Using the Linux Shell

You can redirect the input output of a program

`ps aux | grep biod`
the output of the `ps` command
is the input to the `grep` command.

`ps aux > save.ps`
the output of the `ps` command
goes to the file `save.ps`

`mail joe </etc/printcap`
the input of the `mail` command
is from the file `/etc/printcap`

`ps aux >> save.ps`
Append to the file

`ps aux | tee save.ps | less`
`tee` duplicates the input stream
One copy goes into a file (for permanent record)
the other goes to the pager (for convenient viewing)

Standard error–error messages and some other information goes to `stderr` instead of `stdout`.
`stderr` is not redirected by the above.
`stderr` can be redirected

```
studentprog |& less
```
everything, including `stderr` is redirected

```
studentprog &> save.me
```

Job Control

Most commands can be successfully run in the background:

```
ps aux  > Look &
```

You get to enter more commands while the `ps` is running.

`jobs`–what is running in the background.

`^Z`–stop (suspend) the current job

`bg`–background the suspended job.
`bg %3`–backgroung job 3 on the jobs list

`fg`–foreground the job
`fg %2`–foregroung job 2 on the jobs list

Shell Programming Overview

Principle: Commands can be placed into a file and run.

Good when you have something you do repeatedly.

Each of the different shells has a slightly different syntax.

Must specify the shell to use for the program.
Defaults to the Bourne Shell (`sh`).
(the first shell for Unix.)

Notes:
The default shell for an account (for typing) is specified in the password file;
if the shell field is left blank your login shell is the Bourne shell.

The file should be marked executable.

The file can be run by naming the file
(if marked executable)
or by starting a shell and using the file as input.

Anything you can put in a shell file can be typed at the command line.

We will cover Bash shell programming.

Simple Example

Create a file with a shell program in it (use `vim` or `nano`)

File Name: `runme`

File Contents:

```
#!/bin/bash
# save the output of the ps to a file
ps aux >> save.ps
```

Mark executable: `chmod a+x runme`

Three ways to run:

`runme` (uses search path)

`./runme` (in current directory)

`bash runme`

The last way does not require the file to be executable.

Note the syntax of the top comment.

The shell is specified using a special form of the comment syntax (`#!/bin/bash`).

Shell Variables

Declare variables with the equals sign.

```
count=5
message=Hello
msg2="Hi There"
```

Variables are stored as strings.
Strings with spaces in them must be quoted.
Spaces around the = are wrong.

The variable may be set from the keyboard using the read command:

`read msg3`
where msg3 is the variable name.

Variables are used with the `$`:

```
echo $count
echo ${message}show
```

Braces may be used,
needed if a non-space follows the variable name

Variables can be set from other variables:

```
icount=$count+1
```

`icount` is now the string `5+1`

Using Command Line

Command line parameters can be accessed

Example: `parm hi there`

Contents of file `parm`:

```
#!/bin/bash
echo $0
echo $1
echo $#
echo $@
```

`$0` the command name: `parm`
`$1` the first parameter name: `hi`
`$#` the number of parameters: `2`
`$@` all the parameters: `hi there`

Shell Numeric Operations

Since variables are strings special notations are necessary.

```
let icount=$count+1
```

`count` was the string 5,
`icount` is set using a numeric operation by the "let" keyword
`icount` gets the value 6.

Allowed operators: `+ - * / % ++ -- += -= *= /=`

Numeric comparison operators: `< <= > >=`

Numeric comparison switches: `-lt -le -gt -ge`

Comparison switches are used with single bracket notation. Operators may be used with double bracket notation.

String comparision operators: `== !=`
Note: `03` is not equal to `3`

The comparisons are used with branches and loops

Booleans

Boolean operators: `! && ||`

Branches

“if” and “switch” type branches are available¡

```
if [ $icount -lt $count ]; then
  echo max is $count
else
  echo max is $icount
fi
```

The “else” part may be omitted

```
case $count in
 0)
  echo not there
  ;;
 3|4)
  echo have a three or a four
  ;;
 5)
  echo have a five
  ;;
 *)
  echo default
  ;;
esac
```

The switch/case uses string compares.

Loops

Boolean test loop

```
sum=0
echo -n "Enter a number:"
read num
while [ "$num" != "" ]
do
  let sum+=$num
  echo -n "Enter a number:"
  read num
done
echo sum is $sum
```

Note the use of the string compare here.

Could do: `[ $num -gt 0 ]` for an integer compare

for type loop

```
for i in  '1 2 hi 4'
do
  echo $i Hello
done
```

Notice it uses strings.

Exit Status

Unix and Linux programs use an exit status, for example `exit(3);` or `return 3;`
This status can be used by scripts.
Bash has a special variable called `$?` that contains the exit/return value of the last program that ran.
Suppose `returnit` is an executable (compiled program). Its return value can be used in a script:

```
#!/bin/bash
returnit
if [ $? == 0 ]; then
  echo returned 0
else
  echo returned something else
fi
```

Most Unix programs are careful about what they return. For example the search program (`grep`) will return 0 if it finds a match and non-zero otherwise. This allows you to branch on whether or not you find something in a file.

Note: `grep` generates string output (which will be sent to the screen by default). So you often see such a program called in a script as:

```
grep "sam" /tmp/homework &> /dev/null
```

This discards the string output, but status is still valid.

The Unix Program `test`

The `test` program is used in many scripts.
It is designed to return 0 if something is true; non-zero otherwise.

For example:

`test -f findme`
returns 0 if `findme` is a regular file

`test -d finddir`
returns 0 if `finddir` is a directory

The manual entry for test shows all the options (and there are many).

In Unix the program "`[`" is an alias (actually a hardlink) for `test`.

`[ -f findme ]`
returns 0 if `findme` is a regular file

The grave

A command is run if appears as the left most thing on a line.

To run a command elsewhere, the grave key is used (the key usually below Esc, above tab, looks like a backwards single quote)

Examples:

`found=‘grep joe myfile‘`
`found` will contain the “screen” output of grep.
Note: returns become spaces so this is a long string.

```
for i in  ‘ls‘
do
  echo $i
done
```

Uses the output of `ls` (the file names) as the loop variables. It uses `echo` to print the directories

Summary: if it is the left item on the line, it runs it, otherwise use the graves to run it.

Exit

The `exit` command leaves the shell. Usually found in a branch statement.

Sample Shell Program

Add a user. Basic commands: Add a password entry. Create home directory. Change ownership
Needed information, user, uid, group, home location, User Name

Sample usage:

```
add_user george 1001 30 /home/george  "Big George"
```

The program

```
#!/bin/bash
grep $1 /etc/passwd >/dev/null
if [ $? == 0 ]; then
 echo account name exists
 exit
fi
test -d $4
if [ $? == 0 ]; then
  echo directory exists
  exit
fi
echo $1":"x":"$2":"$3":"$5":"$4":"/bin/bash \
     >> /etc/passwd
cp -pr /usr/skel $4
chown -R $2.$3 $4
```

Sample Shell Program

Show how many lines are in each of the files in the current directory

```
#!/bin/bash
for name in  `ls`
do
  test -f $name
  if [ $? == 0 ]; then
     wc -l $name
  fi
done
```

Note: `wc` prints the number of lines followed by the file name

Sample Shell Program

Kill those processes with a '?' on the TT column and 'gcc' on the command column when the command `ps aux` is executed.

```
pid=‘ps aux | grep ’gcc’ | grep ’?’ | \
  grep -v ’grep’ | cut -c10-14‘
kill -9 $pid
```

Set a variable `pid` to contain a list of process numbers
use that variable list to kill processes

`ps` for all processes, select those with `gcc` and `?` and without the word `grep` (don't kill this process).
It cuts out the pid columns of lines that match the above conditions.