Unix tour - exercise 1 (the shell)

This exercise takes you around some of the features of the shell. Even if you don't need to use them all straight away, it's very useful to be aware of them and to know how to deal with some problems which may arise.

Try out all of the examples given; if they don't behave how you expect, try to work out what went wrong, and ask for assistance if you need it.

Please **DO** write answers and any other notes you care to make on this handout.

Note: where a command prompt of '$' is shown, you do *not* need to be root (and therefore you should be a normal user, for safety reasons)

Only where the command prompt is shown as '#' do you need to be root. You can use '*su*' to start a temporary root shell, and '*exit*' to return from it.

Starting and stopping processes

For the purposes of testing, we will use the command 'sleep'. This does nothing except wait for a specified number of seconds, and then terminate. In real life, the process you start would be a program which does something useful.

Try it:

```
$ sleep 5
<5 second pause>
$
```

Here you are running the program in the *foreground*. The shell waits for the command to finish, before returning you to the prompt.

Now let's try running a longer version, and terminating it early:

```
$ sleep 600
Ctrl-C
$
```

You didn't want to wait 600 seconds for this process to terminate, so you typed Ctrl-C. This sends a "terminate" signal to the process. This tells the process that you want it to stop; some programs may take the opportunity to clean up any temporary files they have generated first, but will normally exit quickly.

Now, instead of running that program in the foreground, let's run it as a *background* task. The shell does this if you put a single ampersand after the command.

```
$ sleep 600 &
[1] 8531
$
```

The shell prompt comes back immediately. It tells you that this is job number 1, and has process ID 8531 (in this example). The process ID is allocated by the kernel and you will most likely see a different number.

The shell keeps track of each of the jobs it started:

```
$ jobs
[1]+  Running                 sleep 600 &
```

and you should also be able to see it in the table of processes:

```
$ ps auxw | grep sleep
brian     8531  0.0  0.1  1188   472  p5  S    11:16PM   0:00.00 sleep 300
brian     8535  0.0  0.2  1484   912  p5  S+   11:16PM   0:00.00 grep sleep
```

The first column is the username the process is running as; the second column is the pid. Other columns are resource utilisation information, and the end shows the command itself.

> You can do 'ps auxw' by itself, but then you'd see *all* processes on the system, and that's a big list. So we filter it through 'grep' to extract only lines which contain the word 'sleep'. grep is itself a process, and you may see it in the process listing, depending on whether it starts before 'ps' has finished.

Suppose you want to kill this process? You have a choice of several commands.

```
$ kill 8531          # using the pid
$ kill %1            # using the job number
$ fg                 # move it from background into foreground, then use Ctrl-C
```

The first way (using the pid) is the most general-purpose, because as long as you have sufficient rights, you can even kill processes which were started by other shells. But of course, you'd first need to find the pid of the process, using 'ps'.

> Starting commands in the background explicitly using '&' is usually not needed. Most daemon processes come with their own startup scripts which do it for you, or else are able to put themselves into the background automatically. But it's important to be able to locate processes with 'ps' and to terminate them using 'kill'.

Environment variables

Have a look at all the environment variables which your shell has set:

```
$ printenv
```

There may well be more than will fit on the screen. If so, how could you see them all?

Firstly, you can pipe the output of 'printenv' into another command which buffers the output and shows it one page at a time. Try:

```
$ printenv | less
```

Use space and 'b' to move forward and backwards, 'q' to quit.

> What you have actually done is to connect the *standard output* of **printenv** into a pipe, connect the other side of the pipe into the *standard input* of **less**, and run both programs at once in separate processes. Many Unix utilities are designed to be chained together in this way.

Another option is to capture the output of the command into a file, which you can then open using a text editor (or you can mail it to someone else to look at).

```
$ printenv >env.txt
$ vi env.txt
(exit by hitting ESC :q! Enter)
```

> Here, you have opened a new file for writing, and connected it to the *standard output* of printenv.

> *QUESTION:* You don't want to leave env.txt lying around. How can you delete it? Now delete it. How can you check that it has actually been deleted?

Now let's demonstrate how settings in the environment can alter the behaviour of a program.

Many programs which need to edit files will use the editor you specify in environment variable 'EDITOR'. Have a look at the current setting: there are two different ways to do it, try both.

```
$ echo $EDITOR
vi
$ printenv EDITOR
vi
$
```

Now let's look at another command. 'vipw' locks the master.passwd file, runs an editor on it, and then rebuilds the various password database files if you have made any changes. It can be used to add/delete or modify users. Let's try it now; you have to be root. When it puts you into vi, don't make any changes, just quit without saving.

```
$ su
Password: <enter root password>
# vipw
:q!
#
(stay as root for now)
```

Now we're back again, let's change the EDITOR environment variable to point to 'ee', check it has changed, then run vipw again:

```
# setenv EDITOR 'ee'
# echo $EDITOR
ee
# vipw
```

Hopefully, you'll find yourself in 'ee' which looks very different. You should notice a help section at the top of the screen. Getting out of this editor is different to getting out of vi:

```
Ctrl-C exit [Enter]
#
```

Once you're back at the command line, type 'exit' again to get out of 'su' and so return to being a normal user. The command prompt should revert to '$'.

```
# exit
$
```

Now you're back again, you should check the EDITOR environment variable again:

```
$ echo $EDITOR
vi
$
```

What's happened here? It looks like the changes have been lost, and yes they have. The reasons are:

- Every shell has its own *copy* of the set of environment variables
- When you change them, you are changing only the shell's local copy
- When the shell exits, any changes you made are lost
- When you do 'su' you are starting a new shell, with its own private copy of the environment. So if you change any environment variables there, when you exit you are returning back to the original shell, and back to its original environment.

So, the only way to make this change permanent is to arrange that every time your shell starts up, it sets the environment how you want it. This is done through shell startup files, sometimes called "rc files" (for Runtime Configuration).

So, how do you make 'ee' the default editor permanently? It's made slightly complicated by the fact that by default FreeBSD uses 'csh' for root's shell, and 'sh' for other users' shells, and these two different shells have different startup scripts and different syntax for setting environment variables.

- For root: edit /root/.cshrc and change the existing EDITOR assignment so it says:

  setenv EDITOR 'ee'

Aside: The quotation marks are not necessary here, but would be if the value contained spaces.

- For other users: go to their home directory, edit **.profile** and change the existing EDITOR assigment so it says:

  EDITOR='ee'; export EDITOR

Why 'export EDITOR'? With **sh**, only environment variables marked as "for export" are made available to programs started from the shell, so you have to explicitly mark them.

Command processing

Now let's have a look at how the shell processes the line you type, splitting it into command and arguments. It's useful to use the command 'echo' for testing: this just prints back its arguments to you. Try it:

```
$ echo hello world
hello world
```

So, where actually is the command 'echo'? Well, you can ask the shell to look for it for you:

```
$ which echo
/bin/echo
```

How did it find it? By looking in a series of standard directories, and this list of places comes from the environment variable 'PATH'. It tries them one after the other, stopping as soon as a match is found.

```
$ echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:
/usr/X11R6/bin:/home/yourname/bin
```

So it tries */sbin/echo* and fails; next it tries */bin/echo* and succeeds. If it wasn't there, it would try the other directories. If you want to override the search, or you want to run a program which is in a directory which is not in $PATH, then you can give it an explicit location:

```
$ /bin/echo hello world
hello world
```

> Actually, this particular example is a little bit of a special case but it's worth noting. A few commands are built-in to the shell itself, and 'echo' happens to be one of those. So if you type 'echo' the shell itself is answering you; if you type '/bin/echo' you are using an external program. Normally you'd expect both the built-in shell command and the external one to behave in the same way, but since they are actually two different programs there might be some minor differences.

Line splitting

When you type

```
$ /bin/echo hello world
```

then the following things happen:

1. the shell splits this into three words: "/bin/echo", "hello" and "world", using spaces as the separator
2. the shell performs various types of expansion on these words (see below)
3. it starts a new process, runs /bin/echo in it, and passes in "hello" and "world" as arguments
4. this program executes. /bin/echo just sends it arguments back down its standard output, separated by a space, and ending with a newline; normally we'd be running something more useful!

You can show this is the case by putting in lots of spaces:

```
$ /bin/echo      hello           world
hello world
```

The shell has still split this into three words, "/bin/echo", "hello" and "world", and echo has joined them together with a single space.

Where this becomes a problem is if you need to use a filename which contains a space (which are usually best avoided, but you will come across them from time to time). If you try this:

```
$ touch my file
$ ls -l
```

you should find that you have actually created *two* files, called "my" and "file". So you need to stop the shell from breaking them into separate arguments. You can do this using quoting, and there are three main ways:

```
$ touch "my file"
$ touch 'my file'
$ touch my\ file
$ ls -l
```

The first two are enclosing the filename in either single or double quotes; the last is to preceed the space with a backslash, which makes it lose its special meaning as an argument separator. This also gives you a way to make echo give you a string with lots of spaces in it:

```
$ echo "hello       world"
hello       world
```

> If you've done all this, you now have three files in your current directory: "my", "file", and "my file". You need to delete them all. You can do this with three separate commands - or see if you can work out how to make a single line which deletes all three.

Note that it's always safe to add quoting to arguments, even if they don't strictly need it. So these two commands are the same:

```
$ rm somefile
$ rm "somefile"
```

Argument expansion

The shell performs additional processing on arguments; it's very useful, but it can cause strange behaviour if you're not aware of it. Try the following command:

```
$ cd /usr/bin
$ echo *what* is your name?
makewhatis what whatis is your name?
```

That's certainly strange! You thought echo was supposed to just return the arguments you gave it? Well it has, but the shell has expanded them first.

Here the shell is doing pattern matching against files in the filesystem, otherwise known as "filename globbing". The character "*" matches any 0 or more characters in the filename. So, "*what*" looks in the current directory for all files whose names contain the string 'what' anywhere in the middle, and replaces it with those filenames.

> Prove to yourself that files /usr/bin/makewhatis, /usr/bin/what and /usr/sbin/whatis do in fact exist on your system.

You can prevent this from happening by using quoting, just as we saw before. You can quote individual words, or quote the whole piece of text so that echo sees it as a single argument:

```
$ echo "*what* is your name?"
*what* is your name?
```

Globbing is really useful; for example you can delete all files in the current drectory with names ending .txt just by typing "rm *.txt"

Filename globbing uses these special characters:

```
*      matches any 0 or more characters
?      matches any 1 character
[abc]  matches 'a', 'b' or 'c' only
[a-z]  matches any character between 'a' and 'z' inclusive
```

> QUESTION: Try these commands. Look carefully at the last word of the result. Explain what has happened.
>
> ```
> $ cd /usr/sbin
> $ echo what is your name?
> ```

Home directory (Tilde) expansion

Arguments which begin with a tilde (~) are expanded into your home directory, or if followed immediately by a username, the home directory of that user (taken from /etc/passwd). There is no checking that the filename produced actually exists or not.

```
$ echo ~
/home/yourname
$ echo ~/wibble
/home/yourname/wibble
$ echo ~root/wibble
/root/wibble
$ echo ~www/wibble
/nonexistent/wibble
```

It's a useful shortcut; e.g. ~/.ssh/authorized_keys refers to a file within directory .ssh within your home directory.

Parameter expansion

Arguments which contain $ are subject to further expansion. There are actually a lot of things you can do with this, but the most common example is substitution of environment variables, as we saw before with $PATH.

```
$ echo $HOME
/home/yourname
```

There are some special shell variables, the most useful being $? which gives the exit status of the last command (0=success, >0=failure) and $$ which gives the process id of the shell itself.

```
$ echo $$
2302
```

However you can do other fancy things, including using the output of one command and inserting it in the command line as an argument to another. There are two syntaxes for this, $(...) and backticks. For example, we can run 'wc' to count the lines in a file, and use the result as an argument to another command, such as 'echo' for testing.

```
$ echo $(wc -l /etc/motd)
24 /etc/motd
$ echo `wc -l /etc/motd`
24 /etc/motd
```

And there's even basic arithmetic, although it's very rarely used:

```
$ echo $((3+4))
7
```

This sort of parameter expansion can be disabled by quoting, *except* that double quotes don't disable it; single quotes and backslash do. So:

```
$ echo "Home is $HOME"
Home is /home/yourname
$ echo 'Home is $HOME'
Home is $HOME
$ echo Home is \$HOME
Home is $HOME
```

I/O redirection

You've already seen one example of I/O redirection: connecting the standard output of a process to write to a file.

```
$ echo "hello" >test.txt
```

But you can also append to a file, instead of overwriting it:

```
$ echo "hello" >>test.txt
$ echo "hello" >>test.txt
```

Equally you can connect the standard input of a process to read from a file:

```
$ less <test.txt
(should show three lines of 'hello')
```

Command grouping

The shell lets you run multiple commands on the same line. Some example are:

1. Run commands one after the other

   ```
   $ echo hello; echo world
   hello
   world
   ```

2. Run command only if the preceeding command succeeded

   ```
   $ echo hello && echo world
   hello
   world
   ```

3. Run command only if the preceeding command failed

   ```
   $ echo hello || echo world
   hello
   ```

4. Run a group of commands in a subshell. They can share I/O redirection.

   ```
   $ (echo hello; echo world) >out.txt
   ```

Again, if you need to use these special symbols as part of an argument (say they are in a filename), you can quote them.

```
$ echo "hello; echo world"
hello; echo world
```

Quoting summary

This is by no means a complete list of shell features, but it should be clear by now that lots of characters have special meanings to the shell. If you are using a filename which contains anything other than letters and numbers, dot, dash and underscore, you'd be wise to quote it to make sure nothing unexpected happens! And you should try to avoid creating files whose names contain special characters (space, question mark, asterisk, ampersand, tilde etc) in the first place, because of the confusion they could cause to others.

Flags and dash

Finally, many commands take options (flags) beginning with a dash. For example:

```
$ less -Mi /etc/motd
```

The -M and -i flags alter the behaviour of less (read 'man less' if you want to know how). So how would you actually create or delete a file whose name begins with a dash? The command will try to interpret it as a flag, not a filename.

```
$ touch -foo
touch: illegal option -- o
usage: touch [-acfhm] [-r file] [-t [[CC]YY]MMDDhhmm[.SS]] file ...
```

Quoting doesn't help us here, because it's not the shell which is expanding anything. But most commands have a convention that if you include '--' as an option, then everything afterwards is *not* considered an option, even if it starts with a dash. So:

```
$ touch -- -foo
$ ls
$ rm -- -foo
```

Warning: because they are a pain, try to avoid creating files whose name begins with a dash!

Extra exercises

If you have spare time available, work on the following problems. Feel free to refer to any handouts, documents or web pages which might help you.

1. You know that printenv lists all the environment variables which are set, one per line. How could you quickly determine *how many* environment variables are set? (That is, without counting them by hand :-)

2. The command "touch -foo" shown above gives a failure message. What exit status value does it return? If you try to run a non-existent command, what exit status do you get?

3. Capture the actual error message which "touch -foo" gives into a file. Make sure that the file does actually contain the error message. (Hint: error messages are sent on the standard error stream, not standard output)

   This is useful in the event that you wanted to mail the error message to someone, for example.

4. List all the files in /etc which have "tab" in their name. Find two or more different ways of doing it, at least one not using asterisk.