f` option that can be used to continuously monitor the last few lines of a (possibly changing) file. This can be used to monitor log files, for example:

```
$ tail -f /var/log/messages
```

continuously outputs the latest additions to the system log file.

`objdump` options binaryfile

`objdump` can be used to disassemble binary files - that is it can show the machine language instructions which make up compiled application programs and system utilities.

`od` options filename (octal dump)

`od` can be used to displays the contents of a binary or text file in a variety of formats, e.g.

```
$ cat hello.txt
hello world
$ od -c hello.txt
0000000  h  e  l  l  o     w  o  r  l  d \n
0000014
$ od -x hello.txt
0000000 6865 6c6c 6f20 776f 726c 640a
0000014
```

There are also several other useful content inspectors that are non-standard (in terms of availability on UNIX systems) but are nevertheless in widespread use. They are summarised in Fig. 3.2.

File type	Typical extension	Content viewer
Portable Document Format	.pdf	acroread

Postscript Document	.ps	ghostview
DVI Document	.dvi	xdvi
JPEG Image	.jpg	xv
GIF Image	.gif	xv
MPEG movie	.mpg	mpeg_play
WAV sound file	.wav	realplayer
HTML document	.html	netscape

Fig 3.2: Other file types and appropriate content viewers.

3.3. Finding Files

There are at least three ways to find files when you don't know their exact location:

`find`

If you have a rough idea of the directory tree the file might be in (or even if you don't and you're prepared to wait a while) you can use `find`:

`$ find` directory `-name` targetfile `-print`

`find` will look for a file called targetfile in any part of the directory tree rooted at directory. targetfile can include wildcard characters. For example:

```
$ find /home -name "*.txt" -print 2>/dev/null
```

will search all user directories for any file ending in "`.txt`" and output any matching files (with a full absolute or relative path). Here the quotes (") are necessary to avoid filename expansion, while the `2>/dev/null` suppresses error messages (arising from errors such as not being able to read the contents of directories for which the user does not have the right permissions).

`find` can in fact do a lot more than just find files by name. It can find files by type (e.g. `-type f` for files, `-type d` for directories), by permissions (e.g. `-perm o=r` for all files and directories that can be read by others), by size (`-size`) etc. You can also execute commands on the files you find. For example,

```
$ find . -name "*.txt" -exec wc -l '{}' ';'
```

counts the number of lines in every text file in and below the current directory. The `'{}'` is replaced by the name of each file found and the `';'` ends the `-exec` clause.

For more information about `find` and its abilities, use `man find` and/or `info find`.

`which` (sometimes also called `whence`) command

If you can execute an application program or system utility by typing its name at the shell prompt, you can use `which` to find out where it is stored on disk. For example:

```
$ which ls
```

```
/bin/ls
```

`locate` string

`find` can take a long time to execute if you are searching a large filespace (e.g. searching from `/` downwards). The `locate` command provides a much faster way of locating all files whose names match a particular search string. For example:

```
$ locate ".txt"
```

will find all filenames in the filesystem that contain "`.txt`" anywhere in their full paths.

One disadvantage of `locate` is it stores all filenames on the system in an index that is usually updated only once a day. This means `locate` will not find files that have been created very recently. It may also report filenames as being present even though the file has just been deleted. Unlike `find`, `locate` cannot track down files on the basis of their permissions, size and so on.

3.4. Finding Text in Files

`grep` (General Regular Expression Print)

`$ grep` options pattern files

`grep` searches the named files (or standard input if no files are named) for lines that match a given pattern. The default behaviour of `grep` is to print out the matching lines. For example:

```
$ grep hello *.txt
```

searches all text files in the current directory for lines containing "hello". Some of the more useful options that `grep` provides are:
`-c` (print a count of the number of lines that match), `-i` (ignore case), `-v` (print out the lines that don't match the pattern) and `-n` (printout the line number before printing the matching line). So

```
$ grep -vi hello *.txt
```

searches all text files in the current directory for lines that do not contain any form of the word hello (e.g. Hello, HELLO, or hELlO).

If you want to search all files in an entire directory tree for a particular pattern, you can combine `grep` with `find` using backward single quotes to pass the output from `find` into `grep`. So

```
$ grep hello `find . -name "*.txt" -print`
```

will search all text files in the directory tree rooted at the current directory for lines containing the word "hello".

The patterns that `grep` uses are actually a special type of pattern known as regular expressions. Just like arithemetic expressions, regular expressions are made up of basic subexpressions combined by operators.

The most fundamental expression is a regular expression that matches a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any other character with special meaning may be quoted by preceding it with a backslash (\). A list of characters enclosed by '`[`' and '`]`'

matches any single character in that list; if the first character of the list is the caret `^', then it matches any character not in the list. A range of characters can be specified using a dash (-) between the first and last items in the list. So `[0-9]` matches any digit and `[^a-z]` matches any character that is not a digit.

The caret `^' and the dollar sign `$' are special characters that match the beginning and end of a line respectively. The dot '.' matches any character. So

```
$ grep ^..[l-z]$ hello.txt
```

matches any line in `hello.txt` that contains a three character sequence that ends with a lowercase letter from l to z.

`egrep` (extended grep) is a variant of grep that supports more sophisticated regular expressions. Here two regular expressions may be joined by the operator `|'; the resulting regular expression matches any string matching either subexpression. Brackets '(' and ')' may be used for grouping regular expressions. In addition, a regular expression may be followed by one of several repetition operators:

`?' means the preceding item is optional (matched at most once).
`*' means the preceding item will be matched zero or more times.
`+' means the preceding item will be matched one or more times.
`{N}' means the preceding item is matched exactly N times.
`{N,}' means the preceding item is matched N or more times.
`{N,M}' means the preceding item is matched at least N times, but not more than M times.

For example, if `egrep` was given the regular expression

```
'(^[0-9]{1,5}[a-zA-Z ]+$)|none'
```

it would match any line that either begins with a number up to five digits long, followed by a sequence of one or more letters or spaces, or contains the word `none`.

You can read more about regular expressions on the `grep` and `egrep` manual pages.

Note that UNIX systems also usually support another `grep` variant called `fgrep` (fixed grep) which simply looks for a fixed string inside a file (but this facility is largely redundant).

3.5. Sorting Files

There are two facilities that are useful for sorting files in UNIX:

`sort` filenames

`sort` sorts lines contained in a group of files alphabetically (or if the `-n` option is specified) numerically. The sorted output is displayed on the screen, and may be stored in another file by redirecting the output. So

```
$ sort input1.txt input2.txt > output.txt
```

outputs the sorted concentenation of files `input1.txt` and `input2.txt` to the file `output.txt`.

`uniq` filename

`uniq` removes duplicate adjacent lines from a file. This facility is most useful when combined with `sort`:

```
$ sort input.txt | uniq > output.txt
```

3.6. File Compression and Backup

UNIX systems usually support a number of utilities for backing up and compressing files. The most useful are:

`tar` (tape archiver)

`tar` backs up entire directories and files onto a tape device or (more commonly) into a single disk file known as an archive. An archive is a file that contains other files plus information about them, such as their filename, owner, timestamps, and access permissions. `tar` does not perform any compression by default.

To create a disk file `tar` archive, use

`$ tar -cvf` archivename filenames

where archivename will usually have a `.tar` extension. Here the `c` option means create, `v` means verbose (output filenames as they are archived), and `f` means file.To list the contents of a `tar` archive, use

`$ tar -tvf` archivename

To restore files from a `tar` archive, use

`$ tar -xvf` archivename

`cpio`

`cpio` is another facility for creating and reading archives. Unlike `tar`, `cpio` doesn't automatically archive the contents of directories, so it's common to combine `cpio` with `find` when creating an archive:

`$ find . -print -depth | cpio -ov -Htar >` archivename

This will take all the files in the current directory and the
directories below and place them in an archive called archivename.The `-depth` option controls the order in which the filenames are produced and is recommended to prevent problems with directory permissions when doing a restore.The `-o` option creates the archive, the `-v` option prints the names of the files archived as they are added and the `-H` option specifies an archive format type (in this case it creates a `tar` archive). Another common archive type is `crc`, a portable format with a checksum for error control.

To list the contents of a `cpio` archive, use

`$ cpio -tv <` archivename

To restore files, use:

`$ cpio -idv <` archivename

Here the `-d` option will create directories as necessary. To force `cpio` to extract files on top of files of the

same name that already exist (and have the same or later modification time), use the `-u` option.

`compress`, `gzip`

`compress` and `gzip` are utilities for compressing and decompressing individual files (which may be or may not be archive files). To compress files, use:

```
$ compress filename
```
or
```
$ gzip filename
```

In each case, filename will be deleted and replaced by a compressed file called filename.Z or filename.gz. To reverse the compression process, use:

```
$ compress -d filename
```
or
```
$ gzip -d filename
```

3.7. Handling Removable Media

UNIX supports tools for accessing removable media such as CDROMs and floppy disks.

`mount, umount`

The `mount` command serves to attach the filesystem found on some device to the filesystem tree. Conversely, the `umount` command will detach it again (it is very important to remember to do this when removing the floppy or CDROM). The file `/etc/fstab` contains a list of devices and the points at which they will be attached to the main filesystem:

```
$ cat /etc/fstab
/dev/fd0   /mnt/floppy  auto    rw,user,noauto  0 0
/dev/hdc   /mnt/cdrom   iso9660 ro,user,noauto  0 0
```

In this case, the mount point for the floppy drive is `/mnt/floppy` and the mount point for the CDROM is `/mnt/cdrom`. To access a floppy we can use:

```
$ mount /mnt/floppy
$ cd /mnt/floppy
$ ls (etc...)
```

To force all changed data to be written back to the floppy and to detach the floppy disk from the filesystem, we use:

```
$ umount /mnt/floppy
```

`mtools`

If they are installed, the (non-standard) `mtools` utilities provide a convenient way of accessing DOS-formatted floppies without having to mount and unmount filesystems. You can use DOS-type commands like "`mdir a:`", "`mcopy a:*.* .`", "`mformat a:`", etc. (see the `mtools` manual pages for more details).

3.8. Processes

A process is a program in execution. Every time you invoke a system utility or an application program from a shell, one or more "child" processes are created by the shell in response to your command. All UNIX processes are identified by a unique process identifier or PID. An important process that is always present is the `init` process. This is the first process to be created when a UNIX system starts up and usually has a PID of 1. All other processes are said to be "descendants" of `init`.

3.9. Pipes

The pipe ('|') operator is used to create concurrently executing processes that pass data directly to one another. It is useful for combining system utilities to perform more complex functions. For example:

```
$ cat hello.txt | sort | uniq
```

creates three processes (corresponding to `cat`, `sort` and `uniq`) which execute concurrently. As they execute, the output of the who process is passed on to the `sort` process which is in turn passed on to the `uniq` process. `uniq` displays its output on the screen (a sorted list of users with duplicate lines removed). Similarly:

```
$ cat hello.txt | grep "dog" | grep -v "cat"
```

finds all lines in `hello.txt` that contain the string "`dog`" but do not contain the string "`cat`".

3.10. Redirecting Input and Output

The output from programs is usually written to the screen, while their input usually comes from the keyboard (if no file arguments are given). In technical terms, we say that processes usually write to standard output (the screen) and take their input from standard input (the keyboard). There is in fact another output channel called standard error, where processes write their error messages; by default error messages are also sent to the screen.

To redirect standard output to a file instead of the screen, we use the `>` operator:

```
$ echo hello
hello
$ echo hello > output
$ cat output
hello
```

In this case, the contents of the file `output` will be destroyed if the file already exists. If instead we want to append the output of the echo command to the file, we can use the >> operator:

```
$ echo bye >> output
$ cat output
hello
bye
```

To capture standard error, prefix the `>` operator with a 2 (in UNIX the file numbers 0, 1 and 2 are assigned to standard input, standard output and standard error respectively), e.g.:

```
$ cat nonexistent 2>errors
```

```
$ cat errors
cat: nonexistent: No such file or directory
$
```

You can redirect standard error and standard output to two different files:

```
$ find . -print 1>errors 2>files
```

or to the same file:

```
$ find . -print 1>output 2>output
```

or

```
$ find . -print >& output
```

Standard input can also be redirected using the `<` operator, so that input is read from a file instead of the keyboard:

```
$ cat < output
hello
bye
```

You can combine input redirection with output redirection, but be careful not to use the same filename in both places. For example:

```
$ cat < output > output
```

will destroy the contents of the file `output`. This is because the first thing the shell does when it sees the `>` operator is to create an empty file ready for the output.

One last point to note is that we can pass standard output to system utilities that require filenames as "`-`":

```
$ cat package.tar.gz | gzip -d | tar tvf -
```

Here the output of the `gzip -d` command is used as the input file to the `tar` command.

3.11. Connecting to Remote Machines

`telnet` machinename

`telnet` provides an insecure mechanism for logging into remote machines. It is insecure because all data (including your username and password) is passed in unencrypted format over the network. For this reason, `telnet` login access is disabled on most systems and where possible it should be avoided in favour of secure alternatives such as `ssh`.

`telnet` is still a useful utility, however, because, by specifying different port numbers, `telnet` can be used to connect to other services offered by remote machines besides remote login (e.g. web pages, email, etc.) and reveal the mechanisms behind how those services are offered. For example,

```
$ telnet www.doc.ic.ac.uk 80
Trying 146.169.1.10...
Connected to seagull.doc.ic.ac.uk (146.169.1.10).
Escape character is '^]'.
```

```
GET / HTTP/1.0


HTTP/1.1 200 OK
Date: Sun, 10 Dec 2000 21:06:34 GMT
Server: Apache/1.3.14 (Unix)
Last-Modified: Tue, 28 Nov 2000 16:09:20 GMT
ETag: "23dcfd-3806-3a23d8b0"
Accept-Ranges: bytes
Content-Length: 14342
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
    <TITLE>Department of Computing, Imperial College, London: Home
Page</TITLE>
</HEAD>
```
(etc)

Here `www.doc.ic.ac.uk` is the name of the remote machine (in this case the web server for the Department of Computing at Imperial College in London). Like most web servers, it offers web page services on port 80 through the daemon `httpd` (to see what other services are potentially available on a machine, have a look at the file `/etc/services`; and to see what services are actually active, see `/etc/inetd.conf`). By entering a valid HTTP GET command (HTTP is the protocol used to serve web pages) we obtain the top-level home page in HTML format. This is exactly the same process that is used by a web browser to access web pages.

`rlogin, rsh`

`rlogin` and `rsh` are insecure facilities for logging into remote machines and for executing commands on remote machines respectively. Along with `telnet`, they have been superseded by `ssh`.

`ssh` machinename (secure shell)

`ssh` is a secure alternative for remote login and also for executing commands in a remote machine. It is intended to replace `rlogin` and `rsh`, and provide secure encrypted communications between two untrusted hosts over an insecure network. X11 connections (i.e. graphics) can also be forwarded over the secure channel (another advantage over `telnet`, `rlogin` and `rsh`). `ssh` is not a standard system utility, although it is a de facto standard. It can be obtained from http://www.ssh.org.

`ssh` clients are also available for Windows machines (e.g. there is a good `ssh` client called `putty`).

3.12. Internet related utilities

`netscape`

`netscape` is a fully-fledged graphical web browser (like Internet Explorer).

`lynx`

`lynx` provides a way to browse the web on a text-only terminal.

`wget` URL

`wget` provides a way to retrieve files from the web (using the HTTP protocol). `wget` is non-interactive, which means it can run in the background, while the user is not logged in (unlike most web browsers). The content retrieved by `wget` is stored as raw HTML text (which can be viewed later using a web browser).

Note that `netscape`, `lynx` and `wget` are not standard UNIX system utilities, but are frequently-installed application packages.

3.13. Printer Control

`lpr -P`printqueue filename

`lpr` adds a document to a print queue, so that the document is printed when the printer is available. Look at `/etc/printcap` to find out what printers are available.

`lpq -P`printqueue

`lpq` checks the status of the specified print queue. Each job will have an associated job number.

`lprm -P`printqueue jobnumber

`lprm` removes the given job from the specified print queue.

Note that `lpr`, `lpq` and `lprm` are BSD-style print management utilities. If you are using a strict SYSV UNIX, you may need to use the SYSV equivalents `lp`, `lpstat` and `cancel`.

3.14. Manual Pages

`man`

More information is available on most UNIX commands is available via the online manual pages, which are accessible through the `man` command. The online documentation is in fact divided into sections. Traditionally, they are

```
1 User-level commands
2 System calls
3 Library functions
4 Devices and device drivers
5 File formats
6 Games
7 Various miscellaneous stuff - macro packages etc.
8 System maintenance and operation commands
```

Sometimes `man` gives you a manual page from the wrong section. For example, say you were writing a program and you needed to use the `rmdir` system call. `man rmdir` gives you the manual page for the user-level command `rmdir`. To force `man` to look in Section 2 of the manual instead, type `man 2 rmdir` (or`man -s2 rmdir` on some systems).

`man` can also find manual pages which mention a particular topic. For example, `man -k postscript` should produce a list of utilities that can produce and manipulate postscript files.

`info`

`info` is an interactive, somewhat more friendly and helpful alternative to `man`. It may not be installed on all systems, however.

4.1. Introduction to vi

`vi` (pronounced "vee-eye", short for visual, or perhaps vile) is a display-oriented text editor based on an underlying line editor called `ex`. Although beginners usually find `vi` somewhat awkward to use, it is useful to learn because it is universally available (being supplied with all UNIX systems). It also uses standard alphanumeric keys for commands, so it can be used on almost any terminal or workstation without having to worry about unusual keyboard mappings. System administrators like users to use `vi` because it uses very few system resources.

To start `vi`, enter:

`$ vi` filename

where filename is the name of the file you want to edit. If the file doesn't exist, `vi` will create it for you.

4.2. Basic Text Input and Navigation in vi

The main feature that makes `vi` unique as an editor is its mode-based operation. `vi` has two modes: command mode and input mode. In command mode, characters you type perform actions (e.g. moving the cursor, cutting or copying text, etc.) In input mode, characters you type are inserted or overwrite existing text.

When you begin `vi`, it is in command mode. To put `vi` into input mode, press `i` (insert). You can then type text which is inserted at the current cursor location; you can correct mistakes with the backspace key as you type.To get back into command mode, press ESC (the escape key). Another way of inserting text, especially useful when you are at the end of a line is to press `a` (append).

In command mode, you are able to move the cursor around your document. `h`, `j`, `k` and `l` move the cursor left, down, up and right respectively (if you are lucky the arrow keys may also work). Other useful keys are `^` and `$` which move you to the beginning and end of a line respectively. `w` skips to the beginning of the next word and `b` skips back to the beginning of the previous word. To go right to the top of the document, press `1` and then `G`. To go the bottom of the document, press `G`. To skip forward a page, press `^F`, and to go back a page, press `^B`. To go to a particular line number, type the line number and press `G`, e.g. `55G` takes you to line 55.

To delete text, move the cursor over the first character of the group you want to delete and make sure you are in command mode. Press `x` to delete the current character, `dw` to delete the next word, `d4w` to delete the next 4 words, `dd` to delete the next line, `4dd` to delete the next 4 lines, `d$` to delete to the end of the line or even `dG` to delete to the end of the document. If you accidentally delete too much, pressing `u` will undo the last change.

Occasionally you will want to join two lines together. Press `J` to do this (trying to press backspace on the beginning of the second line does not have the intuitive effect!).

4.3. Moving and copying text in vi

`vi` uses buffers to store text that is deleted. There are nine numbered buffers (1-9) as well as the undo buffer.

Usually buffer 1 contains the most recent deletion, buffer 2 the next recent, etc.

To cut and paste in `vi`, delete the text (using e.g. `5dd` to delete 5 lines). Then move to the line where you want the text to appear and press `p`. If you delete something else before you paste, you can still retrieve the delete text by pasting the contents of the delete buffers. You can do this by typing `"1p`, `"2p`, etc.

To copy and paste, "yank" the text (using e.g. `5yy` to copy 5 lines). Then move to the line where you want the text to appear and press `p`.

4.4. Searching and Replacing Text in vi

In command mode, you can search for text by specifying regular expressions. To search forward, type `/` and then a regular expression and press . To search backwards, begin with a `?` instead of a `/`. To find the next text that matches your regular expression press `n`.

To search and replace all occurences of pattern1 with pattern2, type `:%s/`pattern1`/`pattern2`/g`. To be asked to confirm each replacement, add a `c` to this substitution command. Instead of the `%` you can also give a range of lines (e.g. 1,17) over which you want the substitution to apply.

4.5. Other Useful vi Commands

Programmers might like the `:set number` command which displays line numbers (`:set nonumber` turns them off).

To save a file, type `:w`. To save and quit, type `:wq` or press `ZZ`. To force a quit without saving type `:q!`.

To start editing another file, type :e filename.

To execute shell commands from within vi, and then return to vi afterwards, type `:!`shellcommand. You can use the letter `%` as a substitute for the name of the file that you are editing (so `:!echo %` prints the name of the current file).

`.` repeats the last command.

4.6. Quick Reference to vi

Inserting and typing text:

```
i           insert text (and enter input mode)
$a          append text (to end of line)
ESC         re-enter command mode
J           join lines
```

Cursor movement:

```
h           left
j           down
k           up
l           right
^           beginning of line
$           end of line
1 G         top of document
```

```
G           end of document
<n> G       go to line <n>
^F          page forward
^B          page backward
w           word forwards
b           word backwards
```

Deleting and moving text:

```
Backspace   delete character before cursor
            (only works in insert mode)
x           delete character under cursor
dw          delete word
dd          delete line (restore with p or P)
<n> dd      delete n lines
d$          delete to end of line
dG          delete to end of file
yy          yank/copy line (restore with p or P)
<n> yy      yank/copy <n> lines
```

Search and replace:

```
%s/<search string>/<replace string>/g
```

Miscellaneous:

```
u           undo
:w          save file
:wq         save file and quit
ZZ          save file and quit
:q!         quit without saving
```

5.1. The Superuser root

The superuser is a privileged user who has unrestricted access to all commands and files on a system regardless of their permissions. The superuser's login is usually `root`. Access to the root account is restricted by a password (the root password). Because the root account has huge potential for destruction, the root password should be chosen carefully, only given to those who need it, and changed regularly.

One way to become `root` is to log in as usual using the username `root` and the root password (usually security measures are in place so that this is only possible if you are using a "secure" console and not connecting over a network). Using `root` as your default login in this way is not recommended, however, because normal safeguards that apply to other user accounts do not apply to `root`. Consequently using `root` for mundane tasks often results in a memory lapse or misplaced keystrokes having catastrophic effects (e.g. forgetting for a moment which directory you are in and accidentally deleting another user's files, or accidentally typing "`rm -rf * .txt`" instead of "`rm -rf *.txt`").

A better way to become root is to use the `su` utility. `su` (switch user) lets you become another user (at least as far as the computer is concerned). If you don't specify the name of the user you wish to become, the system will assume you want to become `root`. Using `su` does not usually change your current directory, unless you specify a "`-`" option which will run the target user's startup scripts and change into their home directory (provided you can supply the right password of course). So:

```
$ su -
```

```
Password: xxxxxxxx
#
```

Note that the root account often displays a different prompt (usually a #). To return to your old self, simply type "exit" at the shell prompt.

You should avoid leaving a root window open while you are not at your machine. Consider this paragraph from a humorous 1986 Computer Language article by Alan Filipski:

"The prudent administrator should be aware of common techniques used to breach UNIX security. The most widely known and practised attack on the security of the UNIX brand operating system is elegant in its simplicity. The perpetrator simply hangs around the system console until the operator leaves to get a drink or go to the bathroom. The intruder lunges for the console and types `rm -rf /` before anyone can pry his or her hands off the keyboard. Amateur efforts are characterised by typing in things such as `ls` or `pwd`. A skilled UNIX brand operating system security expert would laugh at such attempts."

5.2. Shutdown and System Startup

Shutdown: `shutdown`, `halt`, `reboot` (in `/sbin`)

`/sbin/shutdown` allows a UNIX system to shut down gracefully and securely. All logged-in users are notified that the system is going down, and new logins are blocked. It is possible to shut the system down immediately or after a specified delay and to specify what should happen after the system has been shut down:

```
# /sbin/shutdown -r now     (shut down now and reboot)
# /sbin/shutdown -h +5      (shut down in 5 minutes & halt)
# /sbin/shutdown -k 17:00   (fake a shutdown at 5pm)
```

`halt` and `reboot` are equivalent to `shutdown -h` and `shutdown -r` respectively.

If you have to shut a system down extremely urgently or for some reason cannot use `shutdown`, it is at least a good idea to first run the command:

```
# sync
```

which forces the state of the file system to be brought up to date.

System startup:

At system startup, the operating system performs various low-level tasks, such as initialising the memory system, loading up device drivers to communicate with hardware devices, mounting filesystems and creating the `init` process (the parent of all processes). `init`'s primary responsibility is to start up the system services as specified in `/etc/inittab`. Typically these services include gettys (i.e. virtual terminals where users can login), and the scripts in the directory `/etc/rc.d/init.d` which usually spawn high-level daemons such as `httpd` (the web server). On most UNIX systems you can type `dmesg` to see system startup messages, or look in `/var/log/messages`.

If a mounted filesystem is not "clean" (e.g. the machine was turned off without shutting down properly), a system utility `fsck` is automatically run to repair it. Automatic running can only fix certain errors, however, and you may have to run it manually:

```
# fsck filesys
```

where filesys is the name of a device (e.g. `/dev/hda1`) or a mount point (like `/`). "Lost" files recovered during this process end up in the `lost+found` directory. Some more modern filesystems called "journaling" file systems don't require `fsck`, since they keep extensive logs of filesystem events and are able to recover in a similar way to a transactional database.

5.3. Shells and Shell Scripts

A shell is a program which reads and executes commands for the user. Shells also usually provide features such job control, input and output redirection and a command language for writing shell scripts. A shell script is simply an ordinary text file containing a series of commands in a shell command language (just like a "batch file" under MS-DOS).

There are many different shells available on UNIX systems (e.g. `sh`, `bash`, `csh`, `ksh`, `tcsh` etc.), and they each support a different command language. Here we will discuss the command language for the Bourne shell `sh` since it is available on almost all UNIX systems (and is also supported under `bash` and `ksh`).

5.4. Shell Variables and Environment

A shell lets you define variables (like most programming languages). A variable is a piece of data that is given a name. Once you have assigned a value to a variable, you access its value by prepending a `$` to the name:

```
$ bob='hello world'
$ echo $bob
hello world
$
```

Variables created within a shell are local to that shell, so only that shell can access them. The `set` command will show you a list of all variables currently defined in a shell. If you wish a variable to be accessible to commands outside the shell, you can export it into the environment:

```
$ export bob
```

(under `csh` you used `setenv`). The environment is the set of variables that are made available to commands (including shells) when they are executed. UNIX commands and programs can read the values of environment variables, and adjust their behaviour accordingly. For example, the environment variable `PAGER` is used by the `man` command (and others) to see what command should be used to display multiple pages. If you say:

```
$ export PAGER=cat
```

and then try the `man` command (say `man pwd`), the page will go flying past without stopping. If you now say:

```
$ export PAGER=more
```

normal service should be resumed (since now `more` will be used to display the pages one at a time). Another environment variable that is commonly used is the `EDITOR` variable which specifies the default editor to use (so you can set this to `vi` or `emacs` or which ever other editor you prefer). To find out which environment variables are used by a particular command, consult the `man` pages for that command.

Another interesting environment variable is `PS1`, the main shell prompt string which you can use to create your own custom prompt. For example:

```
$ export PS1="(\h) \w> "
(lumberjack) ~>
```

The shell often incorporates efficient mechanisms for specifying common parts of the shell prompt (e.g. in `bash` you can use `\h` for the current host, `\w` for the current working directory, `\d` for the date, `\t` for the time, `\u` for the current user and so on - see the `bash` man page).

Another important environment variable is `PATH`. `PATH` is a list of directories that the shell uses to locate executable files for commands. So if the `PATH` is set to:

```
/bin:/usr/bin:/usr/local/bin:.
```

and you typed `ls`, the shell would look for `/bin/ls`, `/usr/bin/ls` etc. Note that the `PATH` contains'.', i.e. the current working directory. This allows you to create a shell script or program and run it as a command from your current directory without having to explicitly say "`./`filename".

Note that `PATH` has nothing to with filenames that are specified as arguments to commands (e.g. `cat myfile.txt` would only look for `./myfile.txt`, not for `/bin/myfile.txt`, `/usr/bin/myfile.txt` etc.)

5.5. Simple Shell Scripting

Consider the following simple shell script, which has been created (using an editor) in a text file called `simple`:

```
#!/bin/sh
# this is a comment
echo "The number of arguments is $#"
echo "The arguments are $*"
echo "The first is $1"
echo "My process number is $$"
echo "Enter a number from the keyboard: "
read number
echo "The number you entered was $number"
```

The shell script begins with the line "`#!/bin/sh`" . Usually "`#`" denotes the start of a comment, but `#!` is a special combination that tells UNIX to use the Bourne shell (`sh`) to interpret this script. The `#!` must be the first two characters of the script. The arguments passed to the script can be accessed through `$1`, `$2`, `$3` etc. `$*` stands for all the arguments, and `$#` for the number of arguments. The process number of the shell executing the script is given by `$$`. the `read` number statement assigns keyboard input to the variable `number`.

To execute this script, we first have to make the file `simple` executable:

```
$ ls -l simple
-rw-r--r--    1 will  finance  175  Dec 13  simple
$ chmod +x simple
$ ls -l simple
-rwxr-xr-x    1 will  finance  175  Dec 13  simple
$ ./simple hello world
The number of arguments is 2
The arguments are hello world
```

```
The first is hello
My process number is 2669
Enter a number from the keyboard:
5
The number you entered was 5
$
```

We can use input and output redirection in the normal way with scripts, so:

```
$ echo 5 | simple hello world
```

would produce similar output but would not pause to read a number from the keyboard.

Notice:

This content is taken from **http://www.doc.ic.ac.uk/~wjk/UnixIntro/** and it should be translated to Croatian and completed into UNIX/Linux textbook.