Introduction to Linux

The following practicals have not been tested on Ubuntu, so some may not work as described.

The Linux Prompt

In Linux the backslash escapes the newline-character. By escaping the newline-character the operating system knows the current command is a multi-line command.

<u>Example 3</u>

```
$ echo This is the first \
        > of a multi-line command.

        This is the first of a multi-line command.
```

Customising the Prompt

Linix allows you to customise the way its prompts are displayed. Escape sequences can be used to customise the prompt. Note that not all escape sequences may work.

Sequence	Output
\d	date
\H	hostname
\h	hostname up to first .
\n	carriage return line feed
\t	time in HH:MM:SS format
\u	username
\w	current directory
\W	basename of current directory
\!	history number

Profile

Find the profile file (probably in /etc). Try to work out what some of them control.

Alter the prompt:

```
PS1= "\h \d \t yourname > "
```

To start an application type its name at the command prompt e.g. gedit

Try to use the command line whilst the window is open. Close gedit.

Ways to customise your environment

Environment variables

- Typically have upper case names
- Can be seen by running processes you own.
- Are usually set in your .login file.

Type *setenv* to view them all.

Some useful Shell variables

path	a list of directories in which to search for commands. By default the current directory is searched. Usually something like: *path=/bin:/usr/bin:/user1/bin::*
prompt	the prompt. Can be changed, e.g. *set ps1 = $*
user	your username
argv	contains list of command line arguments supplied at invocation of the shell.
history	when set to a number, stores that number of previous commands. e.g. *set history = 20*
home	contains the location of your home directory
ignoreeof	disable CTRL/D to exit a shell. Type exit instead. Just needs to exist. e.g *set ignoreeof*
noclobber	just needs top be set (to anything). Prevents accidental overwriting of files with > and >>. Will tell you if file exists. E.g. *set noclobber*

Aliases

- Are local to your shell
- Usually set in your *.cshrc* file, which is read by the shell every time it starts.
- Aliases set up at the command prompt are lost when that shell is exited.

Type *alias* to view them all

Safety aliases
alias cp cp –i
alias rm rm -i

alias dir=ls
alias l=’ls –ltr’ – alias up=’cd..’
unalias removes named alias

Password file

- Everybody has read access to the password file.
- All you can change is the encrypted password.

/etc/passwd

A typical entry:

mread : k8q23 : 101 : 15 : Martin Read – staff: /home/staff/martin:/bin/csh

this consists of the user id : encrypted password : user no. : group no. : comment : home directory : login programme

The login programme

- Is usually a shell
- When the login programme terminates, the user is logged out.
- If the system administrator wishes a user to execute one application only, then the name of the login programme can be set to the name of that application. When the application finishes, the user is logged out. They have access to no other parts of the system.

The utility *ypcat* knows where to find the password file.

ypcat passwd ¦ *grep* yourname

ypcat passwd ¦ wc

will tell you how many registered users are logged on. Read about *wc* in the manual.

Logging out

Type logout or CTRL+D

The .login File
Many user specific commands can be executed by putting them in a file which is usually called .login, which is read upon logging in. Each different shell has its different .login file. For example, the tcsh shelll uses a .tcshrc file.

The sh shell uses a file called .profile which is read when a user logs on. When bash is run as an interactive login shell, it first reads and executes commands from the file /etc/profile (if that file exists). After reading that file, it reads and executes the first of for ~/.bash profile, ~/.bash login, and /.profile. When an interactive shell that is not a login shell is started, bash reads and executes commands from ~/.bashrc, if that file exists.

Using the Login File
If you put the following lines in your .bashrc file you can automatically customise your primary and secondary prompt if bash is your shell.

Example 4
PS1=" \! \h \w\n$ "
PS2="$ "
export PS1
export PS2

The export statements are to ensure that in every sub-shell (sub-process or programme) within the current shell the value of the variables will be set to the last value that was assigned to them

Useful commands:

finger - Lists information about users
http://www.computerhope.com/unix/ufinger.htm

whois - Internet user name directory service
http://www.computerhope.com/unix/uwhois.htm

who - Displays who is on the system
http://www.computerhope.com/unix/uwho.htm

http://www.computerhope.com/unix.htm

Input and Output

At a very low level input/output is from/to channels. Every command has access to 12 channels. Commands are free to use channel 3{9 as they like.

- Channel 0 is called stdin. It is reserved for the standard input of a command;
- Channel 1 is called stdout. It is reserved for the standard output of a command;
- Channel 2 is called stderr. It is reserved for the standard errors of a command.

Good commands should output questions posed to user onto stderr.

Output and Input Redirection

Two extremely useful concepts are output redirection and input redirection. They are as common as are pipelines. Output redirection lets you redirect from a certain channel to a file or another channel. Input redirection lets you put the contents from a file or a channel onto a certain channel.

Redirection Operators
Construct Semantics
cmd < file cmd gets standard input from file
cmd > file send standard output from cmd to file
cmd 1> file send standard output from cmd to file
cmd 2> file send standard errors from cmd to file
cmd >> file append standard output from cmd to file
cmd 1>> file append standard output from cmd to file
cmd 2>> file append standard errors from cmd to file

Examples

```
$ cat file
        A line of text.
$ echo hello > file
$ cat file
        hello
$

$ ls file nofile
        ls: nofile: no such file or directory file
$ ls file nofile 2> errors
        file
$ cat errors
        ls: nofile: no such file or directory
$
```

Construct Semantics

cmd <&m cmd gets standard input from channel m
cmd <<DELIMITER here document
cmd 2>&1 put standard errors from cmd on stout
cmd >&m put standard output from cmd on channel m

```
$ echo b > file
$ echo a >> file
$ sort < file
        a
        b
$ sort < file > sortedfile
$ cat sortedfile
        a
        b
$

$ ls file nofile
        ls: file: no such file or directory
        ls: nofile: no such file or directory
$ >file
$ ls file nofile
        s: nofile: no such file or directory file
$ ls file nofile 2>&1 > alloutput
$ cat alloutput
        ls: nofile: no such file or directory file
$
```

Practical II

The find Command

Find is very useful if you have forgotten about the location of a file, to find all files created before/after the modification time of another file (useful for backups) or want to run a command for certain files in a certain directory.

The default syntax for find is:

```
find directory -options [-print -exec -ok]
```

Examples. Try to work out what they mean:

```
find mystuff -name appointments

find ~ -name "*.txt"

find / -name "[a-z]*"

find / -user root

find / -group root
```

We have covered permission bits. Find all the files in the current directory whose permission bits (-perm) have the octal value 740 (what does this mean?).

How would you find all the files in the current directory whose permission bits have the octal value 744 and whose names start with a B?

Other Options

-print tells find to print the name of the file (default).

The following two find commands do exactly the same:

```
find ~ -name "*.txt"
find ~ -name "*.txt" -print
```

-exec lets you apply commands to the files that are found.
-ok is the same as -exec but asks if you're sure you want to apply the command.

Apply the command ls –l to each of these files:

```
find ~ -name "*.txt" -exec ls -l fg \;
```

Filename substitution using metacharacters

Filename substitution transforms shell commands by replacing specifications of filenames to existing filenames. The sequence ‘a*’ means any sequence of characters which starts with an ‘a’ followed by zero or more other characters.

Given this list of filenames (a aa ab ba b bb bc ccc) what will the following commands return:

```
ls
ls a*
ls b*
ls *b
ls *c*
```

The symbol ‘?’ will match any single character. What will the following return?

```
ls ?
ls ?a
ls a?a
ls c?c
```

Using [...]
The sequence [character sequence] will match any character in the sequence enclosed by the square brackets (assuming the first character differs from ! or (if your shell supports it) ^).

e.g. [a-z] indicates the characters in the range a to z and [axc-fz] is equivalent to [axcdefz].

```
ls [ab]
ls b[ac]
ls [ab]*
```

The construct [!chars] will match any character which is not in the sequence enclosed by the square brackets.

```
ls [!ac]*
ls ?[!ab]*
ls *[!a]
```

Run the *man* command on *find*. Study the -newer option of find.

Use the *mkdir* command (make directory) to create a directory called tmp in you home directory. Use *cd* (change directory) to make it your current directory. Create each of the following files: a, b, c, aa, ab, ba, cc, bab. Try selected the files using ls with the following?

```
a*
a?
*a
?a
?
??
[a-z]*
[!a-z]*
```

The find Command - answers

Find all files called appointments in the directory mystuff.

```
find todo -name appointments
```

Find all files whose names ends in .txt in your home directory.

```
find ~ -name "*.txt"
```

Find all files whose names start with a lower case letter in the root directory (the / directory).

```
find / -name "[a-z]*"
```

Find all files in the root directory belonging to root (the super-user) as follows:

```
find / -user root
```

Find all files in the root directory belonging to the group root as follows:

```
find / -group root
```

Find all the files in the current directory (called .) whose permission bits have the octal value 740.

```
find . -perm 740
```

Find all the files in the current directory whose permission bits have the octal value 744 and whose names start with a B.

```
find . -name "B*" -perm 744
```

Practical III

Processes and programmes

ps – process status. Gives information on one or all processes running.

Look up the options in the manual.

```
ps –l
```

(F process flags, uid process owner, PID – process number, PPID – process number of parent, PRI – maximum computer time assigned to the process in milliseconds, SIZE - virtual memory used by the programme, RSS – programme size in main memory, WCHAN – even the process is waiting for, STAT – process status R for running, S for sleeping, TTY – terminal process started from, TIME – the computer time used)

top – reads the proc file system at regular intervals and gives information on current system data and status of processes. Can also be used to send signals to processes.

The lines at the top refer to system data:
Uptime – shows the current time, time passed last system restart, number of active users, average CPU load in the last 1, 5 & 15 minutes. Can also be given by the commands *uptime* and *w*.

Processes – shows all the processes on the system. Idle, executable, scheduled and halted.

CPU states – the current computer performance by users and the system.

Memory – current memory load. Available main memory and current usage – used, free, shared and buffer. Can also be given by the command *free*.

Swap – statistics for the swap area (if present). Can also be given by the command *free*.

Press *q* to quit.

time – gives information on the system resources used by a programme over its' running time e.g. execution time, operating system kernel time for system calls, etc.

There are 2 versions. The command interpreter for shell tcsh contains a time function and the directory /usr/bin has a time version.

The version of time in tcsh called without options provides information on the computer time used by the current shell.

/usr/bin/time always has to be used with a further command:

```
time command
```

Output the time used, time taken for processing the programme code and system calls and percentage of CPU load given to the programme. Also gives average size of the memory used (divided, undivided, resident) and number of accesses to the file system, number of accesses to swapped pages. The number of swaps executed is always 0.

Creating processes

To create a new process, the operating system kernel uses the system call fork(), which creates a child process as a copy of the parent process, and then uses exec() to overlay the code segment of the child process with the programme code of the new programme.

Try the following C programme. Compile it using cc (or similar compiler e.g. gcc) and run it.

```
#include <stdio.h>
void main( )
{
int pid;
        /* fork another process */
        pid = fork( );
        if (pid == 0) { /* child process */
                printf(“This is from child process\n”);
                execlp(“/bin/ls”, “ls”, NULL);
        } else { /* parent process */
                printf(“This is from parent process\n”);
                wait(NULL);
                printf(“Child process completes”);
        }
        printf(“process ends %d\n”, pid);
}
```

Locate
There is a faster way to find files than using the find command (which looks in every directory & subdirectory in turn to see if the files you want are there). This is by using the locate command.

This searches an index to rapidly locate files. Locate searches partial filenames by default, so wildcards are unnecessary.

Locate all files containing the word ‘file’:

```
locate file
```

To generate the index:

```
updatedb
```

this will take a while to build the database. You will need to be logged on as root or superuser otherwise there are some directories you cannot scan due to permissions.

Open a shell as root user and create the index.

Grep

General **R**egular **E**xpression **P**arser is a useful Linux command. It is a utility for finding strings in files. You can run it directly on a file or use it as a filter in a pipeline (more on pipelines in another practical).

```
grep “this” myfile.txt
```

-c prints a count of matching lines
-f only shows matching lines – not filenames
-l only show filenames – not matching lines
-I ignore case
-n print line numbers
-v find lines which don’t match
-w only match whole words
-x only match whole lines
-q no output – for use in shell scripts. Exit status is 0 if any matches found, otherwise 1.

```
$ grep is <<EOF
> A horse is a horse
> Of course of course
> And no one can talk to a horse,
> Of course
> That is, of course
> Unless the horse
> Is the famous Mister Ed!
> EOF
        A horse is a horse
        That is, of course
$
```

Special characters
Some commands understand a further set of special regular expression characters. These commands are *egrep* & *gawk*.

Egrep is extended grep. Grep can also be used with –e option to do the same thing.

Gawk is the GNU/Linux version of the Unix tool “awk” which is a scripting language used for text file processing. See online documentation.

+ match one or more
? match zero or one
| means “or”, as in match A or B
(and) used to group expressions together

e.g.
Standard [a-zA-Z] extended [a-z] | [A-Z]
Standard [A-Z][a-zA-Z]* means a word which *must* begin with a capital letter
Extended [A-Z]?[a-zA-Z]+ means a word which *may* begin with a capital letter.

More on Processes & programmes
Every background programme you start is called a *job*, which can consist of several *processes*.

```
ls –l | pr –h “Myfiles” | lp &
```

Consists of 3 different programmes, and 3 co-operating processes, it is all one job. The *jobs* command will tell you which jobs are active. The current job has a + sign in front.

If you start a job in the foreground and want to make it a background process, press CTRL+Z (to pause the job) followed by *bg*. You could pause the job, run another programme, then type *fg* to continue the original job. Jobs can be stopped using the kill command (see below) using the job identifier from *jobs*.

Try the above by starting an emacs editor in the foreground. Pause the job, run a listing, then start the job again in the foreground. Pause the job and run it in the background.

More on process status
There are several versions of *ps* depending on the version of Linux/unix you are running.

The boot process runs a number of processes known as ‘daemons’ to look after the print queue, networking, etc.

```
ps -uax
```

Shows all the processes on the system (or use -e) and the username they are running under. The process with pid of 1 is called init. This process starts all the others. Using the -l option you can see the parent process (ppid) of each process.

To look for the processes being run by a particular user, you can pipeline the output into grep.

```
        ps uax | grep rsmith
```

(could also be done using ps –u rsmith on some systems)

You can produce a sorted output using the ‘-sort’ option.

```
        ps uax –sort “uid, -size, +stime”
```

sort by user ID, then the memory used, then in order of system time.

You can also tell ps more explicitly what to output and how to sort it. The -o flag is for user defined format.

```
        ps -e -o format_output_modifiers

        ps -e -o pid,user,%cpu,%mem,comm --sort user
```

Some ps versions also support AIX format descriptors.

Sometimes you may want to see what processes are doing – e.g. one is taking up too much memory or process time. The command to do this is *top*, because it shows the top processes in terms of CPU usage.

Priorities and the *nice* command
Some tasks require a lot of CPU time which can seriously degrade the performance of the system. You can adjust the percentage of CPU time commands get using the *nice* command.

Each process has a priority which determines how much share of the CPU it gets. You can find the priority of a process in several way.

```
ps -e -o "%p %u %c %n"
ps -e -o pid,user,comm,nice
```

The priority is usually shown as PRI using *top*.

You can change the priority of a process using the *nice* command.

```
nice myprogram&
```

Run myprogram with a lower priority than usual.

The nice command allows you to run a command with a certain nice level.

```
nice -n level command
```

Each process that is carried out has a priority level, usually a number between -20 & 19. Usually, only root is allowed to run processes with a negative level.

You can also change the priority of a running process using the ‘renice’ command. You can only affect your own processes, unless you are the superuser, and users can normally only increase their nice levels.

```
renice -2 –p 301
```
Reduce the priority of process id 301 by 2

```
renice 15 –p 302
```
Change the priority of process id 3023 to 15

Stopping processes
The ‘kill’ command is used to stop a process. This sends the process a ‘SIGTERM’ signal, meaning terminate.

```
kill 1502
```

meaning terminate pid 1502

Some processes will not respond (if they are in an error condition, or are programmed to ignore SIGTERM) or close down slowly. Use -9 option to kill the process immediately

```
kill -9 1502
```

Practical IV

Shell scripts

Shells allow you to create simple (or very complex) "shell scripts" to make your life easier when you are performing highly repetitive tasks, or when you want to do some programming. These scripts contain a sequence of commands, very similar to the way you would use the commands if you typed them from the keyboard interactively.

UNIX shell scripts are similar to DOS "batch" files (.bat extension), but the operating system tools within UNIX are much greater in number and more powerful than those available in DOS.

A shell is any collection of commands stored in a file.

A shell script does not have to be marked executable, but will have to be invoked in a subshell using ksh script

Use chmod +x script to allow direct execution
Script runs the script in your current shell environment

Shell Input and Output

echo
The echo command is used to print things.

```
$ echo "This is a string"
        This is a string
$ echo "This is a string"
        This is a string
$
```

The echo command normally adds a newline after the last character of the string. By putting the two characters \c at the end of the string, you can tell echo that it should not print the newline.

```
$ echo "This is your prompt:\c"
        This is your prompt:$ echo "Ah. I see."
        Ah. I see.
$
```

The following escape sequence causes a beep.

```
$ echo -e "\aWake up!"
        Wake up!
$
```

The -e ag is to enable interpretation of certain special escape sequences like the \a (alarm).

read
The read command allows you to read things.

```
        read var1 ... varn
```

gets one line from standard input. Let w1, . . . , wn be the words that were in that line. The words w1, . . . , wn are assigned to var1, . . . , varm one word at a time (from left to right). If there are more words than variables then the last variable gets assigned the words that are left.

```
$ read TRUTH
        Linux. Probably the best operating system in the world.
$ echo $TRUTH
        Linux. Probably the best operating system in the world.
$ read A B C
        How are you today?
$ echo $A
        How
$ echo $B
        are
$ echo $C
        you today?
$
```

cat
The cat command is similar to echo in that it outputs things. Echo outputs the values of its arguments, cat does not!

There are two ways to use cat:
1. cat file1 ... fileN outputs the contents of the files file1 through file N.
2. cat outputs whatever it reads on its standard input channel (in interactive mode this is usually the keyboard).

```
$ cat file
        A line.
        And another line.
$ cat
        A line.
        A line.
        And another line.
        And another line.
<CTRL-D>
$
```

Commands like cat are called filters. Such commands read their input, do something with it, and output their output.

Other useful filters are sort (to sort things) and grep (to find lines that have certain strings in them).

Most filters have two modes. The filter is used with a list of arguments. The arguments are filenames and the filter takes its input from these files. In the other mode there are no arguments and the filter takes its input from standard input. Filters always output their output on standard output (in interactive mode this is usually the screen).

```
$ sort
          b
          a
          c
<CTRL-D>
          a
          b
          c
$

$ grep word
          blah blah blah
          a word is a word
          a word is a word
          blah blah blah
<CTRL-D>
$
```

The Pipeline Construct
One of the most important constructs, pipelines occur everywhere. A pipeline lets you combine two programmes p1 and p2 such that the output of p1 is used as the input of p2.

```
program1 | program2
```

creates a pipe between program1 and program2. This results in a programme which combines program1 and program2. The resulting programme simultaneously runs program1 and program2 and uses the output from program1 as the input of program2.

```
$ cat file
          3: A line with text.
          1: Another line.
          2: The last line with text.
$ sort file | grep text
          2: The last line with text.
          3: A line with text.
$
```

tee
The tee command is another example of a filter. It is usually run as *tee filename*. tee outputs whatever appears on its standard input (like cat) and saves the input in the file. The tee command is especially useful if you are debugging a programme or if you are installing a large software package and want to save the output just in case an error occurs.

```
$ ls -l
        total 0
        -rw------- 1 dongen 93 0 Nov 6 11:37 a
        -rw------- 1 dongen 93 0 Nov 6 11:37 b

$ ls -l | tee lsoutput
        total 0
        -rw------- 1 dongen 93 0 Nov 6 11:37 a
        -rw------- 1 dongen 93 0 Nov 6 11:37 b

$ cat lsoutput
        total 0
        -rw------- 1 dongen 93 0 Nov 6 11:37 a
        -rw------- 1 dongen 93 0 Nov 6 11:37 b
$
```

Practical V

Carry out the following examples (they may need slight modification).

Exit Status

Every process returns an exit status. The exit status can be used by the calling environment to find out if the command completed successfully. By convention a zero exit status means success. A non-zero exit status means failure.

The exit status returned by the last command is stored in the (read-only) variable ?.

Try the following, you should get the same results:

```
$ ls
$ echo $?
        0
$ ls b
        b: No such file or directory
$ echo $?
        1

$ mkdir b
$ echo $?
        0
$ mkdir b
        mkdir: cannot create directory 'b': Directory exists
$ echo $?
        1
```

The exit status returned by a script is the exit status returned by the last command. Shell scripts can also return an exit statuses with the exit statement. A statement of the form *exit i* will stop the current command and return *i* as its exit status. Try the following:

```
$ echo exit 0 > exitZero
$ echo exit 1 > exitOne
$ chmod +x exitZero exitOne
$ exitZero
$ echo $?
        0
$ exitOne
$ echo $?
        1
```

Conditional Statement
Conditional or if/else statement allows for the execution of commands depending on the exit statuses of commands.

```
if condition
then
        statements1
[else
        statements2]
fi

If ($fred == 6) then
        ------------
        else if (----) then
                -----------------
        else
                --------------
        endif
```

conditions

```
            <= >=
            ==
            && logical and
            ¦¦ logical or
            (single & and ¦ are bit operators)
            -n str - not null
            -z str - is null (zero)
```

Example
create a file ‘a’.

```
$ ls
        a
$ if ls a; then
        >   echo a exists
        > else
        >   echo a does not exist
        > fi
a
a exists
```

Example 2

```
$ ls
        a
$ if ls c; then
        >   echo c exists
        > else
        >   echo c does not exist
        > fi
        c: No such file or directory
        c does not exist
```

Example 3

```
$ ls
        a
$ if ls c 1> /dev/null 2>&1; then
        >   echo c exists
        > else
        >   echo c does not exist
        > fi
        c does not exist
```

While

```
        while (condition)
                -----------
        end
```

or

```
        while condition
        do
                statements
        done
```

or

```
        while condition, do
                statements
        done
```

Example 4

Alter ‘a’ to the following, then run it.

```
while ($1 > 1)
        echo “boring”
        sleep 5
        $1 = $1 - 1
        end
```

Logical Operators

There are three logical operators which can be used to change/combine exit statuses. The operators correspond to the classical Boolean operators for negation (logical not), conjunction (logical and) and disjunction (logical or). Unlike their counterparts in other languages they operate on programmes.

Negation

The expression ! programme returns an exit status of:

    1 If programme returns a zero exit status
    0 If programme returns a non-zero exit status

Example 5

```
$ ls
        a b
$ if ! ls a; then echo a does not exist; fi
        a
$ if ! ls c; then echo c does not exist; fi
        c: No such file or directory
        c does not exist
```

Conjunction

The expression program1 && program2 does the following:

- If program1 fails (returns a non-zero exit status) it stops and returns the exit status returned by program1
- If program1 succeeds (returns a zero exit status) it will carry out program2 and will return the exit status returned by program2

Example 6

```
$ ls
        a b
$ if ls a && ls b; then echo yes; else echo no; fi
        a
        b
        yes
$ if ls c && ls b; then echo yes; else echo no; fi
        c: No such file or directory
        no
```

Disjunction

The expression program1 || program2 does the following:

- If program1 succeeds (returns a zero exit status) it stops and returns the exit status returned by program1 (i.e. it returns 0)
- If program1 fails (returns a non-zero exit status) it will carry out program2 and will return the exit status returned by program2.

Example 7

```
$ ls
        a b
$ if ls a || ls b; then echo yes; else echo no; fi
        a
        yes
$ if ls c || ls b; then echo yes; else echo no; fi
        c: No such file or directory
        b
        yes
```

Grouping

Commands can be grouped using parentheses or by curly brackets.

Example 8

```
$ ls
$ if ! (ls a || ls b); then echo hello; fi
        a: No such file or directory
        b: No such file or directory
        hello
$ echo $?
        0
```

Grouping with Parentheses

If you group commands with parentheses they are executed in a sub-shell. Make sure you separate commands with semicolons.

Example 9

```
$ ( LOCAL=local; echo LOCAL = $LOCAL; )
        LOCAL = local
$ echo LOCAL = $LOCAL
        LOCAL =
```

Grouping with Curly Brackets

If you group commands with curly brackets they are executed in the current shell. Make sure you put a semicolon after the last command and a space before the closing curly bracket.

Example 10

```
$ { GLOBAL=global; echo GLOBAL = $GLOBAL; }
        GLOBAL = global
$ echo GLOBAL = $GLOBAL
        GLOBAL = global
```

Exercise

Write a shell script which does the following:

- returns an exit status of 1 if and only if there is no file called input in the current directory
- returns an exit status of 2 if and only if there is a file called input in the current directory and the file does not contain the word string
- returns an exit status of 0 if and only if there is a file called input in the current directory and the file contains the word string

In the first two cases the script should output a proper error message. (Hint, use ls, grep, if, and exit).

Practical VI

Shell Variables

You can assign values to them using a statement of the form VAR=VAL (There should be no spaces before and after the = operator. If the value contains spaces enclose them with double quotes). You can dereference a SHELL variable by prefixing it with a $.

Example 1

```
$ echo VAR = \"$VAR\"
        VAR = ""
$ VAR="my favourite value"
$ echo VAR = \"$VAR\"
        VAR = "my favourite value"
$ VAR=
$ echo VAR = \"$VAR\"
        VAR = ""
```

Create a script ‘directory’ containing the following:

```
echo “Enter Directory: “
read DIRNAME
cd $DIRNAME
```

Run the script and see what happens with real and fictitious directory names.

Exporting SHELL Variables

Each time you execute a script or a programme, it inherits a copy of the environment. The environment consists of the values of environment (SHELL) variables and user defined (SHELL) variables which have been made explicitly visible to other programs and scripts. A SHELL variable VAR (and its value) will only be added to the copy of the environment if the variable has been exported with an export statement.

Create a shell script ‘script’ containing

```
echo VAR = \"$VAR\"
```

Try the following:

```
$ VAR="a nice value"
$ echo VAR = \"$VAR\"
VAR = "a nice value"
$ ./script
VAR = ""
$ export VAR
$ ./script
VAR = "a nice value"
```

Positional Variables
In a shell script, you can get hold of the values of arguments using positional variables. Positional variables are read-only SHELL variables. They can only be used in a script. The variables 1, . . . , 9 store the values of the first nine arguments. The variable 0 stores the name of the script that is being executed and # stores the number of arguments.

Example 2
Change ‘script’ to contain the following:

```
        echo My name is $0.
        echo I have $# argument\(s\).
        echo The first argument is $1.
```

Try the following:

```
$ ./script A B
        My name is ./script.
        I have 2 argument(s).
        The first argument is A.

$ mv script test
$ ./test B
        My name is ./test.
        I have 1 argument(s).
        The first argument is B.
```

Change ‘script’ to contain the following:

```
        echo My name is \"$0\"
```

Try the following:

```
$ ./script
        My name is \"./script\"
$ mv script my\ name
$ ./my\ name
        My name is "./my name"
```

Exercise

Implement a script to do the following:

- If it is not called with three arguments it will output an error message.
- Otherwise, let *file* be the value of its first argument. If *file* does not exist then the script will output an error message.

Shell Arithmetic
Used to perform arithmetic operations.

Syntax:
expr op1 math-operator op2

<u>Examples:</u>
```
$ expr 1 + 3
```

```
$ expr 2 - 1
```

```
$ expr 10 / 2
```

```
$ expr 20 % 3
```

Remainder read as 20 mod 3 and remainder is 2.

```
$ expr 10 \* 3
```

For multiplication use * and not * since its wild card.

```
$ echo `expr 6 + 3`
```

Use ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboard OR above TAB key. If you use double quote or single quote, it will NOT work

```
$ echo "expr 6 + 3" # It will print expr 6 + 3
$ echo 'expr 6 + 3' # It will print expr 6 + 3
```

```
$ echo "Sum of $1 and $2 is `expr $1 + $2`"
```

Questions

- Find all the users that use bash as their login shell. Hint: Use grep and have a look at /etc/passwd.
- Use the same grep command and redirect the output in a file called bash users.
- Use grep to see if root is in bash users.
- Use /etc/passwd, a pipeline and two grep commands to see if the login shell of root is bash. You are not allowed to use redirection.