THE UNIVERSITY OF SYDNEY

COMP2129 Week 2 Tutorial

Introduction to UNIX and C

Getting into Linux

The lab machines are set up with a dual-boot system, of Windows and Linux. For these tutorials, we will be using Linux. If your computer is not already running Linux, follow these steps:

- Reboot your machine.
- While it's starting up, a blue screen will appear. Select "Fedora Linux" with the arrow keys, and press enter.
- When the login screen appears, log in with your unikey and password.

You may have trouble accessing external `https://` sites through Firefox. To fix this, go to Preferences, Advanced, Networking, Connection, and select the bottom option (Automatic Proxy configuration URL), entering the URL `http://sydney.edu.au/proxy.pac`.

Exercise 1: In the beginning was the command line

Although modern *nix distributions have a graphical interface for “pretty much everything” these days, in this course we encourage you to use the command line (specifically the bash shell) as much as possible. It may seem a little confusing at first, but once you get used to it, you’ll find that many tasks are much faster and easier with a powerful set of tools at your fingertips.

Open a terminal (Applications, Accessories, Terminal). As mentioned in lectures, this terminal is running a shell (probably bash). Try typing the following command (without the $ sign) and pressing enter:

```
$ echo "Hello world"
```

(It’s a convention to write $ to show people that the following text should be entered into a shell).

One of the learning outcomes for the course is that you understand the Unix philosophy and the approach and concepts behind common Unix tools http://wikipedia.org/en/Unix_philosophy.

Some well-known sayings include “small is beautiful”, “make each program do one thing well” and “no need to reinvent the wheel”. In the spirit of the latter, we refer you now to the Unix tutorial by William Knottenbelt: http://www.doc.ic.ac.uk/~wjk/UnixIntro/ (also available on eLearning). If you are new to a UNIX shell, please work through Exercises set 2 to familiarise yourself with navigating a directory structure from the command line. If you are comfortable with a shell, then have a look at Exercise set 1 to see how many of the commands you recognise. Some of them are a little obscure, and may not be installed on the undergraduate servers, but hopefully you will learn something useful.

Some of the basic commands you will definitely need for moving around and using the file system include: `cd, pwd, mkdir, rm, rmdir, man, ls, cat, cp, mv, less.`

Before moving on to the next section, make yourself a local temporary directory, by typing `cd` to move yourself to your home directory, `mkdir tmp`, to make yourself a new folder called `tmp`, and then `cd tmp` to move yourself into it. You can confirm you’re there by running `pwd`. (print working directory).

Exercise 2: Grepping for lines

There are many small tools deployed with a UNIX system, along with a shell, which can coordinate these tools into large useful programs. In this section we'll pull a few of these tools together into pipelines to do interesting things.

Modern Linux systems contain a dictionary of known English words, stored at `/usr/share/dict/words`. Copy this to your local directory:

```
$ cp /usr/share/dict/words .
```

You can use `$ ls` to make sure that the `words` file is saved to your local directory.

The command `cat` will output the contents of a file, or if multiple files are given, will con`cat`enate them together (output them in sequence). Try running

```
$ cat words
```

A bunch of words should have flashed before your eyes, you may not have caught them all. Unix systems also come with paging programs `less` and `more`, which allow you to page through long files or outputs. Try running

```
$ less words
```

and use the arrow keys, or page up and down, to move up and down the file. Type `q` when you're done to close the paging program.

`grep` is a program which can isolate lines from a file matching a certain search pattern. Let's try looking at all words containing the substring "fish", in the file `words` you have in your local directory:

```
$ grep fish words | wc -l
```

Wow, that was a lot of words, right? How many? Use the `wc` "word count" tool to find out:

```
$ grep fish words | wc -l
```

You just "piped" the output of `grep` into `wc`, giving it the `-l` flag (which tells it to count the number of lines). The beauty of UNIX tools is that the output of each program can be piped into more and more programs, and little programs can be pulled together in new ways by the shell user. The overall philosophy is for each program to do only *one* thing, and to do it *well* - this allows the shell user to pull together a collection of small, well-functioning programs into small parts of a larger system.

You can find out about the `wc` by using the UNIX manual `man`.

```
$ man wc
```

You can navigate around manpages like less windows - using up, down, and q to quit. You should now know what wc will do if you give it the -c or -w flags, instead of the -l flag.

You can also chain multiple grep commands together. At the moment we've been telling grep to read from a file, but it can also act as part of a pipeline, just like wc did. Try running

```
$ grep fish words | grep sword
```

Here the first grep instance is reading from the file words, but the second instance has no file mentioned to it, and so is just reading the output which is piped into it. A lot of unix commands work like this: they can either read from one or more files, or they will read from *standard in* (the pipeline).

This filters the entire dictionary down to lines containing the substring "fish", then filtered *those* down further to words containing "sword". Isn't that neat?

Linux has special files stored in the /proc directory which have information about the currently running processes, the CPU(s), and RAM of your machine. Try looking at the

/proc/cpuinfo file:

```
$ less /proc/cpuinfo
```

You should see multiple processors there (if you're on one of the lab machines). Let's try to extract their model names:

```
$ cat /proc/cpuinfo | grep name
```

You should now see four identical lines, each one representing a single core inside your machine's processor. Since having four lines saying the same thing is somewhat redundant information, let's limit it by running it through the uniq command:

```
$ cat /proc/cpuinfo | grep name | uniq
```

Check the manpages to see what uniq does.

Exercise 3: Shell redirection

Suppose you want to save the output of a command to a file so that you can refer back to it later. You can do this using *shell redirection* with the > character:

```
$ grep fish words > myfile
```

This tells the shell to run the command `grep fish words`, and to write the output to the file `myfile`. Run `$ cat myfile` to make sure they're all there.

If you run that command again, it will overwrite the contents of `myfile`. By using the >> command, you can tell the shell to write to the file in *append mode*, leaving any new line(s) at the end of the file, instead of clobbering the whole file.

```
$ echo "aardvark" >> myfile
```

Look at the last 10 lines by running

```
$ tail myfile
```

Of course, the word "aardvark" is now no longer in dictionary order, so sort the file and save it as a new file.

```
$ sort < myfile > mysorted
```

The < symbol tells the shell to open `myfile` and send it into `sort`. The > symbol then redirected the output into the file `mysorted`.

Equivalent commands

The shell and standard tools are very flexible: they will allow you to write pipelines which do the same thing in many different ways.

The following shell lines *should* get all of the words from `/usr/share/dict/words` containing the substrings "table" and "po", and save them to a local file called `potable`. Classify the following commands as being in group 1, 2, or 3, where the groups are:

1. The shell line does what is described above.
2. The shell line does not do what is described above.
3. The shell line does not make sense (syntax error).

```
$ cat /usr/share/dict/words > | grep table | grep po | > potable
$ grep table /usr/share/dict/words | grep po | cat > potable
$ grep /usr/share/dict/words table | grep po > potable
$ grep table < /usr/share/dict/words | grep po > potable
$ cat /usr/share/dict/words > grep po | grep table < potable
$ cat < /usr/share/dict/words | grep po | cat | grep table | cat > potable
```

Exercise 4: Hello, C

To write C programs, you need an editor. We recommend using `nano` if you've never tried a terminal-based text editor before. Run it now:

```
$ nano hello.c
```

You can now write text, and move around with the arrow keys. `Ctrl+o` will save your file: if it prompts you about the file name, press enter. `Ctrl+x` will leave the editor and return you to the shell (nano will alert you if you have unsaved changes). Write the following C program using nano:

```c
#include <stdio.h>

int main(void) {
  printf("Hello, world!\n");
  return 0;
}
```

Once you have it saved, and are back in the shell, make sure it's there by running

```
$ cat hello.c
```

You can compile it using `gcc`, the GNU C Compiler:

```
$ gcc hello.c -o hello
```

The `-o hello` tells `gcc` to write the output to the file `hello`. Now you can run your program:

```
$ ./hello
```

You should see a line in the terminal saying "hello, world!".

You can look up the C function `printf` in the manpages, however if you just run `man printf`, you'll come across the documentation for a *different* `printf`. The manpages are divided up into sections - the ones we'll come across in this course are:

- Section 1: Shell commands.
- Section 2: Linux system calls.
- Section 3: C Standard Library routines.

To show which sections of the manual the page `printf` appears in, run:

```
$ man -f printf
```

You should see sections 1, and 3. Since the `printf` we're looking for is part of the C standard library, let's access the `printf` in section 3:

```
$ man 3 printf
```

Exercise 5: Let's shout

I have always been a fan of CAPITAL LETTERS ON THE INTERNET. I want to make a little C program which will translate anything I say into a version consisting of all capital letters — then I can use it before I post anything on an internet forum.

I know that C has a function called `toupper`, and I also know about the functions `getchar` and `putchar`. (You should look these up in the manpages). Unfortunately, I only have a program called `whisper.c` which converts everything to lower case.

```c
#include <stdio.h>
#include <ctype.h> // Check 'man tolower' to find the includes.

int main(void) {
  int c;
  while ((c = getchar()) != EOF) {
    c = tolower(c);
    putchar(c);
  }

  return 0;
}
```

Make a new file, `whisper.c`, and enter the code above. You can compile and run the program like the last one:

```
$ gcc whisper.c -o whisper
$ ./whisper
```

You can now write text, and each line will be echoed back at you, in lower case. You can press `Ctrl+d` to indicate you're done entering input, and return to the shell. You can also pipe the output of UNIX commands into the whisper program - compare the following two commands:

```
$ date
$ date | ./whisper
```

Make a copy of `whisper.c` to `shout.c`, and change the program so that it will output all upper case, rather than lower case.

```
$ cp whisper.c shout.c
$ nano shout.c
```

Once you're done with that, try using the `isspace` function to replace any space characters with an underscore.