Exercise Manual
For

SC3103
Embedded Programming

Practical Exercise #3:
Familiarization with
Linux Environment
and
GNU toolchain

Venue: Hardware Laboratory 2
(Location: N4-01b-05)

COLLEGE OF COMPUTING AND DATA SCIENCE
NANYANG TECHNOLOGICAL UNIVERSITY

Learning Objectives

These exercises are to introduce students to the text-based Linux (Bash) shell commands and to use them to explore the environment of a Linux system. The students then practice program development techniques that include the usage of Makefile, the GDB debugger and the GPROF profiler. (These exercises can be done either on an Ubuntu based PC, or on the RPi board.)

Equipment and accessories required

i) One desktop computer with Ubuntu installed.
ii) One RPi3 (Optional)

Introduction

You will go through a sequence of tasks designed to improve your familiarity with Linux environment. Make notes here if you need to remember anything, as we will not go over them again in subsequent lab exercises and we will assume that you are already familiar with it all from this lab onwards.

1. Preparation

Turn on the PC and start the (VMware) virtualization software. Choose the Linux OS (Ubuntu) and start it. (Check with the Lab assistant if there is any login information required, or deviation from the preceding description.)

1.1. Open a terminal window by running the Terminal program:
- the application may be called “Terminal”, or “XTerm” etc.
- alternatively, press **Ctrl-Alt-T** to start the Terminal program.

Resize the terminal window such that it is of a reasonable size for you to type into.

Alternatively, you can do the exercises on a RPi board by logging remotely to the RPi board using the Secure Shell (i.e., ssh) protocol from the PC.
- In the Terminal, issue the ssh command as follows: (Check with the Lab assistant to confirm the IP address of RPi board if needed.)

```
ssh pi@192.168.3.1
```

and login using the default login & password (which are ……..?)

1.2 Create a working directory for storing your work (e.g., `ce3103/sep1` if you are in lab group SEP1) by typing the following commands in the Terminal:

```
mkdir ce3103
cd ce3103
mkdir sep1
cd sep1
```

2. Finding information in Linux

2.1 Type each of the following commands into the Terminal and observe the respective outputs shown.

- `whoami`
- `date`
- `time`
- `pwd`
- `ps`
- `ps -A`

2.2 You can find the information regarding a command by adding `man` ahead of the command, such as `man pwd`. Try these with the above commands. (Albeit you probably would just google them on Internet nowadays.)

2.3 Not all the commands used so far are Shell commands. How do you find out? The `help` command can be used to identify the commands provided by the shell.

- Type help `cd`, and then `help man`. Notice the different output.
- Try these will the commands used above.
- In fact, typing `help` will list out all the Shell commands available. Try it.

2.4 All other commands that do not belong to Shell are provided by utility programs residing in the Linux's file system, which can be found by adding the `which` command in front of the command.

- choose one of the non-Shell commands and use the `which` command to find out its detail (i.e., its path).

2.5 You can list the options and command line switches by appending –help to the command. Try the following:

- `man ps` (notice that it indicates there are option for this command)
- `ps -help`

3. Shell Basic

In practice, almost everything we do on (embedded) Linux based systems development will be issued through the Shell's command line. It is the most powerful, flexible and quickest way to accomplish anything.

3.1 Creating directories is something you had already done. You can create an empty file* in the directory by 'touching' it. (This will update the time stamp if the file already exists):

- `touch scse`

(*But the normal way of creating a file is to use a text editor – see later).

3.2 Let's view the files in the current directory: `ls`
To get more detail about the file, list the files using the long format: `ls -l`
Note the difference in the file structure that you have created above.

3.3 There is another way of getting information about files that is sometimes quite useful:

```
file *
```

3.4 There are actually some hidden files that are not shown in the directory, which you can see by using: `ls -a`
and their detail: `ls -al`

Without the -a flag, the hidden files (i.e., files whose name begins with a full stop) will not be shown.

3.5 Notice there is a '`..`' entry in the list.
Try this command and see what happen:

`cd ..` and then `ls -a`

Use the command `pwd` to understand what you have just done.
Try `cd ..` a few times and check where you end up with.

The symbol '`/`' means that you are at the root of your directory.
Now try to go back to your working directory.

3.6 You can use the following commands to rename(move) and copy file, even across directory. Try them, such as using the scse file you created earlier.

```
mv
cp
```

Tip: the <**Tab**> key is very useful in Linux operation, such as for you to select a file from the directory without entering the full file name. Try it.

3.7 *Another useful tip*: The <**Up-arrow**> and <**Down-arrow**>keys let you scroll through the commands that you have previously issued. Try it!

4 Redirection and Viewing

4.1 Type the following commands in the Terminal. Observe their output and understand what each command does.

- `echo "hello"`
- `echo "hello" > hello.txt`
- `cat hello.txt`
- `echo "world" >> hello.txt`
- `cat hello.txt`

4.2 Some useful utilities that will be useful for looking for content of file. Check out the following commands and how they can be useful.

- `less`
- `more`
- `tail`
- `head`
- `grep`

5. Text Editing and Shell Programming

Next is to learn how to use a text editor to create file (and program code). The 'conventional' text editor used in Linux/Unix is the **vi** (visual interface), and its improved version **vim**(vi IMproved), but this is definitely not for everyone's taste.

The easiest editor is probably the one built into the GUI (e.g., gedit, kate, emacs), but we will use the very simple `nano` command-line text editor to create text file known as shell script.

Shell programming can be accomplished by directly executing shell commands at the shell prompt (similar to what you have done so far). For more flexibility, the commands can be stored, in the order of required executions, in a text file called a shell script, which we can then execute.

The first line of the shell script file always begins with a #! (sha-bang) followed by the full path where the shell interpreter is located. (Note: Apart from the first line, any line starting with "#" is considered a comment line.) It is common to name the shell script with the ".sh" extension (but actual extension used is not important in Linux).

Also, you might want to open additional Terminal window(s) to do this exercise, to appreciate the conveniences of having multiple Terminals that you can easily switch between when doing various tasks.

5.1 Create a new directory `work` for the following exercises. Open a new Terminal window and type `nano`. Once the text editor starts, type the following shell script commands and then save it as "`myscript.sh`" in the `work` directory.

```
#!/bin/bash
function f1
{
  echo "hello world"
}
echo "one"
f1
echo "two"
```

5.2 Back at the command line, check that the file is really there. If not, you would have done something wrong in step 5.1. Try to figure out the mistake.

5.3 The shell script can be executed by using the command: `bash myscript.sh`

5.4 Note the need to specify the command `bash` in order to execute the shell script. But we can change the file permissions to make it (more easily) executable.

First, use the command

```
ls -l m*
```

to note the file permissions it is currently set to. (Guess what the `m*` means in the command).

5.5 Now make the file readable, writable and executable by everyone (user, group and others) as follows:

```
chmod ugo+rwx myscript.sh
```

Check how the file permissions have become now.

5.6 Finally, you can now execute the shell script directly:

```
./myscript.sh
```

Note: The '.' indicates that the file to be run is to be found in the current directory

6 Developing a simple C program

Let's use the text editor again, this time to write a C program which we will compile and run.

6.1. Enter the following program as **myprog.c** and save in the usual place.

```
include <stdio.h>
int main()
{ char i, c;
  while(1)
  { I = fread(&c, 1, 1, stdin);
    if (i>0)
       fwrite(&C, 1, 1, stdout);
  }
}
```

6.2 It is easy to compile and link (i.e., build) this program using the GCC compiler as below (and fix any error indicated in the output):

```
gcc -o myprog  mprog.c
```

6.3 Run the executable that has been produced by using the command:

```
./myprog
```

Press a few keys and see if it works the way it should be.

Then exit with Ctrl-C.

6.4 What is the size, in bytes, of your executable?

6.5 Find out the meaning of the various parameters used in the function fread() and fwrite(). (using ….?).

7 Process Control in Linux

7.1 Observe what happen when you launch the program by using the command:

```
./myprog &
```

7.2 Type the command `ps -l` and observe the output. This is the process identification (PID) number.

7.3 Use the command `fg` to bring the program back to the foreground.

7.4 You can also launch the same program multiple times, i.e., run multiple instances or processes of the same program by issuing commands such as the following:

```
./myprog &
./myprog &
./myprog &
```

7.5 Use the command `jobs -l` to check the status of processes launched, and their respective job number - shown as `[x]`.

7.6 You can bring a particular process back to the foreground by using the `fg` command with their job number:

```
fg %x
```

7.7 You can terminate a process by using the following command with its PID number (the one shown by the `ps` command):

```
kill -9 pid
```

8. Makefile

A Makefile is a text file contains instructions for the GNU make utility on how to build a program. It helps you to organize the code compilation of your program, especially useful with the program code store in multiple files. A correctly setup Makefile only compiles items that have changed as well as those that have a dependency which in turn has changed. In this exercise, you will develop a Makefile for a simple program that has code located in multiple files.

8.1 Write a C program `helloworld` that prints a message to the screen during execution. The program is to consist of three files:

hello-main.c, hello-funct1.c and a header file hello.h.

- `hello-main.c` contains the ***`main()`*** function, which calls a function ***`helloprint()`***.
- `hello-funct1.c` contains the function ***`helloprint()`*** which print the message "`Hello World from funct1!`".
- `hello.h` contains the declaration of the ***`helloprint()`***.

8.2. Create a Makefile that contains the following rules/commands, consisting of the components: Comments, macros, targets, dependencies and commands.

- Macro defines an "`object`" variable.
- Compiling rules with targets that create the respective object file for each of the C file.
- Linking rule with an '`all`' target to create the program executive helloworld, based on the dependency of each of the C program file.
- A target '`clean`' to remove appropriate files such that compilation can be redone afresh.

8.3 Execute the **`make`** command to check that your Makefile can properly support the various commands listed in the file:

- Produce the executable file and test that it executes as expected.
- Delete one of the object files, and run **`make`** to recompile the executable (how do you know that the output is the recompiled version, and not the previous version?).
- 'Update' one of the C file by using the command touch. Check that the recompilation is actually performed.
- Execute the 'clean' command to delete all the object files.

9. GNU Debugger - GDB

Debugging is a very important step during the development cycle of a software. A good debugging tool is hence an important asset to have. In this exercise, you will practice how to use the GNU debugging tool, GDB to investigate the execution detail of a program, such as by single stepping the code and examining the various variables used in the program.

9.1 Using the **`helloworld`** program developed in section 8, modify the code to meet the following specifications:

- The program prints a message when the user presses a key on the **`stdin`**.
- The message shows a number indicating the number of times the user has pressed the keys.
- A new C file **`hello-funct2.c`** with a function that uses a variable '**`count`**' to keep track of the number of times the user has pressed the key.

9.2 In the Makefile add a new linking target '**`debug`**' to enable GDB debugging of the program executable, which is to be called helloworld-d.

9.3 Execute **make** to generate helloworld and helloworld-d. Note the size of their executable files.

9.4 Use the GDB to launch the helloworld-d executable file (i.e., using the command **`gdb helloworld-d`**). Then run the program under the control of GDB, using its various commands such as those shown below.

- List the program code using the command '**`l`**'.
- Identify a suitable code line *`xx`* to place a break point (e.g., the **`main()`**), using the command '**`break`** *`xx`*'
- Start the program using '**`run`**'.
- Single-step the program using '**`n`**' **`or`** '**`step`**'. (observe whether there is any difference between the two).
- Resume executing the program using '**`cont`**'.
- Monitor the value of the variable *`count`* using '**`watch`** *`count`*'

10 GNU Profiler - GPROF

Profiling is another very useful function to perform during the development of a program. A profiling tool enables one to analyse and determine the time consumed by various parts used in the program code. This allows one to re-code the parts that are most time consuming in order to improve program performance. In this exercise, you will practice how to use the GNU profiling tool, GPROF to analyse the execution pattern of a program.

10.1 Using the program code developed in section 8 and 9, change the code to perform the following operations.

- In the `main()` function, print a message "`Hello World from main!`", which then execute a delay using a counter with value of **`0x5fffff`**. It then calls the functions in **`hello-funct1.c`** and **`hello-funct2.c`** files. Print a message "`Bye!`" before the program terminates.
- For the function in **`hello-funct1.c`**, print a message "`Hello World from funct1!`", then execute a delay using a counter with value of **`0x6fffffff`**.
- For the function in **`hello-funct2.c`**, print a message "`Hello World from funct2!`", then execute a delay using a counter with value of **`0x7fffffff`**.

10.2 Change the Makefile to include the profiling option during the compilation of the program.

- Compile the program (and fix any error).
- Execute the program and check that the outputs are generated accordingly.

10.3 Use the GPROF tool to generate an analysis file that contains the profiling information. Study the analysis file and check that the detail corresponds to what is expected from the program execution.