Lab 1

COMP9021, Session 1, 2013

Part I

Working in Unix

The purpose of this part is to help you become familiar with the basics of the Unix operating system. Skim through this part so you can refer to it in the second part of the lab, devoted to configuring your programming environment and testing it. Use this first part as a reference whenever you need it in this or future labs.

1 Introduction

Using the *Terminal* application, open an *x-term window*. You type your commands in an x-term window.

- Many commands take one or more *arguments*.
- Many commands can take one or more *options*.
- The *short* options start with one hyphen (`-`) followed by one letter, and sometimes an argument for the option.
- The *long* options start with one or two hyphens (`-` or `--`) followed by a string (word or sequence of words usually separated with a hyphen), and sometimes an argument for the option.
- Many short options that do not take an argument can be combined together, with one hyphen followed by the letters of the options.
- Many arguments are *optional*.

2 A few Unix commands

Try the following commands.

- `date`.
- `cal` (no argument), `cal 2013` (one argument), `cal 3 2013` (two arguments).

- `pwd` to **p**rint the **w**orking **d**irectory, which after you have logged in and before you have done anything, is also your *home directory*.
- `ls` to **l**i**s**t the *files* (of which *directories* are a particular case) in the working directory, excluding the *hidden files* whose name starts with a dot (these are usually configuration files that are seldom modified or read).
- `ls -a` (one short option) to list **a**ll files in the working directory.
- `ls -l` to get a **l**ong listing of the files in the working directory, excluding the hidden files.
  - The most common characters for the first character are `-` for a regular file, and `d` for a directory.
  - The next three characters indicate whether the file is *readable* (`r`) or not (`-`), *writable* (`w`) or not (`-`), and *executable* (`x`) or not (`-`) by the *owner* of the file.
  - The next three characters provide the same information for the users who belong to the same *group* as the owner of the file.
  - The next three characters provide the same information for all other users.
- `ls -l -a` or `ls -a -l` or `ls -la` or `ls -al` to use `ls` with both short options.
- `mkdir` followed by a number of directory names to **m**a**k**e (create) some **dir**ectories. The names can be either
  - *absolute paths*, that start with the string that `pwd` returns when it is executed in your home directory;
  - paths that are *implicitly relative* to the working directory;
  - paths that are *explicitly relative* to the working directory, starting with `./`;
  - paths that are explicitly relative to the *parent* of the working directory, starting with `../`. More generally, `..` can be used in paths to go one level higher in the *hierarchy* of directories;
  - paths that are explicitly relative to your home directory, starting with (`~/`).

  For instance, assume that `pwd`, when executed in your home directory, prints out

  ```
  /import/kamen/1/aussie278
  ```

  So your user name is `aussie278`. Assume that your home directory contains a subdirectory named `Letters` which itself contains a subdirectory named `Friends`. Finally, assume that your working directory is the subdirectory `Letters` of your home directory. So `pwd`, executed in this working directory, outputs

  ```
  /import/kamen/1/aussie278/Letters
  ```

  Now assume that you want to create the subdirectories `Family`, `Work` and `Council` of the subdirectory `Letters` of your home directory, a subdirectory `User_manuals` of the home directory, and a subdirectory `iPad` of the directory `User_manuals`. Then corresponding to the 5 options listed above, you could execute:

  - `mkdir /import/kamen/1/aussie278/Letters/Family`

  - `mkdir Work`
  - `mkdir ./../User_manuals`
  - `mkdir ../Letters/Council`
  - `mkdir ~/User_manuals/iPad`

  Of course, rather than the first, third and fourth commands above, it would be more natural and effective to execute instead:

  - `mkdir Family`
  - `mkdir ../User_manuals`
  - `mkdir Council`

- `touch` followed by a number of file names to create empty files or to modify the *last modification date* of existing files. When creating empty files, `>` is a simpler alternative to `touch`. For instance, to create two files `file_name1` and `file_name2` in the working directory you can type either `touch file_name1 file_name2` or `>file_name1 >file_name2` (with or without spaces after `>`).

- `cd` to **c**hange (go to another) **d**irectory. This command can be followed by:

  - no argument, in which case the new directory is the home directory;
  - an absolute path name;
  - a pathname that is implicitly relative to the working directory;
  - a pathname that, starting with `./`, is explicitly relative to the working directory;
  - a pathname that, starting with `../`, is explicitly relative to the parent of the working directory.
  - a pathname that, starting with `~/`, is explicitly relative to your home directory.

  For instance, assume that your working directory is `~/Letters/Friends`, that is, the subdirectory `Friends` of the subdirectory `Letters` of your home directory. Also assume that you first want to go to your home directory, and from there to the directory `~/Letters`, and from there to `~/Letters/Family`, and from there to `~/User_manuals/iPads`, and from there to `~/Letters/Council`, and from there to `~/Letters`. Then corresponding to the 6 options listed above, you could execute:

  - `cd`
  - `cd /import/kamen/1/aussie278/Letters`
  - `cd Family`
  - `cd ./../User_manuals/iPad`
  - `cd ../Letters/Council`
  - `cd ~/Letters`

  Of course, rather than the second, fourth and sixth commands above, it would be more natural and effective to execute instead:

  - `cd Letters`
  - `cd ../User_manuals/iPad`

  - `cd ..`

- `mv` *`file_name directory_name`* to move the file *`file_name`* to the directory *`directory_name`*, where *`file_name`* and *`directory_name`* can be either relative or absolute paths.
- `cp` *`file_name1 file_name2`* to copy the file *`file_name1`* and give it the name *`file_name2`*, where *`file_name1`* and *`file_name2`* can be either relative or absolute paths.
- `cp` *`file_name directory_name`* to copy the file *`file_name`* in the directory *`directory_name`*, where *`file_name`* and *`directory_name`* can be either relative or absolute paths.
- `rmdir` followed with some directory paths to **rem**ove those **dir**ectories, provided that they are *empty*, *i.e.*, do not contain any file.
- `rm` followed with some regular file paths to **rem**ove those files.
- `rm -r` followed with some directory paths, *i.e.*, the previous command provided with one short option and directory paths as arguments, to **r**ecursively remove those directories and everything they contain, down to any depth. To be used with utmost care…

Command completion is a useful feature of the *bash shell*, the command-line interpretation we are using. By pressing the **tab** key, you let bash complete what you are typing. For instance, suppose that you want to go from the working directory to a subdirectory whose name starts with `User`. Suppose that you type `cd User`.

- If only one subdirectory has a name that starts with `User`, say `User_manuals`, then pressing the tab key after `cd User` automatically completes the command to `cd User_manuals`.
- If no subdirectory has a name that starts with `User`, then pressing the tab key again and again after `cd User` will just make your computer beep, or flash, or complain in some way.
- If many subdirectories have a name that starts with `User`, then pressing the tab key once after `cd User` will make your computer complain, but pressing the tab key a second time will display the list of all subdirectories whose name starts with `User`, and let display `cd` with its incomplete argument again, giving you hints on how to complete it partially or totally.

You can also use the uparrow and the downarrow of your keyboard to retrieve commands you have typed previously.

2.1 The chmod command

Recall from previous section what the `ls -l` command outputs. When you want to change the permissions of some file, you use the `chmod` command to **ch**ange the **mod**e of the file.

- With the options, `+r`, `+w` or `+x`, you make (or keep) the file readable, writable or executable, respectively.
- With the options, `-r`, `-w` or `-x`, you make (or keep) the file nonreadable, nonwritable or nonexecutable, respectively.

- Depending on which system you work on, the previous options might change the permissions for everyone, or for just the owner of the file. To restrict the change to the owner of the file, to the members of the group to which the owner of the file belongs, and to the other users, prefix the option with `u` (like **u**ser), `g` (like **g**roup), or `o` (like **o**ther), respectively.

- The options can be combined. For instance, `chmod go-wx` *`file_name`* will prevent the members of the group and the other users to write and execute the file *`file_name`*.

2.2 The tar command

`tar` is used to put together a number of files into a single file, called an *archive*, possibly compressed so that it takes less space. It is also used to perform the inverse operation of creating a hierarchical structure of files from a single, possibly compressed, archive. Finally, it can be used to display the contents of an archive.

- You **c**reate a compressed (**z**ipped) archive
  - of all files stored in a directory `directory_name`, by executing

    `tar czf` *`archive_name`*`.tar.gz` *`directory_name`*

  - of the files *`filename_1`* ... *`filename_n`*, by executing

    `tar czf` *`archive_name`*`.tar.gz` *`filename_1`* ... *`filename_n`*

  which will create a **f**ile whose name is *`archive_name`*`.tar.gz`.

- You display the contents of an archive *`archive_name`*`.tar.gz` by executing the command `tar tzf` *`file_name`*`.tar.gz`, where `t` stands for **t**able of contents.

- You obtain the files from which an archive *`archive_name`*`.tar.gz` has been created by executing `tar xzf` *`archive_name`*`.tar.gz`, where `x` stands for e**x**tract.

Note the extensions we have been using: `.tar.gz` that indicates a compressed (`.gz`) archive (`.tar`). Sometimes, you will only want to compress or uncompress a single file; the commands `gzip` and `gunzip` will do the job, respectively.

3 Wildcards

Wildcard save you from typing too many characters. Here are some example of uses of the wildcards `*`, `?` and `[`*`numbers_or_range_of_numbers`*`]`:

- `ls *` gives a listing of all files and directories in the working directory.

- `ls file*.c` gives a listing of all files whose name starts with `file` and ends in `.c`, with any characters (possibly none) in between (so it would match `file.c`, `file2.c`, ...).

- `ls file?3.c` gives a listing of all files whose name starts with `file` and ends in `3.c`, with exactly one character in between (so it would match `file13.c`, `fileA3.c`, ...).
- `ls file??.c` gives a listing of all files whose name starts with `file` and ends in `.c`, with exactly two characters in between (so it would match `file12.c`, `file1B.c`, ...).
- `ls file[13].c` gives a listing of all files whose name starts with `file` and ends in `.c`, with either `1` or `3` in between (so it would match `file1.c` and `file3.c`).
- `ls file[1-3].c` gives a listing of all files whose name starts with `file` and ends in `.c`, with either `1`, `2` or `3` in between (so it would match `file1.c`, `file2.c` and `file3.c`).

Of course, wildcard can be used with any command, not just `ls`.

Part II

Programming in Unix

The purpose of this part is to help you configure your programming environment, test it, and start practicing. Many steps consist in configuring Emacs so it can be used as a lean IDE (Integrated Programming Environment). But remember that using Emacs customised that way is only an option, and you might want to explore and see whether you would prefer using other tools amongst those installed on the School machines.

4 Setting up your documentation and programming environment

Recall that you can check in which directory you currently are with the command `pwd` and you can list the contents of the current directory with the command `ls`. Use these commands every time you have doubts on "where you are" or "what you possess".

- Go to or remain in your home directory by executing a plain

  ```
  cd
  ```

- Create a subdirectory `COMP9021` of your home directory by executing

  ```
  mkdir COMP9021
  ```

- Go to the directory you have just created with the command

  ```
  cd COMP9021
  ```

- In `~/COMP9021`, create subdirectories `Lectures`, `Labs` and `Assignments` of `~/COMP9021` by executing

  ```
  mkdir Lectures Labs Assignments
  ```

  to store the relevant material. It is important to be organised and tidy…

- Still in `~/COMP9021`, create subdirectories `scripts`, `include` and `lib` of `~/COMP9021` by executing

  ```
  mkdir scripts include lib
  ```

  to store further material.

- From WebCMS, save the tarred compressed archives `Notes_1.tar.gz` and `Notes_2.tar.gz`, under those names in `~/COMP9021/Lectures`.

- Still in `~/COMP9021`, move to `~/COMP9021/Lectures` with the command

```
cd Lectures
```

In `~/COMP9021/Lectures`, decompress and untar all of `Notes_1.tar.gz` and `Notes_2.tar.gz` by executing (recall the `*` wildcard)

```
tar xzf Notes_1*
tar xzf Notes_2*
```

and then delete the archives with the command

```
rm Notes*gz
```

- Move to your home directory with a plain

  ```
  cd
  ```

  Execute

  ```
  cp COMP9021/Lectures/Notes_2/emacs.el .emacs.el
  ```

  to save a copy of the file `emacs.el` provided in the second set of notes in your home directory under the name `.emacs.el` (note that the name of the file you retrieve from WebCMS does not start with a dot, but the name of the copy you save in your home directory starts with a dot).

- Still in your home directory, execute

  ```
  cd COMP9021/Lectures/Notes_2
  ```

  to go to the directory which contains all the material provided with the second set of notes. Save a copy of all scripts you have been provided with, namely `_getfilenames`, `_mctemplate`, `_mmakefile`, `mmakefile` and `mycstyle`, in the dedicated directory you have created earlier, namely `~/COMP9021/scripts`, by executing

  ```
  cp _* m* ../../scripts
  ```

- Still in `~/COMP9021/Lectures/Notes_2`, save in directory `~/COMP9021` a copy of the file `style_sheet.txt` provided in the second set of notes by executing

  ```
  cp s* ../..
  ```

- Still in `~/COMP9021/Lectures/Notes_2`, save in `~/COMP9021/include` a copy of `p_io.h` by executing

  ```
  cp *.h ../../include
  ```

- Still in `~/COMP9021/Lectures/Notes_2`, execute the following commands (ignore the warnings produced by the first invocation to `gcc`).

  ```
  gcc -std=gnu99 -c p_io.c
  gcc -shared -o ~/COMP9021/lib/libp_io.so p_io.o
  rm p_io.o
  ```

  This will create a file named `p_io.o` in the current directory, then create a file named `libp_io.so` in `~/COMP9021/lib`, and finally remove the file `p_io.o` from the current directory (you are not expected to understand what the first two commands really do. . .).

- Move to `~/COMP9021` by executing

  ```
  cd ../..
  ```

- In `~/COMP9021`, make all scripts e**x**ecutable by **a**ll, using the command

  ```
  chmod a+x scripts/*
  ```

- Go back to your your home directory with a plain

  ```
  cd
  ```

  Check whether you have a file named `.profile` by typing

  ```
  ls -a .profile
  ```

  If you do, make a backup copy of that file, as we are going to modify it, but it is easy to mess things up, with possibly annoying consequences. . . . To make a backup copy, execute the command

  ```
  cp .profile .profile.original
  ```

  Then open the file `.profile` with an editor such as `nedit`, executing the command

  ```
  nedit .profile
  ```

  and make the following changes to `.profile`.

  – If there is no line that starts with `export PATH=` then insert somewhere the line

    ```
    export PATH=$PATH:$HOME/COMP9021/scripts:
    ```

    If there is a line that starts with `export PATH=` then change that line so that it ends in `:$HOME/COMP9021/scripts:`, and so that the sequence of symbols on the right hand side of the `=` sign does not start with a colon nor with a colon preceded with a dot and does not contain two successive colons possibly with a dot in between (so reduce any sequence of the form `::` or `:.:` to `:`). The purpose of this change is that you can execute anywhere any command whose code is stored in either `~/COMP9021/scripts` or in the working directory.

  - Add a line that reads
    `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/COMP9021/lib`

  Don't try and understand what this does. Now save your changes and exit the editor.

- Still in your home directory, execute

  ```
  . .profile
  ```

  This will have the effect of making sure that the changes you have just made to `.profile` take effect now, without having to log out.

Congratulations, you are all set!

5 Compiling and running programs from the command line

5.1 By issuing commands to the compiler

Make `~/COMP9021/Lectures/Notes_1` your working directory, and compile and run the four programs in this directory, namely,

`input_output.c`, `two_worlds.c`, `zeros_and_ones` and `perfect_square_palindromes.c`

from the command line. To compile, you execute the command `gcc -std=gnu99 -Wall` *`file_name`* with or without the `-o` option to generate either the default `a.out` file, or a file name of your choice to store the executable. For instance, type

```
gcc -std=gnu99 -Wall input_output.c
```

to execute `input_output.c` as `a.out` (followed by some command line argument; otherwise the program will crash...), and type

```
gcc -std=gnu99 -Wall two_worlds.c -o cool
```

to execute `two_worlds.c` as `cool`.

When compiling the program `zeros_and_ones.c`, you also need to provide `-lm` as an option to `gcc` (to **l**ink to the **m**ath library functions, which is needed because this program uses the `pow()` function from that library). To run the program you have compiled, you either type `a.out` or the name of the executable you have chosen if you have used to `-o` option. (Refer to slides 12 and 20 of the first set of lecture notes.)

5.2 By using the mmakefile script

Another way to compile a program from the command line is by using the provided `mmakefile` script. For each of the four programs stored in `~/COMP9021/Lectures/Notes_1`, type `mmakefile` *`file_name`*. This produces a file called *Makefile* in the working directory; then type `make -B` to execute this Makefile and generate `a.out`, and finally type `a.out` to execute the program.

6 Using customised Emacs

Remember the two important *key bindings* (that you can redefine) of our customisation of Emacs:

- `\C-c o` (that is 'Control C' followed by 'o') to open a new or an existing `.c` file (you can but do not have to type the `.c` extension).
- `\C-c p` (that is 'Control C' followed by 'p') to compile, run and debug the program you are working on.

Whenever Emacs behaves strangely, typing Control G, possibly a couple of times, is likely to fix things.

Create a subdirectory `Lab_1` of `~/COMP9021/Labs`. Make `~/COMP9021/Labs/Lab_1` your working directory. Launch Emacs using either the menu or from the command line, typing `emacs&`. The `&` at the end of the command executes the command *in the background*, so that the x-term window where you typed that command gives you the prompt back and allows you to type in new commands.

Reproduce the steps shown on slides 8 to 11 of notes2.pdf , creating a file `hello.c` in the directory `~/COMP9021/Labs/Lab_1`, adding a `printf("Hello!\n");` statement, making a syntactic error and fixing it.

Then use `\C-c o` to display, compile and run the program `hello_world.c` stored in the directory `~/COMP9021/Lectures/Notes_2`.

7 Start practicing

7.1 ☞ Palindromes revisited

Copy the file `perfect_square_palindromes.c` stored in `~/COMP9021/Lectures/Notes_1` into the directory `~/COMP9021/Labs/Lab_1` and give it a new name, namely `perfect_cube_palindrome.c`. Modify this copy so that it displays all 7-digit palindromes that are perfect cubes. Do nor forget to modify the comments!

7.2 Temperatures

Copy, compile and run the following program.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int lower = 0, upper = 300, step = 20;
    /* \t denotes a tab */
    printf("fahr\tcelsius\n");
    for (int fahr = lower; fahr <= upper; fahr += step) {
        double celsius = 5. * (fahr - 32) / 9;
        /* far is printed out as a decimal number, followed by a tab,
         * followed by the floating point number celsius displayed
         * with 2 digits after the decimal point. */
        printf("%d\t%.2f\n", fahr, celsius);
    }
    return EXIT_SUCCESS;
}
```

Figure out how it works. Then modify it so that it displays instead a conversion table from celsius degrees to fahrenheit degrees, with the former ranging from 0 to 100 in steps of 10.