

Main Features of UNIX

This section contains a brief overview of the main features of UNIX. At the time UNIX was introduced, some of these features set the UNIX operating system apart from other systems available at that time. Today, many of these features are common place.

- multi-user
more than one user can use the machine at a time
supported via terminals (serial or network connection)
- multi-tasking, more than one program can be run at a time
- hierarchical directory structure, to support the organisation and maintenance of files
- portability
only the kernel (<10%) written in assembler. This meant the operating system could be easily converted to run on different hardware
- tools for program development, a wide range of support tools (debuggers, compilers)

The basic structure of the UNIX operating system as a division of three parts.

Kernel

- schedules programs
manages data/file access and storage
enforces security mechanisms
performs all hardware access

shell

- presents each user with a prompt
interprets commands types by a user
executes user commands
supports a custom environment for each user

utilities

- file management (rm, cat, ls, rmdir, mkdir)
user management (passwd, chmod, chgrp)
process management (kill, ps)
printing (lp, troff, pr)
program development tools

Multi-User Operating Systems

A multi-user operating system allows more than one user to share the same computer system at the same time.

It does this by time-slicing the computer processor at regular intervals between the various users.



The switching between user programs is done by part of the kernel. To switch from one program to another requires,

- a regular timed interrupt event
- saving the interrupted programs state and data
- restoring the next programs state and data
- running that program till the next timed interrupt occurs

The timed event is usually about 1 to 10 milliseconds apart. It is generated by a real-time clock.

Handling Programs

Each computer has a maximum amount of memory (RAM) that is installed in the computer. Some of this memory is required by the operating system. The remainder is available to user programs.

The more memory that can be provided in total the better. Where there is insufficient main memory to run a users program, some other users program residing in main memory must be written out to the disk unit to create some free memory space.

This process is called **swapping**. When the system becomes overloaded (where there are more users than the system can handle), the operating system spends most of its time shuttling users programs between main memory and the disk unit, and users response time degrades. This is called **disk thrashing** and is overcome by installing more main memory.

Processes

Each program running on a UNIX system is called a process. When a user types a command, UNIX constructs a Process Control Block (PCB) for the process that process. Each process has a PCB that holds its priority, the process state, register information and additional details.

UNIX provides a set of utilities for managing processes.

ps	list processes
kill	kill a process
&	run a process in the background

If the system administrator found that a particular user was performing an operation that was consuming too much computing time or dominating a system resource such as a printer, they could use the **ps** command to identify the offending users process and then use the **kill** command to terminate that process.

Each program is assigned a **priority level**. Higher priority tasks (like reading and writing to the disk) are performed more regularly. User programs may have their priority adjusted dynamically, upwards or downwards, depending upon their activity and available system resources.

Multi-tasking systems support **foreground** and **background** tasks. A foreground task is one that the user interacts directly with using the keyboard and screen. A background task is one that runs in the background (it does not have access to the screen or keyboard). Background tasks are usually used for printing.

Running Programs In The Background

- lets the user carry on with more important tasks
- examples are printing and formatting documents
- the ampersand symbol (&) is appended to the command
- the shell assigns a process number (pid) to the command
- background jobs can be deleted using the kill command

```
$ sort -n catalog > /dev/lp &
3786
$
```

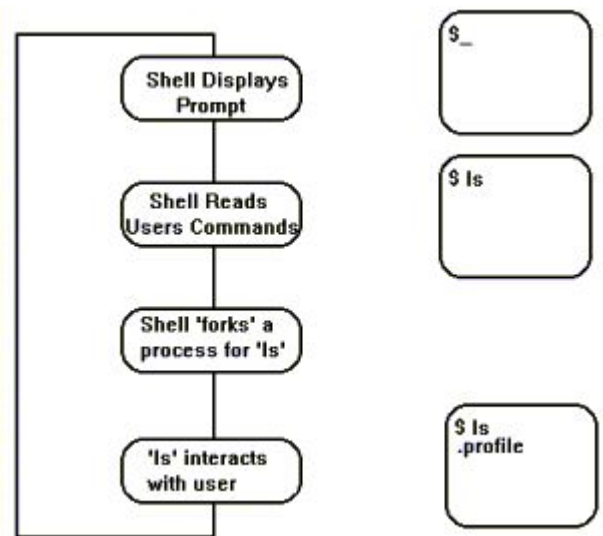
When a user logs on

When the UNIX system starts up, it also starts a system process (getty) which monitors the state of each terminal input line. When getty detects a user has turned their terminal on, it presents the login prompt and once the password is validated, the UNIX system associates the shell program (/bin/sh) with that terminal.

Each user is presented with a shell. This is a program which displays the users prompt, handles user input and displays output on the terminal.

The shell program provides a mechanism for customising each users setup requirements, and storing this information for re-use (in the file .profile).

The user interacts with /bin/sh, which interprets each command typed. Internal commands are handled within the shell (set, unset), external commands are invoked as programs (ls, grep, sort, ps).



There are a number of different command line shells (user interfaces).

- Bourne (sh)
- Korn (ksh)
- C shell (csh)
- Bash (an improved Bourne shell with history and aliases)

The shell is often called a command line shell, since it presents a single prompt for the user. The user types a command, the shell invokes that command, then presents the prompt again when the command has finished. This is done on a line by line basis, hence the term 'command line'.

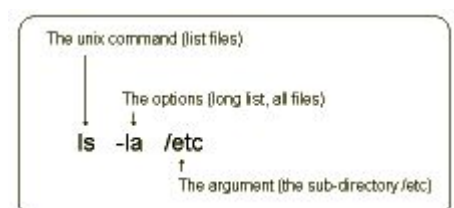
Recent enhancements turn the command line interface into a graphical one (X, MOTIF, OPENVIEW), where programs are represented as objects or icons on a screen. By use of a mouse these icons are selected and run. This is similar to operating systems like Windows or OS/2.

Unix Command Syntax

This section

outlines the standard format for all UNIX commands.

Commands are short two or three character names, and accept additional options that control their actions.



- all commands have a similar format

- commands are generally two or three characters long
- commands are case sensitive (use lowercase)
- options always precede filenames
- options are prefixed using a - symbol
- the **man** command can be used to display the correct syntax

The UNIX File System

This section discusses UNIX files and directories. The file system refers to the way in which UNIX implements files and directories. In UNIX, a file system has the following features,

- hierarchical structure (support for directories)
- files are expandable (may grow as required)
- files are treated as byte streams (can contain any characters)
- security rights are associated with files and directories (read/write/execute privilege for owner/group/others)
- files may be shared (concurrent access)
- hardware devices are treated just like files

The File

A file is a collection of information, which can be data, an application, documents; in fact, under UNIX, a file can contain anything. When a file is created, UNIX assigns the file a unique internal number (called an inode).

File Links

A file link is a directory entry which points to an original file somewhere else. A link is made to an existing file using the **ln** command. This creates a directory entry which points to the existing file (it does not make a copy of the existing file). This allows more than one reference to an existing file.

For instance, a person can give another access to a file and let them create a link to it. In this way they both can access and work with the same file, ensuring that the information they enter into the file is up-to-date. Only the original owner of the file may delete the file.

The security rights associated with files and directories

UNIX provides three sets of security rights for each file and directory entry. These sets are for the owner, the group to the owner belongs, and any other users on the system.

Each user belongs to a group (only one at a time). Group membership facilitates the sharing of common files. A user can change their membership to another group by using the **newgrp** command.

The security rights are

- read (read, display, copy the file contents)
- write (modify and append to the file contents)
- execute (run the file as a program)

The security bits are grouped as a series of three bits for each of the owner, group and other access rights.

owner	group	others
rwx	rw-	---

The owner can read write and execute the file
 Group members can read and write the file
 Other users have no access

```

$ls -l
total 1
-rwx----- 1 joe bc1a 8 oct 1 20:10 .profile
$chmod 770 .profile
$ls -l
total 1
-rwxrwx--- 1 joe bc1a 8 oct 1 20:10 .profile
$chgrp bc2 .profile
-rwxrwx--- 1 joe bc2 8 oct 1 20:10 .profile
$chown sam .profile
-rwxrwx--- 1 sam bc2 8 oct 1 20:10 .profile
$chmod +x .profile
-rwxrwx--x 1 sam bc2 8 oct 1 20:10 .profile
$chmod -w .profile
-r-xr-x--x 1 sam bc2 8 oct 1 20:10 .profile
$

```

The group to which the file belongs is changed using the **chgrp** command.

The owner of the file is changed using the **chown** command. Security rights for a file or directory are modified by using the **chmod** utility.

Listing files

Files are listed using the **ls** command.

This picture shows the security rights associated with the file, as well as the owner (joe), the group that the owner belongs to, the size of the file and other information.

```

$ls -l
total 1
-rwx----- 1 joe bc1a 8 oct 1 20:10 .profile

```

Diagram illustrating the components of the `ls -l` output for the file `.profile`:

- `-rwx-----`: security bits
- `1`: number of links
- `joe`: owner of the file
- `bc1a`: group membership
- `8`: file size in bytes
- `oct 1 20:10`: time of last modification
- `.profile`: filename

file type(directory = d, file = -)

Wild Card Characters

Wild card characters are used when working with a number of files at once.

- used to match characters in filenames
- the asterisk `*` matches any sequence of zero or more characters
- the question mark `?` matches exactly one character

```

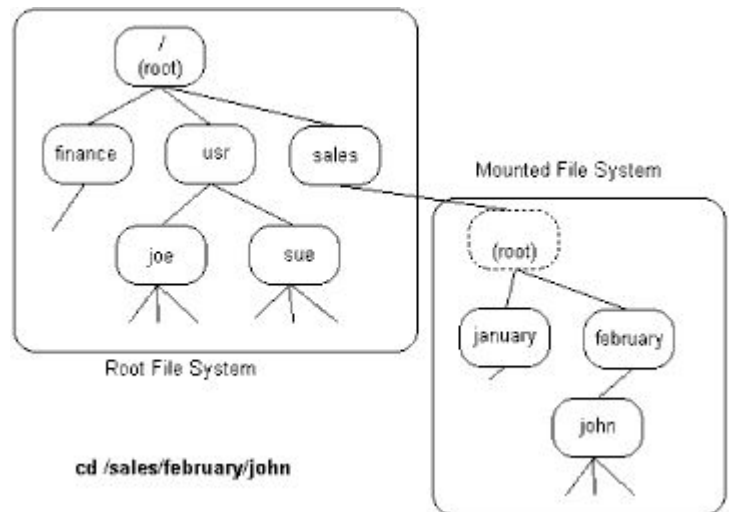
$ls
data1
data10
data12
data2
data3
$rm data1?
$ls
data1
data2
data3
$rm *
$ls
$

```

Mountable File Systems

All UNIX systems have at least one permanent non-removable hard disk system. The root directory and the directories below it are stored on this disk. Its structure is known as the root file system.

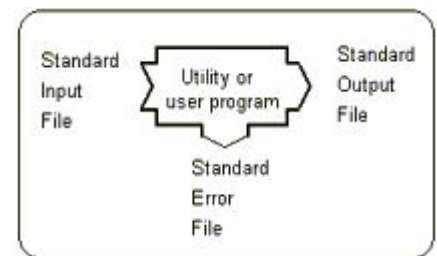
If an additional hard disk is added, UNIX creates a separate new filesystem on this disk. Before it can be used, it must be attached to the root file system. This is called mounting an additional filesystem.



An empty directory on the root file system is selected as the mount point, and using the UNIX mount command, the new file system will appear under this directory.

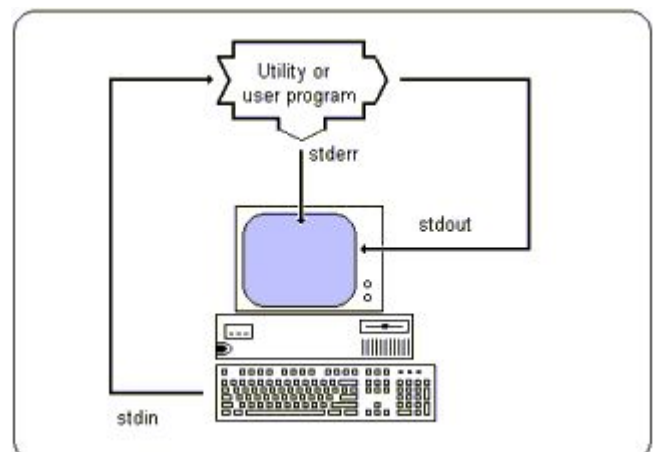
UNIX Standard Devices

There are THREE standard devices supported by the UNIX shell, stdin, stdout and stderr. Each program that runs has allocated to it a stdin, stdout and stderr device.



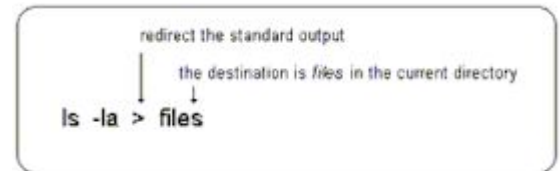
- these devices may be redirected to another device or file
- standard output (stdout) is associated with the users terminal display
- standard input (stdin) is associated with the users terminal keyboard
- standard error (stderr) is associated with the users terminal display

When a program is executed, it has associated with it each of three standard UNIX devices.

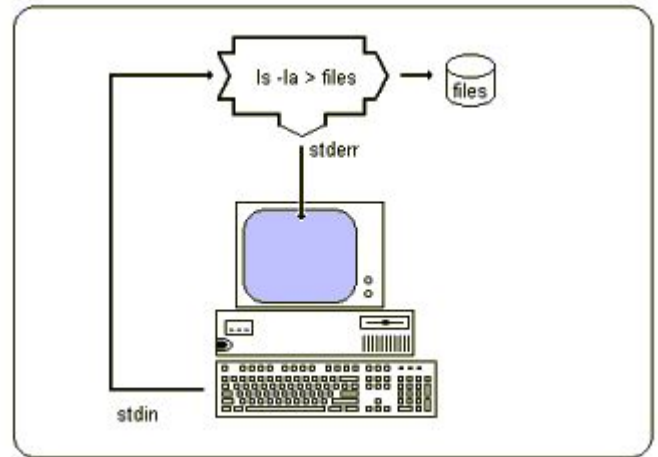


Device Redirection

The input or output devices associated with a program may be redirected to another device.



For example, a program that normally reads from stdin (the keyboard) can be redirected to read from a file. In a similar manner, a program that normally writes to stdout (the screen) can redirect its output to a file or printer. In this example, the `ls` command redirects stdout to **files**.



- the standard devices may be redirected to a device or file
- other devices can be printers, terminals
- devices appear as files in the `/dev` subdirectory
- the symbol `>` changes the standard output device, creating a new file
- the symbol `<` changes the standard input device
- `>>` redirects stdout, appending to an existing file

Shell Device Redirection

It is also possible to redirect any device using the file descriptor number assigned to that device.

By default, the `c` compiler sends the error output to the screen. To redirect the error output to a file, it is necessary to use the file descriptor.

Example of device redirection

compile the program `test.c` and redirect stderr to the file `'errors.log'`

```
cc test.c 2>> errors.log
```

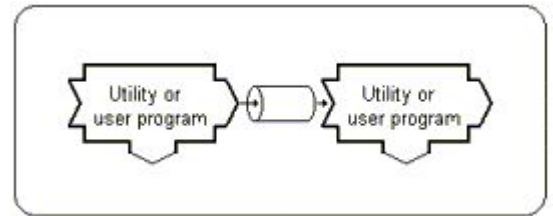
- stdin has a file descriptor of 0
- stdout has a file descriptor of 1
- stderr has a file descriptor of 2

Pipes

Sometimes, the use of intermediary files is undesirable. Consider an example where you are required to generate a report, but this involves a number of steps. First you have to concatenate all the log files. The next step after that is strip out comments, and then to sort the information. Once it is sorted, the results must be printed.

A typical approach is to do each operation as a separate step, writing the results to a file that then becomes an input to the next step.

Pipelining allows you to connect two programs together so that the output of one program becomes the input to the next program.



- allow commands to be combined in a sequential order
- connects stdout of one program to the stdin of the next program
- the symbol | (vertical bar) represents a pipe
- any number of commands can be connected in sequence, forming a pipeline
- all programs in a pipeline execute at the same time
- complex operations are easily performed by piping commands

Shell Scripts

To automate many routine tasks, a system administrator will create a file that contains the commands to be executed. This is known as a shell script file.

This is an example of a shell script file.

It is a file called **testlogin** that accepts a username and then tests to see if that user is logged on.

```
$ cat testlogin
#testlogin
useron(){
    if ( who | grep $1 > /dev/null )
    then echo $1 is logged in
    else echo $1 is not logged in
    fi
}
if test $# != 1
then echo testlogin: username
else useron
fi
$testlogin joe
joe is logged in
$
```

Shell scripts

- contain command sequences
- have execute permission (using chmod)
- simplify repetitious command sequences
- are run by the shell as if it has been typed at the terminal

Create A Shell Script

Shell scripts must be in ASCII text format, and can be created using a simple editor such as vi.

This example uses the cat command (and redirect output) to create the script file that echos the current date when executed.

```
$ cat -> sample
echo The date is
date
$
```

indicates stdin
↓
redirect output to

} entered by user
ctrl-d terminates

Set 'execute' And Run A Shell Script

Once an administrator creates a shell script, its rights must be changed to executable before it can be run.

The administrator used the **chmod** command to alter permission settings for files.

Once the script file has execute rights, the shell script is run by typing its name.

```
$ chmod +x sample
$ sample
The date is
Thu Nov 19 13:40:38 NZT 1992
$
```

Shell Variables

When working with shell scripts, it is often desirable to use variables. These can be either user-defined or in-built variables that the shell supports.

Commonly used shell variables are,

HOME	pathname of users home directory
PATH	search path used to find programs
PS1	the shell prompt
TERM	the type of terminal being used

- the shell supports internal variables
- application and user defined variables are also supported
- the set command displays all currently defined shell variables
- variables are used to find programs (path) or set terminal types

Defining Shell Variables

Shell variables are created by assigning a string (sequence of characters) to a variable name. Variables can also contain valid UNIX commands.

To access the value of a shell variable, the variable is prefixed with the \$ symbol.

```
$ tmp=/usr/tmp
$ cd $tmp
$ pwd
/usr/tmp
$ PS1=`pwd`
/usr/u1a/joe echo $TERM
vt100
/usr/u1a/joe PS1=$
$
```

Shell Programming Language

Commands can be combined in shell scripts using a mini programming language

supported by the shell.

Sample shell script

```
#!/list all files in the root directory
for files in /*
do
    echo $files
done
```

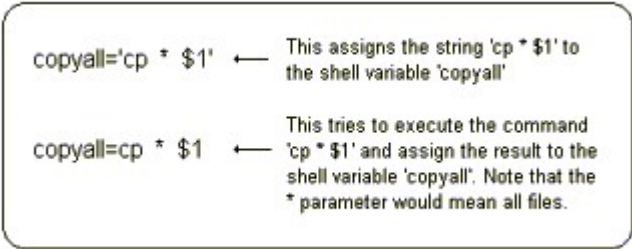
Support for repetitive tasks (for and while statements) and conditions (if and select statements) allow system administrators to write scripts that are reasonably powerful.

- simple statements which when combined, create powerful scripts
- support for conditional execution and conditional repetition
- command substitution (the output of a command can be fed back into the script for execution)
- support for shell variables

Shell Arguments and Quotes

When using arguments to shell scripts or variables, the system administrator must be careful to ensure that the resultant action is valid. Arguments are often interpreted by the shell in a manner that is unintentional.

This is due to the shell applying certain rules about special characters, so system administrators should exercise care when using these in arguments to scripts and variables.



```
copyall='cp * $1' ← This assigns the string 'cp * $1' to the shell variable 'copyall'
```

```
copyall=cp * $1 ← This tries to execute the command 'cp * $1' and assign the result to the shell variable 'copyall'. Note that the * parameter would mean all files.
```

- the shell supports pattern matching and recognition
- the asterisk (*) symbol matches all strings
- the question mark (?) symbol matches a single character
- the special symbols < > * ? [] have special meaning
- to use special symbols, they are enclosed in single or double quotes, else the shell will try to execute them as commands or use them as parameters to commands
- within double quotes, the symbols \$ \ ` " retain their special meaning. To use these characters inside a double quoted string, precede these symbols with a backslash (\) symbol